Power ISA™
Version 3.0 B

March 29, 2017

IBM®

© Copyright International Business Machines Corporation 1994 - 2017. All rights reserved.

Printed in the United States of America March, 2017

By downloading the POWER® Instruction set Architecture (“ISA”) Specification, you agree to be bound by the terms and conditions of this agreement.

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Other company, product, and service names may be trademarks or service marks of others.

All information contained in this document is subject to change without notice. The products described in this document are NOT intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury, or catastrophic property damage. The information contained in this document does not affect or change IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments may vary.

While the information contained herein is believed to be accurate, such information is preliminary, and should not be relied upon for accuracy or completeness, and no representations or warranties of accuracy or completeness are made.

Note: This document contains information on products in the design, sampling and/or initial production phases of development. This information is subject to change without notice. Verify with your IBM field applications engineer that you have the latest version of this document before finalizing a design.

You may use this documentation solely for developing technology products compatible with Power Architecture® in support of growing the POWER ecosystem. You may not modify this documentation. You may distribute the documentation to suppliers and other contractors hired by you solely to produce your technology products compatible with Power Architecture® technology and to your customers (either directly or indirectly through your resellers) in conjunction with their use and instruction of your technology products compatible with Power Architecture® technology. This agreement does not include rights to create a CPU design to run the POWER ISA unless such rights have been granted by IBM under a separate agreement. The POWER ISA specification is protected by copyright and the practice or implementation of the information herein may be protected by one or more patents or pending patent applications. No other license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN “AS IS” BASIS. IBM makes no representations or warranties, either express or implied, including but not limited to, warranties of merchantability, fitness for a particular purpose, or non-infringement, or that any practice or implementation of the IBM documentation will not infringe any third party patents, copyrights, trade secrets, or other rights. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

IBM Systems and Technology Group
2070 Route 52, Bldg. 330
Hopewell Junction, NY 12533-6351

The IBM home page can be found at **ibm.com®**.

The following paragraph does not apply to the United Kingdom or any country or state where such provisions are inconsistent with local law.

The specifications in this manual are subject to change without notice. This manual is provided “AS IS”. International Business Machines Corp. makes no warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

International Business Machines Corp. does not warrant that the contents of this publication or the accompanying source code examples, whether individually or as one or more groups, will meet your requirements or that the publication or the accompanying source code examples are error-free.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication.

Address comments to IBM Corporation, 11400 Burnett Road, Austin, Texas 78758-3493. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

IBM®
Power ISA
PowerPC®
Power Architecture
PowerPC Architecture
Power Family
RISC/System 6000®
POWER®
POWER2
POWER4
POWER4+
POWER5
POWER5+
POWER6®
POWER7®
POWER8®
POWER9™
System/370
System z

Notice to U.S. Government Users—Documentation Related to Restricted Rights—Use, duplication or disclosure is subject to restrictions set fourth in GSA ADP Schedule Contract with IBM Corporation.

Preface

The roots of the Power ISA (Instruction Set Architecture) extend back over a quarter of a century, to IBM Research. The POWER (Performance Optimization With Enhanced RISC) Architecture was introduced with the RISC System/6000 product family in early 1990. In 1991, Apple, IBM, and Motorola began the collaboration to evolve to the PowerPC Architecture, expanding the architecture's applicability. In 1997, Motorola and IBM began another collaboration, focused on optimizing PowerPC for embedded systems, which produced Book E.

In 2006, Freescale and IBM collaborated on the creation of the Power ISA Version 2.03, which represented the reunification of the architecture by combining Book E content with the more general purpose PowerPC Version 2.02. The resulting architecture included environment-specific privileged architecture optimizations (two Book IIIs) and optional application-specific facilities (categories) as extensions to a pervasive base architecture.

Power ISA Version 3.0 B focuses this integration by choosing a single Book III and a set of widely used categories to become part of the base architecture for all forward-looking Power implementations. All other optional architecture categories have been eliminated to ensure increased application portability between Power processors. Legacy embedded applications that require the eliminated material will continue to use V. 2.07B.

The Power ISA Version 3.0 B consists of three books and a set of appendices.

Book I, *Power ISA User Instruction Set Architecture*, covers the base instruction set and related facilities available to the application programmer.

Book II, *Power ISA Virtual Environment Architecture,* defines the storage model and other instructions and facilities that enable the application programmer to create multithreaded programs and programs that interact with certain physical realities of the computing environment.

Book III, *Power ISA Operating Environment Architecture,* defines the supervisor instructions and related facilities.

As used in this document, the term "Power ISA" refers to the instructions and facilities described in Books I, II, and III.

Change bars have been included in the body of this document to indicate changes from the Power ISA Version 2.07B. Change bars may be omitted for changes associated with removing obsolete categories and the second Book III.

Summary of Changes in Power ISA Version 3.0 B

This document is Version 3.0 B of the Power ISA. It is intended to supersede and replace version 2.07B. Any product descriptions that reference a version of the architecture are understood to reference the latest version. This version was created by making miscellaneous corrections and by applying the following requests for change (RFCs) to Power ISA Version 2.07B. Change bars in this summary of changes indicate new, changed, or removed changes relative to V. 3.0.

Instruction Fusion: Specifies instruction sequences that, when placed consecutively in the program, are expected to provide improved performance.

Hashing Support Operations: Adds new Count Trailing Zeros and Modulo instructions

Decimal Integer Support Operations: Adds new BCD support instructions, including variable-length load/store instructions for bcd values, new format conversion instructions between BCD and National decimal, zoned decimal, and 128-bit signed integer formats. new BCDtruncate, round, and shift instructions, new BCD sign digit manipulation instructions. Also adds multiply-by-10 instructions to faciliate binary-to-decimal conversion for printf. Corrected functionality of Decimal Shift and Round (bcdsr.) instruction.

Decimal Floating-Point Support Operations: Add immediate forms of DFP Test Significance instructions.

Binary Floating-Point Support Operations: Adds new binary floating-point support instructions (e.g., exponent and significand extraction and insertion) to enhance implementation of math libraries.

Quad-Precision Binary Floating-Point Operations: Add new instructions to support IEEE-754-2008 binary128 floating-point.

String Operations (FXU option): Adds instructions to accelerate character testing functions.

String Operations (VSU option): Adds instructions to accelerate string processing and targeted character extraction.

Vector Half-Precision Floating-Point Support Operations: Adds support for IEEE-754-2008 binary16 floating-point as a transport format.

128-bit SIMD Video Compression Operations: Adds instructions to accelerate motion estimation.

128-bit SIMD FXU Operations: Adds remaining 32-bit and 64-bit FXU functionality to vector instruction set.

128-bit SIMD Miscellaneous Operations: Enhances support for Little-Endian processing with new load/store instructions and new permute-class instructions, new byte and halfword element load/store instructions, and vector element insertion/extraction.

System Call Extension: Provides a new form of system call that can direct execution to one of a number of locations and that provides other enhancements.

PC-Relative Addressing: Specifies a new instruction that adds an immediate value to the program counter and writes it to the destination register in preparation for use with a D-Form *Load* instructon.

Hypervisor ***msgsnd*** Instruction Enhancements: Extends the ***msgsnd*** instruction so that messages can be sent throughout the system.

Performance Monitor Enhancements: Reserves a special no-op instruction for use by the Performance Monitor, and increases the scope of control of the Performance Monitor bit of the Hypervisor Facility Status and Control register.

Radix Tree and Related MMU Extensions: Adds support for the radix tree style of MMU with full virtualization and related control mechanisms that manage its coexistence with the HPT. Also adds a ***tlbie*** variant that invalidates multiple consecutive translations.

Copy-Paste Facility: Adds support for a new facility that enables an application to initiate accelerator operations.

Optimizing ***mtspr*** Sequences: Reserves an SPR to be used in a no-op ***mtspr*** to indicate the beginning of a sequence of ***mtspr***s that can be done without synchronizing each one independently.

Atomic Memory Operations: Adds support for a new facility that performs simple atomic operations directly in memory to avoid bringing the line through the cache hierarchy when another core is likely to be the next user.

Event-Based Branch Extension: Adds External Event-Based Branch exception and status bits to the BESCR.

Processor Compatibility Register: Adds a new V 2.07 bit to the PCR that controls the availability facilities in problem state that are introduced in this level of the architecture.

Atomicity and Alignment Enhancements: Limits the number of disjoint atomic storage accesses that are allowed for various non-atomic storage accesses.

Power-Saving Mode: Replaces the existing power-saving mode instructions with a single ***stop*** instruction, and enables the operating system to enter a limited set of power-saving levels without hypervisor involvement.

D-form VSX Floating-Point Storage Access Instructions: Adds base+displacement forms of VSR load and store instructions.

Integer Multiply-Add Instructions: Adds new integer multiply-add instructions to accelerate arbitrary-length multiplication.

msgsndp Hypervisor Facility Availability Interrupt: Adds a new HFSCR bit to control the availability of the ***msgsndp*** instruction and the associated control registers.

VSX Permute: Adds new pernute instructions that can address all 64 VSRs.

Array Index Support: Enhance support for mixed-data-type addressing into arrays (e.g., base + 32-bit index)

Hypervisor Virtualization Interrupt: Defines a new exception and corresponding interrupt that is caused by events external to the processor that relate to virtualization.

wait Instruction Enhancements: Improves the capabilities of the ***wait*** instruction so that resumption of processing can occur due to event-based branches and external signals.

Decrementer and Hypervisor Decrementer Enahncements: Defines a new mode bit in the LPCR that enables additional Decrementer and Hypervisor Decrementer bits in order to increase the time between the associated interrupts.

Deliver A Random Number: Adds a new instruction to place a random number in a GPR in one of three formats.

Data Storage Interrupt Status Register for Alignment Interrupt: Simplifies the Alignment interrupt by removing the Data Storage Interrupt Status Register (DSISR) from the set of registers modified by the Alignment interrupt.

CA32 & OV32 and Move XER to CR Extended: Added support for 32-bit CA & OV status in 64-bit mode for dynamically-typed languages.

VSX Shift Variable: Accelerate parallel element extraction from packed vectors of arbitrary-width-element values.

Enhanced Virtualization for Linux: Delivers exceptions caused by the OS attempting to use hypervisor instructions and SPRs to the hypervisor instead of the OS. Accesses to unimplemented SPRs by the OS newly cause interrupts that are also directed to the hypervisor.

Synchronizing Messages and Storage Updates: Adds a new instruction to make latent storage updates from another thread accessible after receiving a Directed Hypervisor Doorbell interrupt from that thread.

VSX Conditional: Adds new instruction to accelerate conditional, maximum, and minimum operations. Withdrew xscmpnedp, xvcmpnesp[.], and xvcmpnedp[.] instructions introduced in v3.0.

FXU & Vector Extensions for Blockchain Support: Two new instructions (addex and vmsumudm) introduced to accelerate arbitrary-precision integer arithmetic, and specifically to accelerate Blockchain's implementation of elliptical curve encryption signature algorithm. The OV bit is employed to provide an additional, independent carry status bit, allowing software to parallelize carry propagation.

Miscellaneous Changes: Makes minor clarifications, corrections, and editorial enhancements.

FX/VSX/Vector Miscellaneous: Editorial cleanup of Book I chapters 4, 5, and 7.

TM Multithread Overflow: Adds a bit to TEXASR to enable software to differentiate single thread footprint overflow from that aggravated by multiple threads competing for footprint.

Lightweight mffs: Modifications of mffs to accelerate saving/setting/restoring floating-point environments (e.g., rounding modes, exception trapping enables) common in math libraries that require overriding the environment.

Table of Contents

Preface. v
Summary of Changes in Power ISA Version 3.0 B vi

Table of Contents ix

Book I:

Power ISA User Instruction Set Architecture. 1

Chapter 1. Introduction 3
1.1 Overview. 3
1.2 Instruction Mnemonics and Operands 3
1.3 Document Conventions 3
1.3.1 Definitions 3
1.3.2 Notation 4
1.3.3 Reserved Fields, Reserved Values, and Reserved SPRs 5
1.3.4 Description of Instruction Operation 6
1.3.5 Phased-Out Facilities 8
1.4 Processor Overview 9
1.5 Computation modes 10
1.6 Instruction Formats. 11
1.6.1 A-FORM 12
1.6.2 B-FORM 12
1.6.3 D-FORM 12
1.6.4 DQ-FORM 12
1.6.5 DS-FORM 12
1.6.6 DX-FORM 12
1.6.7 I-FORM 12
1.6.8 M-FORM 12
1.6.9 MD-FORM 12
1.6.10 MDS-FORM. 12
1.6.11 SC-FORM 12
1.6.12 VA-FORM 12
1.6.13 VC-FORM 12
1.6.14 VX-FORM 13
1.6.15 X-FORM 13
1.6.16 XFL-FORM 15
1.6.17 XFX-FORM 15
1.6.18 XL-FORM 15
1.6.19 XO-FORM 15
1.6.20 XS-FORM. 15
1.6.21 XX2-FORM. 15
1.6.22 XX3-FORM. 15
1.6.23 XX4-FORM. 15
1.6.24 Z22-FORM 15
1.6.25 Z23-FORM 16
1.7 Instruction Fields 16
1.8 Classes of Instructions 22
1.8.1 Defined Instruction Class 22
1.8.2 Illegal Instruction Class 22
1.8.3 Reserved Instruction Class 22
1.9 Forms of Defined Instructions 23
1.9.1 Preferred Instruction Forms 23
1.9.2 Invalid Instruction Forms 23
1.9.3 Reserved-no-op Instructions 23
1.10 Exceptions. 23
1.11 Storage Addressing 24
1.11.1 Storage Operands 24
1.11.2 Instruction Fetches 26
1.11.3 Effective Address Calculation . . . 27

Chapter 2. Branch Facility 29
2.1 Branch Facility Overview. 29
2.2 Instruction Execution Order. 29
2.3 Branch Facility Registers 30
2.3.1 Condition Register 30
2.3.2 Link Register 32
2.3.3 Count Register 32
2.3.4 Target Address Register. 32
2.4 Branch Instructions 33
2.5 Condition Register Instructions 40
2.5.1 Condition Register Logical Instructions 40
2.5.2 Condition Register Field Instruction . 41
2.6 System Call Instructions. 42

Chapter 3. Fixed-Point Facility. . . . 45
3.1 Fixed-Point Facility Overview 45
3.2 Fixed-Point Facility Registers 45
3.2.1 General Purpose Registers 45
3.2.2 Fixed-Point Exception Register 45
3.2.3 VR Save Register. 46
3.3 Fixed-Point Facility Instructions . . . 47

3.3.1 Fixed-Point Storage Access Instructions 47
3.3.1.1 Storage Access Exceptions 47
3.3.2 Fixed-Point Load Instructions 47
3.3.2.1 64-bit Fixed-Point Load Instructions 52
3.3.3 Fixed-Point Store Instructions 54
3.3.3.1 64-bit Fixed-Point Store Instructions 57
3.3.4 Fixed Point Load and Store Quadword Instructions 58
3.3.5 Fixed-Point Load and Store with Byte Reversal Instructions 60
3.3.5.1 64-Bit Load and Store with Byte Reversal Instructions 61
3.3.6 Fixed-Point Load and Store Multiple Instructions 62
3.3.7 Fixed-Point Move Assist Instructions [Phased Out]. 63
3.3.8 Other Fixed-Point Instructions. . . . 66
3.3.9 Fixed-Point Arithmetic Instructions 67
3.3.9.1 64-bit Fixed-Point Arithmetic Instructions 79
3.3.10 Fixed-Point Compare Instructions. . 84
3.3.10.1 Character-Type Compare Instructions 87
3.3.11 Fixed-Point Trap Instructions. . . . 89
3.3.11.1 64-bit Fixed-Point Trap Instructions 91
3.3.12 Fixed-Point Select. 91
3.3.13 Fixed-Point Logical Instructions .92
3.3.13.1 64-bit Fixed-Point Logical Instructions 99
3.3.14 Fixed-Point Rotate and Shift Instructions 101
3.3.14.1 Fixed-Point Rotate Instructions . . 101
3.3.14.1.1 64-bit Fixed-Point Rotate Instructions 104
3.3.14.2 Fixed-Point Shift Instructions .107
3.3.14.2.1 64-bit Fixed-Point Shift Instructions 109
3.3.15 Binary Coded Decimal (BCD) Assist Instructions. 111
3.3.16 Move To/From Vector-Scalar Register Instructions 112
3.3.17 Move To/From System Register Instructions 117

Chapter 4. Floating-Point Facility 123

4.1 Floating-Point Facility Overview. . . 123
4.2 Floating-Point Facility Registers. . . 124
4.2.1 Floating-Point Registers 124
4.2.2 Floating-Point Status and Control Register 124
4.3 Floating-Point Data. 127
4.3.1 Data Format. 127
4.3.2 Value Representation 127
4.3.3 Sign of Result 129
4.3.4 Normalization and Denormalization 129
4.3.5 Data Handling and Precision . . . 129
4.3.5.1 Single-Precision Operands. . . 129
4.3.5.2 Integer-Valued Operands 130
4.3.6 Rounding 131
4.4 Floating-Point Exceptions. 132
4.4.1 Invalid Operation Exception. . . . 134
4.4.1.1 Definition. 134
4.4.1.2 Action 134
4.4.2 Zero Divide Exception 134
4.4.2.1 Definition. 134
4.4.2.2 Action 135
4.4.3 Overflow Exception 135
4.4.3.1 Definition. 135
4.4.3.2 Action 135
4.4.4 Underflow Exception 136
4.4.4.1 Definition. 136
4.4.4.2 Action 136
4.4.5 Inexact Exception 136
4.4.5.1 Definition. 136
4.4.5.2 Action 136
4.5 Floating-Point Execution Models . 137
4.5.1 Execution Model for IEEE Operations 137
4.5.2 Execution Model for Multiply-Add Type Instructions 139
4.6 Floating-Point Facility Instructions 140
4.6.1 Floating-Point Storage Access Instructions 140
4.6.1.1 Storage Access Exceptions . . 140
4.6.2 Floating-Point Load Instructions 140
4.6.3 Floating-Point Store Instructions 144
4.6.4 Floating-Point Load and Store Double Pair Instructions [Phased-Out] . . . 148
4.6.5 Floating-Point Move Instructions 150
4.6.6 Floating-Point Arithmetic Instructions 152
4.6.6.1 Floating-Point Elementary Arithmetic Instructions 152
4.6.6.2 Floating-Point Multiply-Add Instructions 157
4.6.7 Floating-Point Rounding and Conversion Instructions 159
4.6.7.1 Floating-Point Rounding Instruction 159
4.6.7.2 Floating-Point Convert To/From Integer Instructions 159
4.6.7.3 Floating Round to Integer Instructions 165
4.6.8 Floating-Point Compare Instructions 167

4.6.9 Floating-Point Select Instruction 168
4.6.10 Floating-Point Status and Control Register Instructions 170

Chapter 5. Decimal Floating-Point . . 175

5.1 Decimal Floating-Point (DFP) Facility Overview 175
5.2 DFP Register Handling 176
5.2.1 DFP Usage of Floating-Point Registers 176
5.3 DFP Support for Non-DFP Data Types 178
5.4 DFP Number Representation 179
5.4.1 DFP Data Format. 179
5.4.1.1 Fields Within the Data Format 179
5.4.1.2 Summary of DFP Data Formats . . 180
5.4.1.3 Preferred DPD Encoding 181
5.4.2 Classes of DFP Data 181
5.5 DFP Execution Model 182
5.5.1 Rounding 182
5.5.2 Rounding Mode Specification . . 183
5.5.3 Formation of Final Result. 183
5.5.3.1 Use of Ideal Exponent 183
5.5.4 Arithmetic Operations 184
5.5.4.1 Sign of Arithmetic Result 184
5.5.5 Compare Operations 184
5.5.6 Test Operations 184
5.5.7 Quantum Adjustment Operations 184
5.5.8 Conversion Operations 185
5.5.8.1 Data-Format Conversion 185
5.5.8.2 Data-Type Conversion 185
5.5.9 Format Operations. 185
5.5.10 DFP Exceptions. 185
5.5.10.1 Invalid Operation Exception . 187
5.5.10.2 Zero Divide Exception 188
5.5.10.3 Overflow Exception. 189
5.5.10.4 Underflow Exception. 189
5.5.10.5 Inexact Exception 190
5.5.11 Summary of Normal Rounding And Range Actions. 191
5.6 DFP Instruction Descriptions 193
5.6.1 DFP Arithmetic Instructions 193
5.6.2 DFP Compare Instructions 197
5.6.3 DFP Test Instructions. 200
5.6.4 DFP Quantum Adjustment Instructions 203
5.6.5 DFP Conversion Instructions . . . 212
5.6.5.1 DFP Data-Format Conversion Instructions 212
5.6.5.2 DFP Data-Type Conversion Instructions 215
5.6.6 DFP Format Instructions 217
5.6.7 DFP Instruction Summary 221

Chapter 6. Vector Facility 223

6.1 Vector Facility Overview 223
6.2 Chapter Conventions 223
6.2.1 Description of Instruction Operation. 223
6.3 Vector Facility Registers 232
6.3.1 Vector Registers. 232
6.3.2 Vector Status and Control Register . 232
6.3.3 VR Save Register. 233
6.4 Vector Storage Access Operations 234
6.4.1 Accessing Unaligned Storage Operands. 236
6.5 Vector Integer Operations 237
6.5.1 Integer Saturation. 237
6.6 Vector Floating-Point Operations . 239
6.6.1 Floating-Point Overview. 239
6.6.2 Floating-Point Exceptions 239
6.6.2.1 NaN Operand Exception 239
6.6.2.2 Invalid Operation Exception . . 240
6.6.2.3 Zero Divide Exception 240
6.6.2.4 Log of Zero Exception 240
6.6.2.5 Overflow Exception 240
6.6.2.6 Underflow Exception 240
6.7 Vector Storage Access Instructions 241
6.7.1 Storage Access Exceptions 241
6.7.2 Vector Load Instructions. 242
6.7.3 Vector Store Instructions 245
6.7.4 Vector Alignment Support Instructions 247
6.8 Vector Permute and Formatting Instructions 248
6.8.1 Vector Pack and Unpack Instructions 248
6.8.2 Vector Merge Instructions 255
6.8.3 Vector Splat Instructions 258
6.8.4 Vector Permute Instruction. 260
6.8.5 Vector Select Instruction 261
6.8.6 Vector Shift Instructions 262
6.8.7 Vector Extract Element Instructions . 267
6.8.8 Vector Insert Element Instructions . . 268
6.9 Vector Integer Instructions 269
6.9.1 Vector Integer Arithmetic Instructions 269
6.9.1.1 Vector Integer Add Instructions 269
6.9.1.2 Vector Integer Subtract Instructions 275
6.9.1.3 Vector Integer Multiply Instructions 281
6.9.1.4 Vector Integer Multiply-Add/Sum Instructions 285
6.9.1.5 Vector Integer Sum-Across Instructions 290

6.9.1.6 Vector Integer Negate Instructions. 293
6.9.2 Vector Extend Sign Instructions .294
6.9.2.1 Vector Integer Average Instructions 295
6.9.2.2 Vector Integer Absolute Difference Instructions 297
6.9.2.3 Vector Integer Maximum and Minimum Instructions 299
6.9.3 Vector Integer Compare Instructions. 303
6.9.4 Vector Logical Instructions 312
6.9.5 Vector Parity Byte Instructions . .314
6.9.6 Vector Integer Rotate and Shift Instructions 315
6.10 Vector Floating-Point Instruction Set . 321
6.10.1 Vector Floating-Point Arithmetic Instructions 321
6.10.2 Vector Floating-Point Maximum and Minimum Instructions 323
6.10.3 Vector Floating-Point Rounding and Conversion Instructions 324
6.10.4 Vector Floating-Point Compare Instructions 328
6.10.5 Vector Floating-Point Estimate Instructions 331
6.11 Vector Exclusive-OR-based Instructions 333
6.11.1 Vector AES Instructions. 333
6.11.2 Vector SHA-256 and SHA-512 Sigma Instructions 335
6.11.3 Vector Binary Polynomial Multiplication Instructions 336
6.11.4 Vector Permute and Exclusive-OR Instruction. 338
6.12 Vector Gather Instruction 339
6.13 Vector Count Leading Zeros Instructions 340
6.14 Vector Count Trailing Zeros Instructions 341
6.14.1 Vector Count Leading/Trailing Zero LSB Instructions 342
6.14.2 Vector Extract Element Instructions 343
6.15 Vector Population Count Instructions. 345
6.16 Vector Bit Permute Instruction . . .346
6.17 Decimal Integer Instructions. 347
6.17.1 Decimal Integer Arithmetic Instructions 347
6.17.2 Decimal Integer Format Conversion Instructions 350
6.17.3 Decimal Integer Sign Manipulation Instructions 356
6.17.4 Decimal Integer Shift and Round Instructions 357
6.17.5 Decimal Integer Truncate Instructions 360
6.18 Vector Status and Control Register Instructions 362

Chapter 7. Vector-Scalar Floating-Point Operations 363

7.1 Introduction. 363
7.1.1 Overview of the Vector-Scalar Extension 363
7.1.1.1 Compatibility with Floating-Point and Decimal Floating-Point Operations 363
7.1.1.2 Compatibility with Vector Operations 363
7.2 VSX Registers 364
7.2.1 Vector-Scalar Registers. 364
7.2.1.1 Floating-Point Registers 364
7.2.1.2 Vector Registers 366
7.2.2 Floating-Point Status and Control Register. 367
7.3 VSX Operations 372
7.3.1 VSX Floating-Point Arithmetic Overview. 372
7.3.2 VSX Floating-Point Data 373
7.3.2.1 Data Format 373
7.3.2.2 Value Representation 375
7.3.2.3 Sign of Result 376
7.3.2.4 Normalization and Denormalization 377
7.3.2.5 Data Handling and Precision . 377
7.3.2.6 Rounding 381
7.3.3 VSX Floating-Point Execution Models 384
7.3.3.1 VSX Execution Model for IEEE Operations 384
7.3.3.2 VSX Execution Model for Multiply-Add Type Instructions 385
7.4 VSX Floating-Point Exceptions. . . 387
7.4.1 Floating-Point Invalid Operation Exception 390
7.4.1.1 Definition. 390
7.4.1.2 Action for VE=1. 390
7.4.1.3 Action for VE=0. 392
7.4.2 Floating-Point Zero Divide Exception 401
7.4.2.1 Definition. 401
7.4.2.2 Action for ZE=1. 401
7.4.2.3 Action for ZE=0. 402
7.4.3 Floating-Point Overflow Exception . 404
7.4.3.1 Definition. 404
7.4.3.2 Action for OE=1 404
7.4.3.3 Action for OE=0 407

7.4.4 Floating-Point Underflow Exception. 409
7.4.4.1 Definition. 409
7.4.4.2 Action for UE=1 409
7.4.4.3 Action for UE=0 411
7.4.5 Floating-Point Inexact Exception 414
7.4.5.1 Definition. 414
7.4.5.2 Action for XE=1. 414
7.4.5.3 Action for XE=0. 417
7.5 VSX Storage Access Operations . 420
7.5.1 Accessing Aligned Storage Operands 420
7.5.2 Accessing Unaligned Storage Operands 421
7.5.3 Storage Access Exceptions 422
7.6 VSX Instruction Set 423
7.6.1 VSX Instruction Set Summary . . 423
7.6.1.1 VSX Storage Access Instructions . 423
7.6.1.2 VSX Binary Floating-Point Sign Manipulation Instructions 425
7.6.1.3 VSX Binary Floating-Point Arithmetic Instructions. 425
7.6.1.4 VSX Binary Floating-Point Compare Instructions 428
7.6.1.5 VSX Binary Floating-Point Round to Shorter Precision Instructions 429
7.6.1.6 VSX Binary Floating-Point Convert to Shorter Precision Instructions 429
7.6.1.7 VSX Binary Floating-Point Convert to Longer Precision Instructions 429
7.6.1.8 VSX Binary Floating-Point Round to Integral Instructions. 430
7.6.1.9 VSX Binary Floating-Point Convert To Integer Instructions. 430
7.6.1.10 VSX Binary Floating-Point Convert From Integer Instructions 431
7.6.1.11 VSX Binary Floating-Point Math Support Instructions 431
7.6.1.12 VSX Vector Logical Instructions . 432
7.6.1.13 VSX Vector Permute-class Instructions 432
7.6.2 VSX Instruction Description Conventions 434
7.6.2.1 VSX Instruction RTL Operators 434
7.6.2.2 VSX Instruction RTL Function Calls 435
7.6.3 VSX Instruction Descriptions . . . 480

Appendix A. Suggested Floating-Point Models 775
A.1 Floating-Point Round to Single-Precision Model. 775
A.2 Floating-Point Convert to Integer Model 779
A.3 Floating-Point Convert from Integer Model. 782
A.4 Floating-Point Round to Integer Model 784

Appendix B. Densely Packed Decimal 787
B.1 BCD-to-DPD Translation. 787
B.2 DPD-to-BCD Translation. 787
B.3 Preferred DPD encoding. 788

Appendix C. Assembler Extended Mnemonics 791
C.1 Symbols 791
C.2 Branch Mnemonics. 792
C.2.1 BO and BI Fields 792
C.2.2 Simple Branch Mnemonics 792
C.2.3 Branch Mnemonics Incorporating Conditions 793
C.2.4 Branch Prediction 794
C.3 Condition Register Logical Mnemonics 795
C.4 Subtract Mnemonics. 795
C.4.1 Subtract Immediate 795
C.4.2 Subtract 795
C.5 Compare Mnemonics 796
C.5.1 Doubleword Comparisons 796
C.5.2 Word Comparisons 796
C.6 Trap Mnemonics. 797
C.7 Integer Select Mnemonics 798
C.8 Rotate and Shift Mnemonics 799
C.8.1 Operations on Doublewords . . . 799
C.8.2 Operations on Words. 800
C.9 Move To/From Special Purpose Register Mnemonics 801
C.10 Miscellaneous Mnemonics 802

Book II:

Power ISA Virtual Environment Architecture 807

Chapter 1. Storage Model 809
1.1 Definitions 809
1.2 Introduction 810
1.3 Virtual Storage 810
1.4 Single-Copy Atomicity 811
1.5 Cache Model. 812
1.6 Storage Control Attributes 812
1.6.1 Write Through Required 813
1.6.2 Caching Inhibited 813
1.6.3 Memory Coherence Required . 813
1.6.4 Guarded 813
1.6.5 Strong Access Order 814

1.7 Shared Storage 814
1.7.1 Storage Access Ordering 815
1.7.2 Storage Ordering of Copy/Paste-Initiated Data Transfers 817
1.7.3 Storage Ordering of I/O Accesses. . . 817
1.7.4 Atomic Update. 817
1.7.4.1 Reservations 818
1.7.4.2 Forward Progress 820
1.8 Transactions. 821
1.8.1 Rollback-Only Transactions 823
1.9 Instruction Storage. 823
1.9.1 Concurrent Modification and Execution of Instructions 825

Chapter 2. Performance Considerations and Instruction Restart 827
2.1 Performance-Optimized Instruction Sequences 827
2.1.1 Load and Store Operations. 828
2.1.2 32-Bit Constant Generation. 831
2.1.3 Sign and Zero Extension 831
2.1.4 Load/Store Addressing Relative to Program Counter 832
2.1.5 Destructive Operation Operand Preservation 833
2.2 Instruction Restart 834

Chapter 3. Management of Shared Resources 835
3.1 Program Priority Registers. 835
3.2 “or” Instruction 835

Chapter 4. Storage Control Instructions 837
4.1 Parameters Useful to Application Programs 837
4.2 Data Stream Control Register (DSCR) 837
4.3 Cache Management Instructions .839
4.3.1 Instruction Cache Instructions. . .840
4.3.2 Data Cache Instructions 841
4.3.2.1 Obsolete Data Cache Instructions . 852
4.3.3 “or” Instruction. 853
4.4 Copy-Paste Facility 854
4.5 Atomic Memory Operations 857
4.5.1 Load Atomic 857
4.5.2 Store Atomic 861
4.6 Synchronization Instructions 863
4.6.1 Instruction Synchronize Instruction . . 863
4.6.2 Load and Reserve and Store Conditional Instructions 863
4.6.2.1 64-Bit Load and Reserve and Store Conditional Instructions. 869
4.6.2.2 128-bit Load and Reserve Store Conditional Instructions. 871
4.6.3 Memory Barrier Instructions . . . 873
4.6.4 Wait Instruction 876

Chapter 5. Transactional Memory Facility 877
5.1 Transactional Memory Facility Overview. 877
5.1.1 Definitions 878
5.2 Transactional Memory Facility States. 880
5.2.1 The TDOOMED Bit 882
5.3 Transaction Failure. 882
5.3.1 Causes of Transaction Failure. . 882
5.3.2 Recording of Transaction Failure 885
5.3.3 Handling of Transaction Failure. 885
5.4 Transactional Memory Facility Registers 886
5.4.1 Transaction Failure Handler Address Register (TFHAR) 886
5.4.2 Transaction EXception And Status Register (TEXASR). 886
5.4.3 Transaction Failure Instruction Address Register (TFIAR). 889
5.5 Transactional Facility Instructions. 890

Chapter 6. Time Base 897
6.1 Time Base Instructions. 898

Chapter 7. Event-Based Branch Facility 901
7.1 Event-Based Branch Overview. . . 901
7.2 Event-Based Branch Registers . . 902
7.2.1 Branch Event Status and Control Register. 902
7.2.2 Event-Based Branch Handler Register 903
7.2.3 Event-Based Branch Return Register 904
7.3 Event-Based Branch Instructions . 905

Chapter 8. Branch History Rolling Buffer. 907
8.1 Branch History Rolling Buffer Entry Format. 908
8.2 Branch History Rolling Buffer Instructions 909

Appendix A. Assembler Extended Mnemonics 911
A.1 Data Cache Block Touch [for Store] Mnemonics 911
A.2 Data Cache Block Flush Mnemonics . 911
A.3 Or Mnemonics 911
A.4 Load and Reserve Mnemonics 911
A.5 Synchronize Mnemonics 912
A.6 Wait Mnemonics 912
A.7 Transactional Memory Instruction Mnemics 912
A.8 Move To/From Time Base Mnemonics 912
A.9 Return From Event-Based Branch Mnemonic 912

Appendix B. Programming Examples for Sharing Storage 913
B.1 Atomic Update Primitives 913
B.2 Lock Acquisition and Release, and Related Techniques. 915
B.2.1 Lock Acquisition and Import Barriers 915
B.2.1.1 Acquire Lock and Import Shared Storage 915
B.2.1.2 Obtain Pointer and Import Shared Storage 915
B.2.2 Lock Release and Export Barriers. . 916
B.2.2.1 Export Shared Storage and Release Lock 916
B.2.2.2 Export Shared Storage and Release Lock using lwsync 916
B.2.3 Safe Fetch 916
B.3 List Insertion 917
B.4 Notes 917
B.5 Transactional Lock Elision 917
B.5.1 Enter Critical Section. 918
B.5.2 Handling Busy Lock 918
B.5.3 Handling TLE Abort 918
B.5.4 TLE Exit Section Critical Path . . 918
B.5.5 Acquisition and Release of TLE Locks. 918

Book III:

Power ISA Operating Environment Architecture. 921

Chapter 1. Introduction 923
1.1 Overview. 923
1.2 Document Conventions 923
1.2.1 Definitions and Notation 923
1.2.2 Reserved Fields 924
1.3 General Systems Overview. 925
1.4 Exceptions. 925
1.5 Synchronization. 925
1.5.1 Context Synchronization 925
1.5.2 Execution Synchronization. 926

Chapter 2. Logical Partitioning (LPAR) and Thread Control. 927
2.1 Overview 927
2.2 Logical Partitioning Control Register (LPCR). 927
2.3 Hypervisor Real Mode Offset Register (HRMOR). 931
2.4 Logical Partition Identification Register (LPIDR) 931
2.5 Processor Compatibility Register (PCR). 932
2.6 Other Hypervisor Resources. 941
2.7 Sharing Hypervisor Resources . . . 941
2.8 Sub-Processors. 942
2.9 Thread Identification Register (TIR) . . 942
2.10 Hypervisor Interrupt Little-Endian (HILE) Bit 942

Chapter 3. Branch Facility 943
3.1 Branch Facility Overview. 943
3.2 Branch Facility Registers 943
3.2.1 Machine State Register 943
3.2.2 State Transitions Associated with the Transactional Memory Facility 946
3.2.3 Processor Stop Status and Control Register (PSSCR) 949
3.3 Branch Facility Instructions 952
3.3.1 System Linkage Instructions . . . 952
3.3.2 Power-Saving Mode 957
3.3.2.1 Power-Saving Mode Instruction . . 958
3.3.2.2 Entering and Exiting Power-Saving Mode 958
3.4 Event-Based Branch Facility and Instruction 960

Chapter 4. Fixed-Point Facility. . . 961
4.1 Fixed-Point Facility Overview 961
4.2 Special Purpose Registers 961
4.3 Fixed-Point Facility Registers 961
4.3.1 Processor Version Register 961
4.3.2 Chip Information Register 961
4.3.3 Processor Identification Register 961
4.3.4 Process Identification Register. . 962
4.3.5 Thread ID Register. 962
4.3.6 Control Register 962

4.3.7 Program Priority Register 963
4.3.8 Problem State Priority Boost Register 963
4.3.9 Relative Priority Register. 963
4.3.10 Software-use SPRs. 964
4.4 Fixed-Point Facility Instructions . . .965
4.4.1 Fixed-Point Load and Store Caching Inhibited Instructions. 965
4.4.2 OR Instruction 968
4.4.3 Transactional Memory Instructions . . 969
4.4.4 Move To/From System Register Instructions 970

Chapter 5. Storage Control 981

5.1 Overview 981
5.2 Storage Exceptions 981
5.3 Instruction Fetch 981
5.3.1 Implicit Branch. 981
5.3.2 Address Wrapping Combined with Changing MSR Bit SF 981
5.4 Data Access 982
5.5 Performing Operations Out-of-Order 982
5.6 Invalid Real Address 982
5.7 Storage Addressing 983
5.7.1 32-Bit Mode. 983
5.7.2 Virtualized Partition Memory (VPM) Mode. 984
5.7.3 Hypervisor Real And Virtual Real Addressing Modes 984
5.7.3.1 Hypervisor Offset Real Mode Address 984
5.7.3.2 Storage Control Attributes for Accesses in Hypervisor Real Addressing Mode. 984
5.7.3.2.1 Hypervisor Real Mode Storage Control 985
5.7.3.3 Virtual Real Mode Addressing Mechanism 985
5.7.3.4 Storage Control Attributes for Implicit Storage Accesses. 986
5.7.4 Definitions 986
5.7.5 Address Ranges Having Defined Uses 987
5.7.5.1 Effective Address Space Structure for Radix-using Partitions 987
5.7.6 In-Memory Tables 988
5.7.6.1 Partition Table 989
5.7.6.2 Process Table. 991
5.7.7 Address Translation Overview . .991
5.7.8 Segment Translation 994
5.7.8.1 Segment Lookaside Buffer (SLB) . 994
5.7.8.2 SLB Search 995
5.7.8.3 Segment Table Description and Search. 995
5.7.8.3.1 Primary Hash for 256MB Segment 996
5.7.8.3.2 Primary Hash for 1TB Segment. 996
5.7.8.3.3 Secondary Hash for 256MB Segment 996
5.7.8.3.4 Secondary Hash for 1TB Segment 996
5.7.9 Hashed Page Table Translation. 996
5.7.9.1 Hashed Page Table 998
5.7.9.2 Page Table Search 999
5.7.10 Radix Tree Translation. 1001
5.7.10.1 Radix Tree Page Directory Entry 1002
5.7.10.2 Radix Tree Page Table Entry1003
5.7.10.3 Nested Translation 1003
5.7.11 Translation Process 1005
5.7.11.1 Fully-Qualified Address. . . . 1005
5.7.11.2 Finding the Page Tables . . . 1006
5.7.11.3 Obtaining Host Real Address, Radix on Radix 1006
5.7.11.4 Obtaining Host Real Address, HPT. 1007
5.7.12 Reference and Change Recording 1007
5.7.13 Storage Protection. 1011
5.7.13.1 Virtual Page Class Key Protection 1011
5.7.13.2 Basic Storage Protection, Address Translation Enabled 1015
5.7.13.3 Basic Storage Protection, Address Translation Disabled 1016
5.7.13.4 Radix Tree Translation Storage Protection 1016
5.8 Storage Control Attributes 1017
5.8.1 Guarded Storage 1017
5.8.1.1 Out-of-Order Accesses to Guarded Storage 1018
5.8.2 Storage Control Bits 1018
5.8.2.1 Storage Control Bit Restrictions . . 1019
5.8.2.2 Altering the Storage Control Bits . 1019
5.9 Storage Control Instructions. . . . 1021
5.9.1 Cache Management Instructions. . . 1021
5.9.2 Synchronize Instruction 1021
5.9.3 Lookaside Buffer Management. 1022
5.9.3.1 Thread-Specific Segment Translations 1023
5.9.3.2 SLB Management Instructions . . 1023

5.9.3.3 TLB Management Instructions . . . 1033
5.10 Translation Table Update Synchronization Requirements 1043
5.10.1 Translation Table Updates . . . 1044
5.10.1.1 Adding a Page Table Entry . 1045
5.10.1.2 Modifying a Translation Table Entry 1045

Chapter 6. Interrupts 1049
6.1 Overview 1049
6.2 Interrupt Registers 1049
6.2.1 Machine Status Save/Restore Registers 1049
6.2.2 Hypervisor Machine Status Save/ Restore Registers 1049
6.2.3 Access Segment Descriptor Register 1049
6.2.4 Data Address Register 1050
6.2.5 Hypervisor Data Address Register. . 1050
6.2.6 Data Storage Interrupt Status Register 1050
6.2.7 Hypervisor Data Storage Interrupt Status Register 1050
6.2.8 Hypervisor Emulation Instruction Register. 1050
6.2.9 Hypervisor Maintenance Exception Register. 1051
6.2.10 Hypervisor Maintenance Exception Enable Register 1051
6.2.11 Facility Status and Control Register 1051
6.2.12 Hypervisor Facility Status and Control Register. 1052
6.3 Interrupt Synchronization 1057
6.4 Interrupt Classes 1057
6.4.1 Precise Interrupt 1057
6.4.2 Imprecise Interrupt. 1057
6.4.3 Interrupt Processing 1059
6.4.4 Implicit alteration of HSRR0 and HSRR1 1061
6.5 Interrupt Definitions 1063
6.5.1 System Reset Interrupt 1065
6.5.2 Machine Check Interrupt 1067
6.5.3 Data Storage Interrupt 1069
6.5.4 Data Segment Interrupt 1071
6.5.5 Instruction Storage Interrupt . . 1071
6.5.6 Instruction Segment Interrupt. 1072
6.5.7 External Interrupt 1073
6.5.7.1 Direct External Interrupt 1073
6.5.7.2 Mediated External Interrupt . 1073
6.5.8 Alignment Interrupt 1073
6.5.9 Program Interrupt 1074
6.5.10 Floating-Point Unavailable Interrupt 1076
6.5.11 Decrementer Interrupt 1076
6.5.12 Hypervisor Decrementer Interrupt 1077
6.5.13 Directed Privileged Doorbell Interrupt 1077
6.5.14 System Call Interrupt 1077
6.5.15 Trace Interrupt 1077
6.5.16 Hypervisor Data Storage Interrupt . 1078
6.5.17 Hypervisor Instruction Storage Interrupt 1082
6.5.18 Hypervisor Emulation Assistance Interrupt 1083
6.5.19 Hypervisor Maintenance Interrupt . 1086
6.5.20 Directed Hypervisor Doorbell Interrupt 1086
6.5.21 Hypervisor Virtualization Interrupt . 1087
6.5.22 Performance Monitor Interrupt 1087
6.5.23 Vector Unavailable Interrupt. . 1087
6.5.24 VSX Unavailable Interrupt . . . 1087
6.5.25 Facility Unavailable Interrupt . 1088
6.5.26 Hypervisor Facility Unavailable Interrupt 1088
6.5.27 System Call Vectored Interrupt1088
6.6 Partially Executed Instructions 1090
6.7 Exception Ordering 1091
6.7.1 Unordered Exceptions 1091
6.7.2 Ordered Exceptions 1091
6.8 Event-Based Branch Exception Ordering 1092
6.9 Interrupt Priorities 1092
6.10 Relationship of Event-Based Branches to Interrupts 1095
6.10.1 EBB Exception Priority 1095
6.10.2 EBB Synchronization 1095
6.10.3 EBB Classes 1095

Chapter 7. Timer Facilities 1097
7.1 Overview 1097
7.2 Time Base (TB) 1097
7.2.1 Writing the Time Base 1098
7.3 Virtual Time Base 1098
7.4 Decrementer 1099
7.4.1 Writing and Reading the Decrementer 1100
7.5 Hypervisor Decrementer 1100
7.6 Processor Utilization of Resources Register (PURR) 1100
7.7 Scaled Processor Utilization of Resources Register (SPURR) 1101

7.8 Instruction Counter 1102

Chapter 8. Debug Facilities 1103
8.1 Overview 1103
8.2 Come-From Address Register . . . 1103
8.3 Completed Instruction Address Breakpoint 1103
8.4 Data Address Watchpoint. 1104

Chapter 9. Performance Monitor Facility 1107
9.1 Overview 1107
9.2 Performance Monitor Operation. . 1107
9.3 No-op Instructions Reserved for the Performance Monitor 1108
9.4 Performance Monitor Facility Registers 1108
9.4.1 Performance Monitor SPR Numbers. 1108
9.4.2 Performance Monitor Counters . 1109
9.4.2.1 Event Counting and Sampling 1109
9.4.3 Threshold Event Counter 1110
9.4.4 Monitor Mode Control Register 0 . . . 1111
9.4.5 Monitor Mode Control Register 1 . . . 1116
9.4.6 Monitor Mode Control Register 2 . . . 1118
9.4.7 Monitor Mode Control Register A . . . 1119
9.4.8 Sampled Instruction Address Register 1122
9.4.9 Sampled Data Address Register 1122
9.4.10 Sampled Instruction Event Register 1123
9.5 Branch History Rolling Buffer 1125
9.6 Interaction With Other Facilities . . 1125

Chapter 10. Processor Control . 1127
10.1 Overview 1127
10.2 Programming Model. 1127
10.3 Processor Control Registers . . . 1127
10.3.1 Directed Privileged Doorbell Exception State 1127
10.4 Processor Control Instructions . . 1129

Chapter 11. Synchronization Requirements for Context Alterations 1133

Power ISA Book I-III Appendices .1139

Appendix A. Illegal Instructions .1141

Appendix B. Reserved Instructions . 1143

Appendix C. Opcode Maps 1145

Appendix D. Power ISA Instruction Set Sorted by Opcode 1179

Appendix E. Power ISA Instruction Set Sorted by Version 1199

Appendix F. Power ISA Instruction Set Sorted by Mnemonic 1219

Last Page - End of Document . . . 1239

Book I:

Power ISA User Instruction Set Architecture

Chapter 1. Introduction

1.1 Overview

This chapter describes computation modes,document conventions, a processor overview, instruction formats, storage addressing, and instruction fetching.

1.2 Instruction Mnemonics and Operands

The description of each instruction includes the mnemonic and a formatted list of operands. Some examples are the following.

```
stw     RS,D(RA)
addis   RT,RA,SI
```

Power ISA-compliant Assemblers will support the mnemonics and operand lists exactly as shown. They should also provide certain extended mnemonics, such as the ones described in Appendix C of Book I.

1.3 Document Conventions

1.3.1 Definitions

The following definitions are used throughout this document.

- **program**
  A sequence of related instructions.
- **application program**
  A program that uses only the instructions and resources described in Books I and II.
- **processor**
  The hardware component that implements the instruction set, storage model, and other facilities defined in the Power ISA architecture, and executes the instructions specified in a program.
- **quadword, doubleword, word, halfword, and byte**
  128 bits, 64 bits, 32 bits, 16 bits, and 8 bits, respectively.
- **positive**
  Means greater than zero.
- **negative**
  Means less than zero.
- **floating-point single format** (or simply **single format**)
  Refers to the representation of a single-precision binary floating-point value in a register or storage.
- **floating-point double format** (or simply **double format**)
  Refers to the representation of a double-precision binary floating-point value in a register or storage.
- **system library program**
  A component of the system software that can be called by an application program using a *Branch* instruction.
- **system service program**
  A component of the system software that can be called by an application program using a *System Call* or *System Call Vectored* instruction.
- **system trap handler**
  A component of the system software that receives control when the conditions specified in a *Trap* instruction are satisfied.
- **system error handler**
  A component of the system software that receives control when an error occurs. The system error handler includes a component for each of the various kinds of error. These error-specific components are referred to as the system alignment error handler, the system data storage error handler, etc.
- **latency**
  Refers to the interval from the time an instruction begins execution until it produces a result that is available for use by a subsequent instruction.
- **unavailable**
  Refers to a resource that cannot be used by the program. For example, storage is unavailable if access to it is denied. See Book III.

- **undefined value**
  May vary between implementations, and between different executions on the same implementation, and similarly for register contents, storage contents, etc., that are specified as being undefined.
- **boundedly undefined**
  The results of executing a given instruction are said to be boundedly undefined if they could have been achieved by executing an arbitrary finite sequence of instructions (none of which yields boundedly undefined results) in the state the processor was in before executing the given instruction. Boundedly undefined results may include the presentation of inconsistent state to the system error handler as described in Section 1.9.1 of Book II. Boundedly undefined results for a given instruction may vary between implementations, and between different executions on the same implementation.
- **“must”**
  If software violates a rule that is stated using the word “must” (e.g., “this field must be set to 0”), the results are boundedly undefined unless otherwise stated.
- **sequential execution model**
  The model of program execution described in Section 2.2, “Instruction Execution Order” on page 29.

1.3.2 Notation

The following notation is used throughout the Power ISA documents.

- All numbers are decimal unless specified in some special way.
  - 0bnnnn means a number expressed in binary format.
  - 0xnnnn means a number expressed in hexadecimal format.

  Underscores may be used between digits.
- RT, RA, R1, ... refer to General Purpose Registers.
- FRT, FRA, FR1, ... refer to Floating-Point Registers.
- FRTp, FRAp, FRBp, ... refer to an even-odd pair of Floating-Point Registers. Values must be even, otherwise the instruction form is invalid.
- VRT, VRA, VR1, ... refer to Vector Registers.
- (x) means the contents of register x, where x is the name of an instruction field. For example, (RA) means the contents of register RA, and (FRA) means the contents of register FRA, where RA and FRA are instruction fields. Names such as LR and CTR denote registers, not fields, so parentheses are not used with them. Parentheses are also omitted when register x is the register into which the result of an operation is placed.
- (RA|0) means the contents of register RA if the RA field has the value 1-31, or the value 0 if the RA field is 0.
- Bytes in instructions, fields, and bit strings are numbered from left to right, starting with byte 0 (most significant).
- Bits in registers, instructions, fields, and bit strings are specified as follows. In the last three items (definition of X_p etc.), if X is a field that specifies a GPR, FPR, or VR (e.g., the RS field of an instruction), the definitions apply to the register, not to the field.
  - Bits in instructions, fields, and bit strings are numbered from left to right, starting with bit 0
  - For all registers except the Vector registers, bits in registers that are less than 64 bits start with bit number 64-L, where L is the register length; for the Vector registers, bits in registers that are less than 128 bits start with bit number 128-L.
  - The leftmost bit of a sequence of bits is the most significant bit of the sequence.
  - X_p means bit p of register/instruction/field/bit_string X.
  - $X_{p:q}$ means bits p through q of register/instruction/field/bit_string X.
  - $X_{p\ q\ ...}$ means bits p, q, ... of register/instruction/field/bit_string X.
- ¬(RA) means the one’s complement of the contents of register RA.
- A period (.) as the last character of an instruction mnemonic means that the instruction records status information in certain fields of the Condition Register as a side effect of execution.
- The symbol || is used to describe the concatenation of two values. For example, 010 || 111 is the same as 010111.
- x^n means x raised to the n^{th} power.
- $^n x$ means the replication of x, n times (i.e., x concatenated to itself n-1 times). $^n 0$ and $^n 1$ are special cases:
  - $^n 0$ means a field of n bits with each bit equal to 0. Thus $^5 0$ is equivalent to 0b00000.
  - $^n 1$ means a field of n bits with each bit equal to 1. Thus $^5 1$ is equivalent to 0b11111.
- Each bit and field in instructions, and in status and control registers (e.g., XER, FPSCR) and Special Purpose Registers, is either defined or reserved. Some defined fields contain reserved values. In such cases when this document refers to the specific field, it refers only to the defined values, unless otherwise specified.

- /, //, ///, ... denotes a reserved field, in a register, instruction, field, or bit string.
- ?, ??, ???, ... denotes an implementation-dependent field in a register, instruction, field or bit string.

1.3.3 Reserved Fields, Reserved Values, and Reserved SPRs

Reserved fields in instructions are ignored by the processor.

In some cases a defined field of an instruction has certain values that are reserved. This includes cases in which the field is shown in the instruction layout as containing a particular value; in such cases all other values of the field are reserved. In general, if an instruction is coded such that a defined field contains a reserved value the instruction form is invalid; see Section 1.9.2 on page 23. The only exception to the preceding rule is that it does not apply to Reserved and Illegal classes of instructions (see Section 1.8) or to portions of defined fields that are specified, in the instruction description, as being treated as reserved fields.

To maximize compatibility with future architecture extensions, software must ensure that reserved fields in instructions contain zero and that defined fields of instructions do not contain reserved values.

The handling of reserved bits in System Registers (e.g., XER, FPSCR) depends on whether the processor is in problem state. Unless otherwise stated, software is permitted to write any value to such a bit. In problem state, a subsequent reading of the bit returns 0 regardless of the value written; in privileged states, a subsequent reading of the bit returns 0 if the value last written to the bit was 0 and returns an undefined value (0 or 1) otherwise.

In some cases, a defined field of a System Register has certain values that are reserved. Software must not set a defined field of a System Register to a reserved value. References elsewhere in this document to a defined field (in an instruction or System Register) that has reserved values assume the field does not contain a reserved value, unless otherwise stated or obvious from context.

In some cases, a given bit of a System Register is specified to be set to a constant value by a given instruction or event. Unless otherwise stated or obvious from context, software should not depend on this constant value because the bit may be assigned a meaning in a future version of the architecture.

The reserved SPRs include SPRs 808, 809, 810, and 811. ***mtspr*** and ***mfspr*** instructions specifying these SPRs are treated as no-ops. Reserved SPRs are provided in the architecture to anticipate the eventual adoption of performance hint functionality that must be controlled by SPRs. Control of these capabilities using reserved SPRs will allow software to use these new capabilities on new implementations that support them while remaining compatible with existing implementations that may not support the new functionality.

Reserved SPRs are not assigned names. There are no individual descriptions of reserved SPRs in this document.

Assembler Note

Assemblers should report uses of reserved values of defined fields of instructions as errors.

Programming Note

It is the responsibility of software to preserve bits that are now reserved in System Registers, because they may be assigned a meaning in some future version of the architecture.

In order to accomplish this preservation in implementation-independent fashion, software should do the following.

- Initialize each such register supplying zeros for all reserved bits.
- Alter (defined) bit(s) in the register by reading the register, altering only the desired bit(s), and then writing the new value back to the register.

The XER and FPSCR are partial exceptions to this recommendation. Software can alter the status bits in these registers, preserving the reserved bits, by executing instructions that have the side effect of altering the status bits. Similarly, software can alter any defined bit in the FPSCR by executing a Floating-Point Status and Control Register instruction. Using such instructions is likely to yield better performance than using the method described in the second item above.

1.3.4 Description of Instruction Operation

Instruction descriptions (including related material such as the introduction to the section describing the instructions) mention that the instruction may cause a system error handler to be invoked, under certain conditions, if and only if the system error handler may treat the case as a programming error. (An instruction may cause a system error handler to be invoked under other conditions as well; see Chapter 6 of Book III).

A formal description is given of the operation of each instruction. In addition, the operation of most instructions is described by a semiformal language at the register transfer level (RTL). This RTL uses the notation given below, in addition to the notation described in Section 1.3.2. Some of this notation is also used in the formal descriptions of instructions. RTL notation not summarized here should be self-explanatory.

The RTL descriptions cover the normal execution of the instruction, except that "standard" setting of status registers, such as the Condition Register, is not shown. ("Non-standard" setting of these registers, such as the setting of the Condition Register by the *Compare* instructions, is shown.) The RTL descriptions do not cover cases in which the system error handler is invoked, or for which the results are boundedly undefined.

The RTL descriptions specify the architectural transformation performed by the execution of an instruction. They do not imply any particular implementation.

Notation	Meaning
$\leftarrow$	Assignment
$\leftarrow_{iea}$	Assignment of an instruction effective address. In 32-bit mode the high-order 32 bits of the 64-bit target address are set to 0.
$\neg$	NOT logical operator
+	Two's complement addition
-	Two's complement subtraction, unary minus
$\times$	Multiplication
$\times_{si}$	Signed-integer multiplication
$\times_{ui}$	Unsigned-integer multiplication
/	Division
$\div$	Division, with result truncated to integer
%	Remainder of integer division
$\sqrt{}$	Square root
$=, \neq$	Equals, Not Equals relations
$<, \leq, >, \geq$	Signed comparison relations
$<^u, >^u$	Unsigned comparison relations
?	Unordered comparison relation
&, |	AND, OR logical operators
$\oplus, \equiv$	Exclusive OR, Equivalence logical operators ($(a \equiv b) = (a \oplus \neg b)$)
ABS(x)	Absolute value of x
BCD_TO_DPD(x)	The low-order 24 bits of x contain six, 4-bit BCD fields which are converted to two declets; each set of two declets is placed into the low-order 20 bits of the result. See Section B.1, "BCD-to-DPD Translation".
CEIL(x)	Least integer $\geq$ x
DOUBLE(x)	Result of converting x from floating-point single format to floating-point double format, using the model shown on page 140
DPD_TO_BCD(x)	The low-order 20 bits of x contain two declets which are converted to six, 4-bit BCD fields; each set of six, 4-bit BCD fields is placed into the low-order 24 bits of the result. See Section B.2, "DPD-to-BCD Translation".
EXTS(x)	Result of extending x on the left with sign bits
FLOOR(x)	Greatest integer $\leq$ x
GPR(x)	General Purpose Register x
MASK(x, y)	Mask having 1s in positions x through y (wrapping if x > y) and 0s elsewhere

MEM(x, y) Contents of a sequence of y bytes of storage. The sequence depends on the byte ordering used for storage access, as follows.
Big-Endian byte ordering:
The sequence starts with the byte at address x and ends with the byte at address x+y-1.
Little-Endian byte ordering:
The sequence starts with the byte at address x+y-1 and ends with the byte at address x.

$ROTL_{64}$(x, y)
Result of rotating the 64-bit value x left y positions

$ROTL_{32}$(x, y)
Result of rotating the 64-bit value x||x left y positions, where x is 32 bits long

SINGLE(x) Result of converting x from floating-point double format to floating-point single format, using the model shown on page 144

SPR(x) Special Purpose Register x

TRAP Invoke the system trap handler

characterization
Reference to the setting of status bits, in a standard way that is explained in the text

undefined An undefined value.

CIA Current Instruction Address, which is the 64-bit address of the instruction being described by a sequence of RTL. Used by relative branches to set the Next Instruction Address (NIA), and by Branch instructions with LK=1 to set the Link Register. Does not correspond to any architected register. The CIA is sometimes referred to as the Program Counter (PC).

NIA Next Instruction Address, which is the 64-bit address of the next instruction to be executed. For a successful branch, the next instruction address is the branch target address: in RTL, this is indicated by assigning a value to NIA. For other instructions that cause non-sequential instruction fetching (see Book III), the RTL is similar. For instructions that do not branch, and do not otherwise cause instruction fetching to be non-sequential, the next instruction address is CIA+4. Does not correspond to any architected register.

if... then... else...
Conditional execution, indenting shows range; else is optional.

do Do loop, indenting shows range. “To” and/or “by” clauses specify incrementing an iteration variable, and a “while” clause gives termination conditions.

leave Leave innermost do loop, or do loop described in leave statement.

for For loop, indenting shows range. Clause after “for” specifies the entities for which to execute the body of the loop.

switch/case/default
switch/case/default statement, indenting shows range. The clause after “switch” specifies the expression to evaluate. The clause after “case” specifies individual values for the expression, followed by a colon, followed by the actions that are taken if the evaluated expression has any of the specified values. “default” is optional. If present, it must follow all the “case” clauses. The clause after “default” starts with a colon, and specifies the actions that are taken if the evaluated expression does not have any of the values specified in the preceding case statements.

The precedence rules for RTL operators are summarized in Table 1. Operators higher in the table are applied before those lower in the table. Operators at the same level in the table associate from left to right, from right to left, or not at all, as shown. (For example, `-` associates from left to right, so `a-b-c` = `(a-b)-c`.) Parentheses are used to override the evaluation order implied by the table or to increase clarity; parenthesized expressions are evaluated before serving as operands.

Table 1: Operator precedence

Operators	**Associativity**
subscript, function evaluation	left to right
pre-superscript (replication), post-superscript (exponentiation)	*right to left*
unary `-`, ¬	*right to left*
×, ÷	left to right
+, -,	left to right
||	left to right
=, ≠, <, ≤, >, ≥, $<^u$, $>^u$, ?	left to right
&, ⊕, ≡	left to right
|	left to right
: (range)	none
←, $\leftarrow_{iea}$	none

1.3.5 Phased-Out Facilities

Phased-Out Facilities

These are facilities and instructions that, in some future version of the architecture, will be dropped out of the architecture. System developers should develop a migration plan to eliminate use of them in new systems. These facilities are marked with a [Phased-Out] marker.

Phased-Out facilities and instructions must be implemented.

Programming Note

Warning: Instructions and facilities being phased out of the architecture are likely to perform poorly on future implementations. New programs should not use them.

1.4 Processor Overview

The basic classes of instructions are as follows:

- branch instructions (Chapter 2)
- GPR-based scalar fixed-point instructions (Chapter 3)
- FPR-based scalar floating-point instructions (Chapter 4)
- FPR-based scalar decimal floating-point instructions (Chapter 5)
- VR-based vector fixed-point and floating-point instructions (Chapter 6)
- VSR-based scalar and vector floating-point instructions (Chapter 7)

Scalar fixed-point instructions operate on byte, halfword, word, doubleword, and quadword operands, where each operand contained in a GPR. Vector fixed-point instructions operate on vectors of byte, halfword, and word operands, where each vector is contained in a VR. Scalar floating-point instructions operate on single-precision or double-precision floating-point operands, where each operand is contained in an FPR or VSR. Vector floating-point instructions operate on vectors of single-precision and double-precision floating-point operands, where each vector is contained in a VR or VSR.

The Power ISA uses instructions that are four bytes long and word-aligned. It provides for byte, halfword, word, doubleword, and quadword operand loads and stores between storage and a set of 32 General Purpose Registers (GPRs). It provides for word and doubleword operand loads and stores between storage and a set of 32 Floating-Point Registers (FPRs). It also provides for byte, halfword, word, and quadword operand loads and stores between storage and a set of 32 Vector Registers (VRs). It provides for doubleword and quadword operand loads and stores between storage and a set of 64 Vector-Scalar Registers (VSRs).

Signed integers are represented in two's complement form.

There are no computational instructions that modify storage; instructions that reference storage may reformat the data (e.g. load halfword algebraic). To use a storage operand in a computation and then modify the same or another storage location, the contents of the storage operand must be loaded into a register, modified, and then stored back to the target location. Figure 1 is a logical representation of instruction processing. Figure 2 shows the registers that are defined in Book I. (A few additional registers that are available to application programs are defined in other Books, and are not shown in the figure.)

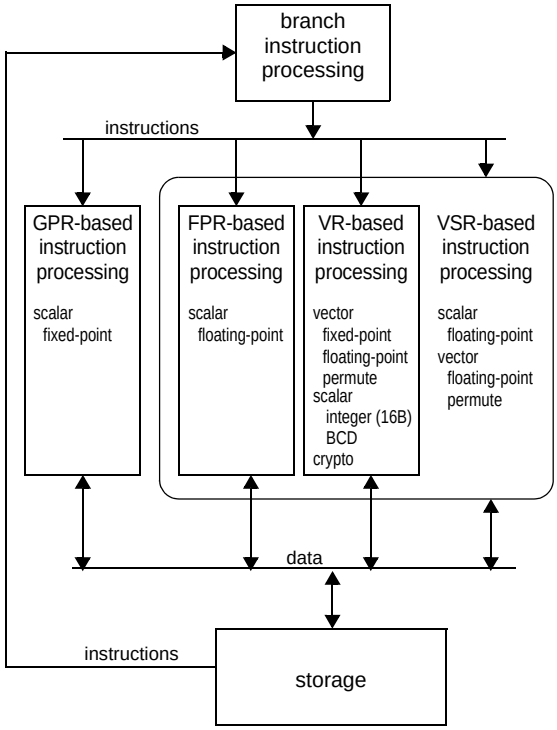
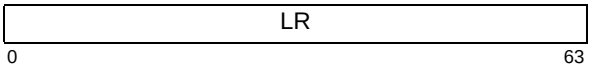
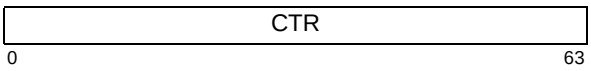


Figure 1. Logical processing model

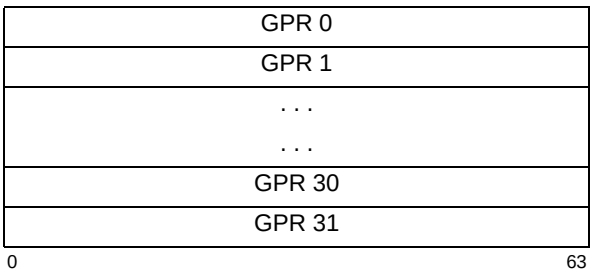
"Condition Register" on page 30



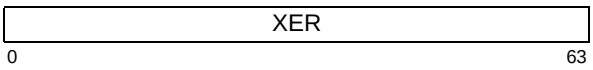
"Link Register" on page 32



"Count Register" on page 32

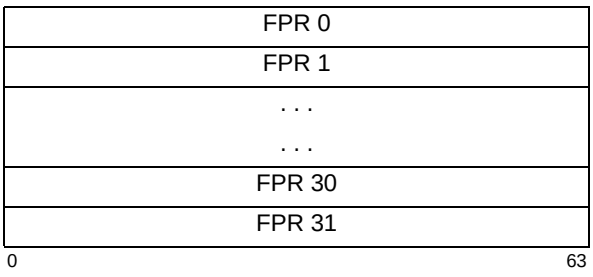


"General Purpose Registers" on page 45



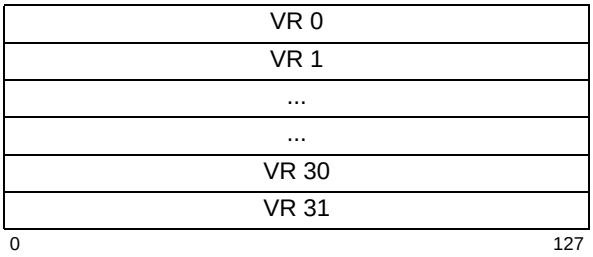
"Fixed-Point Exception Register" on page 45

"VR Save Register" on page 233

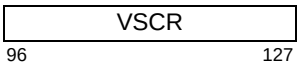


"Floating-Point Registers" on page 124

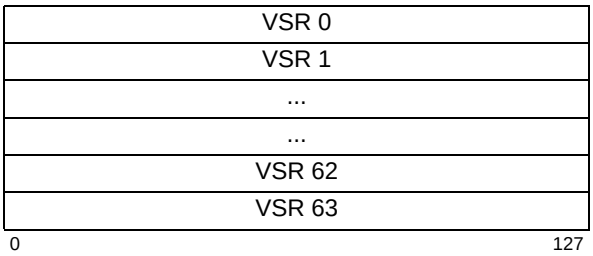
"Floating-Point Status and Control Register" on page 124



"Vector Registers" on page 232



"Vector Status and Control Register" on page 232



"Vector-Scalar Registers" on page 364

Figure 2. Registers that are defined in Book I

1.5 Computation modes

Processors provide two execution modes, 64-bit mode and 32-bit mode. In both of these modes, instructions that set a 64-bit register affect all 64 bits. The computational mode controls how the effective address is interpreted, how Condition Register bits and XER bits are set, how the Link Register is set by *Branch* instructions in which LK=1, and how the Count Register is tested by *Branch Conditional* instructions. Nearly all instructions are available in both modes (the only exceptions are a few instructions that are defined in Book III). In both modes, effective address computations use all 64 bits of the relevant registers (General Purpose Registers,

Link Register, Count Register, etc.) and produce a 64-bit result. However, in 32-bit mode the high-order 32 bits of the computed effective address are ignored for the purpose of addressing storage; see Section 1.11.3 for additional details.

Programming Note

Although instructions that set a 64-bit register affect all 64 bits in both 32-bit and 64-bit modes, operating systems often do not preserve the upper 32-bits of all registers across context switches done in 32-bit mode. For this reason, application programs operating in 32-bit mode should not assume that the upper 32 bits of the GPRs are preserved from instruction to instruction unless the operating system is known to preserve these bits.

1.6 Instruction Formats

All instructions are four bytes long and word-aligned. Thus, whenever instruction addresses are presented to the processor (as in *Branch* instructions) the low-order two bits are ignored. Similarly, whenever the processor develops an instruction address the low-order two bits are zero.

Bits 0:5 always specify the primary opcode (PO, below). Many instructions also have an extended opcode (XO, below). The remaining bits of the instruction contain one or more fields as shown below for the different instruction formats.

The format diagrams given below show horizontally all valid combinations of instruction fields. The diagrams include instruction fields that are used only by instructions defined in Book II or in Book III.

Split Field Notation

In some cases an instruction field occupies more than one contiguous sequence of bits, or occupies one contiguous sequence of bits that are used in permuted order. Such a field is called a *split field*. In the format diagrams given below and in the individual instruction layouts, the name of a split field is shown in small letters, once for each of the contiguous sequences. In the RTL description of an instruction having a split field, and in certain other places where individual bits of a split field are identified, the name of the field in small letters represents the concatenation of the sequences from left to right. In all other places, the name of the field is capitalized and represents the concatenation of the sequences in some order, which need not be left to right, as described for each affected instruction.

1.6.1 A-FORM

0	6	11	16	21	26	31
PO	FRT	///	FRB	///	XO	Rc
PO	FRT	FRA	///	FRC	XO	Rc
PO	FRT	FRA	FRB	///	XO	Rc
PO	FRT	FRA	FRB	FRC	XO	Rc
PO	RT	RA	RB	BC	XO	/

Figure 3. A instruction format

1.6.2 B-FORM

0	6	11	16	30	31
PO	BO	BI	BD	AA	LK

Figure 4. B instruction format

1.6.3 D-FORM

0	6	11	16	31
PO	BF / L	RA	SI	
PO	BF / L	RA	UI	
PO	FRS	RA	D	
PO	FRT	RA	D	
PO	RS	RA	D	
PO	RS	RA	UI	
PO	RT	RA	D	
PO	RT	RA	SI	
PO	TO	RA	SI	

Figure 5. D instruction format

1.6.4 DQ-FORM

0	6	11	16	28	29	31
PO	RTp	RA	DQ	PT		
PO	S	RA	DQ	SX	XO	
PO	T	RA	DQ	TX	XO	

Figure 6. DQ instruction format

1.6.5 DS-FORM

0	6	11	16	30	31
PO	FRSp	RA	DS	XO	
PO	FRTp	RA	DS	XO	
PO	RS	RA	DS	XO	
PO	RSp	RA	DS	XO	
PO	RT	RA	DS	XO	
PO	VRS	RA	DS	XO	
PO	VRT	RA	DS	XO	

Figure 7. DS instruction format

1.6.6 DX-FORM

0	6	11	16	26	31
PO	RT	d1	d0	XO	d2

Figure 8. DX instruction format

1.6.7 I-FORM

0	6	30	31
PO	LI	AA	LK

Figure 9. I instruction format

1.6.8 M-FORM

0	6	11	16	21	26	31
PO	RS	RA	RB	MB	ME	Rc
PO	RS	RA	SH	MB	ME	Rc

Figure 10. M instruction format

1.6.9 MD-FORM

0	6	11	16	21	27	30	31
PO	RS	RA	sh	mb	XO	sh	Rc
PO	RS	RA	sh	me	XO	sh	Rc

Figure 11. MD instruction format

1.6.10 MDS-FORM

0	6	11	16	21	25	27	31
PO	RS	RA	RB	mb		XO	Rc
PO	RS	RA	RB	me		XO	Rc

Figure 12. MDS instruction format

1.6.11 SC-FORM

0	6	11	16	20	27	30	31
PO	///	///	///	LEV	///	1	/

Figure 13. SC instruction format

1.6.12 VA-FORM

0	6	11	16	21	22	26	31
PO	RT	RA	RB	RC		XO	
PO	VRT	VRA	VRB	/	SHB	XO	
PO	VRT	VRA	VRB	VRC		XO	

Figure 14. VA instruction format

1.6.13 VC-FORM

0	6	11	16	21	22	31
PO	VRT	VRA	VRB	Rc	XO	

Figure 15. VC instruction format

1.6.14 VX-FORM

0	6	11	12	13	14	16	21	22	23	31
PO	///	///				VRB	XO			
PO	RT	EO				VRB	XO			
PO	VRT	///				///	XO			
PO	VRT	///				VRB	XO			
PO	VRT	///			UIM	VRB	XO			
PO	VRT	//			UIM	VRB	XO			
PO	VRT	/			UIM	VRB	XO			
PO	VRT	EO				VRB	1	/	XO	
PO	VRT	EO				VRB	1	PS	XO	
PO	VRT	EO				VRB	XO			
PO	VRT	RA				VRB	XO			
PO	VRT	SIM				///	XO			
PO	VRT	UIM				VRB	XO			
PO	VRT	VRA				///	XO			
PO	VRT	VRA				VRB	1	/	XO	
PO	VRT	VRA				VRB	1	PS	XO	
PO	VRT	VRA				VRB	XO			

Figure 16. VX instruction format

1.6.15 X-FORM

0	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
PO	///				///					///					XO											/
PO	///				///					///					XO											1
PO	///				///					RB					XO											/
PO	///				RA					///					XO											/
PO	///				RA					///					XO											1
PO	///				RA					RB					XO											/
PO	///			L	///					///					XO											/
PO	///			L	///					RB					XO											/
PO	///			1	RA					RB					XO											/
PO	///			L	RA					RB					XO											Rc
PO	///		L		///					///					XO											/
PO	///		L		RA					RB					XO											/
PO	///		WC		///					///					XO											/
PO	//		IH		///					///					XO											/
PO	/		CT		RA					RB					XO											/
PO	A		///		///					///					XO											/
PO	A		///	R	///					///					XO											/
PO	BF		//		///					///					XO											/
PO	BF		//		///					FRB					XO											/
PO	BF		//		///				W	U				/	XO											Rc
PO	BF		//		BFA				//	///					XO											/
PO	BF		//		FRA					FRB					XO											/

Figure 17. X instruction format

0	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
PO	BF		//		FRA					FRBp					XO											/
PO	BF		//		FRAp					FRBp					XO											/
PO	BF		//		RA					RB					XO											/
PO	BF		//		UIM					FRB					XO											/
PO	BF		//		UIM					FRBp					XO											/
PO	BF		//		VRA					VRB					XO											/
PO	BF		/	1	RA					RB					XO											/
PO	BF		/	L	RA					RB					XO											/
PO	BF		DCMX							VRB					XO											/
PO	BT				///					///					XO											Rc
PO	FRS				RA					RB					XO											/
PO	FRSp				RA					RB					XO											/
PO	FRT				///					///					XO											Rc
PO	FRT				///					FRB					XO											Rc
PO	FRT				///					FRBp					XO											Rc
PO	FRT				EO					///					XO											Rc
PO	FRT				EO					///					XO											/
PO	FRT				EO					///				RM	XO											/
PO	FRT				EO					//				DRM	XO											/
PO	FRT				EO					FRB					XO											/
PO	FRT				FRA					FRB					XO											/
PO	FRT				FRA					FRB					XO											Rc
PO	FRT				RA					RB					XO											/
PO	FRT		S		///					FRB					XO											Rc
PO	FRT		SP		///					FRB					XO											Rc
PO	FRTp				///					FRB					XO											Rc
PO	FRTp				///					FRBp					XO											Rc
PO	FRTp				FRA					FRBp					XO											Rc
PO	FRTp				FRAp					FRBp					XO											Rc
PO	FRTp				RA					RB					XO											/
PO	FRTp		S		///					FRBp					XO											Rc
PO	FRTp		SP		///					FRBp					XO											Rc
PO	RS				///					RB					XO											/
PO	RS				///				L	///					XO											/
PO	RS		/		RIC				PR	R					RB											XO
PO	RS		/		SR					///					XO											/
PO	RS		BFA		//					///					XO											/
PO	RS				RA					///					XO											/
PO	RS				RA					///					XO											1
PO	RS				RA					///					XO											Rc
PO	RS				RA					FC					XO											/
PO	RS				RA					NB					XO											/
PO	RS				RA					SH					XO											Rc
PO	RS				RA					RB					XO											/

Figure 17. X instruction format

0	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
PO	RS				RA					RB						XO										1
PO	RS				RA					RB						XO										Rc
PO	RSp				RA					RB						XO										1
PO	RT				///					///						XO										/
PO	RT				///					RB						XO										/
PO	RT				///					RB						XO										1
PO	RT				///				L	///						XO										/
PO	RT				/	SR				///						XO										/
PO	RT				RA					FC						XO										/
PO	RT				RA					NB						XO										/
PO	RT				RA					RB						XO										/
PO	RT				RA					RB						XO										EH
PO	RTp				RA					RB						XO										EH
PO	S				RA					///						XO										SX
PO	S				RA					RB						XO										SX
PO	T				EO	IMM8										XO										TX
PO	T				RA					///						XO										TX
PO	T				RA					RB						XO										TX
PO	TH				RA					RB						XO										/
PO	TO				RA					SI						XO										1
PO	TO				RA					RB						XO										/
PO	TO				RA					RB						XO										1
PO	VRS				RA					RB						XO										/
PO	VRT				EO					VRB						XO										/
PO	VRT				EO					VRB						XO										RO
PO	VRT				RA					RB						XO										/
PO	VRT				VRA					VRB						XO										/
PO	VRT				VRA					VRB						XO										RO

Figure 17. X instruction format

1.6.16 XFL-FORM

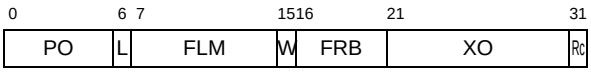


Figure 18. XFL instruction format

1.6.17 XFX-FORM

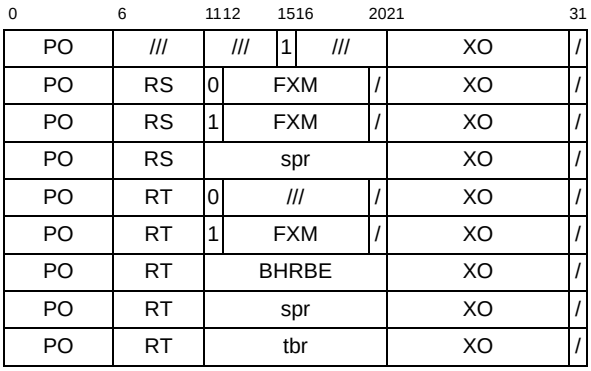


Figure 19. XFX instruction format

1.6.18 XL-FORM

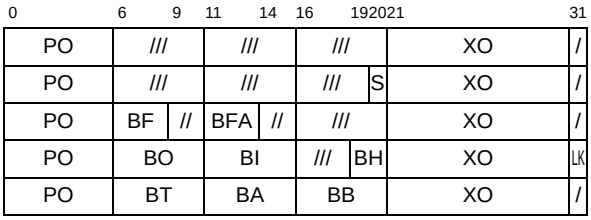


Figure 20. XL instruction format

1.6.19 XO-FORM

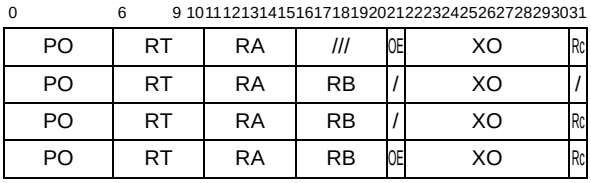


Figure 21. XO instruction format

1.6.20 XS-FORM

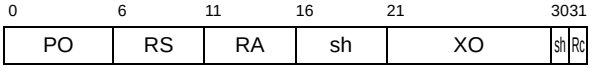


Figure 22. XS instruction format

1.6.21 XX2-FORM

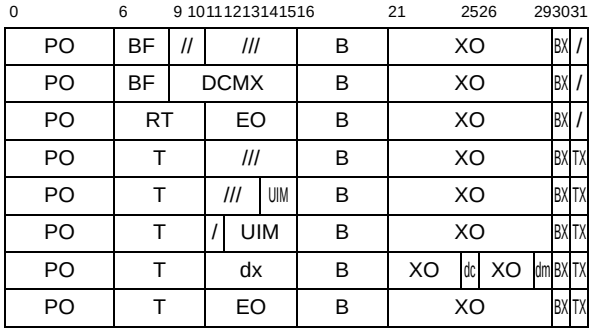


Figure 23. XX2 instruction format

1.6.22 XX3-FORM

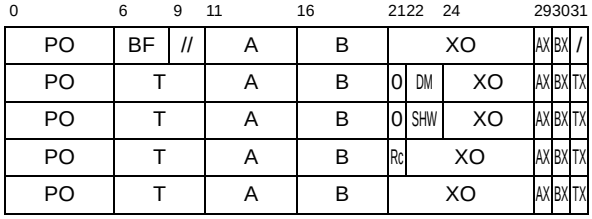


Figure 24. XX3 instruction format

1.6.23 XX4-FORM

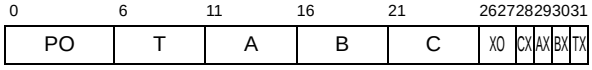


Figure 25. XX4 instruction format

1.6.24 Z22-FORM

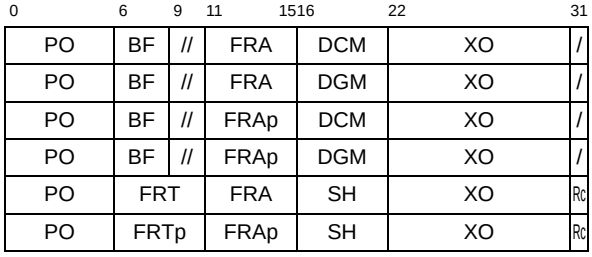


Figure 26. Z22 instruction format

1.6.25 Z23-FORM

0	6	11	15	16	21	23	31
PO	FRT	///	R	FRB	RMC	XO	Rc
PO	FRT	FRA		FRB	RMC	XO	Rc
PO	FRT	TE		FRB	RMC	XO	Rc
PO	FRTp	///	R	FRBp	RMC	XO	Rc
PO	FRTp	FRA		FRBp	RMC	XO	Rc
PO	FRTp	FRAp		FRBp	RMC	XO	Rc
PO	FRTp	TE		FRBp	RMC	XO	Rc
PO	VRT	///	R	VRB	RMC	XO	/
PO	VRT	///	R	VRB	RMC	XO	EX

Figure 27. Z23 instruction format

1.7 Instruction Fields

A (6)

Field used by the ***tbegin.*** instruction to specify an implementation-specific function.

Field used by the ***tend.*** instruction to specify the completion of the outer transaction and all nested transactions.

Formats: X

AA (30)

Absolute Address.

0 The immediate field represents an address relative to the current instruction address. For I-form branches the effective address of the branch target is the sum of the LI field sign-extended to 64 bits and the address of the branch instruction. For B-form branches the effective address of the branch target is the sum of the BD field sign-extended to 64 bits and the address of the branch instruction.

1 The immediate field represents an absolute address. For I-form branches the effective address of the branch target is the LI field sign-extended to 64 bits. For B-form branches the effective address of the branch target is the BD field sign-extended to 64 bits.

Formats: B, I

AX,A (29,11:15)

Fields that are concatenated to specify a VSR to be used as a source.

Formats: XX3, XX4

BA (11:15)

Field used to specify a bit in the CR to be used as a source.

Formats: XL

BB (16:20)

Field used to specify a bit in the CR to be used as a source.

Formats: XL

BC (21:25)

Field used to specify a bit in the CR to be used as a source.

Formats: A

BD (16:29)

Immediate field used to specify a 14-bit signed two's complement branch displacement which is concatenated on the right with 0b00 and sign-extended to 64 bits.

Formats: B

BF (6:8)

Field used to specify one of the CR fields or one of the FPSCR fields to be used as a target.

Formats: D, X, XL, XX2, XX3, Z22

BFA (11:13)

Field used to specify one of the CR fields or one of the FPSCR fields to be used as a source.

Formats: X, XL

BH (19:20)

Field used to specify a hint in the *Branch Conditional to Link Register* and *Branch Conditional to Count Register* instructions. The encoding is described in Section 2.4, "Branch Instructions".

Formats: XL

BHRBE (11:20)

Field used to identify the BHRB entry to be used as a source by the *Move From Branch History Rolling Buffer* instruction.

Formats: X

BI (11:15)

Field used to specify a bit in the CR to be tested by a *Branch Conditional* instruction.

Formats: B, XL

BO (6:10)

Field used to specify options for the *Branch Conditional* instructions. The encoding is described in Section 2.4, "Branch Instructions".

Formats: B, XL, X, XL

BT (6:10)

Field used to specify a bit in the CR or in the FPSCR to be used as a target.

Formats: XL

BX,B (30,16:20)
Fields that are concatenated to specify a VSR to be used as a source.

Formats: XX2, XX3, XX4

CT (7:10)
Field used in X-form instructions to specify a cache target (see Section 4.3.2 of Book II).

Formats: X

CX,C (28,21:25)
Fields that are concatenated to specify a VSR to be used as a source.

Formats: XX4

D (16:31)
Immediate field used to specify a 16-bit signed two's complement integer which is sign-extended to 64 bits.

Formats: D

d0,d1,d2 (16:25,11:15,31)
Immediate fields that are concatenated to specify a 16-bit signed two's complement integer which is sign-extended to 64 bits.

Formats: DX

dc,dm,dx (25,29,11:15)
Immediate fields that are concatenated to specify Data Class Mask.

Formats: XX2

DCM (16:21)
Immediate field used to specify Data Class Mask.

Formats: Z22

DCMX (9:15)
Immediate field used to specify Data Class Mask.

Formats: X, XX2

DGM (16:21)
Immediate field used as the Data Group Mask.

Formats: Z22

DM (22:23)
Immediate field used by ***xxpermdi*** instruction as doubleword permute control.

Formats: XX3

DRM (18:20)
Immediate operand field used to specify new decimal floating-point rounding mode.

Formats: X

DQ (16:27)
Immediate field used to specify a 12-bit signed two's complement integer which is concatenated on the right with 0b0000 and sign-extended to 64 bits.

Formats: DQ

DS (16:29)
Immediate field used to specify a 14-bit signed two's complement integer which is concatenated on the right with `0b00` and sign-extended to 64 bits.

Formats: DS

EH (31)
Field used to specify a hint in the *Load and Reserve* instructions. The meaning is described in Section 4.6.2, "Load and Reserve and Store Conditional Instructions", in Book II.

Formats: X

EO (11:12)
Expanded opcode field

Formats: X

EO (11:15)
Expanded opcode field

Formats: VX, X, XX2

EX (31)
Field used to specify Inexact form of round to quad-precision integer.

Formats: X

FC (16:20)
Field used to specify the function code in *Load/ Store Atomic* instructions.

Formats: X

FLM (7:14)
Field mask used to identify the FPSCR fields that are to be updated by the ***mtfsf*** instruction.

Formats: XFL

FRA (11:15)
Field used to specify a FPR to be used as a source.

Formats: A, X, Z22, Z23

FRAp (11:15)
Field used to specify an even/odd pair of FPRs to be concatenated and used as a source.

Formats: X, Z22, Z23

FRB (16:20)
Field used to specify an FPR to be used as a source.

Formats: A, X, XFL, Z23

FRBp (16:20)
Field used to specify an even/odd pair of FPRs to be concatenated and used as a source.

Formats: X, Z23

FRC (21:25)
Field used to specify an FPR to be used as a source.

Formats: A

FRS (6:10)
Field used to specify an FPR to be used as a source.

Formats: D, X

FRSp (6:10)
Field used to specify an even/odd pair of FPRs to be concatenated and used as a source.

Formats: DS, X

FRT (6:10)
Field used to specify an FPR to be used as a target.

Formats: A, D, X, Z22, Z23

FRTp (6:10)
Field used to specify an even/odd pair of FPRs to be concatenated and used as a target.

Formats: DS, X, Z22, Z23

FXM (12:19)
Field mask used to identify the CR fields that are to be written by the ***mtcrf*** and ***mtocrf*** instructions, or read by the ***mfocrf*** instruction.

Formats: XFX

IB (16:20)
Immediate field used to specify a 5-bit signed integer.

Formats: MDS

IH (8:10)
Field used to specify a hint in the *SLB Invalidate All* instruction. The meaning is described in Section 5.9.3.2, “SLB Management Instructions”, in Book III.

Formats: X

IMM8 (13:20)
Immediate field used to specify an 8-bit integer.

Formats: X

IS (6:10)
Immediate field used to specify a 5-bit signed integer.

Formats: MDS

L (6)
Field used to specify whether the ***mtfsf*** instruction updates the entire FPSCR.

Formats: XFL

L (9:10)
Field used by the *Data Cache Block Flush* instruction (see Section 4.3.2 of Book II) and also by the *Synchronize* instruction (see Section 4.6.3 of Book II).

Formats: X

L (10)
Field used to specify whether a fixed-point Compare instruction is to compare 64-bit numbers or 32-bit numbers.

Field used by the *Compare Range Byte* instruction to indicate whether to compare against 1 or 2 ranges of bytes.

Formats: D, X

L (15)
Field used by the *Move To Machine State Register* instruction (see Book III).

Field used by the *SLB Move From Entry VSID* and *SLB Move From Entry ESID* instructions for implementation-specific purposes.

Formats: X

L (14:15)
Field used by the *Deliver A Random Number* instruction (see Section 3.3.9, “Fixed-Point Arithmetic Instructions”) to choose the random number format.

Formats: X

LEV (20:26)
Field used by the *System Call* instructions.

Formats: SC

LI (6:29)
Immediate field used to specify a 24-bit signed two’s complement integer which is concatenated on the right with `0b00` and sign-extended to 64 bits.

Formats: I

LK (31)
LINK bit.

0 Do not set the Link Register.

1 Set the Link Register. The address of the instruction following the *Branch* instruction is placed into the Link Register.

Formats: B, I, XL

MB (21:25)
Field used in M-form instructions to specify the first 1-bit of a 64-bit mask, as described in Section 3.3.14, “Fixed-Point Rotate and Shift Instructions” on page 101.

Formats: M

mb (21:26)
Field used in MD-form and MDS-form instructions to specify the first 1-bit of a 64-bit mask, as described in Section 3.3.14, “Fixed-Point Rotate and Shift Instructions” on page 101.

Formats: MD, MDS

me (21:26)
Field used in MD-form and MDS-form instructions to specify the last 1-bit of a 64-bit mask, as described in Section 3.3.14, “Fixed-Point Rotate and Shift Instructions” on page 101.

Formats: MD, MDS

ME (26:30)
Field used in M-form instructions to specify the last 1-bit of a 64-bit mask, as described in Section 3.3.14, “Fixed-Point Rotate and Shift Instructions” on page 101.

Formats: M

NB (16:20)
Field used to specify the number of bytes to move in an immediate *Move Assist* instruction.

Formats: X

OE (21)
Field used by XO-form instructions to enable setting OV and SO in the XER.

Formats: XO

PO (0:5)
Primary opcode.

Formats: all

PRS (14)
Field used to specify whether to invalidate process- or partition-scoped entries for ***tlbie***[***l***].

Formats: X

PS (22)
Field used to specify preferred sign for BCD operations.

Formats: VX

PT (28:31)
Immediate field used to specify a 4-bit unsigned value.

Formats: DQ

R (10)
Field used by the ***tbegin.*** instruction to specify the start of a ROT.

Formats: X

R (15)
Immediate field that specifies whether the RMC is specifying the primary or secondary encoding

Field used to specify whether to invalidate Radix Tree or HPT entries for ***tlbie[l***].

Formats: X, Z23

RA (11:15)
Field used to specify a GPR to be used as a source or as a target.

Formats: A, D, DQ, DQE, DS, M, MD, MDS, TX, VA, VX, X, XO, XS

RB (16:20)
Field used to specify a GPR to be used as a source.

Formats: A, M, MDS, VA, X, XO

Rc (21)
RECORD bit.

0 Do not alter the Condition Register.

1 Set Condition Register Field 6 as described in Section 2.3.1, “Condition Register” on page 30.

Formats: VC, XX3

RC (21:25)
Field used to specify a GPR to be used as a source.

Formats: VA

Rc (31)
RECORD bit.

0 Do not alter the Condition Register.

1 Set Condition Register Field 0 or Field 1 as described in Section 2.3.1, “Condition Register” on page 30.

Formats: A, M, MD, MDS, X, XFL, XO, XS, Z22, Z23

RIC (12:13)
Field used to specify what types of entries to invalidate for ***tlbie[l]***.

Formats: X

RM (19:20)
Immediate operand field used to specify new binary floating-point rounding mode.

Formats: X

RMC (21:22)
Immediate field used for DFP rounding mode control.

Formats: Z23

RO (31)
Round to Odd override

Formats: X

RS (6:10)
Field used to specify a GPR to be used as a source.

Formats: D, DS, M, MD, MDS, X, XFX, XS

RSp (6:10)
Field used to specify an even/odd pair of GPRs to be concatenated and used as a source.

Formats: DS, X

RT (6:10)
Field used to specify a GPR to be used as a target.

Formats: A, D, DQE, DS, DX, VA, VX, X, XFX, XO, XX2

RTp (6:10)
Field used to specify an even/odd pair of GPRs to be concatenated and used as a target.

Formats: DQ, X

S (11)
Immediate field that specifies signed versus unsigned conversion.

Formats: X

S (20)
Immediate field that specifies whether or not the ***rfebb*** instruction re-enables event-based branches.

Formats: XL

SH (16:20)
Field used to specify a shift amount.

Formats: M, X

SH (16:21)
Field used to specify a shift amount.

Formats: Z22

sh (30,16:20)
Fields that are concatenated to specify a shift amount.

Formats: MD, XS

SHB (22:25)
Field used to specify a shift amount in bytes.

Formats: VA

SHW (22:23)
Field used to specify a shift amount in words.

Formats: XX3

SI (16:20)
Immediate field used to specify a 5-bit signed integer.

Formats: X

SI (16:31)
Immediate field used to specify a 16-bit signed integer.

Formats: D

SIM (11:15)
Immediate field used to specify a 5-bit signed integer.

Formats: VX

SP (11:12)
Immediate field that specifies signed versus unsigned conversion.

Formats: X

SPR (11:20)
Field used to specify a Special Purpose Register for the ***mtspr*** and ***mfspr*** instructions.

Formats: X

SR (12:15)
Field used by the *Segment Register Manipulation* instructions (see Book III).

Formats: X

SX,S (28,6:10)
Fields SX and S are concatenated to specify a VSR to be used as a source.

Formats: DQ

SX,S (31,6:10)
Fields SX and S are concatenated to specify a VSR to be used as a source.

Formats: X

TBR (11:20)
Field used by the *Move From Time Base* instruction (see Section 6.1 of Book II).

Formats: X

TE (11:15)

Immediate field that specifies a DFP exponent.

Formats: Z23

TH (6:10)
Field used by the data stream variant of the ***dcbt*** and ***dcbtst*** instructions (see Section 4.3.2 of Book II).

Formats: X

TO (6:10)
Field used to specify the conditions on which to trap. The encoding is described in Section 3.3.10.1, “Character-Type Compare Instructions” on page 87.

Formats: TX, X

TX,T (28,6:10)
Fields that are concatenated to specify a VSR to be used as either a target.

Formats: DQ

TX,T (31,6:10)

Fields that are concatenated to specify a VSR to be used as either a target or a source.

Formats: X, XX2, XX3, XX4

U (16:19)

Immediate field used as the data to be placed into a field in the FPSCR.

Formats: X

UI (16:20)

Immediate field used to specify a 5-bit unsigned integer.

Formats: TX

UI (16:31)
Immediate field used to specify a 16-bit unsigned integer.

Formats: D

UIM (11:15)
Immediate field used to specify a 5-bit unsigned integer.

Formats: VX, X

UIM (12:15)
Immediate field used to specify a 4-bit unsigned integer.

Formats: VX, XX2

UIM (13:15)
Immediate field used to specify a 3-bit unsigned integer.

Formats: VX

UIM (14:15)

Immediate field used to specify a 2-bit unsigned integer.

Formats: VX, XX2

VRA (11:15)
Field used to specify a VR to be used as a source.

Formats: VA, VC, VX

VRB (16:20)
Field used to specify a VR to be used as a source.

Formats: VA, VC, VX

VRC (21:25)
Field used to specify a VR to be used as a source.

Formats: VA

VRS (6:10)
Field used to specify a VR to be used as a source.

Formats: DS, X

VRT (6:10)
Field used to specify a VR to be used as a target.

Formats: DS, VA, VC, VX, X

W (15)
Field used by the ***mtfsfi*** and ***mtfsf*** instructions to specify the target word in the FPSCR.

Formats: X, XFL

WC (9:10)
Field used to specify the condition or conditions that cause instruction execution to resume after executing a ***wait*** instruction (see Section 4.6.4 of Book II).

Formats: X

XBI (21:24)
Field used to specify a bit in the XER.

Formats: MDS, MDS, TX

XO (21,23:31)
Extended opcode field.

Formats: VX

XO (21:24,26:28)
Extended opcode field.

Formats: XX2

XO (21:24:28)
Extended opcode field.

Formats: XX3

XO (21:28)
Extended opcode field.

Formats: XX3

XO (21:29)
Extended opcode field.

Formats: XS, XX2

XO (21:30)
Extended opcode field.

Formats: X, XFL, XFX, XL

XO (21:31)
Extended opcode field.

Formats: VX

XO (22:30)
Extended opcode field.

Formats: XO, XX3, Z22

XO (22:31)
Extended opcode field.

Formats: VC

XO (23:30)
Extended opcode field.

Formats: X, Z23

XO (25:30)
Extended opcode field.

Formats: TX

XO (26:27)
Extended opcode field.

Formats: XX4

XO (26:30)
Extended opcode field.

Formats: A, DX

XO (26:31)
Extended opcode field.

Formats: VA

XO (27:29)
Extended opcode field.

Formats: MD

XO (27:30)
Extended opcode field.

Formats: MDS

XO (29:31)
Extended opcode field.

Formats: DQ

XO (30)
Extended opcode field.

Formats: SC

XO (30:31)
Extended opcode field.

Formats: DQE, DS, SC

1.8 Classes of Instructions

An instruction falls into exactly one of the following three classes:

Defined
Illegal
Reserved

The class is determined by examining the opcode, and the extended opcode if any. If the opcode, or combination of opcode and extended opcode, is not that of a defined instruction or a reserved instruction, the instruction is illegal.

1.8.1 Defined Instruction Class

This class of instructions contains all the instructions defined in this document.

A defined instruction can have preferred and/or invalid forms, as described in Section 1.9.1, “Preferred Instruction Forms” and Section 1.9.2, “Invalid Instruction Forms”.

1.8.2 Illegal Instruction Class

This class of instructions contains the set of instructions described in Appendix A of Book Appendices. Illegal instructions are available for future extensions of the Power ISA ; that is, some future version of the Power ISA may define any of these instructions to perform new functions.

Any attempt to execute an illegal instruction will cause the system illegal instruction error handler to be invoked and will have no other effect.

An instruction consisting entirely of binary 0s is guaranteed always to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized storage will result in the invocation of the system illegal instruction error handler.

1.8.3 Reserved Instruction Class

This class of instructions contains the set of instructions described in Appendix B of Book Appendices.

Reserved instructions are allocated to specific purposes that are outside the scope of the Power ISA.

Any attempt to execute a reserved instruction will:

- perform the actions described by the implementation if the instruction is implemented; or
- cause the system illegal instruction error handler to be invoked if the instruction is not implemented.

1.9 Forms of Defined Instructions

1.9.1 Preferred Instruction Forms

Some of the defined instructions have preferred forms. For such an instruction, the preferred form will execute in an efficient manner, but any other form may take significantly longer to execute than the preferred form.

Instructions having preferred forms are:

- the *Condition Register Logical* instructions
- the *Load Quadword* instruction
- the *Move Assist* instructions
- the *Or Immediate* instruction (preferred form of no-op)
- the *Move To Condition Register Fields* instruction

1.9.2 Invalid Instruction Forms

Some of the defined instructions can be coded in a form that is invalid. An instruction form is invalid if one or more fields of the instruction, excluding the opcode field(s), are coded incorrectly in a manner that can be deduced by examining only the instruction encoding.

In general, any attempt to execute an invalid form of an instruction will either cause the system illegal instruction error handler to be invoked or yield boundedly undefined results. Exceptions to this rule are stated in the instruction descriptions.

Some instruction forms are invalid because the instruction contains a reserved value in a defined field (see Section 1.3.3 on page 5); these invalid forms are not discussed further. All other invalid forms are identified in the instruction descriptions.

References to instructions elsewhere in this document assume the instruction form is not invalid, unless otherwise stated or obvious from context.

Assembler Note

Assemblers should report uses of invalid instruction forms as errors.

1.9.3 Reserved-no-op Instructions

Reserved-no-op instructions include the following extended opcodes under primary opcode 31: 530, 562, 594, 626, 658, 690, 722, and 754.

Reserved-no-op instructions are provided in the architecture to anticipate the eventual adoption of performance hint instructions to the architecture. For these instructions, which cause no visible change to architected state, employing a reserved-no-op opcode will allow software to use this new capability on new implementations that support it while remaining compatible with existing implementations that may not support the new function.

When a reserved-no-op instruction is executed, no operation is performed.

Reserved-no-op instructions are not assigned instruction names or mnemonics. There are no individual descriptions of reserved-no-op instructions in this document.

1.10 Exceptions

There are two kinds of exception, those caused directly by the execution of an instruction and those caused by an asynchronous event. In either case, the exception may cause one of several components of the system software to be invoked.

The exceptions that can be caused directly by the execution of an instruction include the following:

- an attempt to execute an illegal instruction, or an attempt by an application program to execute a “privileged” instruction (see Book III) (system illegal instruction error handler or system privileged instruction error handler)
- the execution of a defined instruction using an invalid form (system illegal instruction error handler or system privileged instruction error handler)
- an attempt to execute an instruction that is not provided by the implementation (system illegal instruction error handler)
- an attempt to access a storage location that is unavailable (system instruction storage error handler or system data storage error handler)
- an attempt to access storage with an effective address alignment that is invalid for the instruction (system alignment error handler)
- the execution of a *System Call* or *System Call Vectored* instruction (system service program)
- the execution of a *Trap* instruction that traps (system trap handler)
- the execution of a floating-point instruction that causes a floating-point enabled exception to exist (system floating-point enabled exception error handler)
- the execution of an auxiliary processor instruction that causes an auxiliary processor enabled exception to exist (system auxiliary processor enabled exception error handler)

The exceptions that can be caused by an asynchronous event are described in Book III.

The invocation of the system error handler is precise, except that the invocation of the auxiliary processor enabled exception error handler may be imprecise, and

if one of the imprecise modes for invoking the system floating-point enabled exception error handler is in effect (see page 133), then the invocation of the system floating-point enabled exception error handler may also be imprecise. When the system error handler is invoked imprecisely, the excepting instruction does not appear to complete before the next instruction starts (because one of the effects of the excepting instruction, namely the invocation of the system error handler, has not yet occurred).

Additional information about exception handling can be found in Book III.

1.11 Storage Addressing

A program references storage using the effective address computed by the processor when it executes a *Storage Access* or *Branch* instruction (or certain other instructions described in Book II and Book III), or when it fetches the next sequential instruction.

Bytes in storage are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

The byte ordering (Big-Endian or Little-Endian) for a storage access is specified by the operating system. This byte ordering is also referred to as the Endian mode and it applies to both data accesses and instruction fetches. The Endian mode is specified by the LE mode bit (see Section 3.2.1 of Book III), which applies to all of storage.

1.11.1 Storage Operands

A storage operand may be a byte, a halfword, a word, a doubleword, or a quadword, or, for the *Load/Store Multiple* and *Move Assist* instructions, a sequence of bytes (*Move Assist*) or words (*Load/Store Multiple*). The address of a storage operand is the address of its first byte (i.e., of its lowest-numbered byte). An instruction for which the storage operand is a byte is said to cause a byte access, and similarly for halfword, word, doubleword, and quadword.

The length of the storage operand is the number of bytes (of the storage operand) that the instruction would access in the absence of invocations of the system error handler. The length is generally implied by the name of the instruction (equivalently, by the opcode, and extended opcode if any). For example, the length of the storage operand of a *Load Word and Zero*, *Load Floating-Point Single*, and *Load Vector Element Word* instruction is four bytes (one word), and the length of a *Store Quadword*, *Store Floating-Point Double Pair*, and *Store VSX Vector Word*4* instruction is 16 bytes (one quadword). The only exceptions are the *Load/Store Multiple* and *Move Assist* instructions, for which the length of the storage operand is implied by the identity of the specified source or target register (*Load/Store Multiple*), or by an immediate field in the instruction or the contents of a field in the XER (*Move Assist*), as well as by the name of the instruction. For example, the length of the storage operand of a *Load Multiple Word* instruction for which the specified target register is GPR 20 is 48 bytes ((32-20)x4), and the length of the storage operand of a *Load String Word Immediate* instruction for which the immediate field contains the number 20 is 20 bytes.

The storage operand of a *Load* or *Store* instruction other than a *Load/Store Multiple* or *Move Assist* instruction is said to be aligned if the address of the storage operand is an integral multiple of the storage operand length; otherwise it is said to be unaligned. See the following table. (The storage operand of a *Load/Store Multiple* or *Move Assist* instruction is neither said to be aligned nor said to be unaligned. Its alignment properties are described, when necessary, using terms such as “word-aligned”, which are defined below.)

Operand	Length	$Addr_{60:63}$ if aligned
Byte	8 bits	xxxx
Halfword	2 bytes	xxx0
Word	4 bytes	xx00
Doubleword	8 bytes	x000
Quadword	16 bytes	0000

Note: An “x” in an address bit position indicates that the bit can be 0 or 1 independent of the contents of other bits in the address.

The concept of alignment is also applied more generally, to any datum in storage.

- A datum having length that is an integral power of 2 is said to be aligned if its address is an integral multiple of its length.
- A datum of any length is said to be halfword-aligned (or aligned at a halfword boundary) if its address is an integral multiple of 2, word-aligned (or aligned at a word boundary) if its address is an integral multiple of 4, etc. (All data in storage is byte-aligned.)

The concept of alignment can also be applied to data in registers, with the "address" of the datum interpreted as the byte number of the datum in the register. E.g., a word element (4 bytes) in a Vector Register is said to be aligned if its byte number is an integral multiple of 4.

Programming Note

The technical literature sometimes uses the term “naturally aligned” to mean “aligned.”

Versions of the architecture that precede Version 2.07 also used “naturally aligned” as defined above. The term was dropped from the architecture in Version 2.07 because it seemed to mean different things to different readers and is not needed.

Some instructions require their storage operands to have certain alignments. In addition, alignment may affect performance. In general, the best performance is obtained when storage operands are aligned.

When a storage operand of length N bytes starting at effective address EA is copied between storage and a register that is R bytes long (i.e., the register contains bytes numbered from 0, most significant, through R-1, least significant), the bytes of the operand are placed into the register or into storage in a manner that depends on the byte ordering for the storage access as shown in Figure 28, unless otherwise specified in the instruction description.

Big-Endian Byte Ordering	
Load	Store
for i=0 to N-1: $RT_{(R-N)+i} \leftarrow MEM(EA+i,1)$	for i=0 to N-1: $MEM(EA+i,1) \leftarrow (RS)_{(R-N)+i}$
Little-Endian Byte Ordering	
Load	Store
for i=0 to N-1: $RT_{(R-1)-i} \leftarrow MEM(EA+i,1)$	for i=0 to N-1: $MEM(EA+i,1) \leftarrow (RS)_{(R-1)-i}$
Notes: 1. In this table, subscripts refer to bytes in a register rather than to bits as defined in Section 1.3.2. 2. This table does not apply to the ***lvebx***, ***lvehx***, ***lvewx, stvebx***, ***stvehx***, and ***stvewx*** instructions.	

Figure 28. Storage operands and byte ordering

Figure 29 shows an example of a C language structure **s** containing an assortment of scalars and one character string. The value assumed to be in each structure element is shown in hex in the C comments; these values are used below to show how the bytes making up each structure element are mapped into storage. It is assumed that structure s is compiled for 32-bit mode or for a 32-bit implementation. (This affects the length of the pointer to c.)

C structure mapping rules permit the use of padding (skipped bytes) in order to align the scalars on desirable boundaries. Figures 30 and 31 show each scalar as aligned. This alignment introduces padding of four bytes between **a** and **b**, one byte between **d** and **e**, and two bytes between **e** and **f**. The same amount of padding is present for both Big-Endian and Little-Endian mappings.

The Big-Endian mapping of structure **s** is shown in Figure 30. Addresses are shown in hex at the left of each doubleword, and in small figures below each byte. The contents of each byte, as indicated in the C example in Figure 29, are shown in hex (as characters for the elements of the string).

The Little-Endian mapping of structure s is shown in Figure 31. Doublewords are shown laid out from right to left, which is the common way of showing storage maps for processors that implement only Little-Endian byte ordering.

```
struct {
    int       a;     /*  0x1112_1314                  word           */
    double    b;     /*  0x2122_2324_2526_2728        doubleword     */
    char *    c;     /*  0x3132_3334                  word           */
    char      d[7];  /*  'A', 'B', 'C', 'D', 'E', 'F', 'G'  array of bytes */
    short     e;     /*  0x5152                       halfword       */
    int       f;     /*  0x6162_6364                  word           */
} s;
```

Figure 29. C structure 's', showing values of elements

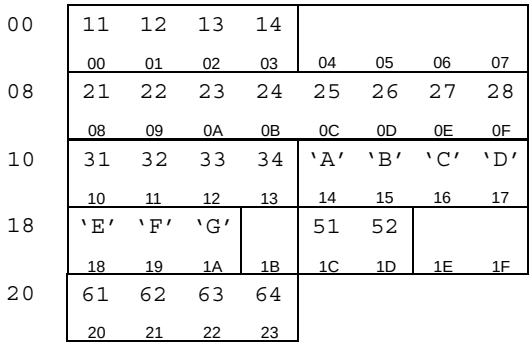


Figure 30. Big-Endian mapping of structure 's'

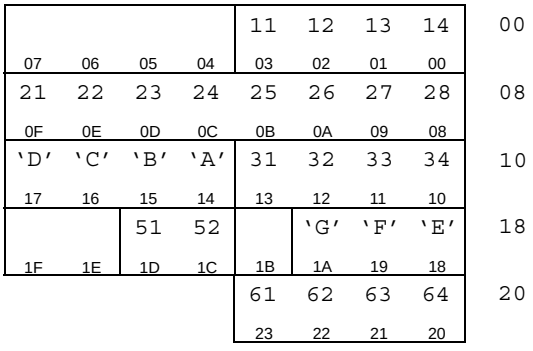


Figure 31. Little-Endian mapping of structure 's'

1.11.2 Instruction Fetches

Instructions are always four bytes long and word-aligned.

When an instruction starting at effective address EA is fetched from storage, the relative order of the bytes within the instruction depend on the byte ordering for the storage access as shown in Figure 32.

Big-Endian Byte Ordering
for i=0 to 3: $inst_i \leftarrow MEM(EA+i,1)$
Little-Endian Byte Ordering
for i=0 to 3: $inst_{3-i} \leftarrow MEM(EA+i,1)$
Note: In this table, subscripts refer to bytes of the instruction rather than to bits as defined in Section 1.3.2.

Figure 32. Instructions and byte ordering

Figure 33 shows an example of a small assembly language program **p**.

```
loop:
          cmplwi      r5,0
          beq         done
          lwzux       r4,r5,r6
          add         r7,r7,r4
          subi        r5,r5,4
          b           loop

done:
          stw         r7,total
```

Figure 33. Assembly language program 'p'

The Big-Endian mapping of program **p** is shown in Figure 34 (assuming the program starts at address 0).

00	loop: cmplwi r5,0				beq done			
	00	01	02	03	04	05	06	07
08	lwzux r4,r5,r6				add r7,r7,r4			
	08	09	0A	0B	0C	0D	0E	0F
10	subi r5,r5,4				b loop			
	10	11	12	13	14	15	16	17
18	done: stw r7,total							
	18	19	1A	1B	1C	1D	1E	1F

Figure 34. Big-Endian mapping of program 'p'

The Little-Endian mapping of program **p** is shown in Figure 35.

beq done				loop: cmplwi r5,0				00
07	06	05	04	03	02	01	00	
add r7,r7,r4				lwzux r4,r5,r6				08
0F	0E	0D	0C	0B	0A	09	08	
b loop				subi r5,r5,4				10
17	16	15	14	13	12	11	10	
				done: stw r7,total				18
1F	1E	1D	1C	1B	1A	19	18	

Figure 35. Little-Endian mapping of program 'p'

Programming Note

The terms *Big-Endian* and *Little-Endian* come from Part I, Chapter 4, of Jonathan Swift's *Gulliver's Travels*. Here is the complete passage, from the edition printed in 1734 by George Faulkner in Dublin.

> ... our Histories of six Thousand Moons make no Mention of any other Regions, than the two great Empires of *Lilliput* and *Blefuscu*. Which two mighty Powers have, as I was going to tell you, been engaged in a most obstinate War for six and thirty Moons past. It began upon the following Occasion. It is allowed on all Hands, that the primitive Way of breaking Eggs before we eat them, was upon the larger End: But his present Majesty's Grand-father, while he was a Boy, going to eat an Egg, and breaking it according to the ancient Practice, happened to cut one of his Fingers. Whereupon the Emperor his Father, published an Edict, commanding all his Subjects, upon great Penalties, to break the smaller End of their Eggs. The People so highly resented this Law, that our Histories tell us, there have been six Rebellions raised on that Account; wherein one Emperor lost his Life, and another his Crown. These civil Commotions were constantly fomented by the Monarchs of *Blefuscu*; and when they were quelled, the Exiles always fled for Refuge to that Empire. It is computed that eleven Thousand Persons have, at several Times, suffered Death, rather than submit to break their Eggs at the smaller End. Many hundred large Volumes have been published upon this Controversy: But the Books of the *Big-Endians* have been long forbidden, and the whole Party rendered incapable by Law of holding Employments. During the Course of these Troubles, the Emperors of *Blefuscu* did frequently expostulate by their Ambassadors, accusing us of making a Schism in Religion, by offending against a fundamental Doctrine of our great Prophet *Lustrog*, in the fifty-fourth Chapter of the *Brundrecal*, (which is their *Alcoran*.) This, however, is thought to be a mere Strain upon the text: For the Words are these; *That all true Believers shall break their Eggs at the convenient End*: and which is the convenient End, seems, in my humble Opinion, to be left to every Man's Conscience, or at least in the Power of the chief Magistrate to determine. Now the *Big-Endian* Exiles have found so much Credit in the Emperor of *Blefuscu's* Court; and so much private Assistance and Encouragement from their Party here at home, that a bloody War has been carried on between the two Empires for six and thirty Moons with various Success; during which Time we have lost Forty Capital Ships, and a much greater Number of smaller Vessels, together with thirty thousand of our best Seamen and Soldiers; and the Damage received by the Enemy is reckoned to be somewhat greater than ours. However, they have now equipped a numerous Fleet, and are just preparing to make a Descent upon us: and his Imperial Majesty, placing great Confidence in your Valour and Strength, hath commanded me to lay this Account of his Affairs before you.

1.11.3 Effective Address Calculation

An effective address is computed by the processor when executing a *Storage Access* or *Branch* instruction (or certain other instructions described in Book II and Book III) when fetching the next sequential instruction, or when invoking a system error handler. The following provides an overview of this process. More detail is provided in the individual instruction descriptions.

Effective address calculations, for both data and instruction accesses, use 64-bit two's complement addition. All 64 bits of each address component participate in the calculation regardless of mode (32-bit or 64-bit). In this computation one operand is an address (which is by definition an unsigned number) and the second is a signed offset. Carries out of the most significant bit are ignored.

In 64-bit mode, the entire 64-bit result comprises the 64-bit effective address. The effective address arithmetic wraps around from the maximum address, $2^{64} - 1$, to address 0, except that if the current instruction is at effective address $2^{64} - 4$ the effective address of the next sequential instruction is undefined.

In 32-bit mode, the low-order 32 bits of the 64-bit result, preceded by 32 0 bits, comprise the 64-bit effective address for the purpose of addressing storage, except that if the current instruction is at effective address $2^{32} - 4$ the 64-bit effective address of the next sequential instruction is undefined. Thus, as used to address storage, the effective address arithmetic appears to wrap around from the maximum address $2^{32} - 1$, to address 0, except when the resulting 64-bit effective address is undefined as just described. When an effective address is placed into a register by an instruction or event, the value placed into the register is as follows.

- Register RA when set by *Load with Update* and *Store with Update* instructions: the entire 64-bit result.
- All other cases (e.g., the Link Register when set by *Branch* instructions having LK=1, Special Purpose

Registers when set to an effective address by invocation of a system error handler): the low-order 32 bits of the 64-bit result preceded by 32 0 bits, except that if the intended effective address is that of the NIA of the instruction at effective address $2^{32}-4$ the value placed into the register is undefined.

RA is a field in the instruction which specifies an address component in the computation of an effective address. A zero in the RA field indicates the absence of the corresponding address component. A value of zero is substituted for the absent component of the effective address computation. This substitution is shown in the instruction descriptions as (RA|0).

Effective addresses are computed as follows. In the descriptions below, it should be understood that "the contents of a GPR" refers to the entire 64-bit contents, independent of mode, but that in 32-bit mode only bits 32:63 of the 64-bit result of the computation are used to address storage.

- With X-form instructions, in computing the effective address of a data element, the contents of the GPR designated by RB (or the value zero for ***lswi*** and ***stswi***) are added to the contents of the GPR designated by RA or to zero if RA=0 or RA is not used in forming the EA.
- With D-form instructions, the 16-bit D field is sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.
- With DS-form instructions, the 14-bit DS field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.
- With DQ-form instructions, the 12-bit DQ field is concatenated on the right with 0b0000 and sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.
- With I-form Branch instructions, the 24-bit LI field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. If AA=0, this address component is added to the address of the Branch instruction to form the effective address of the target instruction. If AA=1, this address component is the effective address of the target instruction.
- With B-form Branch instructions, the 14-bit BD field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. If AA=0, this address component is added to the address of the Branch instruction to form the effective address of the target instruction. If AA=1, this address component is the effective address of the target instruction.
- With XL-form Branch instructions, bits 0:61 of the Link Register or the Count Register are concatenated on the right with 0b00 to form the effective address of the target instruction.
- With sequential instruction fetching, the value 4 is added to the address of the current instruction to form the effective address of the next instruction, except that if the current instruction is at the maximum instruction effective address for the mode ($2^{64}-4$ in 64-bit mode, $2^{32}-4$ in 32-bit mode) the effective address of the next sequential instruction is undefined.

If the size of the operand of a *Storage Access* instruction is more than one byte, the effective address for each byte after the first is computed by adding 1 to the effective address of the preceding byte.

Chapter 2. Branch Facility

2.1 Branch Facility Overview

This chapter describes the registers and instructions that make up the Branch Facility.

2.2 Instruction Execution Order

In general, instructions appear to execute sequentially, in the order in which they appear in storage. The exceptions to this rule are listed below.

- *Branch* instructions for which the branch is taken cause execution to continue at the target address specified by the Branch instruction.
- *Trap* instructions for which the trap conditions are satisfied, and *System Call* and *System Call Vectored* instructions, cause the appropriate system handler to be invoked.
- Transaction failure will eventually cause the transaction's failure handler, implied by the ***tbegin.*** instruction, to be invoked. See the programming note following the ***tbegin.*** description in Section 5.5 of Book II.
- Event-based exceptions can cause the event-based branch handler to be invoked, as described in Chapter 7 of Book II.
- Exceptions can cause the system error handler to be invoked, as described in Section 1.10, "Exceptions" on page 23.
- Returning from a system service program, system trap handler, or system error handler causes execution to continue at a specified address.

The model of program execution in which the processor appears to execute one instruction at a time, completing each instruction before beginning to execute the next instruction is called the "sequential execution model". In general, the processor obeys the sequential execution model. For the instructions and facilities defined in this Book, the only exceptions to this rule are the following.

- A floating-point exception occurs when the processor is running in one of the Imprecise floating-point exception modes (see Section 4.4). The instruction that causes the exception need not complete before the next instruction begins execution, with respect to setting exception bits and (if the exception is enabled) invoking the system error handler.
- A *Store* instruction modifies one or more bytes in an area of storage that contains instructions that will subsequently be executed. Before an instruction in that area of storage is executed, software synchronization is required to ensure that the instructions executed are consistent with the results produced by the *Store* instruction.

Programming Note

This software synchronization will generally be provided by system library programs (see Section 1.9 of Book II). Application programs should call the appropriate system library program before attempting to execute modified instructions.

2.3 Branch Facility Registers

2.3.1 Condition Register

The Condition Register (CR) is a 32-bit register which reflects the result of certain operations, and provides a mechanism for testing (and branching).

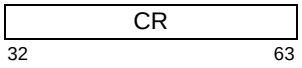


Figure 36. Condition Register

The bits in the Condition Register are grouped into eight 4-bit fields, named CR Field 0 (CR0), ..., CR Field 7 (CR7), which are set in one of the following ways.

- Specified fields of the CR can be set by a move to the CR from a GPR (***mtcrf***, ***mtocrf***).
- A specified field of the CR can be set by a move to the CR from another CR field (***mcrf***), from OV, CA, OV32, and CA32 (***mcrxrx***), or from the FPSCR (***mcrfs***).
- CR Field 0 can be set as the implicit result of a fixed-point instruction.
- CR Field 1 can be set as the implicit result of a floating-point instruction.
- CR Field 1 can be set as the implicit result of a decimal floating-point instruction.
- CR Field 6 can be set as the implicit result of a vector instruction.
- A specified CR field can be set as the result of a *Compare* instruction or of a ***tcheck*** instruction (see Book II).

Instructions are provided to perform logical operations on individual CR bits and to test individual CR bits.

For all fixed-point instructions in which Rc=1, and for ***addic.***, ***andi.***, and ***andis.***, the first three bits of CR Field 0 (bits 32:34 of the Condition Register) are set by signed comparison of the result to zero, and the fourth bit of CR Field 0 (bit 35 of the Condition Register) is copied from the SO field of the XER. “Result” here refers to the entire 64-bit value placed into the target register in 64-bit mode, and to bits 32:63 of the 64-bit value placed into the target register in 32-bit mode.

```
if (64-bit mode)
  then M ← 0
  else M ← 32
if      (target_register)M:63 < 0 then c ← 0b100
else if (target_register)M:63 > 0 then c ← 0b010
else                                   c ← 0b001
CR0 ← c || XERSO
```

If any portion of the result is undefined, then the value placed into the first three bits of CR Field 0 is undefined.

The bits of CR Field 0 are interpreted as follows.

Bit	Description
0	***Negative*** (LT) The result is negative.
1	***Positive*** (GT) The result is positive.
2	***Zero*** (EQ) The result is zero.
3	***Summary Overflow*** (SO) This is a copy of the contents of XER_{SO} at the completion of the instruction.

With the exception of ***tcheck***, the Transactional Memory instructions set $CR0_{0:2}$ indicating the state of the facility prior to instruction execution, or transaction failure. A complete description of the meaning of these bits is given in the instruction descriptions in Section 5.5 of Book II. These bits are interpreted as follows:

CR0	Description
000 || 0	Transaction state of Non-transactional prior to instruction
010 || 0	Transaction state of Transactional prior to instruction
001 || 0	Transaction state of Suspended prior to instruction
101 || 0	Transaction failure

The ***tcheck*** instruction similarly sets bits 1 and 2 of CR field BF to indicate the transaction state, and additionally sets bit 0 to TDOOMED, as defined in Section 5.5 of Book II.

CR field BF	Description
TDOOMED || 00 || 0	Transaction state of Non-transactional prior to instruction
TDOOMED || 10 || 0	Transaction state of Transactional prior to instruction
TDOOMED || 01 || 0	Transaction state of Suspended prior to instruction

Programming Note

Setting of bit 3 of the specified CR field to zero by ***tcheck*** and of field $CR0_3$ to zero by other TM instructions is intended to preserve these bits for future function. Software should not depend on the bits being zero.

The ***paste.*** instruction (see Section 4.4, “Copy-Paste Facility”, in Book II) and the ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx***., and ***stqcx.*** instructions (see Section 4.6.2, “Load and Reserve and Store Conditional Instructions”, in Book II) also set CR Field 0.

For all floating-point instructions in which Rc=1, CR Field 1 (bits 36:39 of the Condition Register) is set to the Floating-Point exception status, copied from bits 32:35 of the Floating-Point Status and Control Register. This occurs regardless of whether any exceptions are enabled, and regardless of whether the writing of the result is suppressed (see Section 4.4, “Floating-Point Exceptions” on page 132). These bits are interpreted as follows.

Bit	Description
32	***Floating-Point Exception Summary*** (FX) This is a copy of the contents of $FPSCR_{FX}$ at the completion of the instruction.
33	***Floating-Point Enabled Exception Summary*** (FEX) This is a copy of the contents of $FPSCR_{FEX}$ at the completion of the instruction.
34	***Floating-Point Invalid Operation Exception Summary*** (VX) This is a copy of the contents of $FPSCR_{VX}$ at the completion of the instruction.
35	***Floating-Point Overflow Exception*** (OX) This is a copy of the contents of $FPSCR_{OX}$ at the completion of the instruction.

For *Compare* instructions, a specified CR field is set to reflect the result of the comparison. The bits of the specified CR field are interpreted as follows. A complete description of how the bits are set is given in the instruction descriptions in Section 3.3.10, “Fixed-Point Compare Instructions” on page 84, and Section 4.6.8, “Floating-Point Compare Instructions” on page 167.

Bit	Description
0	***Less Than, Floating-Point Less Than*** (LT, FL) For fixed-point Compare instructions, (RA) < SI or (RB) (signed comparison) or (RA) $<^u$ UI or (RB) (unsigned comparison). For floating-point Compare instructions, (FRA) < (FRB).
1	***Greater Than, Floating-Point Greater Than*** (GT, FG) For fixed-point Compare instructions, (RA) > SI or (RB) (signed comparison) or (RA) $>^u$ UI or (RB) (unsigned comparison). For floating-point Compare instructions, (FRA) > (FRB).
2	***Equal, Floating-Point Equal*** (EQ, FE) For fixed-point Compare instructions, (RA) = SI, UI, or (RB). For floating-point Compare instructions, (FRA) = (FRB).
3	***Summary Overflow, Floating-Point Unordered*** (SO,FU) For fixed-point *Compare* instructions, this is a copy of the contents of XER_{SO} at the completion of the instruction. For floating-point *Compare* instructions, one or both of (FRA) and (FRB) is a NaN.

The *Vector Integer Compare* instructions (see Section 6.9.3, “Vector Integer Compare Instructions”) compare two Vector Registers element by element, interpreting the elements as unsigned or signed integers depending on the instruction, and set the corresponding element of the target Vector Register to all 1s if the relation being tested is true and 0s if the relation being tested is false.

If Rc=1, CR Field 6 is set to reflect the result of the comparison, as follows

Bit	Description
0	The relation is true for all element pairs (i.e., VRT is set to all 1s).
1	0
2	The relation is false for all element pairs (i.e., VRT is set to all 0s).
3	0

The *Vector Floating-Point Compare* instructions compare two Vector Registers word element by word element, interpreting the elements as single-precision floating-point numbers. With the exception of the *Vector Compare Bounds Floating-Point* instruction, they set the target Vector Register, and CR Field 6 if Rc=1, in the same manner as do the *Vector Integer Compare* instructions.

Bit	Description
0	The relation is true for all element pairs (i.e., VRT is set to all 1s).
1	0
2	The relation is false for all element pairs (i.e., VRT is set to all 0s).
3	0

The *Vector Compare Bounds Floating-Point* instruction on page 328 sets CR Field 6 if Rc=1, to indicate whether the elements in VRA are within the bounds specified by the corresponding element in VRB, as explained in the instruction description. A single-precision floating-point value x is said to be “within the bounds” specified by a single-precision floating-point value y if $-y \leq x \leq y$.

Bit	Description
0	0
1	0
2	Set to indicate whether all four elements in VRA are within the bounds specified by the corresponding element in VRB, otherwise set to 0.
3	0

<BHRB material moved to Chapter 8 of Book II.>

2.3.2 Link Register

The Link Register (LR) is a 64-bit register. It can be used to provide the branch target address for the *Branch Conditional to Link Register* instruction, and it holds the return address after Branch instructions for which LK=1 and after *System Call Vectored* instructions.

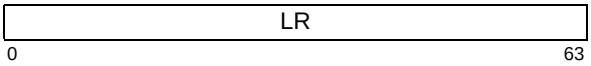


Figure 37. Link Register

2.3.3 Count Register

The Count Register (CTR) is a 64-bit register. It can be used to hold a loop count that can be decremented during execution of Branch instructions that contain an appropriately coded BO field. If the value in the Count Register is 0 before being decremented, it is -1 afterward. The Count Register can also be used to provide the branch target address for the *Branch Conditional to Count Registe*r instruction. The Count Register is modified by the *System Call Vectored* instruction.

Figure 38. Count Register

2.3.4 Target Address Register

The Target Address Register (TAR) is a 64-bit register. It can be used to provide bits 0:61 of the branch target address for the *Branch Conditional to Branch Target Address Registe*r instruction. Bits 62:63 are ignored by the hardware but can be set and reset by software.

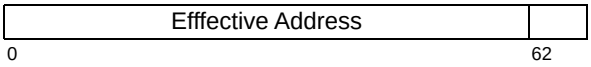


Figure 39. Target Address Register

Programming Note

The TAR is reserved for system software.

2.4 Branch Instructions

The sequence of instruction execution can be changed by the *Branch* instructions. Because all instructions are on word boundaries, bits 62 and 63 of the generated branch target address are ignored by the processor in performing the branch.

The *Branch* instructions compute the effective address (EA) of the target in one of the following five ways, as described in Section 1.11.3, “Effective Address Calculation” on page 27.

1. Adding a displacement to the address of the *Branch* instruction (*Branch* or *Branch Conditiona*l with AA=0).
2. Specifying an absolute address (*Branch* or *Branch Conditional* with AA=1).
3. Using the address contained in the Link Register (*Branch Conditional to Link Register*).
4. Using the address contained in the Count Register (*Branch Conditional to Count Register*).
5. Using the address contained in the Target Address Register (*Branch Conditional to Target Address Register*).

In all five cases, in 32-bit mode the final step in the address computation is setting the high-order 32 bits of the target address to 0.

For the first two methods, the target addresses can be computed sufficiently ahead of the *Branch* instruction that instructions can be prefetched along the target path. For the third through fifth methods, prefetching instructions along the target path is also possible provided the Link Register or the Count Register is loaded sufficiently ahead of the *Branch* instruction.

Branching can be conditional or unconditional, and the return address can optionally be provided. If the return address is to be provided (LK=1), the effective address of the instruction following the *Branch* instruction is placed into the Link Register after the branch target address has been computed; this is done regardless of whether the branch is taken.

For *Branch Conditional* instructions, the BO field specifies the conditions under which the branch is taken, as shown in Figure 40. In the figure, M=0 in 64-bit mode and M=32 in 32-bit mode.

BO	Description
0000z	Decrement the CTR, then branch if the decremented $CTR_{M:63}\neq 0$ and $CR_{BI}=0$
0001z	Decrement the CTR, then branch if the decremented $CTR_{M:63}=0$ and $CR_{BI}=0$
001at	Branch if $CR_{BI}=0$
0100z	Decrement the CTR, then branch if the decremented $CTR_{M:63}\neq 0$ and $CR_{BI}=1$
0101z	Decrement the CTR, then branch if the decremented $CTR_{M:63}=0$ and $CR_{BI}=1$
011at	Branch if $CR_{BI}=1$
1a00t	Decrement the CTR, then branch if the decremented $CTR_{M:63}\neq 0$
1a01t	Decrement the CTR, then branch if the decremented $CTR_{M:63}=0$
1z1zz	Branch always

Notes:
1. “z” denotes a bit that is ignored.
2. The “a” and “t” bits are used as described below.

Figure 40. BO field encodings

The “a” and “t” bits of the BO field can be used by software to provide a hint about whether the branch is likely to be taken or is likely not to be taken, as shown in Figure 41.

at	Hint
00	No hint is given
01	Reserved
10	The branch is very likely not to be taken
11	The branch is very likely to be taken

Figure 41. “at” bit encodings

Programming Note

Many implementations have dynamic mechanisms for predicting whether a branch will be taken. Because the dynamic prediction is likely to be very accurate, and is likely to be overridden by any hint provided by the “at” bits, the “at” bits should be set to 0b00 unless the static prediction implied by at=0b10 or at=0b11 is highly likely to be correct.

For *Branch Conditional to Link Register, Branch Conditional to Count Register,* and *Branch Conditional to Target Address Register* instructions, the BH field provides

a hint about the use of the instruction, as shown in Figure 42.

BH	Hint
00	***bclr***[***l***]: The instruction is a subroutine return ***bcctr***[***l***] and ***bctar***[***l***]:The instruction is not a subroutine return; the target address is likely to be the same as the target address used the preceding time the branch was taken
01	***bclr***[***l***]: The instruction is not a subroutine return; the target address is likely to be the same as the target address used the preceding time the branch was taken ***bcctr***[***l***] and ***bctar***[***l***]:Reserved
10	Reserved
11	***bclr***[***l***], ***bcctr***[***l***], and ***bctar***[l]: The target address is not predictable

Figure 42. BH field encodings

Programming Note

The hint provided by the BH field is independent of the hint provided by the “at” bits (e.g., the BH field provides no indication of whether the branch is likely to be taken).

Extended mnemonics for branches

Many extended mnemonics are provided so that *Branch Conditional* instructions can be coded with portions of the BO and BI fields as part of the mnemonic rather than as part of a numeric operand. Some of these are shown as examples with the Branch instructions. See Appendix C for additional extended mnemonics.

Programming Note

The hints provided by the “at” bits and by the BH field do not affect the results of executing the instruction.

The “z” bits should be set to 0, because they may be assigned a meaning in some future version of the architecture.

Programming Note

Many implementations have dynamic mechanisms for predicting the target addresses of ***bclr***[***l***] and ***bcctr***[***l***] instructions. These mechanisms may cache return addresses (i.e., Link Register values set by *Branch* instructions for which LK=1 and for which the branch was taken, other than the special form shown in the first example below) and recently used branch target addresses. To obtain the best performance across the widest range of implementations, the programmer should obey the following rules.

- Use *Branch* instructions for which LK=1 only as subroutine calls (including function calls, etc.), or in the special form shown in the first example below.
- Pair each subroutine call (i.e., each *Branch* instruction for which LK=1 and the branch is taken, other than the special form shown in the first example below) with a ***bclr*** instruction that returns from the subroutine and has BH=0b00.
- Do not use ***bclrl*** as a subroutine call. (Some implementations access the return address cache at most once per instruction; such implementations are likely to treat ***bclrl*** as a subroutine return, and not as a subroutine call.)
- For ***bclr***[***l***] and ***bcctr***[***l***], use the appropriate value in the BH field.

The following are examples of programming conventions that obey these rules. In the examples, BH is assumed to contain 0b00 unless otherwise stated. In addition, the "at" bits are assumed to be coded appropriately.

Let A, B, and Glue be specific programs.

- Obtaining the address of the next instruction:
  Use the following form of *Branch and Link*.

```
bcl   20,31,$+4
```

- Loop counts:
  Keep them in the Count Register, and use a ***bc*** instruction (LK=0) to decrement the count and to branch back to the beginning of the loop if the decremented count is nonzero.
- Computed goto's, case statements, etc.:
  Use the Count Register to hold the address to branch to, and use a ***bcctr*** instruction (LK=0, and BH=0b11 if appropriate) to branch to the selected address.
- Direct subroutine linkage:
  Here A calls B and B returns to A. The two branches should be as follows.
  - A calls B: use a ***bl*** or ***bcl*** instruction (LK=1).
  - B returns to A: use a ***bclr*** instruction (LK=0) (the return address is in, or can be restored to, the Link Register).
- Indirect subroutine linkage:
  Here A calls Glue, Glue calls B, and B returns to A rather than to Glue. (Such a calling sequence is common in linkage code used when the subroutine that the programmer wants to call, here B, is in a different module from the caller; the Binder inserts "glue" code to mediate the branch.) The three branches should be as follows.
  - A calls Glue: use a ***bl*** or ***bcl*** instruction (LK=1).
  - Glue calls B: place the address of B into the Count Register, and use a ***bcctr*** instruction (LK=0).
  - B returns to A: use a ***bclr*** instruction (LK=0) (the return address is in, or can be restored to, the Link Register).
- Function call:
  Here A calls a function, the identity of which may vary from one instance of the call to another, instead of calling a specific program B. This case should be handled using the conventions of the preceding two bullets, depending on whether the call is direct or indirect, with the following differences.
  - If the call is direct, place the address of the function into the Count Register, and use a ***bcctrl*** instruction (LK=1) instead of a ***bl*** or ***bcl*** instruction.
  - For the ***bcctr***[***l***] instruction that branches to the function, use BH=0b11 if appropriate.

Compatibility Note

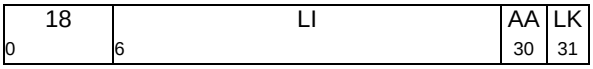
The bits corresponding to the current “a” and “t” bits, and to the current “z” bits except in the “branch always” BO encoding, had different meanings in versions of the architecture that precede Version 2.00.

- The bit corresponding to the “t” bit was called the “y” bit. The “y” bit indicated whether to use the architected default prediction (y=0) or to use the complement of the default prediction (y=1). The default prediction was defined as follows.
  - If the instruction is ***bc***[***l***][***a***] with a negative value in the displacement field, the branch is taken. (This is the only case in which the prediction corresponding to the “y” bit differs from the prediction corresponding to the “t” bit.)
  - In all other cases (***bc***[***l***][***a***] with a nonnegative value in the displacement field, ***bclr***[***l***], or ***bcctr***[***l***]), the branch is not taken.
- The BO encodings that test both the Count Register and the Condition Register had a “y” bit in place of the current “z” bit. The meaning of the “y” bit was as described in the preceding item.
- The “a” bit was a “z” bit.

Because these bits have always been defined either to be ignored or to be treated as hints, a given program will produce the same result on any implementation regardless of the values of the bits. Also, because even the “y” bit is ignored, in practice, by most processors that comply with versions of the architecture that precede Version 2.00, the performance of a given program on those processors will not be affected by the values of the bits.

Branch *I-form*

b	target_addr	(AA=0 LK=0)
ba	target_addr	(AA=1 LK=0)
bl	target_addr	(AA=0 LK=1)
bla	target_addr	(AA=1 LK=1)




```
if AA then NIA ←iea EXTS(LI || 0b00)
else        NIA ←iea CIA + EXTS(LI || 0b00)
if LK then LR ←iea CIA + 4
```

target_addr specifies the branch target address.

If AA=0 then the branch target address is the sum of LI || 0b00 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If AA=1 then the branch target address is the value LI || 0b00 sign-extended, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

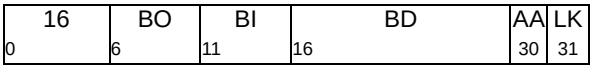
If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

Special Registers Altered:

LR	(if LK=1)

Branch Conditional *B-form*

bc	BO,BI,target_addr	(AA=0 LK=0)
bca	BO,BI,target_addr	(AA=1 LK=0)
bcl	BO,BI,target_addr	(AA=0 LK=1)
bcla	BO,BI,target_addr	(AA=1 LK=1)




```
if (64-bit mode)
  then M ← 0
  else M ← 32
if ¬BO2 then CTR ← CTR - 1
ctr_ok ← BO2 | ((CTRM:63 ≠ 0) ⊕ BO3)
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if ctr_ok & cond_ok then
  if AA then NIA ←iea EXTS(BD || 0b00)
  else       NIA ←iea CIA + EXTS(BD || 0b00)
if LK then LR ←iea CIA + 4
```

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 40. *target_addr* specifies the branch target address.

If AA=0 then the branch target address is the sum of BD || 0b00 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If AA=1 then the branch target address is the value BD || 0b00 sign-extended, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Special Registers Altered:

CTR	(if BO_2=0)
LR	(if LK=1)

Extended Mnemonics:

Examples of extended mnemonics for *Branch Conditional*:

Extended:		**Equivalent to:**	
blt	target	bc	12,0,target
bne	cr2,target	bc	4,10,target
bdnz	target	bc	16,0,target

Branch Conditional to Link Register *XL-form*

bclr	BO,BI,BH	(LK=0)
bclrl	BO,BI,BH	(LK=1)

19	BO	BI	///	BH	16	LK
0	6	11	16	19	21	31

```
if (64-bit mode)
  then M ← 0
  else M ← 32
if ¬BO2 then CTR ← CTR - 1
ctr_ok ← BO2 | ((CTRM:63 ≠ 0) ⊕ BO3
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if ctr_ok & cond_ok then NIA ←iea LR0:61 || 0b00
if LK then LR ←iea CIA + 4
```

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 40. The BH field is used as described in Figure 42. The branch target address is $LR_{0:61}$ || 0b00, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Special Registers Altered:

CTR	(if BO_2=0)
LR	(if LK=1)

Extended Mnemonics:

Examples of extended mnemonics for *Branch Conditional to Link Register*:

Extended:		Equivalent to:	
bclr	4,6	bclr	4,6,0
bltlr		bclr	12,0,0
bnelr	cr2	bclr	4,10,0
bdnzlr		bclr	16,0,0

Programming Note

bclr, ***bclrl***, ***bcctr***, and ***bcctrl*** each serve as both a basic and an extended mnemonic. The Assembler will recognize a ***bclr***, ***bclrl***, ***bcctr***, or ***bcctrl*** mnemonic with three operands as the basic form, and a ***bclr***, ***bclrl***, ***bcctr***, or ***bcctrl*** mnemonic with two operands as the extended form. In the extended form the BH operand is omitted and assumed to be 0b00.

Branch Conditional to Count Register *XL-form*

bcctr	BO,BI,BH	(LK=0)
bcctrl	BO,BI,BH	(LK=1)

19	BO	BI	///	BH	528	LK
0	6	11	16	19	21	31

```
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if cond_ok then NIA ←iea CTR0:61 || 0b00
if LK then LR ←iea CIA + 4
```

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 40. The BH field is used as described in Figure 42. The branch target address is $CTR_{0:61}$ || 0b00, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

If the “decrement and test CTR” option is specified (BO_2=0), the instruction form is invalid.

Special Registers Altered:

LR	(if LK=1)

Extended Mnemonics:

Examples of extended mnemonics for *Branch Conditional to Count Register.*

Extended:		Equivalent to:	
bcctr	4,6	bcctr	4,6,0
bltctr		bcctr	12,0,0
bnectr	cr2	bcctr	4,10,0

Branch Conditional to Branch Target Address Register

XL-form

bctar	BO,BI,BH	(LK=0)
bctarl	BO,BI,BH	(LK=1)

19	BO	BI	///	BH	560	LK
0	6	11	16	19	21	31

```
if (64-bit mode)
  then M ← 0
  else M ← 32
if ¬BO2 then CTR ← CTR - 1
ctr_ok ← BO2 | ((CTRM:63 ≠ 0) ⊕ BO3
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if ctr_ok & cond_ok then NIA ←iea TAR0:61 || 0b00
if LK then LR ←iea CIA + 4
```

BI+32 specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 40. The BH field is used as described in Figure 42. The branch target address is $TAR_{0:61}$ || 0b00, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Special Registers Altered:

CTR	(if BO_2=0)
LR	(if LK=1)

Programming Note

In some systems, the system software will restrict usage of the ***bctar[l]*** instruction to only selected programs. If an attempt is made to execute the instruction when it is not available, the system error handler will be invoked. See Book III for additional information.

2.5 Condition Register Instructions

2.5.1 Condition Register Logical Instructions

The *Condition Register Logical* instructions have preferred forms; see Section 1.9.1. In the preferred forms, the BT and BB fields satisfy the following rule.

- The bit specified by BT is in the same Condition Register field as the bit specified by BB.

Extended mnemonics for Condition Register logical operations

A set of extended mnemonics is provided that allow additional Condition Register logical operations, beyond those provided by the basic *Condition Register Logical* instructions, to be coded easily. Some of these are shown as examples with the *Condition Register Logical* instructions. See Appendix C for additional extended mnemonics.

Condition Register AND *XL-form*

crand BT,BA,BB

19	BT	BA	BB	257	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32}$ & CR_{BB+32}

The bit in the Condition Register specified by BA+32 is ANDed with the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Condition Register NAND *XL-form*

crnand BT,BA,BB

19	BT	BA	BB	225	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow \neg(CR_{BA+32}$ & $CR_{BB+32})$

The bit in the Condition Register specified by BA+32 is ANDed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Condition Register OR *XL-form*

cror BT,BA,BB

19	BT	BA	BB	449	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \mid CR_{BB+32}$

The bit in the Condition Register specified by BA+32 is ORed with the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Extended Mnemonics:

Example of extended mnemonics for *Condition Register OR*:

Extended:	Equivalent to:
crmove Bx,By	cror Bx,By,By

Condition Register XOR *XL-form*

crxor BT,BA,BB

19	BT	BA	BB	193	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \oplus CR_{BB+32}$

The bit in the Condition Register specified by BA+32 is XORed with the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Extended Mnemonics:

Example of extended mnemonics for *Condition Register XOR*:

Extended:	Equivalent to:
crclr Bx	crxor Bx,Bx,Bx

Condition Register NOR *XL-form*

crnor BT,BA,BB

19	BT	BA	BB	33	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow \neg(CR_{BA+32} \mid CR_{BB+32})$

The bit in the Condition Register specified by BA+32 is ORed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Extended Mnemonics:

Example of extended mnemonics for *Condition Register NOR*:

Extended:	Equivalent to:
crnot Bx,By	crnor Bx,By,By

Condition Register Equivalent *XL-form*

creqv BT,BA,BB

19	BT	BA	BB	289	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \equiv CR_{BB+32}$

The bit in the Condition Register specified by BA+32 is XORed with the bit in the Condition Register specified by BB+32, and the complemented result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Extended Mnemonics:

Example of extended mnemonics for *Condition Register Equivalent*:

Extended:	Equivalent to:
crset Bx	creqv Bx,Bx,Bx

Condition Register AND with Complement *XL-form*

crandc BT,BA,BB

19	BT	BA	BB	129	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \ \& \ \neg CR_{BB+32}$

The bit in the Condition Register specified by BA+32 is ANDed with the complement of the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

Condition Register OR with Complement *XL-form*

crorc BT,BA,BB

19	BT	BA	BB	417	/
0	6	11	16	21	31

$CR_{BT+32} \leftarrow CR_{BA+32} \mid \neg CR_{BB+32}$

The bit in the Condition Register specified by BA+32 is ORed with the complement of the bit in the Condition Register specified by BB+32, and the result is placed into the bit in the Condition Register specified by BT+32.

Special Registers Altered:

CR_{BT+32}

2.5.2 Condition Register Field Instruction

Move Condition Register Field *XL-form*

mcrf BF,BFA

19	BF	//	BFA	//	///	0	/
0	6	9	11	14	16	21	31

$CR_{4\times BF+32:4\times BF+35} \leftarrow CR_{4\times BFA+32:4\times BFA+35}$

The contents of Condition Register field BFA are copied to Condition Register field BF.

Special Registers Altered:

CR field BF

2.6 System Call Instructions

These instructions provide the means by which a program can call upon the system to perform a service.

System Call *SC-form*

sc LEV

17	///	///	//	LEV	//	1	/
0	6	11	16	20	27	30	31

System Call Vectored *SC-form*

scv LEV

17	///	///	//	LEV	//	0	1
0	6	11	16	20	27	30	31

These instructions call the system to perform a service. A complete description of these instructions can be found in Section 3.3.1 of Book III.

The first form of the instruction (***sc***) provides a single system call. The second form of the instruction (***scv***) provides the capability for 128 unique system calls.

The use of the LEV field is described in Book III. In the first form of the instruction the LEV values greater than 1 are reserved, and bits 0:5 of the LEV field (instruction bits 20:25) are treated as a reserved field.

When control is returned to the program that executed the *System Call* or *System Call Vectored* instruction, the contents of the registers will depend on the register conventions used by the program providing the system service.

These instructions are context synchronizing (see Book III).

Special Registers Altered:

Dependent on the system service

Programming Note

sc serves as both a basic and an extended mnemonic. The Assembler will recognize an ***sc*** mnemonic with one operand as the basic form, and an ***sc*** mnemonic with no operand as the extended form. In the extended form the LEV operand is omitted and assumed to be 0.

In application programs the value of the LEV operand for ***sc*** should be 0.

Programming Note

Since the ***scv*** instruction modifies the Count Register, programs should treat the contents of the Count Register as undefined after executing this instruction. See Section 3.3 of Book III.

<BHRB material moved to Chapter 8 of Book II.>

Chapter 3. Fixed-Point Facility

3.1 Fixed-Point Facility Overview

This chapter describes the registers and instructions that make up the Fixed-Point Facility.

3.2 Fixed-Point Facility Registers

3.2.1 General Purpose Registers

All manipulation of information is done in registers internal to the Fixed-Point Facility. The principal storage internal to the Fixed-Point Facility is a set of 32 General Purpose Registers (GPRs). See Figure 43.

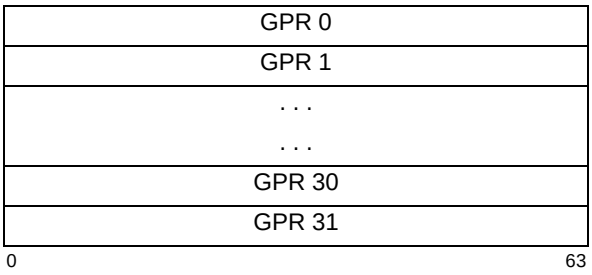


Figure 43. General Purpose Registers

Each GPR is a 64-bit register.

3.2.2 Fixed-Point Exception Register

The Fixed-Point Exception Register (XER) is a 64-bit register.

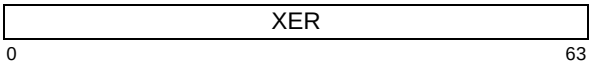


Figure 44. Fixed-Point Exception Register

The bit definitions for the Fixed-Point Exception Register are shown below. Here M=0 in 64-bit mode and M=32 in 32-bit mode.

The bits are set based on the operation of an instruction considered as a whole, not on intermediate results (e.g., the *Subtract From Carrying* instruction, the result of which is specified as the sum of three values, sets bits in the Fixed-Point Exception Register based on the entire operation, not on an intermediate sum).

Bit(s	Description
0:31	Reserved
32	***Summary Overflow*** (SO) The Summary Overflow bit is set to 1 whenever an instruction (except ***mtspr*** and ***addex***) sets the Overflow bit. Once set, the SO bit remains set until it is cleared by an ***mtspr*** instruction (specifying the XER). It is not altered by *Compare* instructions, or by other instructions (except ***mtspr*** to the XER and ***addex*** with operand `CY=0`) that cannot overflow. Executing an ***mtspr*** instruction to the XER, supplying the values 0 for SO and 1 for OV, causes SO to be set to 0 and OV to be set to 1. ***addex*** does not alter the contents of SO.
33	***Overflow*** (OV) The Overflow bit is set to indicate that an overflow has occurred during execution of an instruction. The Overflow bit can also used as an independent Carry bit by using the ***addex*** with operand `CY=0` instruction and avoiding other instructions that modify the Overflow bit (e.g., any XO-form instruction with `OE=1`). XO-form *Add*, *Subtract From*, and *Negate* instructions having `OE=1` set it to 1 if the carry out of bit `M` is not equal to the carry out of bit `M+1`, and set it to 0 otherwise.

XO-form *Multiply Low* and *Divide* instructions having OE=1 set it to 1 if the result cannot be represented in 64 bits (***mulld***, ***divd***, ***divde***, ***divdu***, ***divdeu***) or in 32 bits (***mullw***, ***divw***, ***divwe***, ***divwu***, ***divweu***), and set it to 0 otherwise.

addex with operand CY=0 sets OV to 1 if there is a carry out of bit M, and sets it to 0 otherwise.

The OV bit is not altered by *Compare* instructions, or by other instructions (except ***mtspr*** to the XER) that cannot overflow.

34 ***Carry*** (CA)
The Carry bit is set as follows, during execution of certain instructions. *Add Carrying*, *Subtract From Carrying*, *Add Extended*, and *Subtract From Extended* types of instructions set it to 1 if there is a carry out of bit M, and set it to 0 otherwise. *Shift Right Algebraic* instructions set it to 1 if any 1-bits have been shifted out of a negative operand, and set it to 0 otherwise. The CA bit is not altered by *Compare* instructions, or by other instructions (except *Shift Right Algebraic*, ***mtspr*** to the XER) that cannot carry.

35:43 Reserved

44 ***Overflow32*** (OV32)

OV32 is set whenever OV is implicitly set, and is set to the same value that OV is defined to be set to in 32-bit mode.

45 ***Carry32*** (CA32)

CA32 is set whenever CA is implicitly set, and is set to the same value that CA is defined to be set to in 32-bit mode.

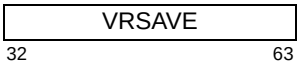
46:56 Reserved
Bits 48:55 are implemented, and can be read and written by software as if the bits contained a defined field.

57:63 This field specifies the number of bytes to be transferred by a *Load String Indexed* or *Store String Indexed* instruction.

Programming Note

Bits 48:55 of the XER correspond to bits 16:23 of the XER in the POWER Architecture. In the POWER Architecture bits 16:23 of the XER contain the comparison byte for the ***lscbx*** instruction. Power ISA lacks the ***lscbx*** instruction, but some application programs that run on processors that implement Power ISA may still use ***lscbx***, and privileged software may emulate the instruction. $XER_{48:55}$ may be assigned a meaning in a future version of the architecture, when POWER compatibility for ***lscbx*** is no longer needed, so these bits should not be used for purposes other than the ***lscbx*** comparison byte.

3.2.3 VR Save Register



The VR Save Register (VRSAVE) is a 32-bit register that can be used as a software use SPR; see Section 6.3.3.

3.3 Fixed-Point Facility Instructions

3.3.1 Fixed-Point Storage Access Instructions

The *Storage Access* instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.11.3 on page 27.

Programming Note

The ***la*** extended mnemonic permits computing an effective address as a *Load* or *Store* instruction would, but loads the address itself into a GPR rather than loading the value that is in storage at that address.

Programming Note

The DS field in DS-form *Storage Access* instructions is a word offset, not a byte offset like the D field in D-form *Storage Access* instructions. However, for programming convenience, Assemblers should support the specification of byte offsets for both forms of instruction.

3.3.1.1 Storage Access Exceptions

Storage accesses will cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (*Store* only), or if the program attempts to access storage that is unavailable.

3.3.2 Fixed-Point Load Instructions

The byte, halfword, word, or doubleword in storage addressed by EA is loaded into register RT.

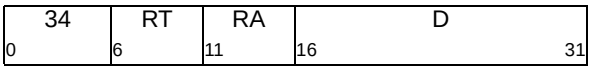
Many of the *Load* instructions have an “update” form, in which register RA is updated with the effective address. For these forms, if RA≠0 and RA≠RT, the effective address is placed into register RA and the storage element (byte, halfword, word, or doubleword) addressed by EA is loaded into RT.

Programming Note

In some implementations, the *Load Algebraic* and *Load with Update* instructions may have greater latency than other types of *Load* instructions. Moreover, *Load with Update* instructions may take longer to execute in some implementations than the corresponding pair of a non-update *Load* instruction and an *Add* instruction.

Load Byte and Zero *D-form*

lbz RT,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
RT ← 56 0 || MEM(EA, 1)
```

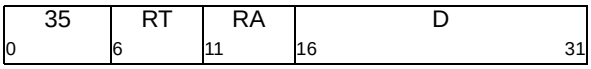
Let the effective address (EA) be the sum (RA|0)+ D. The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

Special Registers Altered:

None

Load Byte and Zero with Update *D-form*

lbzu RT,D(RA)




```
EA ← (RA) + EXTS(D)
RT ← 56 0 || MEM(EA, 1)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ D. The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

EA is placed into register RA.

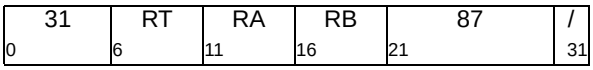
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Byte and Zero Indexed *X-form*

lbzx RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 56 0 || MEM(EA, 1)
```

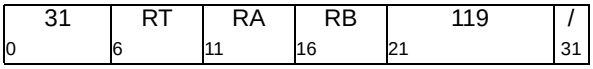
Let the effective address (EA) be the sum (RA|0)+ (RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

Special Registers Altered:

None

Load Byte and Zero with Update Indexed *X-form*

lbzux RT,RA,RB




```
EA ← (RA) + (RB)
RT ← 56 0 || MEM(EA, 1)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Halfword and Zero D-form

lhz RT,D(RA)

40	RT	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
RT ← 48 0 || MEM(EA, 2)
```

Let the effective address (EA) be the sum (RA|0)+ D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:

None

Load Halfword and Zero Indexed X-form

lhzx RT,RA,RB

31	RT	RA	RB	279	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 48 0 || MEM(EA, 2)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:

None

Load Halfword and Zero with Update D-form

lhzu RT,D(RA)

41	RT	RA	D
0	6	11	16 31

```
EA ← (RA) + EXTS(D)
RT ← 48 0 || MEM(EA, 2)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Halfword and Zero with Update Indexed X-form

lhzux RT,RA,RB

31	RT	RA	RB	311	/
0	6	11	16	21	31

```
EA ← (RA) + (RB)
RT ← 48 0 || MEM(EA, 2)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

EA is placed into register RA.

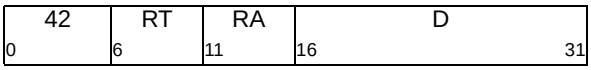
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Halfword Algebraic D-form

lha RT,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
RT ← EXTS(MEM(EA, 2))
```

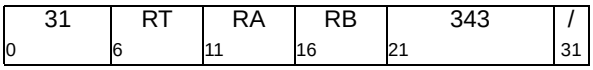
Let the effective address (EA) be the sum (RA|0)+ D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

Special Registers Altered:

None

Load Halfword Algebraic Indexed X-form

lhax RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← EXTS(MEM(EA, 2))
```

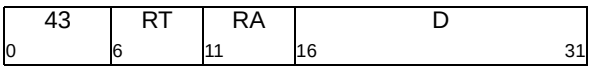
Let the effective address (EA) be the sum (RA|0)+ (RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

Special Registers Altered:

None

Load Halfword Algebraic with Update D-form

lhau RT,D(RA)




```
EA ← (RA) + EXTS(D)
RT ← EXTS(MEM(EA, 2))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

EA is placed into register RA.

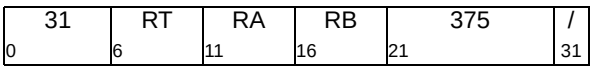
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Halfword Algebraic with Update Indexed X-form

lhaux RT,RA,RB




```
EA ← (RA) + (RB)
RT ← EXTS(MEM(EA, 2))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

EA is placed into register RA.

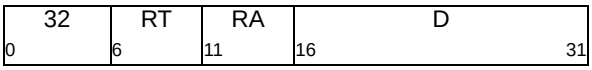
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Word and Zero D-form

lwz RT,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
RT ← 32 0 || MEM(EA, 4)
```

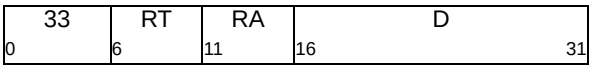
Let the effective address (EA) be the sum (RA|0)+ D. The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Load Word and Zero with Update D-form

lwzu RT,D(RA)




```
EA ← (RA) + EXTS(D)
RT ← 32 0 || MEM(EA, 4)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ D. The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

EA is placed into register RA.

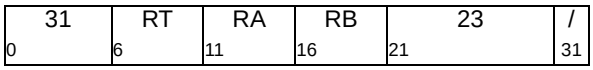
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Word and Zero Indexed X-form

lwzx RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 32 0 || MEM(EA, 4)
```

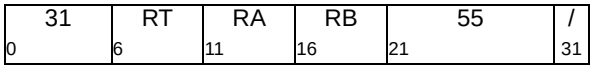
Let the effective address (EA) be the sum (RA|0)+ (RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Load Word and Zero with Update Indexed X-form

lwzux RT,RA,RB




```
EA ← (RA) + (RB)
RT ← 32 0 || MEM(EA, 4)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

3.3.2.1 64-bit Fixed-Point Load Instructions

Load Word Algebraic *DS-form*

lwa RT,DS(RA)

58	RT	RA	DS	2
0	6	11	16	30 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(DS || 0b00)
RT ← EXTS(MEM(EA, 4))
```

Let the effective address (EA) be the sum (RA|0)+ (DS||0b00). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

Special Registers Altered:

None

Load Word Algebraic Indexed *X-form*

lwax RT,RA,RB

31	RT	RA	RB	341	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
RT ← EXTS(MEM(EA, 4))
```

Let the effective address (EA) be the sum (RA|0)+ (RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

Special Registers Altered:

None

Load Word Algebraic with Update Indexed *X-form*

lwaux RT,RA,RB

31	RT	RA	RB	373	/
0	6	11	16	21	31

```
EA ← (RA) + (RB)
RT ← EXTS(MEM(EA, 4))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

EA is placed into register RA.

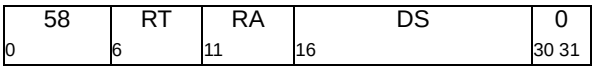
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Doubleword *DS-form*

ld RT,DS(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(DS || 0b00)
RT ← MEM(EA, 8)
```

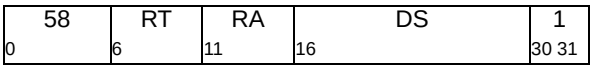
Let the effective address (EA) be the sum (RA|0)+ (DS||0b00). The doubleword in storage addressed by EA is loaded into RT.

Special Registers Altered:

None

Load Doubleword with Update *DS-form*

ldu RT,DS(RA)




```
EA ← (RA) + EXTS(DS || 0b00)
RT ← MEM(EA, 8)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (DS||0b00). The doubleword in storage addressed by EA is loaded into RT.

EA is placed into register RA.

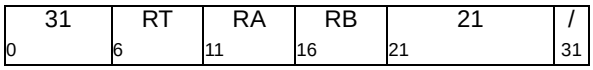
If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Doubleword Indexed *X-form*

ldx RT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← MEM(EA, 8)
```

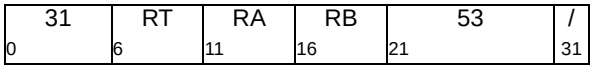
Let the effective address (EA) be the sum (RA|0)+ (RB). The doubleword in storage addressed by EA is loaded into RT.

Special Registers Altered:

None

Load Doubleword with Update Indexed *X-form*

ldux RT,RA,RB




```
EA ← (RA) + (RB)
RT ← MEM(EA, 8)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). The doubleword in storage addressed by EA is loaded into RT.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

3.3.3 Fixed-Point Store Instructions

The contents of register RS are stored into the byte, halfword, word, or doubleword in storage addressed by EA.

Many of the *Store* instructions have an "update" form, in which register RA is updated with the effective address. For these forms, the following rules apply.

- If RA≠0, the effective address is placed into register RA.
- If RS=RA, the contents of register RS are copied to the target storage element and then EA is placed into RA (RS).

Store Byte *D-form*

stb RS,D(RA)

38	RS	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 1) ← (RS)56:63
```

Let the effective address (EA) be the sum (RA|0)+ D. $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

Special Registers Altered:

None

Store Byte Indexed *X-form*

stbx RS,RA,RB

31	RS	RA	RB	215	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 1) ← (RS)56:63
```

Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

Special Registers Altered:

None

Store Byte with Update *D-form*

stbu RS,D(RA)

39	RS	RA	D
0	6	11	16 31

```
EA ← (RA) + EXTS(D)
MEM(EA, 1) ← (RS)56:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ D. $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Byte with Update Indexed *X-form*

stbux RS,RA,RB

31	RS	RA	RB	247	/
0	6	11	16	21	31

```
EA ← (RA) + (RB)
MEM(EA, 1) ← (RS)56:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Halfword *D-form*

sth RS,D(RA)

44	RS	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 2) ← (RS)48:63
```

Let the effective address (EA) be the sum (RA|0)+ D. $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

Special Registers Altered:

None

Store Halfword with Update *D-form*

sthu RS,D(RA)

45	RS	RA	D
0	6	11	16 31

```
EA ← (RA) + EXTS(D)
MEM(EA, 2) ← (RS)48:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ D. $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Halfword Indexed *X-form*

sthx RS,RA,RB

31	RS	RA	RB	407	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 2) ← (RS)48:63
```

Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

Special Registers Altered:

None

Store Halfword with Update Indexed *X-form*

sthux RS,RA,RB

31	RS	RA	RB	439	/
0	6	11	16	21	31

```
EA ← (RA) + (RB)
MEM(EA, 2) ← (RS)48:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

EA is placed into register RA.

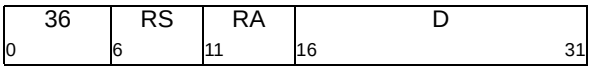
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Word *D-form*

stw RS,D(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 4) ← (RS)32:63
```

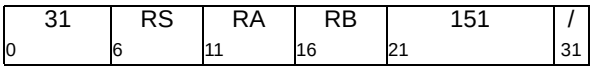
Let the effective address (EA) be the sum (RA|0)+ D. $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

Special Registers Altered:

None

Store Word Indexed *X-form*

stwx RS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (RS)32:63
```

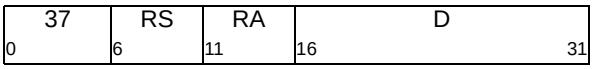
Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

Special Registers Altered:

None

Store Word with Update *D-form*

stwu RS,D(RA)




```
EA ← (RA) + EXTS(D)
MEM(EA, 4) ← (RS)32:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ D. $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

EA is placed into register RA.

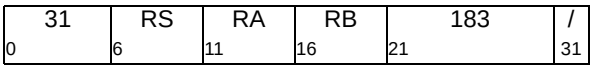
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Word with Update Indexed *X-form*

stwux RS,RA,RB




```
EA ← (RA) + (RB)
MEM(EA, 4) ← (RS)32:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

3.3.3.1 64-bit Fixed-Point Store Instructions

Store Doubleword *DS-form*

std RS,DS(RA)

62	RS	RA	DS	0
0	6	11	16	30 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)
```

Let the effective address (EA) be the sum (RA|0)+ (DS||0b00). (RS) is stored into the doubleword in storage addressed by EA.

Special Registers Altered:

None

Store Doubleword Indexed *X-form*

stdx RS,RA,RB

31	RS	RA	RB	149	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← (RS)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). (RS) is stored into the doubleword in storage addressed by EA.

Special Registers Altered:

None

Store Doubleword with Update *DS-form*

stdu RS,DS(RA)

62	RS	RA	DS	1
0	6	11	16	30 31

```
EA ← (RA) + EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (DS||0b00). (RS) is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Doubleword with Update Indexed *X-form*

stdux RS,RA,RB

31	RS	RA	RB	181	/
0	6	11	16	21	31

```
EA ← (RA) + (RB)
MEM(EA, 8) ← (RS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). (RS) is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

3.3.4 Fixed Point Load and Store Quadword Instructions

For ***lq***, the quadword in storage addressed by EA is loaded into an even-odd pair of GPRs as follows. In Big-Endian mode, the even-numbered GPR is loaded with the doubleword from storage addressed by EA and the odd-numbered GPR is loaded with the doubleword addressed by EA+8. In Little-Endian mode, the even-numbered GPR is loaded with the byte-reversed doubleword from storage addressed by EA+8 and the odd-numbered GPR is loaded with the byte-reversed doubleword addressed by EA.

In the preferred form of the *Load Qudword* instruction RA ≠ RTp+1.

For ***stq***, the contents of an even-odd pair of GPRs is stored into the quadword in storage addressed by EA as follows. In Big-Endian mode, the even-numbered GPR is stored into the doubleword in storage addressed by EA and the odd-numbered GPR is stored into the doubleword addressed by EA+8. In Little-Endian mode, the even-numbered GPR is stored byte-reversed into the doubleword in storage addressed by EA+8 and the odd-numbered GPR is stored byte-reversed into the doubleword addressed by EA.

Programming Note

The ***lq*** and ***stq*** instructions exist primarily to permit software to access quadwords in storage "atomically"; see Section 1.4 of Book II. Because GPRs are 64 bits long, the Fixed-Point Facility on many designs is optimized for storage accesses of at most eight bytes. On such designs, the quadword atomicity required for ***lq*** and ***stq*** makes these instructions complex to implement, with the result that the instructions may perform less well on these designs than the corresponding two *Load Doubleword* or *Store Doubleword* instructions.

The complexity of providing quadword atomicity may be especially great for storage that is Write Through Required or Caching Inhibited (see Section 1.6 of Book II). This is why ***lq*** and ***stq*** are permitted to cause the data storage error handler to be invoked if the specified storage location is in either of these kinds of storage (see Section 3.3.1.1).

Load Quadword *DQ-form*

lq RTp,DQ(RA)

56	RTp	RA	DQ	///
0	6	11	16	28 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(DQ || 0b0000)
RTp ← MEM(EA, 16)
```

Let the effective address (EA) be the sum (RA|0)+(DQ||0b0000). The quadword in storage addressed by EA is loaded into register pair RTp.

If RTp is odd or RTp=RA, the instruction form is invalid. If RTp=RA, an attempt to execute this instruction will invoke the system illegal instruction error handler. (The RTp=RA case includes the case of RTp=RA=0.)

The quadword in storage addressed by EA is loaded into an even-odd pair of GPRs as follows. In Big-Endian mode, the even-numbered GPR is loaded with the doubleword from storage addressed by EA and the odd-numbered GPR is loaded with the doubleword addressed by EA+8. In Little-Endian mode, the even-numbered GPR is loaded with the byte-reversed doubleword from storage addressed by EA+8 and the odd-numbered GPR is loaded with the byte-reversed doubleword addressed by EA.

Programming Note

In versions of the architecture prior to V. 2.07, this instruction was privileged.

Special Registers Altered:

None

Store Quadword *DS-form*

stq RSp,DS(RA)

62	RSp	RA	DS	2
0	6	11	16	30 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(DS || 0b00)
MEM(EA, 16) ← RSp
```

Let the effective address (EA) be the sum (RA|0)+ (DS||0b00). The contents of register pair RSp are stored into the quadword in storage addressed by EA.

If RSp is odd, the instruction form is invalid.

The contents of an even-odd pair of GPRs is stored into the quadword in storage addressed by EA as follows. In Big-Endian mode, the even-numbered GPR is stored into the doubleword in storage addressed by EA and the odd-numbered GPR is stored into the doubleword addressed by EA+8. In Little-Endian mode, the even-numbered GPR is stored byte-reversed into the doubleword in storage addressed by EA+8 and the odd-numbered GPR is stored byte-reversed into the doubleword addressed by EA.

Programming Note

In versions of the architecture prior to V. 2.07, this instruction was privileged.

Special Registers Altered:

None

3.3.5 Fixed-Point Load and Store with Byte Reversal Instructions

Programming Note

These instructions have the effect of loading and storing data in the opposite byte ordering from that which would be used by other *Load* and *Store* instructions.

Programming Note

In some implementations, the *Load Byte-Reverse* instructions may have greater latency than other *Load* instructions.

Load Halfword Byte-Reverse Indexed X-form

lhbrx RT,RA,RB

31	RT	RA	RB	790	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
load_data ← MEM(EA, 2)
RT ← 48 0 || load_data8:15 || load_data0:7
```

Let the effective address (EA) be the sum (RA|0)+(RB). Bits 0:7 of the halfword in storage addressed by EA are loaded into $RT_{56:63}$. Bits 8:15 of the halfword in storage addressed by EA are loaded into $RT_{48:55}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:

None

Store Halfword Byte-Reverse Indexed X-form

sthbrx RS,RA,RB

31	RS	RA	RB	918	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 2) ← (RS)56:63 || (RS)48:55
```

Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{56:63}$ are stored into bits 0:7 of the halfword in storage addressed by EA. $(RS)_{48:55}$ are stored into bits 8:15 of the halfword in storage addressed by EA.

Special Registers Altered:

None

Load Word Byte-Reverse Indexed X-form

lwbrx RT,RA,RB

31	RT	RA	RB	534	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
load_data ← MEM(EA, 4)
RT ← 32 0 || load_data24:31 || load_data16:23
         || load_data8:15 || load_data0:7
```

Let the effective address (EA) be the sum (RA|0)+ (RB). Bits 0:7 of the word in storage addressed by EA are loaded into $RT_{56:63}$. Bits 8:15 of the word in storage addressed by EA are loaded into $RT_{48:55}$. Bits 16:23 of the word in storage addressed by EA are loaded into $RT_{40:47}$. Bits 24:31 of the word in storage addressed by EA are loaded into $RT_{32:39}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Store Word Byte-Reverse Indexed X-form

stwbrx RS,RA,RB

31	RS	RA	RB	662	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (RS)56:63 || (RS)48:55 || (RS)40:47
             ||(RS)32:39
```

Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{56:63}$ are stored into bits 0:7 of the word in storage addressed by EA. $(RS)_{48:55}$ are stored into bits 8:15 of the word in storage addressed by EA. $(RS)_{40:47}$ are stored into bits 16:23 of the word in storage addressed by EA. $(RS)_{32:39}$ are stored into bits 24:31 of the word in storage addressed by EA.

Special Registers Altered:

None

3.3.5.1 64-Bit Load and Store with Byte Reversal Instructions

Load Doubleword Byte-Reverse Indexed X-form

ldbrx RT,RA,RB

31	RT	RA	RB	532	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
load_data ← MEM(EA, 8)
RT ←  load_data56:63 || load_data48:55
   || load_data40:47 || load_data32:39
   || load_data24:31 || load_data16:23
   || load_data8:15  || load_data0:7
```

Let the effective address (EA) be the sum (RA|0)+(RB). Bits 0:7 of the doubleword in storage addressed by EA are loaded into $RT_{56:63}$. Bits 8:15 of the doubleword in storage addressed by EA are loaded into $RT_{48:55}$. Bits 16:23 of the doubleword in storage addressed by EA are loaded into $RT_{40:47}$. Bits 24:31 of the doubleword in storage addressed by EA are loaded into $RT_{32:39}$. Bits 32:39 of the doubleword in storage addressed by EA are loaded into $RT_{24:31}$. Bits 40:47 of the doubleword in storage addressed by EA are loaded into $RT_{16:23}$. Bits 48:55 of the doubleword in storage addressed by EA are loaded into $RT_{8:15}$. Bits 56:63 of the doubleword in storage addressed by EA are loaded into $RT_{0:7}$.

Special Registers Altered:

None

Store Doubleword Byte-Reverse Indexed X-form

stdbrx RS,RA,RB

31	RS	RA	RB	660	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← (RS)56:63 || (RS)48:55
          || (RS)40:47 || (RS)32:39
          || (RS)24:31 || (RS)16:23
          || (RS)8:15  || (RS)0:7
```

Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{56:63}$ are stored into bits 0:7 of the doubleword in storage addressed by EA. $(RS)_{48:55}$ are stored into bits 8:15 of the doubleword in storage addressed by EA. $(RS)_{40:47}$ are stored into bits 16:23 of the doubleword in storage addressed by EA. $(RS)_{32:39}$ are stored into bits 23:31 of the doubleword in storage addressed by EA. $(RS)_{24:31}$ are stored into bits 32:39 of the doubleword in storage addressed by EA. $(RS)_{16:23}$ are stored into bits 40:47 of the doubleword in storage addressed by EA. $(RS)_{8:15}$ are stored into bits 48:55 of the doubleword in storage addressed by EA. $(RS)_{0:7}$ are stored into bits 56:63 of the doubleword in storage addressed by EA.

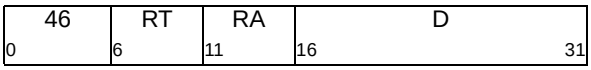
Special Registers Altered:

None

3.3.6 Fixed-Point Load and Store Multiple Instructions

Load Multiple Word *D-form*

lmw RT,D(RA)




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
r ← RT
do while r ≤ 31
   GPR(r) ← 32 0 || MEM(EA, 4)
   r ← r + 1
   EA ← EA + 4
```

Let n = (32 - RT). Let the effective address (EA) be the sum (RA|0)+ D.

n consecutive words starting at EA are loaded into the low-order 32 bits of GPRs RT through 31. The high-order 32 bits of these GPRs are set to zero.

If RA is in the range of registers to be loaded, including the case in which RA=0, the instruction form is invalid.

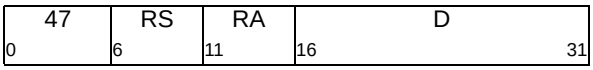
This instruction is not supported in Little-Endian mode. If it is executed in Little-Endian mode, the system alignment error handler is invoked.

Special Registers Altered:

None

Store Multiple Word *D-form*

stmw RS,D(RA)




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
r ← RS
do while r ≤ 31
   MEM(EA, 4) ← GPR(r)32:63
   r ← r + 1
   EA ← EA + 4
```

Let n = (32 - RS). Let the effective address (EA) be the sum (RA|0)+ D.

n consecutive words starting at EA are stored from the low-order 32 bits of GPRs RS through 31.

This instruction is not supported in Little-Endian mode. If it is executed in Little-Endian mode, the system alignment error handler is invoked.

Special Registers Altered:

None

3.3.7 Fixed-Point Move Assist Instructions [Phased Out]

The *Move Assist* instructions allow movement of an arbitrary sequence of bytes from storage to registers or from registers to storage without concern for alignment. These instructions can be used for a short move between arbitrary storage locations or to initiate a long move between unaligned storage fields.

The *Move Assist* instructions have preferred forms; see Section 1.9.1, “Preferred Instruction Forms” on page 23. In the preferred forms, register usage satisfies the following rules.

- RS = 4 or 5
- RT = 4 or 5
- last register loaded/stored ≤ 12

For some implementations, using GPR 4 for RS and RT may result in slightly faster execution than using GPR 5.

Load String Word Immediate *X-form*

lswi RT,RA,NB

31	RT	RA	NB	597	/
0	6	11	16	21	31

```
if RA = 0 then EA ← 0
else           EA ← (RA)
if NB = 0 then n ← 32
else           n ← NB
r ← RT - 1
i ← 32
do while n > 0
   if i = 32 then
     r ← r + 1 (mod 32)
     GPR(r) ← 0
   GPR(r)i:i+7 ← MEM(EA, 1)
   i ← i + 8
   if i = 64 then i ← 32
   EA ← EA + 1
   n ← n - 1
```

Let the effective address (EA) be (RA|0). Let n = NB if NB≠0, n = 32 if NB=0; n is the number of bytes to load. Let nr=CEIL(n/4); nr is the number of registers to receive data.

n consecutive bytes starting at EA are loaded into GPRs RT through RT+nr-1. Data are loaded into the low-order four bytes of each GPR; the high-order four bytes are set to 0.

Bytes are loaded left to right in each register. The sequence of registers wraps around to GPR 0 if required. If the low-order four bytes of register RT+nr-1 are only partially filled, the unfilled low-order byte(s) of that register are set to 0.

If RA is in the range of registers to be loaded, including the case in which RA=0, the instruction form is invalid.

This instruction is not supported in Little-Endian mode. If it is executed in Little-Endian mode, the system alignment error handler is invoked.

Special Registers Altered:

None

Load String Word Indexed *X-form*

lswx RT,RA,RB

31	RT	RA	RB	533	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
n ← XER57:63
r ← RT - 1
i ← 32
RT ← undefined
do while n > 0
   if i = 32 then
    r ← r + 1 (mod 32)
    GPR(r) ← 0
   GPR(r)i:i+7 ← MEM(EA, 1)
   i ← i + 8
   if i = 64 then i ← 32
   EA ← EA + 1
   n ← n - 1
```

Let the effective address (EA) be the sum (RA|0)+ (RB). Let n=$XER_{57:63}$; n is the number of bytes to load. Let nr=CEIL(n/4); nr is the number of registers to receive data.

If n>0, n consecutive bytes starting at EA are loaded into GPRs RT through RT+nr-1. Data are loaded into the low-order four bytes of each GPR; the high-order four bytes are set to 0.

Bytes are loaded left to right in each register. The sequence of registers wraps around to GPR 0 if required. If the low-order four bytes of register RT+nr-1 are only partially filled, the unfilled low-order byte(s) of that register are set to 0.

If n=0, the contents of register RT are undefined.

If RA or RB is in the range of registers to be loaded, including the case in which RA=0, the instruction is treated as if the instruction form were invalid. If RT=RA or RT=RB, the instruction form is invalid.

This instruction is not supported in Little-Endian mode. If it is executed in Little-Endian mode and n>0, the system alignment error handler is invoked.

Special Registers Altered:

None

Store String Word Immediate *X-form*

stswi RS,RA,NB

31	RS	RA	NB	725	/
0	6	11	16	21	31

```
if RA = 0 then EA ← 0
else           EA ← (RA)
if NB = 0 then n ← 32
else           n ← NB
r ← RS - 1
i ← 32
do while n > 0
  if i = 32 then r ← r + 1 (mod 32)
  MEM(EA, 1) ← GPR(r)i:i+7
   i ← i + 8
  if i = 64 then i ← 32
  EA ← EA + 1
  n ← n - 1
```

Let the effective address (EA) be (RA|0). Let n = NB if NB≠0, n = 32 if NB=0; n is the number of bytes to store. Let nr =CEIL(n/4); nr is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs RS through RS+nr-1. Data are stored from the low-order four bytes of each GPR.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR 0 if required.

This instruction is not supported in Little-Endian mode. If it is executed in Little-Endian mode, the system alignment error handler is invoked.

Special Registers Altered:

None

Store String Word Indexed *X-form*

stswx RS,RA,RB

31	RS	RA	RB	661	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
n ← XER57:63
r ← RS - 1
i ← 32
do while n > 0
  if i = 32 then r ← r + 1 (mod 32)
  MEM(EA, 1) ← GPR(r)i:i+7
    i ← i + 8
  if i = 64 then i ← 32
  EA ← EA + 1
  n ← n - 1
```

Let the effective address (EA) be the sum (RA|0)+ (RB). Let n = $XER_{57:63}$; n is the number of bytes to store. Let nr = CEIL(n/4); nr is the number of registers to supply data.

If n>0, n consecutive bytes starting at EA are stored from GPRs RS through RS+nr-1. Data are stored from the low-order four bytes of each GPR.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR 0 if required.

If n=0, no bytes are stored.

This instruction is not supported in Little-Endian mode. If it is executed in Little-Endian mode and n>0, the system alignment error handler is invoked.

Special Registers Altered:

None

3.3.8 Other Fixed-Point Instructions

The remainder of the fixed-point instructions use the contents of the General Purpose Registers (GPRs) as source operands, and place results into GPRs, into the Fixed-Point Exception Register (XER), and into Condition Register fields. In addition, the *Trap* instructions test the contents of a GPR or XER bit, invoking the system trap handler if the result of the specified test is true.

These instructions treat the source operands as signed integers unless the instruction is explicitly identified as performing an unsigned operation.

The X-form and XO-form instructions with Rc=1, and the D-form instructions ***addic.***, ***andi.***, and ***andis.***, set the first three bits of CR Field 0 to characterize the result placed into the target register. In 64-bit mode, these bits are set by signed comparison of the result to zero. In 32-bit mode, these bits are set by signed comparison of the low-order 32 bits of the result to zero.

Unless otherwise noted and when appropriate, when CR Field 0 and the XER are set they reflect the value placed into the target register.

Programming Note

Instructions with the OE bit set or that set CA and CA32 may execute slowly or may prevent the execution of subsequent instructions until the instruction has completed.

3.3.9 Fixed-Point Arithmetic Instructions

The XO-form *Arithmetic* instructions with Rc=1, and the D-form *Arithmetic* instruction ***addic.***, set the first three bits of CR Field 0 as described in Section 3.3.8, “Other Fixed-Point Instructions”.

addic, ***addic.***, ***subfic***, ***addc***, ***subfc***, ***adde***, ***subfe***, ***addme***, ***subfme***, ***addze***, and ***subfze*** always set CA, to reflect the carry out of bit 0 in 64-bit mode and out of bit 32 in 32-bit mode. These instructions also always set CA32 to reflect the carry out of bit 32. The XO-form *Arithmetic* instructions set SO, OV, and OV32 when OE=1 to reflect overflow of the result. Except for the *Multiply Low* and *Divide* instructions, the setting of SO and OV is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode, while OV32 reflects overflow of the low-order 32-bit result independent of the mode. For XO-form *Multiply Low* and *Divide* instructions, the setting of SO, OV, and OV32 is mode-independent, and reflects overflow of the 64-bit result for ***mulld***, ***divd***, ***divde***, ***divdu*** and ***divdeu***, and overflow of the low-order 32-bit result for ***mullw***, ***divw***, ***divwe***, ***divwu***, and ***divweu***.

Programming Note

Notice that CR Field 0 may not reflect the “true” (infinitely precise) result if overflow occurs.

Extended mnemonics for addition and subtraction

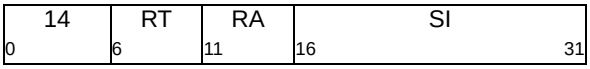
Several extended mnemonics are provided that use the *Add Immediate* and *Add Immediate Shifted* instructions to load an immediate value or an address into a target register. Some of these are shown as examples with the two instructions.

The Power ISA supplies *Subtract From* instructions, which subtract the second operand from the third. A set of extended mnemonics is provided that use the more “normal” order, in which the third operand is subtracted from the second, with the third operand being either an immediate field or a register. Some of these are shown as examples with the appropriate *Add* and *Subtract From* instructions.

See Appendix C for additional extended mnemonics.

Add Immediate — *D-form*

addi RT,RA,SI




```
if RA = 0 then RT ← EXTS(SI)
else           RT ← (RA) + EXTS(SI)
```

The sum (RA|0) + SI is placed into register RT.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Add Immediate*:

Extended:		Equivalent to:	
li	Rx,value	addi	Rx,0,value
la	Rx,disp(Ry)	addi	Rx,Ry,disp
subi	Rx,Ry,value	addi	Rx,Ry,-value

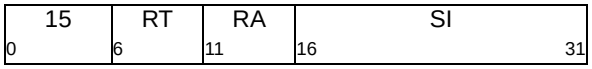
Programming Note

addi, ***addis***, ***add***, and ***subf*** are the preferred instructions for addition and subtraction, because they set few status bits.

Notice that ***addi*** and ***addis*** use the value 0, not the contents of GPR 0, if RA=0.

Add Immediate Shifted — *D-form*

addis RT,RA,SI




```
if RA = 0 then RT ← EXTS(SI || 16 0)
else           RT ← (RA) + EXTS(SI || 16 0)
```

The sum (RA|0) + (SI || 0x0000) is placed into register RT.

Special Registers Altered:

None

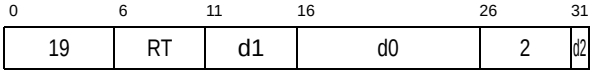
Extended Mnemonics:

Examples of extended mnemonics for *Add Immediate Shifted*:

Extended:		Equivalent to:	
lis	Rx,value	addis	Rx,0,value
subis	Rx,Ry,value	addis	Rx,Ry,-value

Add PC Immediate Shifted *DX-form*

addpcis RT,D




```
D ← d0||d1||d2
RT ← NIA + EXTS(D || 16 0)
```

The sum of NIA + (D || 0x0000) is placed into register RT.

Special Registers Altered:

None

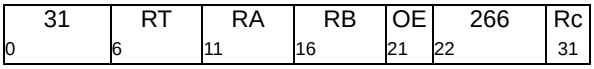
Extended Mnemonics:

Examples of extended mnemonics for *Add PC Immediate Shifted*:

Extended:		Equivalent to:	
lnia	Rx	addpcis	Rx,0
subpcis	Rx,value	addpcis	Rx,-value

Add *XO-form*

add	RT,RA,RB	(OE=0 Rc=0)
add.	RT,RA,RB	(OE=0 Rc=1)
addo	RT,RA,RB	(OE=1 Rc=0)
addo.	RT,RA,RB	(OE=1 Rc=1)



RT ← (RA) + (RB)

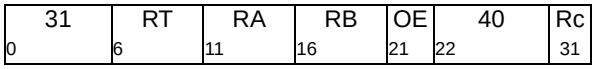
The sum (RA) + (RB) is placed into register RT.

Special Registers Altered:

CR0 (if Rc=1)
SO OV OV32 (if OE=1)

Subtract From *XO-form*

subf	RT,RA,RB	(OE=0 Rc=0)
subf.	RT,RA,RB	(OE=0 Rc=1)
subfo	RT,RA,RB	(OE=1 Rc=0)
subfo.	RT,RA,RB	(OE=1 Rc=1)



RT ← ¬(RA) + (RB) + 1

The sum ¬(RA) + (RB) +1 is placed into register RT.

Special Registers Altered:

CR0 (if Rc=1)
SO OV OV32 (if OE=1)

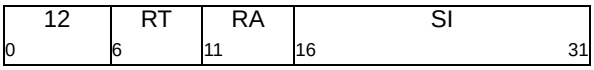
Extended Mnemonics:

Example of extended mnemonics for *Subtract From*:

Extended:	Equivalent to:
sub Rx,Ry,Rz	subf Rx,Rz,Ry

Add Immediate Carrying *D-form*

addic RT,RA,SI



RT ← (RA) + EXTS(SI)

The sum (RA) + SI is placed into register RT.

Special Registers Altered:

CA CA32

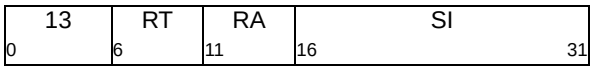
Extended Mnemonics:

Example of extended mnemonics for *Add Immediate Carrying*:

Extended:	Equivalent to:
subic Rx,Ry,value	addic Rx,Ry,-value

Add Immediate Carrying and Record *D-form*

addic. RT,RA,SI



RT ← (RA) + EXTS(SI)

The sum (RA) + SI is placed into register RT.

Special Registers Altered:

CR0 CA CA32

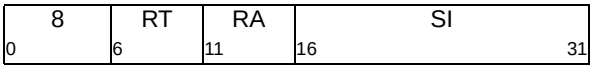
Extended Mnemonics:

Example of extended mnemonics for *Add Immediate Carrying and Record*:

Extended:	Equivalent to:
subic. Rx,Ry,value	addic. Rx,Ry,-value

Subtract From Immediate Carrying *D-form*

subfic RT,RA,SI



RT ← ¬(RA) + EXTS(SI) + 1

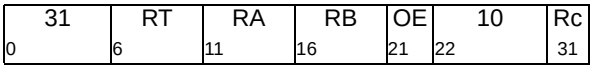
The sum ¬(RA) + SI + 1 is placed into register RT.

Special Registers Altered:

CA CA32

Add Carrying *XO-form*

addc	RT,RA,RB	(OE=0 Rc=0)
addc.	RT,RA,RB	(OE=0 Rc=1)
addco	RT,RA,RB	(OE=1 Rc=0)
addco.	RT,RA,RB	(OE=1 Rc=1)



RT ← (RA) + (RB)

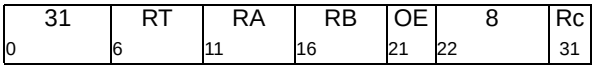
The sum (RA) + (RB) is placed into register RT.

Special Registers Altered:

CA CA32
CR0 (if Rc=1)
SO OV OV32 (if OE=1)

Subtract From Carrying *XO-form*

subfc	RT,RA,RB	(OE=0 Rc=0)
subfc.	RT,RA,RB	(OE=0 Rc=1)
subfco	RT,RA,RB	(OE=1 Rc=0)
subfco.	RT,RA,RB	(OE=1 Rc=1)



RT ← ¬(RA) + (RB) + 1

The sum ¬(RA) + (RB) + 1 is placed into register RT.

Special Registers Altered:

CA CA32
CR0 (if Rc=1)
SO OV OV32 (if OE=1)

Extended Mnemonics:

Example of extended mnemonics for *Subtract From Carrying*:

Extended:	**Equivalent to:**
subc Rx,Ry,Rz	subfc Rx,Rz,Ry

Add Extended *XO-form*

adde	RT,RA,RB	(OE=0 Rc=0)
adde.	RT,RA,RB	(OE=0 Rc=1)
addeo	RT,RA,RB	(OE=1 Rc=0)
addeo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	138	Rc
0	6	11	16	21	22	31

RT ← (RA) + (RB) + CA

The sum (RA) + (RB) + CA is placed into register RT.

Special Registers Altered:

CA CA32
CR0 (if Rc=1)
SO OV OV32 (if OE=1)

Add to Minus One Extended *XO-form*

addme	RT,RA	(OE=0 Rc=0)
addme.	RT,RA	(OE=0 Rc=1)
addmeo	RT,RA	(OE=1 Rc=0)
addmeo.	RT,RA	(OE=1 Rc=1)

31	RT	RA	///	OE	234	Rc
0	6	11	16	21	22	31

RT ← (RA) + CA - 1

The sum (RA) + CA + $^{64}1$ is placed into register RT.

Special Registers Altered:

CA CA32
CR0 (if Rc=1)
SO OV OV32 (if OE=1)

Subtract From Extended *XO-form*

subfe	RT,RA,RB	(OE=0 Rc=0)
subfe.	RT,RA,RB	(OE=0 Rc=1)
subfeo	RT,RA,RB	(OE=1 Rc=0)
subfeo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	136	Rc
0	6	11	16	21	22	31

RT ← ¬(RA) + (RB) + CA

The sum ¬(RA) + (RB) + CA is placed into register RT.

Special Registers Altered:

CA CA32
CR0 (if Rc=1)
SO OV OV32 (if OE=1)

Subtract From Minus One Extended *XO-form*

subfme	RT,RA	(OE=0 Rc=0)
subfme.	RT,RA	(OE=0 Rc=1)
subfmeo	RT,RA	(OE=1 Rc=0)
subfmeo.	RT,RA	(OE=1 Rc=1)

31	RT	RA	///	OE	232	Rc
0	6	11	16	21	22	31

RT ← ¬(RA) + CA - 1

The sum ¬(RA) + CA + $^{64}1$ is placed into register RT.

Special Registers Altered:

CA CA32
CR0 (if Rc=1)
SO OV OV32 (if OE=1)

Add Extended using alternate carry bit Z23-form

addex RT,RA,RB,CY

31	RT	RA	RB	CY	170	/
0	6	11	16	21	23	31

```
if CY=0 then RT ← (RA) + (RB) + OV
```

For CY=0, the sum (RA) + (RB) + OV is placed into register RT.

For CY=0, OV is set to 1 if there is a carry out of bit 0 of the sum in 64-bit mode or there is a carry out of bit 32 of the sum in 32-bit mode, and set to 0 otherwise. OV32 is set to 1 if there is a carry out of bit 32 bit of the sum.

CY=1, CY=2, and CY=3 are reserved.

Special Registers Altered:

OV OV32 (if CY=0)

Programming Note

An ***addc***-equivalent instruction using OV is not provided. An equivalent capability can be emulated by first initializing OV to 0, then using ***addex***. OV can be initialized to 0 using ***subfo***, subtracting any operand from itself.

Add to Zero Extended *XO-form*

addze	RT,RA	(OE=0 Rc=0)
addze.	RT,RA	(OE=0 Rc=1)
addzeo	RT,RA	(OE=1 Rc=0)
addzeo.	RT,RA	(OE=1 Rc=1)

31	RT	RA	///	OE	202	Rc
0	6	11	16	21	22	31

```
RT ← (RA) + CA
```

The sum (RA) + CA is placed into register RT.

Special Registers Altered:

CA CA32
CR0 (if Rc=1)
SO OV OV32 (if OE=1)

Subtract From Zero Extended *XO-form*

subfze	RT,RA	(OE=0 Rc=0)
subfze.	RT,RA	(OE=0 Rc=1)
subfzeo	RT,RA	(OE=1 Rc=0)
subfzeo.	RT,RA	(OE=1 Rc=1)

31	RT	RA	///	OE	200	Rc
0	6	11	16	21	22	31

```
RT ← ¬(RA) + CA
```

The sum ¬(RA) + CA is placed into register RT.

Special Registers Altered:

CA CA32
CR0 (if Rc=1)
SO OV OV32 (if OE=1)

Programming Note

The setting of CA and CA32 by the *Add* and *Subtract From* instructions, including the Extended versions thereof, is mode-dependent. If a sequence of these instructions is used to perform extended-precision addition or subtraction, the same mode should be used throughout the sequence.

Negate *XO-form*

neg	RT,RA	(OE=0 Rc=0)
neg.	RT,RA	(OE=0 Rc=1)
nego	RT,RA	(OE=1 Rc=0)
nego.	RT,RA	(OE=1 Rc=1)

31	RT	RA	///	OE	104	Rc
0	6	11	16	21	22	31

```
RT ← ¬(RA) + 1
```

The sum ¬(RA) + 1 is placed into register RT.

If the processor is in 64-bit mode and register RA contains the most negative 64-bit number (0x8000_0000_0000_0000), the result is the most negative number and, if OE=1, OV and OV32 are set to 1. Similarly, if the processor is in 32-bit mode and $(RA)_{32:63}$ contain the most negative 32-bit number (0x8000_0000), the low-order 32 bits of the result contain the most negative 32-bit number and, if OE=1, OV and OV32 are set to 1.

Special Registers Altered:

CR0 (if Rc=1)
SO OV OV32 (if OE=1)

Multiply Low Immediate *D-form*

mulli RT,RA,SI

7	RT	RA	SI
0	6	11	16 31

$prod_{0:127} \leftarrow (RA) \times EXTS(SI)$
$RT \leftarrow prod_{64:127}$

The 64-bit first operand is (RA). The 64-bit second operand is the sign-extended value of the SI field. The low-order 64 bits of the 128-bit product of the operands are placed into register RT.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

None

Multiply Low Word *XO-form*

mullw	RT,RA,RB	(OE=0 Rc=0)
mullw.	RT,RA,RB	(OE=0 Rc=1)
mullwo	RT,RA,RB	(OE=1 Rc=0)
mullwo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	235	Rc
0	6	11	16	21	22	31

$RT \leftarrow (RA)_{32:63} \times (RB)_{32:63}$

The 32-bit operands are the low-order 32 bits of RA and of RB. The 64-bit product of the operands is placed into register RT.

If OE=1 then OV and OV32 are set to 1 if the product cannot be represented in 32 bits.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0	(if Rc=1)
SO OV OV32	(if OE=1)

Programming Note

For ***mulli*** and ***mullw***, the low-order 32 bits of the product are the correct 32-bit product for 32-bit mode.

For ***mulli*** and ***mulld***, the low-order 64 bits of the product are independent of whether the operands are regarded as signed or unsigned 64-bit integers. For ***mulli*** and ***mullw***, the low-order 32 bits of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

Multiply High Word *XO-form*

mulhw	RT,RA,RB	(Rc=0)
mulhw.	RT,RA,RB	(Rc=1)

31	RT	RA	RB	/	75	Rc
0	6	11	16	21	22	31

$prod_{0:63} \leftarrow (RA)_{32:63} \times (RB)_{32:63}$
$RT_{32:63} \leftarrow prod_{0:31}$
$RT_{0:31} \leftarrow$ undefined

The 32-bit operands are the low-order 32 bits of RA and of RB. The high-order 32 bits of the 64-bit product of the operands are placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)

Multiply High Word Unsigned *XO-form*

mulhwu	RT,RA,RB	(Rc=0)
mulhwu.	RT,RA,RB	(Rc=1)

31	RT	RA	RB	/	11	Rc
0	6	11	16	21	22	31

$prod_{0:63} \leftarrow (RA)_{32:63} \times (RB)_{32:63}$
$RT_{32:63} \leftarrow prod_{0:31}$
$RT_{0:31} \leftarrow$ undefined

The 32-bit operands are the low-order 32 bits of RA and of RB. The high-order 32 bits of the 64-bit product of the operands are placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined.

Both operands and the product are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero.

Special Registers Altered:

CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)

Divide Word *XO-form*

divw	RT,RA,RB	(OE=0 Rc=0)
divw.	RT,RA,RB	(OE=0 Rc=1)
divwo	RT,RA,RB	(OE=1 Rc=0)
divwo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	491	Rc
0	6	11	16	21	22	31

$dividend_{0:31} \leftarrow (RA)_{32:63}$
$divisor_{0:31} \leftarrow (RB)_{32:63}$
$RT_{32:63} \leftarrow dividend \div divisor$
$RT_{0:31} \leftarrow$ undefined

The 32-bit dividend is $(RA)_{32:63}$. The 32-bit divisor is $(RB)_{32:63}$. The 32-bit quotient is placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r < |divisor|$ if the dividend is nonnegative, and $-|divisor| < r \le 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

```
0x8000_0000 ÷ -1
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV and OV32 are set to 1.

Special Registers Altered:

CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
SO OV OV32 (if OE=1)

Programming Note

The 32-bit signed remainder of dividing $(RA)_{32:63}$ by $(RB)_{32:63}$ can be computed as follows, except in the case that $(RA)_{32:63} = -2^{31}$ and $(RB)_{32:63} = -1$.

```
divw   RT,RA,RB      # RT = quotient
mullw  RT,RT,RB      # RT = quotient×divisor
subf   RT,RT,RA      # RT = remainder
```

Divide Word Unsigned *XO-form*

divwu	RT,RA,RB	(OE=0 Rc=0)
divwu.	RT,RA,RB	(OE=0 Rc=1)
divwuo	RT,RA,RB	(OE=1 Rc=0)
divwuo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	459	Rc
0	6	11	16	21	22	31

$dividend_{0:31} \leftarrow (RA)_{32:63}$
$divisor_{0:31} \leftarrow (RB)_{32:63}$
$RT_{32:63} \leftarrow dividend \div divisor$
$RT_{0:31} \leftarrow$ undefined

The 32 bit dividend is $(RA)_{32:63}$. The 32-bit divisor is $(RB)_{32:63}$. The 32-bit quotient is placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r <$ divisor.

If an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In this case, if OE=1 then OV and OV32 are set to 1.

Special Registers Altered:

CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
SO OV OV32 (if OE=1)

Programming Note

The 32-bit unsigned remainder of dividing $(RA)_{32:63}$ by $(RB)_{32:63}$ can be computed as follows.

```
divwu  RT,RA,RB      # RT = quotient
mullw  RT,RT,RB      # RT = quotient×divisor
subf   RT,RT,RA      # RT = remainder
```

Divide Word Extended *XO-form*

divwe	RT,RA,RB	(OE=0 Rc=0)
divwe.	RT,RA,RB	(OE=0 Rc=1)
divweo	RT,RA,RB	(OE=1 Rc=0)
divweo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	427	Rc
0	6	11	16	21	22	31

$dividend_{0:63} \leftarrow (RA)_{32:63} \,||\, {}^{32}0$
$divisor_{0:31} \leftarrow (RB)_{32:63}$
$RT_{32:63} \leftarrow dividend \div divisor$
$RT_{0:31} \leftarrow$ undefined

The 64-bit dividend is $(RA)_{32:63} \,||\, {}^{32}0$. The 32-bit divisor is $(RB)_{32:63}$. If the quotient can be represented in 32 bits, it is placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r < |divisor|$ if the dividend is nonnegative, and $-|divisor| < r \le 0$ if the dividend is negative.

If the quotient cannot be represented in 32 bits, or if an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV and OV32 are set to 1.

Special Registers Altered:

CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
SO OV OV32 (if OE=1)

Divide Word Extended Unsigned *XO-form*

divweu	RT,RA,RB	(OE=0 Rc=0)
divweu.	RT,RA,RB	(OE=0 Rc=1)
divweuo	RT,RA,RB	(OE=1 Rc=0)
divweuo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	395	Rc
0	6	11	16	21	22	31

$dividend_{0:63} \leftarrow (RA)_{32:63} \,||\, {}^{32}0$
$divisor_{0:31} \leftarrow (RB)_{32:63}$
$RT_{32:63} \leftarrow dividend \div divisor$
$RT_{0:31} \leftarrow$ undefined

The 64-bit dividend is $(RA)_{32:63} \,||\, {}^{32}0$. The 32-bit divisor is $(RB)_{32:63}$. If the quotient can be represented in 32 bits, it is placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r <$ divisor.

If $(RA) \ge (RB)$, or if an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV and OV32 are set to 1.

Special Registers Altered:

CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
SO OV OV32 (if OE=1)

Programming Note

Unsigned long division of a 64-bit dividend contained in two 32-bit registers by a 32-bit divisor can be computed as follows. The algorithm is shown first, followed by Assembler code that implements the algorithm. The dividend is Dh || Dl, the divisor is Dv, and the quotient and remainder are Q and R respectively, where these variables and all intermediate variables represent unsigned 32-bit integers. It is assumed that Dv > Dh, and that assigning a value to an intermediate variable assigns the low-order 32 bits of the value and ignores any higher-order bits of the value. (In both the algorithm and the Assembler code, “r1” and “r2” refer to “remainder 1” and “remainder 2”, rather than to GPRs 1 and 2.)

Algorithm:

3. q1 ← **divweu** Dh, Dv
4. r1 ← -(q1 × Dv) # remainder of step 1 divide operation (see Note 1)
5. q2 ← **divwu** Dl, Dv
6. r2 ← Dl - (q2 × Dv) # remainder of step 2 divide operation
7. Q ← q1 + q2
8. R ← r1 + r2
9. if (R < r2) | (R ≥ Dv) then # (see Note 2)
   Q ← Q + 1 # increment quotient
   R ← R - Dv # decrement rem’der

Assembler Code:

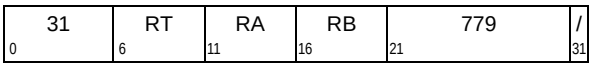
```
# Dh in r4, Dl in r5
# Dv in r6
divweu  r3,r4,r6     # q1
divwu   r7,r5,r6     # q2
mullw   r8,r3,r6     # -r1 = q1 * Dv
mullw   r0,r7,r6     # q2 * Dv
subf    r10,r0,r5    # r2 = Dl - (q2 * Dv)
add     r3,r3,r7     # Q = q1 + q2
subf    r4,r8,r10    # R = r1 + r2
cmplw   r4,r10       # R < r2 ?
blt     *+12         # must adjust Q and R if yes
cmplw   r4,r6        # R ≥ Dv ?
blt     *+12         # must adjust Q and R if yes
addi    r3,r3,1      # Q = Q + 1
subf    r4,r6,r4     # R = R - Dv
# Quotient in r3
# Remainder in r4
```

Notes:

1. The remainder is Dh || $^{32}0$ - (q1 × Dv). Because the remainder must be less than Dv and Dv < 2^{32}, the remainder is representable in 32 bits. Because the low-order 32 bits of Dh || $^{32}0$ are 0s, the remainder is therefore equal to the low-order 32 bits of -(q1 × Dv). Thus assigning -(q1 × Dv) to r1 yields the correct remainder.
2. R is less than r2 (and also less than r1) if and only if the addition at step 6 carried out of 32 bits — i.e., if and only if the correct sum could not be represented in 32 bits — in which case the correct sum is necessarily greater than Dv.
3. For additional information see the book *Hacker's Delight*, by Henry S. Warren, Jr., as potentially amended at the web site http://www.hackersdelight.org.

Modulo Signed Word X-form

modsw RT,RA,RB



$dividend_{0:31} \leftarrow (RA)_{32:63}$
$divisor_{0:31} \leftarrow (RB)_{32:63}$-
$RT_{32:63} \leftarrow$ dividend % divisor
$RT_{0:31} \leftarrow$ undefined

The 32-bit dividend is $(RA)_{32:63}$. The 32-bit divisor is $(RB)_{32:63}$. The 32-bit quotient is placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined. The quotient is not supplied as a result.

Both operands and the remainder are interpreted as signed integers. The remainder is the unique signed integer that satisfies

```
remainder = dividend - (quotient × divisor)
```

where $0 \leq remainder < |divisor|$ if the dividend is nonnegative, and $-|divisor| < remainder \leq 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

```
0x8000_0000 % -1
 <anything> % 0
```

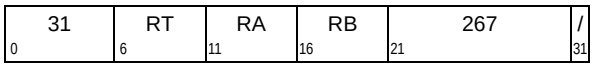
then the contents of register RT are undefined.

Special Registers Altered:

None

Modulo Unsigned Word X-form

moduw RT,RA,RB



$dividend_{0:31} \leftarrow (RA)_{32:63}$
$divisor_{0:31} \leftarrow (RB)_{32:63}$
$RT_{32:63} \leftarrow$ dividend % divisor
$RT_{0:31} \leftarrow$ undefined

The 32-bit dividend is $(RA)_{32:63}$. The 32-bit divisor is $(RB)_{32:63}$. The 32-bit quotient is placed into $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined. The quotient is not supplied as a result.

Both operands and the remainder are interpreted as unsigned integers. The remainder is the unique signed integer that satisfies

```
remainder = dividend - (quotient × divisor)
```

where $0 \leq remainder < divisor$.

If an attempt is made to perform any of the divisions

```
<anything> % 0
```

then the contents of register RT are undefined.

Special Registers Altered:

None

Deliver A Random Number *X-form*

darn RT,L

31	RT	///	L	///	755	/
0	6	11 13	14	16	21	31

RT ← random(L)

A random number is placed into register RT in a format selected by L as shown in the following table. The value 0xFFFFFFFF_FFFFFFFF indicates an error condition. For L=0, the random number range is 0:0xFFFFFFFF. For L=1 and L=2, the random number range is 0:0xFFFFFFFF_FFFFFFFE.

L	Format
0	$^{32}0 \,\|\|\, CRN_{0:31}$
1	$CRN_{0:63}$
2	$RRN_{0:63}$
3	reserved

Format above is for non-error conditions.
0xFFFFFFFF_FFFFFFFF for error conditions.
CRN = conditioned random number
RRN = raw random number

A raw random number is unconditioned noise source output. A conditioned random number has been processed by hardware to reduce bias.

Special Registers Altered:

none

Programming Note

32-bit software running in an environment that does not preserve the high-order 32 bits of GPRs across invocations of the system error handler, signal handlers, event-based branch handlers, etc. may use the L=0 variant of ***darn*** and interpret the value 0xFFFFFFFF to indicate an error condition. The fact that the error condition includes the valid value 0x00000000_FFFFFFFF together with the true error value 0xFFFFFFFF_FFFFFFFF is not a problem.

Programming Note

When the error value is obtained, software is expected to repeat the operation. If a non-error value has not been obtained after several attempts, a software random number generation method should be used. The recommended number of attempts may be implementation specific. In the absence of other guidance, ten attempts should be adequate.

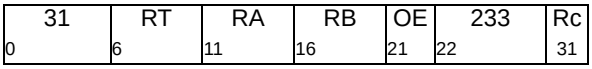
Programming Note

The random number generator provided by this instruction is NIST SP800-90B and SP800-90C compliant to the extent possible given the completeness of the standards at the time the hardware is designed. The random number generator provides a minimum of 0.5 bits of entropy per bit.

3.3.9.1 64-bit Fixed-Point Arithmetic Instructions

Multiply Low Doubleword *XO-form*

mulld	RT,RA,RB	(OE=0 Rc=0)
mulld.	RT,RA,RB	(OE=0 Rc=1)
mulldo	RT,RA,RB	(OE=1 Rc=0)
mulldo.	RT,RA,RB	(OE=1 Rc=1)



$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{64:127}$

The 64-bit operands are (RA) and (RB). The low-order 64 bits of the 128-bit product of the operands are placed into register RT.

If OE=1 then OV and OV32 are set to 1 if the product cannot be represented in 64 bits.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

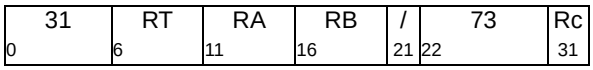
CR0 (if Rc=1)
SO OV OV32 (if OE=1)

Programming Note

The XO-form *Multiply* instructions may execute faster on some implementations if RB contains the operand having the smaller absolute value.

Multiply High Doubleword *XO-form*

mulhd	RT,RA,RB	(Rc=0)
mulhd.	RT,RA,RB	(Rc=1)



$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{0:63}$

The 64-bit operands are (RA) and (RB). The high-order 64 bits of the 128-bit product of the operands are placed into register RT.

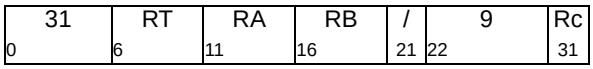
Both operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0 (if Rc=1)

Multiply High Doubleword Unsigned *XO-form*

mulhdu	RT,RA,RB	(Rc=0)
mulhdu.	RT,RA,RB	(Rc=1)



$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{0:63}$

The 64-bit operands are (RA) and (RB). The high-order 64 bits of the 128-bit product of the operands are placed into register RT.

Both operands and the product are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero.

Special Registers Altered:

CR0 (if Rc=1)

Multiply-Add High Doubleword VA-form

maddhd RT,RA.RB,RC

4	RT	RA	RB	RC	48
0	6	11	16	21	26 31

$prod_{0:127} \leftarrow (RA) \times (RB)$
$sum_{0:127} \leftarrow prod + EXTS(RC)$
$RT \leftarrow sum_{0:63}$

The 64-bit operands are (RA), (RB), and (RC). The 128-bit product of the operands (RA) and (RB) is added to (RC). The high-order 64 bits of the 128-bit sum are placed into register RT.

All three operands and the result are interpreted as signed integers.

Special Registers Altered:

None

Multiply-Add High Doubleword Unsigned VA-form

maddhdu RT,RA.RB,RC

4	RT	RA	RB	RC	49
0	6	11	16	21	26 31

$prod_{0:127} \leftarrow (RA) \times (RB)$
$sum_{0:127} \leftarrow prod + EXTZ(RC)$
$RT \leftarrow sum_{0:63}$

The 64-bit operands are (RA), (RB), and (RC). The 128-bit product of the operands (RA) and (RB) is added to (RC). The high-order 64 bits of the 128-bit sum are placed into register RT.

All three operands and the result are interpreted as unsigned integers.

Special Registers Altered:

None

Multiply-Add Low Doubleword VA-form

maddld RT,RA.RB,RC

4	RT	RA	RB	RC	51
0	6	11	16	21	26 31

$prod_{0:127} \leftarrow (RA) \times (RB)$
$sum_{0:127} \leftarrow prod + EXTS(RC)$
$RT \leftarrow sum_{64:127}$

The 64-bit operands are (RA), (RB), and (RC). The 128-bit product of the operands (RA) and (RB) is added to (RC). The low-order 64 bits of the 128-bit sum are placed into register RT.

All three operands and the result are interpreted as signed integers.

Special Registers Altered:

None

Divide Doubleword *XO-form*

divd	RT,RA,RB	(OE=0 Rc=0)
divd.	RT,RA,RB	(OE=0 Rc=1)
divdo	RT,RA,RB	(OE=1 Rc=0)
divdo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	489	Rc
0	6	11	16	21	22	31

```
dividend0:63 ← (RA)
divisor0:63 ← (RB)
RT ← dividend ÷ divisor
```

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit quotient is placed into register RT. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r < |divisor|$ if the dividend is nonnegative, and $-|divisor| < r \le 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

```
0x8000_0000_0000_0000 ÷ -1
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV and OV32 are set to 1.

Special Registers Altered:

CR0	(if Rc=1)
SO OV OV32	(if OE=1)

Programming Note

The 64-bit signed remainder of dividing (RA) by (RB) can be computed as follows, except in the case that (RA) = -2^{63} and (RB) = -1.

```
divd   RT,RA,RB      # RT = quotient
mulld  RT,RT,RB      # RT = quotient×divisor
subf   RT,RT,RA      # RT = remainder
```

Divide Doubleword Unsigned *XO-form*

divdu	RT,RA,RB	(OE=0 Rc=0)
divdu.	RT,RA,RB	(OE=0 Rc=1)
divduo	RT,RA,RB	(OE=1 Rc=0)
divduo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	457	Rc
0	6	11	16	21	22	31

```
dividend0:63 ← (RA)
divisor0:63 ← (RB)
RT ← dividend ÷ divisor
```

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit quotient is placed into register RT. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r < divisor$.

If an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In this case, if OE=1 then OV and OV32 are set to 1.

Special Registers Altered:

CR0	(if Rc=1)
SO OV OV32	(if OE=1)

Programming Note

The 64-bit unsigned remainder of dividing (RA) by (RB) can be computed as follows.

```
divdu  RT,RA,RB      # RT = quotient
mulld  RT,RT,RB      # RT = quotient×divisor
subf   RT,RT,RA      # RT = remainder
```

Divide Doubleword Extended *XO-form*

divde	RT,RA,RB	(OE=0 Rc=0)
divde.	RT,RA,RB	(OE=0 Rc=1)
divdeo	RT,RA,RB	(OE=1 Rc=0)
divdeo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	425	Rc
0	6	11	16	21	22	31

$dividend_{0:127} \leftarrow (RA) \mid\mid {}^{64}0$
$divisor_{0:63} \leftarrow (RB)$
$RT \leftarrow dividend \div divisor$

The 128-bit dividend is (RA) || $^{64}0$. The 64-bit divisor is (RB). If the quotient can be represented in 64 bits, it is placed into register RT. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r < |divisor|$ if the dividend is nonnegative, and $-|divisor| < r \le 0$ if the dividend is negative.

If the quotient cannot be represented in 64 bits, or if an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV and OV32 are set to 1.

Special Registers Altered:

CR0	(if Rc=1)
SO OV OV32	(if OE=1)

Divide Doubleword Extended Unsigned *XO-form*

divdeu	RT,RA,RB	(OE=0 Rc=0)
divdeu.	RT,RA,RB	(OE=0 Rc=1)
divdeuo	RT,RA,RB	(OE=1 Rc=0)
divdeuo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	393	Rc
0	6	11	16	21	22	31

$dividend_{0:127} \leftarrow (RA) \mid\mid {}^{64}0$
$divisor_{0:63} \leftarrow (RB)$
$RT \leftarrow dividend \div divisor$

The 128-bit dividend is (RA) || $^{64}0$. The 64-bit divisor is (RB). If the quotient can be represented in 64 bits, it is placed into register RT. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r <$ divisor.

If (RA) $\ge$ (RB), or if an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0. In these cases, if OE=1 then OV and OV32 are set to 1.

Special Registers Altered:

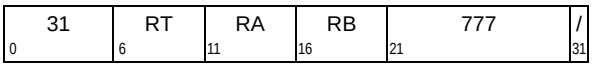
CR0	(if Rc=1)
SO OV OV32	(if OE=1)

Programming Note

Unsigned long division of a 128-bit dividend contained in two 64-bit registers by a 64-bit divisor can be accomplished using the technique described in the Programming Note with the ***divweu*** instruction description: ***divd[e]u*** would be used instead of ***divw[e]u*** (and ***cmpld*** instead of ***cmplw***, etc.).

Modulo Signed Doubleword X-form

modsd RT,RA,RB




```
dividend ← (RA)
divisor  ← (RB)
RT       ← dividend % divisor
```

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit remainder is placed into register RT. The quotient is not supplied as a result.

Both operands and the remainder are interpreted as signed integers. The remainder is the unique signed integer that satisfies

```
remainder = dividend - (quotient × divisor)
```

where $0 \leq$ `remainder` $<$ `|divisor|` if the dividend is nonnegative, and `-|divisor|` $<$ `remainder` ≤ 0 if the dividend is negative.

If an attempt is made to perform any of the divisions

```
<anything> % 0
0x8000_0000_0000_0000 % -1
```

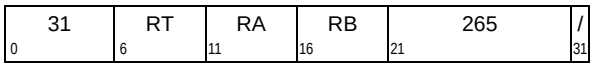
then the contents of register `RT` are undefined.

Special Registers Altered:

None

Modulo Unsigned Doubleword X-form

modud RT,RA,RB




```
dividend ← (RA)
divisor  ← (RB)
RT       ← dividend % divisor
```

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit remainder is placed into register RT. The quotient is not supplied as a result.

Both operands and the remainder are interpreted as unsigned integers. The remainder is the unique signed integer that satisfies

```
remainder = dividend - (quotient × divisor)
```

where $0 \leq$ `remainder` $<$ `divisor`.

If an attempt is made to perform any of the divisions

```
<anything> % 0
```

then the contents of register `RT` are undefined.

Special Registers Altered:

None

3.3.10 Fixed-Point Compare Instructions

The fixed-point *Compare* instructions compare the contents of register RA with (1) the sign-extended value of the SI field, (2) the zero-extended value of the UI field, or (3) the contents of register RB. The comparison is signed for ***cmpi*** and ***cmp***, and unsigned for ***cmpli*** and ***cmpl***.

The L field controls whether the operands are treated as 64-bit or 32-bit quantities, as follows:

L	Operand length
0	32-bit operands
1	64-bit operands

When the operands are treated as 32-bit signed quantities, bit 32 of the register (RA or RB) is the sign bit.

The *Compare* instructions set one bit in the leftmost three bits of the designated CR field to 1, and the other two to 0. XER_{SO} is copied to bit 3 of the designated CR field.

The CR field is set as follows

.

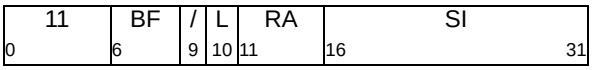
Bit	Name	Description
0	LT	(RA) < SI or (RB) (signed comparison) (RA) $<^{u}$ UI or (RB) (unsigned comparison)
1	GT	(RA) > SI or (RB) (signed comparison) (RA) $>^{u}$ UI or (RB) (unsigned comparison)
2	EQ	(RA) = SI, UI, or (RB)
3	SO	Summary Overflow from the XER

Extended mnemonics for compares

A set of extended mnemonics is provided so that compares can be coded with the operand length as part of the mnemonic rather than as a numeric operand. Some of these are shown as examples with the *Compare* instructions. See Appendix C for additional extended mnemonics.

Compare Immediate *D-form*

cmpi BF,L,RA,SI




```
if L = 0 then a ← EXTS((RA)32:63)
         else a ← (RA)
if      a < EXTS(SI) then c ← 0b100
else if a > EXTS(SI) then c ← 0b010
else                      c ← 0b001
CR4×BF+32:4×BF+35 ← c || XERSO
```

The contents of register RA ($(RA)_{32:63}$ sign-extended to 64 bits if L=0) are compared with the sign-extended value of the SI field, treating the operands as signed integers. The result of the comparison is placed into CR field BF.

Special Registers Altered:

CR field BF

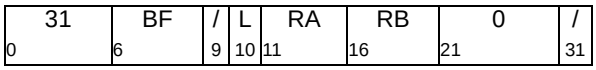
Extended Mnemonics:

Examples of extended mnemonics for Compare Immediate:

Extended:		Equivalent to:	
cmpdi	Rx,value	cmpi	0,1,Rx,value
cmpwi	cr3,Rx,value	cmpi	3,0,Rx,value

Compare *X-form*

cmp BF,L,RA,RB




```
if L = 0 then a ← EXTS((RA)32:63)
                b ← EXTS((RB)32:63)
           else a ← (RA)
                b ← (RB)
if      a < b then c ← 0b100
else if a > b then c ← 0b010
else               c ← 0b001
CR4×BF+32:4×BF+35 ← c || XERSO
```

The contents of register RA ($(RA)_{32:63}$ if L=0) are compared with the contents of register RB ($(RB)_{32:63}$ if L=0), treating the operands as signed integers. The result of the comparison is placed into CR field BF.

Special Registers Altered:

CR field BF

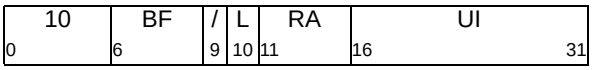
Extended Mnemonics:

Examples of extended mnemonics for Compare:

Extended:		Equivalent to:	
cmpd	Rx,Ry	cmp	0,1,Rx,Ry
cmpw	cr3,Rx,Ry	cmp	3,0,Rx,Ry

Compare Logical Immediate *D-form*

cmpli BF,L,RA,UI




```
if L = 0 then a ← 32 0 || (RA)32:63
         else a ← (RA)
if      a <u (48 0 || UI) then c ← 0b100
else if a >u (48 0 || UI) then c ← 0b010
else                           c ← 0b001
CR4×BF+32:4×BF+35 ← c || XERSO
```

The contents of register RA ($(RA)_{32:63}$ zero-extended to 64 bits if L=0) are compared with $^{48}0$ || UI, treating the operands as unsigned integers. The result of the comparison is placed into CR field BF.

Special Registers Altered:

CR field BF

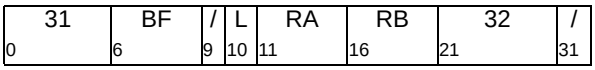
Extended Mnemonics:

Examples of extended mnemonics for *Compare Logical Immediate*:

Extended:		Equivalent to:	
cmpldi	Rx,value	cmpli	0,1,Rx,value
cmplwi	cr3,Rx,value	cmpli	3,0,Rx,value

Compare Logical *X-form*

cmpl BF,L,RA,RB




```
if L = 0 then a ← 32 0 || (RA)32:63
              b ← 32 0 || (RB)32:63
         else a ← (RA)
              b ← (RB)
if      a <u b then c ← 0b100
else if a >u b then c ← 0b010
else               c ← 0b001
CR4×BF+32:4×BF+35 ← c || XERSO
```

The contents of register RA ($(RA)_{32:63}$ if L=0) are compared with the contents of register RB ($(RB)_{32:63}$ if L=0), treating the operands as unsigned integers. The result of the comparison is placed into CR field BF.

Special Registers Altered:

CR field BF

Extended Mnemonics:

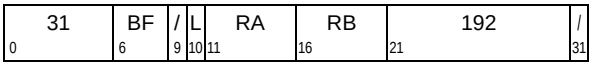
Examples of extended mnemonics for *Compare Logical*:

Extended:		Equivalent to:	
cmpld	Rx,Ry	cmpl	0,1,Rx,Ry
cmplw	cr3,Rx,Ry	cmpl	3,0,Rx,Ry

3.3.10.1 Character-Type Compare Instructions

Compare Ranged Byte X-form

cmprb BF,L,RA,RB




```
src1    ← EXTZ((RA)56:63)

src21hi ← EXTZ((RB)32:39)
src21lo ← EXTZ((RB)40:47)
src22hi ← EXTZ((RB)48:55)
src22lo ← EXTZ((RB)56:63)

if L=0 then
   in_range ←  (src22lo ≤ src1) & (src1 ≤ src22hi)
else
   in_range ← ((src21lo ≤ src1) & (src1 ≤ src21hi)) |
              ((src22lo ≤ src1) & (src1 ≤ src22hi))

CR4×BF+32 ← 0b0
CR4×BF+33 ← in_range
CR4×BF+34 ← 0b0
CR4×BF+35 ← 0b0
```

Let src1 be the unsigned integer value in bits 56:63 of register RA.

Let src21hi be the unsigned integer value in bits 32:39 of register RB.

Let src21lo be the unsigned integer value in bits 40:47 of register RB.

Let src22hi be the unsigned integer value in bits 48:55 of register RB.

Let src22lo be the unsigned integer value in bits 56:63 of register RB.

Let x be considered “in range” of y:z if the value x is greater than or equal to the value y and the value x is less than or equal to the value z.

When L=0, the value in_range is set to 1 if src1 is in range of src22lo:src22hi. Otherwise, the value in_range is set to 0.

When L=1, the value in_range is set to 1 if either src1 is in range of src21lo:src21hi, or src1 is in range of src22lo:src22hi. Otherwise, the value in_range is set to 0.

CR field BF is set to the value 0b0 concatenated with in_range concatenated with 0b00.

Special Registers Altered:

CR field BF

Programming Note

cmprb is useful for implementing character typing functions such as isalpha(), isdigit(), isupper(), and islower() that are implemented using one or two range compares of the character.

A single-range compare can be implemented with an ***addi*** to load the upper and lower bounds in the range, such as isdigit().

```
addi    rRNG,0,0x3930       ; loads ASCII values for ‘9’
                            ;  and ‘0’ into rRNG
cmprb   crTGT,0,rCHAR,rRNG  ; perform range compare
                            ;   sets CR field TGT to
                            ;   indicate in range
```

A combination of ***addi***-***addis*** can be used to set up 2 ranges, such as for isalpha().

```
addi    rRNG,0,0x7A61       ; loads ASCII values for ‘z’
                            ;   and ‘a’ into rRNG
addis   rRNG,rRNG,0x5A41    ; appends ASCII values for ‘Z’
                            ;   and ‘A’ into rRNG
cmprb   crTGT,1,rCHAR,rRNG  ; perform range compare on
                            ;   character in rCHAR,
                            :   setting CR field TGT to
                            ;   indicate in range
```

Compare Equal Byte X-form

cmpeqb BF,RA,RB

31	BF	//	RA	RB	224	/
0	6	9	11	16	21	31

```
src1 ← GPR[RA].bit[56:63]

match ← (src1 = (RB)00:07) |
        (src1 = (RB)08:15) |
        (src1 = (RB)16:23) |
        (src1 = (RB)24:31) |
        (src1 = (RB)32:39) |
        (src1 = (RB)40:47) |
        (src1 = (RB)48:55) |
        (src1 = (RB)56:63)

CR4×BF+32 ← 0b0
CR4×BF+33 ← match
CR4×BF+34 ← 0b0
CR4×BF+35 ← 0b0
```

CR field BF is set to indicate if the contents of bits 56:63 of register RA are equal to the contents of any of the 8 bytes in register RB.

Results are undefined in 32-bit mode.

Special Registers Altered:

CR field BF

Programming Note

cmpeqb is useful for implementing character typing functions such as isspace() that are implemented by comparing the character to 1 or more values.

A function such as isspace() can be implemented by loading the 6 byte codes corresponding to characters considered as whitespace (HT, LF, VT, FF, CR, and SP) and using the ***cmpeb*** to compare the subject character to those 6 values to determine if any match occurs.

```
  ldx     rSPC,WS_CHARS      ; rSPC = 0x0909_090A_0B0C_0D20
                             ;   load rSPC with all 6 ASCII
                             ;   values corresponding to
                             ;   white spaces
  cmpeqb  2,cr1,rCHAR,rSPC   ; perform match compare on
                             ;   character in rCHAR with
                             :   byte values in rSPC
```

In this case, the byte code for HT (0x09) was replicated to fill the all 8 bytes to avoid a potential miscompare.

3.3.11 Fixed-Point Trap Instructions

The *Trap* instructions are provided to test for a specified set of conditions. If any of the conditions tested by a *Trap* instruction are met, the system trap handler is invoked. If none of the tested conditions are met, instruction execution continues normally.

The contents of register RA are compared with either the sign-extended value of the SI field or the contents of register RB, depending on the *Trap* instruction. For ***tdi*** and ***td***, the entire contents of RA (and RB) participate in the comparison; for ***twi*** and ***tw***, only the contents of the low-order 32 bits of RA (and RB) participate in the comparison.

This comparison results in five conditions which are ANDed with TO. If the result is not 0 the system trap handler is invoked. These conditions are as follows.

TO Bit	ANDed with Condition
0	Less Than, using signed comparison
1	Greater Than, using signed comparison
2	Equal
3	Less Than, using unsigned comparison
4	Greater Than, using unsigned comparison

Extended mnemonics for traps

A set of extended mnemonics is provided so that traps can be coded with the condition as part of the mnemonic rather than as a numeric operand. Some of these are shown as examples with the Trap instructions. See Appendix C for additional extended mnemonics.

Trap Word Immediate *D-form*

twi TO,RA,SI

3	TO	RA	SI
0	6	11	16 31

```
a ← EXTS((RA)32:63)
if (a < EXTS(SI)) & TO0  then TRAP
if (a > EXTS(SI)) & TO1  then TRAP
if (a = EXTS(SI)) & TO2  then TRAP
if (a <u EXTS(SI)) & TO3 then TRAP
if (a >u EXTS(SI)) & TO4 then TRAP
```

The contents of $RA_{32:63}$ are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

If the trap conditions are met, this instruction is context synchronizing (see Book III).

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Trap Word Immediate*:

Extended:		**Equivalent to:**	
twgti	Rx,value	twi	8,Rx,value
twllei	Rx,value	twi	6,Rx,value

Trap Word *X-form*

tw TO,RA,RB

31	TO	RA	RB	4	/
0	6	11	16	21	31

```
a ← EXTS((RA)32:63)
b ← EXTS((RB)32:63)
if (a < b) & TO0 then TRAP
if (a > b) & TO1 then TRAP
if (a = b) & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4 then TRAP
```

The contents of $RA_{32:63}$ are compared with the contents of $RB_{32:63}$. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

If the trap conditions are met, this instruction is context synchronizing (see Book III).

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Trap Word*:

Extended:		**Equivalent to:**	
tweq	Rx,Ry	tw	4,Rx,Ry
twlge	Rx,Ry	tw	5,Rx,Ry
trap		tw	31,0,0

3.3.11.1 64-bit Fixed-Point Trap Instructions

Trap Doubleword Immediate *D-form*

tdi TO,RA,SI

2	TO	RA	SI
0	6	11	16 31

```
a ← (RA)
b ← EXTS(SI)
if (a < b) & TO0 then TRAP
if (a > b) & TO1 then TRAP
if (a = b) & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4  then TRAP
```

The contents of register RA are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

If the trap conditions are met, this instruction is context synchronizing (see Book III).

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Trap Doubleword Immediate*:

Extended:		Equivalent to:	
tdlti	Rx,value	tdi	16,Rx,value
tdnei	Rx,value	tdi	24,Rx,value

Trap Doubleword *X-form*

td TO,RA,RB

31	TO	RA	RB	68	/
0	6	11	16	21	31

```
a ← (RA)
b ← (RB)
if (a < b) & TO0 then TRAP
if (a > b) & TO1 then TRAP
if (a = b) & TO2 then TRAP
if (a <u b) & TO3 then TRAP
if (a >u b) & TO4  then TRAP
```

The contents of register RA are compared with the contents of register RB. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

If the trap conditions are met, this instruction is context synchronizing (see Book III).

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Trap Doubleword*:

Extended:		Equivalent to:	
tdge	Rx,Ry	td	12,Rx,Ry

3.3.12 Fixed-Point Select

Integer Select *A-form*

isel RT,RA,RB,BC

31	RT	RA	RB	BC	15	/
0	6	11	16	21	26	31

```
if RA=0 then a ←0 else a ← (RA)
if CRBC+32=1 then RT ← a
else              RT ← (RB)
```

If the contents of bit BC+32 of the Condition Register are equal to 1, then the contents of register RA (or 0) are placed into register RT. Otherwise, the contents of register RB are placed into register RT.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Integer Select*:

Extended:		Equivalent to:	
isellt	Rx,Ry,Rz	isel	Rx,Ry,Rz,0
iselgt	Rx,Ry,Rz	isel	Rx,Ry,Rz,1
iseleq	Rx,Ry,Rz	isel	Rx,Ry,Rz,2

3.3.13 Fixed-Point Logical Instructions

The *Logical* instructions perform bit-parallel operations on 64-bit operands.

The X-form Logical instructions with Rc=1, and the D-form *Logical* instructions ***andi.*** and ***andis.***, set the first three bits of CR Field 0 as described in Section 3.3.8, “Other Fixed-Point Instructions” on page 66. The Logical instructions do not change the SO, OV, OV32, CA, and CA32 bits in the XER.

Extended mnemonics for logical operations

Extended mnemonics are provided that generate two different types of “no-ops” (instructions that do nothing). The first type is the preferred form, which is optimized to minimize its use of the processor's execution resources. This form is based on the *OR Immediate* instruction. The second type is the executed form, which is intended to consume the same amount of the processor's execution resources as if it were not a no-op. This form is based on the *XOR Immediate* instruction. (There are also no-ops that have other uses, such as affecting program priority, for which extended mnemonics have not been defined.)

Extended mnemonics are provided that use the *OR* and *NOR* instructions to copy the contents of one register to another, with and without complementing. These are shown as examples with the two instructions.

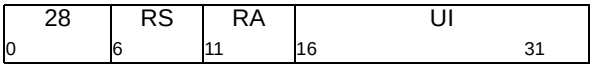
See Appendix C, “Assembler Extended Mnemonics” on page 791 for additional extended mnemonics.

Programming Note

Warning: Some forms of no-op may have side effects such as affecting program priority. Programmers should use the preferred no-op unless the side effects of some other form of no-op are intended.

AND Immediate *D-form*

andi. RA,RS,UI



RA ← (RS) & ($^{48}0$ || UI)

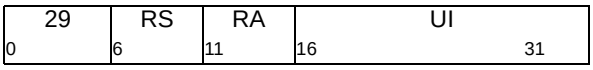
The contents of register RS are ANDed with $^{48}0$ || UI and the result is placed into register RA.

Special Registers Altered:

CR0

AND Immediate Shifted *D-form*

andis. RA,RS,UI



RA ← (RS) & ($^{32}0$ || UI || $^{16}0$)

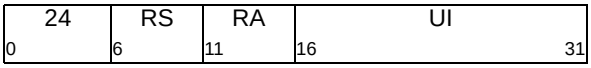
The contents of register RS are ANDed with $^{32}0$ || UI || $^{16}0$ and the result is placed into register RA.

Special Registers Altered:

CR0

OR Immediate *D-form*

ori RA,RS,UI



RA ← (RS) | ($^{48}0$ || UI)

The contents of register RS are ORed with $^{48}0$ || UI and the result is placed into register RA.

The preferred “no-op” (an instruction that does nothing) is:

```
ori    0,0,0
```

Special Registers Altered:

None

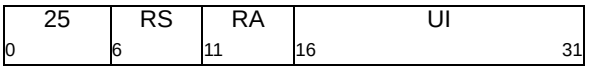
Extended Mnemonics:

Example of extended mnemonics for *OR Immediate*:

Extended:	Equivalent to:
no-op	ori 0,0,0

OR Immediate Shifted *D-form*

oris RA,RS,UI



RA ← (RS) | (^{32}0 || UI || ^{16}0)

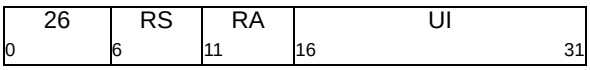
The contents of register RS are ORed with ^{32}0 || UI || ^{16}0 and the result is placed into register RA.

Special Registers Altered:

None

XOR Immediate *D-form*

xori RA,RS,UI



RA ← (RS) XOR (^{48}0 || UI)

The contents of register RS are XORed with ^{48}0 || UI and the result is placed into register RA.

The executed form of a “no-op” (an instruction that does nothing, but consumes execution resources nevertheless) is:

```
xori   0,0,0
```

Special Registers Altered:

None

Extended Mnemonics:

Example of extended mnemonics for *XOR Immediate*:

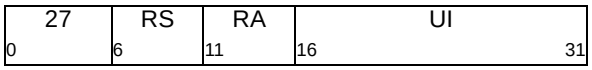
Extended:	**Equivalent to:**
xnop	xori 0,0,0

Programming Note

The executed form of no-op should be used only when the intent is to alter the timing of a program.

XOR Immediate Shifted *D-form*

xoris RA,RS,UI



RA ← (RS) XOR (^{32}0 || UI || ^{16}0)

The contents of register RS are XORed with ^{32}0 || UI || ^{16}0 and the result is placed into register RA.

Special Registers Altered:

None

AND *X-form*

and RA,RS,RB (Rc=0)
and. RA,RS,RB (Rc=1)

31	RS	RA	RB	28	Rc
0	6	11	16	21	31

RA ← (RS) & (RB)

The contents of register RS are ANDed with the contents of register RB and the result is placed into register RA.

Some forms of ***and*** Rx, Rx, Rx provide special functions; see Section 9.3 of Book III.

Special Registers Altered:
CR0 (if Rc=1)

XOR *X-form*

xor RA,RS,RB (Rc=0)
xor. RA,RS,RB (Rc=1)

31	RS	RA	RB	316	Rc
0	6	11	16	21	31

RA ← (RS) ⊕ (RB)

The contents of register RS are XORed with the contents of register RB and the result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

NAND *X-form*

nand RA,RS,RB (Rc=0)
nand. RA,RS,RB (Rc=1)

31	RS	RA	RB	476	Rc
0	6	11	16	21	31

RA ← ¬((RS) & (RB))

The contents of register RS are ANDed with the contents of register RB and the complemented result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

Programming Note

nand or ***nor*** with RS=RB can be used to obtain the one's complement.

OR *X-form*

or RA,RS,RB (Rc=0)
or. RA,RS,RB (Rc=1)

31	RS	RA	RB	444	Rc
0	6	11	16	21	31

RA ← (RS) | (RB)

The contents of register RS are ORed with the contents of register RB and the result is placed into register RA.

Some forms of *or Rx,Rx,Rx* provide special functions; see Section 3.2 and Section 4.3.3, both in Book II.

Special Registers Altered:
CR0 (if Rc=1)

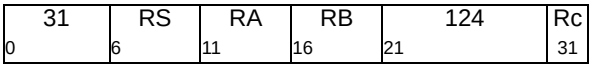
Extended Mnemonics:

Example of extended mnemonics for *OR*:

Extended:		**Equivalent to:**	
mr	Rx,Ry	or	Rx,Ry,Ry

NOR *X-form*

nor RA,RS,RB (Rc=0)
nor. RA,RS,RB (Rc=1)




```
RA ← ¬((RS) | (RB))
```

The contents of register RS are ORed with the contents of register RB and the complemented result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

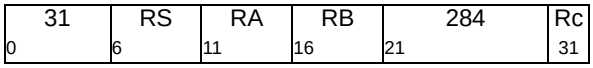
Extended Mnemonics:

Example of extended mnemonics for *NOR*:

Extended:		**Equivalent to:**	
not	Rx,Ry	nor	Rx,Ry,Ry

Equivalent *X-form*

eqv RA,RS,RB (Rc=0)
eqv. RA,RS,RB (Rc=1)



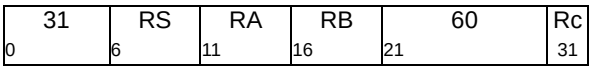

```
RA ← (RS) ≡ (RB)
```

The contents of register RS are XORed with the contents of register RB and the complemented result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

AND with Complement *X-form*

andc RA,RS,RB (Rc=0)
andc. RA,RS,RB (Rc=1)



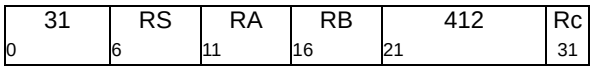

```
RA ← (RS) & ¬(RB)
```

The contents of register RS are ANDed with the complement of the contents of register RB and the result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

OR with Complement *X-form*

orc RA,RS,RB (Rc=0)
orc. RA,RS,RB (Rc=1)



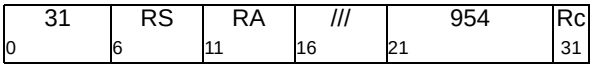

```
RA ← (RS) | ¬(RB)
```

The contents of register RS are ORed with the complement of the contents of register RB and the result is placed into register RA.

Special Registers Altered:
CR0 (if Rc=1)

Extend Sign Byte *X-form*

extsb RA,RS (Rc=0)
extsb. RA,RS (Rc=1)



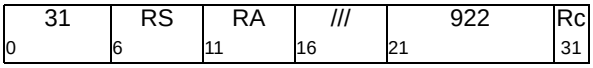
$s \leftarrow (RS)_{56}$
$RA_{56:63} \leftarrow (RS)_{56:63}$
$RA_{0:55} \leftarrow {}^{56}s$

$(RS)_{56:63}$ are placed into $RA_{56:63}$. $RA_{0:55}$ are filled with a copy of $(RS)_{56}$.

Special Registers Altered:
CR0 (if Rc=1)

Extend Sign Halfword *X-form*

extsh RA,RS (Rc=0)
extsh. RA,RS (Rc=1)



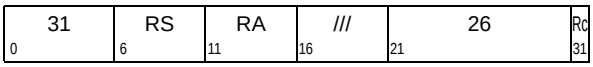
$s \leftarrow (RS)_{48}$
$RA_{48:63} \leftarrow (RS)_{48:63}$
$RA_{0:47} \leftarrow {}^{48}s$

$(RS)_{48:63}$ are placed into $RA_{48:63}$. $RA_{0:47}$ are filled with a copy of $(RS)_{48}$.

Special Registers Altered:
CR0 (if Rc=1)

Count Leading Zeros Word *X-form*

cntlzw RA,RS (Rc=0)
cntlzw. RA,RS (Rc=1)




```
n ← 32

do while n < 64
   if (RS)n = 1 then leave
   n ← n + 1

RA ← n - 32
```

A count of the number of consecutive zero bits starting at bit 32 of register RS is placed into register RA. This number ranges from 0 to 32, inclusive.

If Rc is equal to 1, CR field 0 is set to reflect the result.

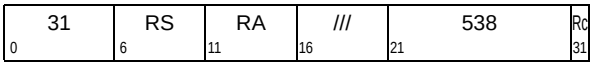
Special Registers Altered:
CR0 (if Rc=1)

Programming Note

For both *Count Leading Zeros* instructions, if Rc=1 then LT is set to 0 in CR Field 0.

Count Trailing Zeros Word *X-form*

cnttzw RA,RS (Rc=0)
cnttzw. RA,RS (Rc=1)




```
n ← 0

do while n < 32
   if (RS)63-n = 0b1 then leave
   n  ← n + 1

RA ← EXTZ64(n)
```

A count of the number of consecutive zero bits starting at bit 63 of the rightmost word of register RS is placed into register RA. This number ranges from 0 to 32, inclusive.

If Rc is equal to 1, CR field 0 is set to reflect the result.

Special Registers Altered:
CR0 (if Rc=1)

Compare Bytes *X-form*

cmpb RA,RS,RB

31	RS	RA	RB	508	/
0	6	11	16	21	31

```
do n = 0 to 7
   if RS8×n:8×n+7 = (RB)8×n:8×n+7 then
      RA8×n:8×n+7 ← 81
   else
      RA8×n:8×n+7 ← 80
```

Each byte of the contents of register RS is compared to each corresponding byte of the contents in register RB. If they are equal, the corresponding byte in RA is set to 0xFF. Otherwise the corresponding byte in RA is set to 0x00.

Special Registers Altered:

None

Population Count Bytes *X-form*

popcntb RA, RS

31	RS	RA	///	122	/
0	6	11	16	21	31

```
do i = 0 to 7
   n ← 0
   do j = 0 to 7
      if (RS)(i×8)+j = 1 then
         n ← n+1
   RA(i×8):(i×8)+7 ← n
```

A count of the number of one bits in each byte of register RS is placed into the corresponding byte of register RA. This number ranges from 0 to 8, inclusive.

Special Registers Altered:

None

Population Count Words *X-form*

popcntw RA, RS

31	RS	RA	///	378	/
0	6	11	16	21	31

```
do i = 0 to 1
   n ← 0
   do j = 0 to 31
      if (RS)(i×32)+j = 1 then
         n ← n+1
   RA(i×32):(i×32)+31 ← n
```

A count of the number of one bits in each word of register RS is placed into the corresponding word of register RA. This number ranges from 0 to 32, inclusive.

Special Registers Altered:

None

Parity Doubleword *X-form*

prtyd RA,RS

31	RS	RA	///	186	/
0	6	11	16	21	31

```
s ← 0
do i = 0 to 7
   s ← s / (RS)i%8+7
RA ← 63 0 || s
```

The least significant bit in each byte of the contents of register RS is examined. If there is an odd number of one bits the value 1 is placed into register RA; otherwise the value 0 is placed into register RA.

Special Registers Altered:

None

Parity Word *X-form*

prtyw RA,RS

31	RS	RA	///	154	/
0	6	11	16	21	31

```
s ← 0
t ← 0
do i = 0 to 3
   s ← s / (RS)i%8+7
do i = 4 to 7
   t ← t / (RS)i%8+7
RA0:31 ← 31 0 || s
RA32:63 ← 31 0 || t
```

The least significant bit in each byte of $(RS)_{0:31}$ is examined. If there is an odd number of one bits the value 1 is placed into $RA_{0:31}$; otherwise the value 0 is placed into $RA_{0:31}$. The least significant bit in each byte of $(RS)_{32:63}$ is examined. If there is an odd number of one bits the value 1 is placed into $RA_{32:63}$; otherwise the value 0 is placed into $RA_{32:63}$.

Special Registers Altered:

None

Programming Note

The *Parity* instructions are designed to be used in conjunction with the *Population Count* instruction to compute the parity of words or a doubleword. The parity of the upper and lower words in (RS) can be computed as follows.

```
popcntb  RA, RS
prtyw    RA, RA
```

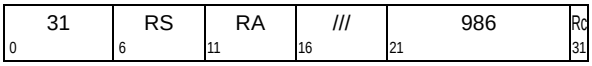
The parity of (RS) can be computed as follows.

```
popcntb  RA, RS
prtyd    RA, RA
```

3.3.13.1 64-bit Fixed-Point Logical Instructions

Extend Sign Word *X-form*

extsw RA,RS (Rc=0)
extsw. RA,RS (Rc=1)




```
s ← (RS)32
RA32:63 ← (RS)32:63
RA0:31 ← 32s
```

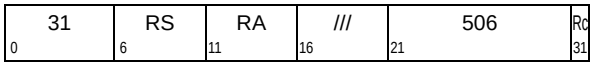
$(RS)_{32:63}$ are placed into $RA_{32:63}$. $RA_{0:31}$ are filled with a copy of $(RS)_{32}$.

Special Registers Altered:

CR0 (if Rc=1)

Population Count Doubleword *X-form*

popcntd RA, RS




```
n ← 0
do i = 0 to 63
   if (RS)i = 1 then
   n ← n+1
RA ← n
```

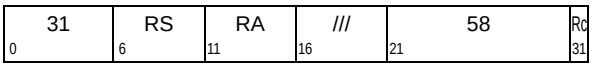
A count of the number of one bits in register RS is placed into register RA. This number ranges from 0 to 64, inclusive.

Special Registers Altered:

None

Count Leading Zeros Doubleword *X-form*

cntlzd RA,RS (Rc=0)
cntlzd. RA,RS (Rc=1)




```
n ← 0
do while n < 64
  if (RS)n = 1 then leave
  n ← n + 1
RA ← n
```

A count of the number of consecutive zero bits starting at bit 0 of register RS is placed into register RA. This number ranges from 0 to 64, inclusive.

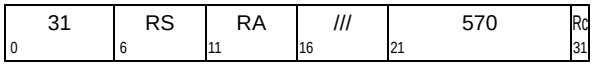
If Rc=1, CR Field 0 is set to reflect the result.

Special Registers Altered:

CR0 (if Rc=1)

Count Trailing Zeros Doubleword *X-form*

cnttzd RA,RS (Rc=0)
cnttzd. RA,RS (Rc=1)




```
n  ← 0
do while n < 64
   if (RS)63-n = 0b1 then leave
   n  ← n + 1
RA ← EXTZ64(n)
```

A count of the number of consecutive zero bits starting at bit 63 of register RS is placed into register RA. This number ranges from 0 to 64, inclusive.

If Rc is equal to 1, CR field 0 is set to reflect the result.

Special Registers Altered:

CR0 (if Rc=1)

Bit Permute Doubleword *X-form*

bpermd RA,RS,RB]

31	RS	RA	RB	252	/
0	6	11	16	21	31

```
For i = 0 to 7
   index ← (RS)8*i:8*i+7
   If index < 64
      then permi ← (RB)index
      else permi ← 0
RA ← 560 || perm0:7
```

Eight permuted bits are produced. For each permuted bit i where i ranges from 0 to 7 and for each byte i of RS, do the following.

> If byte i of RS is less than 64, permuted bit i is set to the bit of RB specified by byte i of RS; otherwise permuted bit i is set to 0.

The permuted bits are placed in the least-significant byte of RA, and the remaining bits are filled with 0s.

Special Registers Altered:

None

Programming Note

The fact that the permuted bit is 0 if the corresponding index value exceeds 63 permits the permuted bits to be selected from a 128-bit quantity, using a single index register. For example, assume that the 128-bit quantity Q, from which the permuted bits are to be selected, is in registers r2 (high-order 64 bits of Q) and r3 (low-order 64 bits of Q), that the index values are in register r1, with each byte of r1 containing a value in the range 0:127, and that each byte of register r4 contains the value 64. The following code sequence selects eight permuted bits from Q and places them into the low-order byte of r6.

```
  bpermd    r6,r1,r2     # select from high-
                           order half of Q
  xor       r0,r1,r4     # adjust index values
  bpermd    r5,r0,r3     # select from low-
                           order half of Q
  or        r6,r6,r5     # merge the two
                           selections
```

3.3.14 Fixed-Point Rotate and Shift Instructions

The Fixed-Point Facility performs rotation operations on data from a GPR and returns the result, or a portion of the result, to a GPR.

The rotation operations rotate a 64-bit quantity left by a specified number of bit positions. Bits that exit from position 0 enter at position 63.

Two types of rotation operation are supported.

For the first type, denoted $rotate_{64}$ or $ROTL_{64}$, the value rotated is the given 64-bit value. The $rotate_{64}$ operation is used to rotate a given 64-bit quantity.

For the second type, denoted $rotate_{32}$ or $ROTL_{32}$, the value rotated consists of two copies of bits 32:63 of the given 64-bit value, one copy in bits 0:31 and the other in bits 32:63. The $rotate_{32}$ operation is used to rotate a given 32-bit quantity.

The *Rotate* and *Shift* instructions employ a mask generator. The mask is 64 bits long, and consists of 1-bits from a start bit, *mstart*, through and including a stop bit, *mstop*, and 0-bits elsewhere. The values of *mstart* and *mstop* range from 0 to 63. If *mstart* > *mstop*, the 1-bits wrap around from position 63 to position 0. Thus the mask is formed as follows:

```
if mstart ≤ mstop then
   mask[mstart:mstop] = ones
   mask[all other bits] = zeros
else
   mask[mstart:63] = ones
   mask[0:mstop] = ones
   mask[all other bits] = zeros
```

There is no way to specify an all-zero mask.

For instructions that use the $rotate_{32}$ operation, the mask start and stop positions are always in the low-order 32 bits of the mask.

The use of the mask is described in following sections.

The *Rotate* and *Shift* instructions with Rc=1 set the first three bits of CR field 0 as described in Section 3.3.8, “Other Fixed-Point Instructions” on page 66. *Rotate* and *Shift* instructions do not change the OV, OV32, and SO bits. *Rotate* and *Shift* instructions, except algebraic right shifts, do not change the CA and CA32 bits.

Extended mnemonics for rotates and shifts

The *Rotate* and *Shift* instructions, while powerful, can be complicated to code (they have up to five operands). A set of extended mnemonics is provided that allow simpler coding of often-used functions such as clearing the leftmost or rightmost bits of a register, left justifying or right justifying an arbitrary field, and performing simple rotates and shifts. Some of these are shown as examples with the *Rotate* instructions. See Appendix C, “Assembler Extended Mnemonics” on page 791 for additional extended mnemonics.

3.3.14.1 Fixed-Point Rotate Instructions

These instructions rotate the contents of a register. The result of the rotation is

- inserted into the target register under control of a mask (if a mask bit is 1 the associated bit of the rotated data is placed into the target register, and if the mask bit is 0 the associated bit in the target register remains unchanged); or
- ANDed with a mask before being placed into the target register.

The *Rotate Left* instructions allow right-rotation of the contents of a register to be performed (in concept) by a left-rotation of 64-n, where n is the number of bits by which to rotate right. They allow right-rotation of the contents of the low-order 32 bits of a register to be performed (in concept) by a left-rotation of 32-n, where n is the number of bits by which to rotate right.

Rotate Left Word Immediate then AND with Mask M-form

rlwinm RA,RS,SH,MB,ME (Rc=0)
rlwinm. RA,RS,SH,MB,ME (Rc=1)

21	RS	RA	SH	MB	ME	Rc
0	6	11	16	21	26	31

```
n ← SH
r ← ROTL32((RS)32:63, n)
m ← MASK(MB+32, ME+32)
RA ← r & m
```

The contents of register RS are rotated$_{32}$ left SH bits. A mask is generated having 1-bits from bit MB+32 through bit ME+32 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Examples of extended mnemonics for *Rotate Left Word Immediate then AND with Mask*:

Extended:		Equivalent to:	
extlwi	Rx,Ry,n,b	rlwinm	Rx,Ry,b,0,n-1
srwi	Rx,Ry,n	rlwinm	Rx,Ry,32-n,n,31
clrrwi	Rx,Ry,n	rlwinm	Rx,Ry,0,0,31-n

Programming Note

Let RSL represent the low-order 32 bits of register RS, with the bits numbered from 0 through 31.

rlwinm can be used to extract an n-bit field that starts at bit position b in RSL, right-justified into the low-order 32 bits of register RA (clearing the remaining 32-n bits of the low-order 32 bits of RA), by setting SH=b+n, MB=32-n, and ME=31. It can be used to extract an n-bit field that starts at bit position b in RSL, left-justified into the low-order 32 bits of register RA (clearing the remaining 32-n bits of the low-order 32 bits of RA), by setting SH=b, MB = 0, and ME=n-1. It can be used to rotate the contents of the low-order 32 bits of a register left (right) by n bits, by setting SH=n (32-n), MB=0, and ME=31. It can be used to shift the contents of the low-order 32 bits of a register right by n bits, by setting SH=32-n, MB=n, and ME=31. It can be used to clear the high-order b bits of the low-order 32 bits of the contents of a register and then shift the result left by n bits, by setting SH=n, MB=b-n, and ME=31-n. It can be used to clear the low-order n bits of the low-order 32 bits of a register, by setting SH=0, MB=0, and ME=31-n.

For all the uses given above, the high-order 32 bits of register RA are cleared.

Extended mnemonics are provided for all of these uses; see Appendix C, “Assembler Extended Mnemonics” on page 791.

Rotate Left Word then AND with Mask *M-form*

rlwnm RA,RS,RB,MB,ME (Rc=0)
rlwnm. RA,RS,RB,MB,ME (Rc=1)

23	RS	RA	RB	MB	ME	Rc
0	6	11	16	21	26	31

```
n ← (RB)59:63
r ← ROTL32((RS)32:63, n)
m ← MASK(MB+32, ME+32)
RA ← r & m
```

The contents of register RS are rotated$_{32}$ left the number of bits specified by (RB)$_{59:63}$. A mask is generated having 1-bits from bit MB+32 through bit ME+32 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Word then AND with Mask*:

Extended:		Equivalent to:	
rotlw	Rx,Ry,Rz	rlwnm	Rx,Ry,Rz,0,31

Programming Note

Let RSL represent the low-order 32 bits of register RS, with the bits numbered from 0 through 31.

rlwnm can be used to extract an n-bit field that starts at variable bit position b in RSL, right-justified into the low-order 32 bits of register RA (clearing the remaining 32-n bits of the low-order 32 bits of RA), by setting RB$_{59:63}$=b+n, MB=32-n, and ME=31. It can be used to extract an n-bit field that starts at variable bit position b in RSL, left-justified into the low-order 32 bits of register RA (clearing the remaining 32-n bits of the low-order 32 bits of RA), by setting RB$_{59:63}$=b, MB = 0, and ME=n-1. It can be used to rotate the contents of the low-order 32 bits of a register left (right) by variable n bits, by setting RB$_{59:63}$=n (32-n), MB=0, and ME=31.

For all the uses given above, the high-order 32 bits of register RA are cleared.

Extended mnemonics are provided for some of these uses; see Appendix C, “Assembler Extended Mnemonics” on page 791.

Rotate Left Word Immediate then Mask Insert *M-form*

rlwimi RA,RS,SH,MB,ME (Rc=0)
rlwimi. RA,RS,SH,MB,ME (Rc=1)

20	RS	RA	SH	MB	ME	Rc
0	6	11	16	21	26	31

```
n ← SH
r ← ROTL32((RS)32:63, n)
m ← MASK(MB+32, ME+32)
RA ← r&m | (RA)&¬m
```

The contents of register RS are rotated$_{32}$ left SH bits. A mask is generated having 1-bits from bit MB+32 through bit ME+32 and 0-bits elsewhere. The rotated data are inserted into register RA under control of the generated mask.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Word Immediate then Mask Insert*:

Extended:		Equivalent to:	
inslwi	Rx,Ry,n,b	rlwimi	Rx,Ry,32-b,b,b+n-1

Programming Note

Let RAL represent the low-order 32 bits of register RA, with the bits numbered from 0 through 31.

rlwimi can be used to insert an n-bit field that is left-justified in the low-order 32 bits of register RS, into RAL starting at bit position b, by setting SH=32-b, MB=b, and ME=(b+n)-1. It can be used to insert an n-bit field that is right-justified in the low-order 32 bits of register RS, into RAL starting at bit position b, by setting SH=32-(b+n), MB=b, and ME=(b+n)-1.

Extended mnemonics are provided for both of these uses; see Appendix C, “Assembler Extended Mnemonics” on page 791.

3.3.14.1.1 64-bit Fixed-Point Rotate Instructions

Rotate Left Doubleword Immediate then Clear Left *MD-form*

rldicl RA,RS,SH,MB (Rc=0)
rldicl. RA,RS,SH,MB (Rc=1)

30	RS	RA	sh	mb	0	sh	Rc
0	6	11	16	21	27	30	31

```
n ← sh5 || sh0:4
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, 63)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left SH bits. A mask is generated having 1-bits from bit MB through bit 63 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Examples of extended mnemonics for *Rotate Left Doubleword Immediate then Clear Left*:

Extended:		Equivalent to:	
extrdi	Rx,Ry,n,b	rldicl	Rx,Ry,b+n,64-n
srdi	Rx,Ry,n	rldicl	Rx,Ry,64-n,n
clrldi	Rx,Ry,n	rldicl	Rx,Ry,0,n

Programming Note

rldicl can be used to extract an n-bit field that starts at bit position b in register RS, right-justified into register RA (clearing the remaining 64-n bits of RA), by setting SH=b+n and MB=64-n. It can be used to rotate the contents of a register left (right) by n bits, by setting SH=n (64-n) and MB=0. It can be used to shift the contents of a register right by n bits, by setting SH=64-n and MB=n. It can be used to clear the high-order n bits of a register, by setting SH=0 and MB=n.

Extended mnemonics are provided for all of these uses; see Appendix C, “Assembler Extended Mnemonics” on page 791.

Rotate Left Doubleword Immediate then Clear Right *MD-form*

rldicr RA,RS,SH,ME (Rc=0)
rldicr. RA,RS,SH,ME (Rc=1)

30	RS	RA	sh	me	1	sh	Rc
0	6	11	16	21	27	30	31

```
n ← sh5 || sh0:4
r ← ROTL64((RS), n)
e ← me5 || me0:4
m ← MASK(0, e)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left SH bits. A mask is generated having 1-bits from bit 0 through bit ME and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Examples of extended mnemonics for *Rotate Left Doubleword Immediate then Clear Right*:

Extended:		Equivalent to:	
extldi	Rx,Ry,n,b	rldicr	Rx,Ry,b,n-1
sldi	Rx,Ry,n	rldicr	Rx,Ry,n,63-n
clrrdi	Rx,Ry,n	rldicr	Rx,Ry,0,63-n

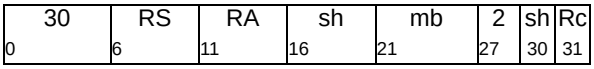
Programming Note

rldicr can be used to extract an n-bit field that starts at bit position b in register RS, left-justified into register RA (clearing the remaining 64-n bits of RA), by setting SH=b and ME=n-1. It can be used to rotate the contents of a register left (right) by n bits, by setting SH=n (64-n) and ME=63. It can be used to shift the contents of a register left by n bits, by setting SH=n and ME=63-n. It can be used to clear the low-order n bits of a register, by setting SH=0 and ME=63-n.

Extended mnemonics are provided for all of these uses (some devolve to ***rldicl***); see Appendix C, “Assembler Extended Mnemonics” on page 791.

Rotate Left Doubleword Immediate then Clear MD-form

rldic RA,RS,SH,MB (Rc=0)
rldic. RA,RS,SH,MB (Rc=1)




```
n ← sh5 || sh0:4
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, ¬n)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left SH bits. A mask is generated having 1-bits from bit MB through bit 63 - SH and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Doubleword Immediate then Clear*:

Extended:		Equivalent to:	
clrlsldi	Rx,Ry,b,n	rldic	Rx,Ry,n,b - n

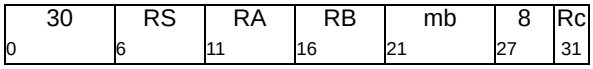
Programming Note

rldic can be used to clear the high-order b bits of the contents of a register and then shift the result left by n bits, by setting SH=n and MB=b - n. It can be used to clear the high-order n bits of a register, by setting SH=0 and MB=n.

Extended mnemonics are provided for both of these uses (the second devolves to ***rldicl***); see Appendix C, “Assembler Extended Mnemonics” on page 791.

Rotate Left Doubleword then Clear Left MDS-form

rldcl RA,RS,RB,MB (Rc=0)
rldcl. RA,RS,RB,MB (Rc=1)




```
n ← (RB)58:63
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, 63)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left the number of bits specified by (RB)$_{58:63}$. A mask is generated having 1-bits from bit MB through bit 63 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Doubleword then Clear Left*:

Extended:		Equivalent to:	
rotld	Rx,Ry,Rz	rldcl	Rx,Ry,Rz,0

Programming Note

rldcl can be used to extract an n-bit field that starts at variable bit position b in register RS, right-justified into register RA (clearing the remaining 64 - n bits of RA), by setting RB$_{58:63}$=b+n and MB=64 - n. It can be used to rotate the contents of a register left (right) by variable n bits, by setting RB$_{58:63}$=n (64 - n) and MB=0.

Extended mnemonics are provided for some of these uses; see Appendix C, “Assembler Extended Mnemonics” on page 791.

Rotate Left Doubleword then Clear Right *MDS-form*

rldcr RA,RS,RB,ME (Rc=0)
rldcr. RA,RS,RB,ME (Rc=1)

30	RS	RA	RB	me	9	Rc
0	6	11	16	21	27	31

```
n ← (RB)58:63
r ← ROTL64((RS), n)
e ← me5 || me0:4
m ← MASK(0, e)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left the number of bits specified by (RB)$_{58:63}$. A mask is generated having 1-bits from bit 0 through bit ME and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 (if Rc=1)

Programming Note

rldcr can be used to extract an n-bit field that starts at variable bit position b in register RS, left-justified into register RA (clearing the remaining 64-n bits of RA), by setting RB$_{58:63}$=b and ME=n-1. It can be used to rotate the contents of a register left (right) by variable n bits, by setting RB$_{58:63}$=n (64-n) and ME=63.

Extended mnemonics are provided for some of these uses (some devolve to ***rldcl***); see Appendix C, “Assembler Extended Mnemonics” on page 791.

Rotate Left Doubleword Immediate then Mask Insert *MD-form*

rldimi RA,RS,SH,MB (Rc=0)
rldimi. RA,RS,SH,MB (Rc=1)

30	RS	RA	sh	mb	3	sh	Rc
0	6	11	16	21	27	30	31

```
n ← sh5 || sh0:4
r ← ROTL64((RS), n)
b ← mb5 || mb0:4
m ← MASK(b, ¬n)
RA ← r&m | (RA)&¬m
```

The contents of register RS are rotated$_{64}$ left SH bits. A mask is generated having 1-bits from bit MB through bit 63-SH and 0-bits elsewhere. The rotated data are inserted into register RA under control of the generated mask.

Special Registers Altered:

CR0 (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Doubleword Immediate then Mask Insert*:

Extended:	**Equivalent to:**
insrdi Rx,Ry,n,b	rldimi Rx,Ry,64-(b+n),b

Programming Note

rldimi can be used to insert an n-bit field that is right-justified in register RS, into register RA starting at bit position b, by setting SH=64-(b+n) and MB=b.

An extended mnemonic is provided for this use; see Appendix C, “Assembler Extended Mnemonics” on page 791.

3.3.14.2 Fixed-Point Shift Instructions

The instructions in this section perform left and right shifts.

Extended mnemonics for shifts

Immediate-form logical (unsigned) shift operations are obtained by specifying appropriate masks and shift values for certain *Rotate* instructions. A set of extended mnemonics is provided to make coding of such shifts simpler and easier to understand. Some of these are shown as examples with the *Rotate* instructions. See Appendix C, “Assembler Extended Mnemonics” on page 791 for additional extended mnemonics.

Programming Note

Any Shift Right Algebraic instruction, followed by ***addze***, can be used to divide quickly by 2^n. The setting of the CA and CA32 bits by the Shift Right Algebraic instructions is independent of mode.

Programming Note

Multiple-precision shifts can be programmed as shown in Section E.1, “Multiple-Precision Shifts” on page 639.

Shift Left Word — *X-form*

slw RA,RS,RB (Rc=0)
slw. RA,RS,RB (Rc=1)

31	RS	RA	RB	24	Rc
0	6	11	16	21	31

```
n ← (RB)59:63
r ← ROTL32((RS)32:63, n)
if (RB)58 = 0 then
     m ← MASK(32, 63-n)
else m ← 640
RA ← r & m
```

The contents of the low-order 32 bits of register RS are shifted left the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into $RA_{32:63}$. $RA_{0:31}$ are set to zero. Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:

CR0 (if Rc=1)

Shift Right Word — *X-form*

srw RA,RS,RB (Rc=0)
srw. RA,RS,RB (Rc=1)

31	RS	RA	RB	536	Rc
0	6	11	16	21	31

```
n ← (RB)59:63
r ← ROTL32((RS)32:63, 64-n)
if (RB)58 = 0 then
    m ← MASK(n+32, 63)
else m ← 640
RA ← r & m
```

The contents of the low-order 32 bits of register RS are shifted right the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into $RA_{32:63}$. $RA_{0:31}$ are set to zero. Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:

CR0 (if Rc=1)

Shift Right Algebraic Word Immediate *X-form*

srawi RA,RS,SH (Rc=0)
srawi. RA,RS,SH (Rc=1)

31	RS	RA	SH	824	Rc
0	6	11	16	21	31

```
n ← SH
r ← ROTL32((RS)32:63, 64-n)
m ← MASK(n+32, 63)
s ← (RS)32
RA ← r&m | (64s)&¬m
carry ← s & ((r&¬m)32:63≠0)
CA    ← carry
CA32  ← carry
```

The contents of the low-order 32 bits of register RS are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 32 of RS is replicated to fill the vacated positions on the left. The 32-bit result is placed into $RA_{32:63}$. Bit 32 of RS is replicated to fill $RA_{0:31}$. CA and CA32 are set to 1 if the low-order 32 bits of (RS) contain a negative number and any 1-bits are shifted out of position 63; otherwise CA and CA32 are set to 0. A shift amount of zero causes RA to receive EXTS($(RS)_{32:63}$), and CA and CA32 to be set to 0.

Special Registers Altered:

CA CA32
CR0 (if Rc=1)

Shift Right Algebraic Word *X-form*

sraw RA,RS,RB (Rc=0)
sraw. RA,RS,RB (Rc=1)

31	RS	RA	RB	792	Rc
0	6	11	16	21	31

```
n ← (RB)59:63
r ← ROTL32((RS)32:63, 64-n)
if (RB)58 = 0 then
    m ← MASK(n+32, 63)
else m ← 640
s ← (RS)32
RA ← r&m | (64s)&¬m
carry ← s & ((r&¬m)32:63≠0)
CA    ← carry
CA32  ← carry
```

The contents of the low-order 32 bits of register RS are shifted right the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 63 are lost. Bit 32 of RS is replicated to fill the vacated positions on the left. The 32-bit result is placed into $RA_{32:63}$. Bit 32 of RS is replicated to fill $RA_{0:31}$. CA and CA32 are set to 1 if the low-order 32 bits of (RS) contain a negative number and any 1-bits are shifted out of position 63; otherwise CA and CA32 are set to 0. A shift amount of zero causes RA to receive EXTS($(RS)_{32:63}$), and CA and CA32 to be set to 0. Shift amounts from 32 to 63 give a result of 64 sign bits, and cause CA and CA32 to receive the sign bit of $(RS)_{32:63}$.

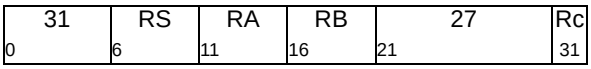
Special Registers Altered:

CA CA32
CR0 (if Rc=1)

3.3.14.2.1 64-bit Fixed-Point Shift Instructions

Shift Left Doubleword *X-form*

sld RA,RS,RB (Rc=0)
sld. RA,RS,RB (Rc=1)




```
n ← (RB)58:63
r ← ROTL64((RS), n)
if (RB)57 = 0 then
     m ← MASK(0, 63-n)
else m ← 640
RA ← r & m
```

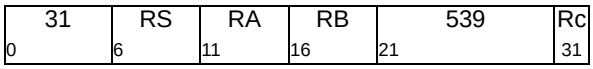
The contents of register RS are shifted left the number of bits specified by $(RB)_{57:63}$. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The result is placed into register RA. Shift amounts from 64 to 127 give a zero result.

Special Registers Altered:

CR0 (if Rc=1)

Shift Right Doubleword *X-form*

srd RA,RS,RB (Rc=0)
srd. RA,RS,RB (Rc=1)




```
n ← (RB)58:63
r ← ROTL64((RS), 64-n)
if (RB)57 = 0 then
    m ← MASK(n, 63)
else m ← 640
RA ← r & m
```

The contents of register RS are shifted right the number of bits specified by $(RB)_{57:63}$. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The result is placed into register RA. Shift amounts from 64 to 127 give a zero result.

Special Registers Altered:

CR0 (if Rc=1)

Shift Right Algebraic Doubleword Immediate XS-form

sradi RA,RS,SH (Rc=0)
sradi. RA,RS,SH (Rc=1)

31	RS	RA	sh	413	sh	Rc
0	6	11	16	21	30	31

```
n ← sh5 || sh0:4
r ← ROTL64((RS), 64-n)
m ← MASK(n, 63)
s ← (RS)0
RA ← r&m | (64s)&¬m
carry ← s & ((r&¬m)≠0)
CA    ← carry
CA32  ← carry
```

The contents of register RS are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 0 of RS is replicated to fill the vacated positions on the left. The result is placed into register RA. CA and CA32 are set to 1 if (RS) is negative and any 1-bits are shifted out of position 63; otherwise CA and CA32 are set to 0. A shift amount of zero causes RA to be set equal to (RS), and CA and CA32 to be set to 0.

Special Registers Altered:

CA CA32
CR0 (if Rc=1)

Shift Right Algebraic Doubleword X-form

srad RA,RS,RB (Rc=0)
srad. RA,RS,RB (Rc=1)

31	RS	RA	RB	794	Rc
0	6	11	16	21	31

```
n ← (RB)58:63
r ← ROTL64((RS), 64-n)
if (RB)57 = 0 then
    m ← MASK(n, 63)
else m ← 640
s ← (RS)0
RA ← r&m | (64s)&¬m
carry ← s & ((r&¬m)≠0)
CA    ← carry
CA32  ← carry
```

The contents of register RS are shifted right the number of bits specified by $(RB)_{57:63}$. Bits shifted out of position 63 are lost. Bit 0 of RS is replicated to fill the vacated positions on the left. The result is placed into register RA. CA and CA32 are set to 1 if (RS) is negative and any 1-bits are shifted out of position 63; otherwise CA and CA32 are set to 0. A shift amount of zero causes RA to be set equal to (RS), and CA and CA32 to be set to 0. Shift amounts from 64 to 127 give a result of 64 sign bits in RA, and cause CA and CA32 to receive the sign bit of (RS).

Special Registers Altered:

CA CA32
CR0 (if Rc=1)

Extend-Sign Word and Shift Left Immediate XS-form

extswsli RA,RS,SH (Rc=0)
extswsli. RA,RS,SH (Rc=1)

31	RS	RA	sh	445	sh	Rc
0	6	11	16	21	30	31

```
n  ← sh5 || sh0:4
r  ← ROTL64(EXTS64(RS32:63), n)
m  ← MASK(0, 63-n)
RA ← r & m
```

The contents of the low order 32 bits of RS are sign-extended to 64 bits and then shifted left SH bits. Bits shifted out of bit 0 are lost. Zeros are supplied to vacated bits on the right. The result is placed in register RA.

Special Registers Altered:

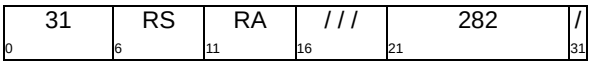
CR0 (if Rc=1)

3.3.15 Binary Coded Decimal (BCD) Assist Instructions

The *Binary Coded Decimal Assist* instructions operate on Binary Coded Decimal operands (***cbcdtd*** and ***addg6s***) and Decimal Floating-Point operands (***cdtbcd***) See Chapter 5. for additional information.

Convert Declets To Binary Coded Decimal X-form

cdtbcd RA, RS




```
do i = 0 to 1
  n ← i x 32
  RA[n+0:n+7] ← 0
  RA[n+8:n+19] ← DPD_TO_BCD( (RS)[n+12:n+21] )
  RA[n+20:n+31] ← DPD_TO_BCD( (RS)[n+22:n+31] )
```

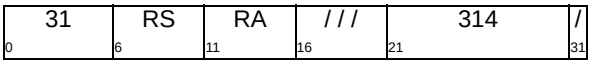
The low-order 20 bits of each word of register RS contain two declets which are converted to six, 4-bit BCD fields; each set of six, 4-bit BCD fields is placed into the low-order 24 bits of the corresponding word in RA. The high-order 8 bits in each word of RA are set to 0.

Special Registers Altered:

None

Convert Binary Coded Decimal To Declets X-form

cbcdtd RA, RS




```
do i = 0 to 1
  n ← i x 32
  RA[n+0:n+11] ← 0
  RA[n+12:n+21] ← BCD_TO_DPD( (RS)[n+8:n+19] )
  RA[n+22:n+31] ← BCD_TO_DPD( (RS)[n+20:n+31] )
```

The low-order 24 bits of each word of register RS contain six, 4-bit BCD fields which are converted to two declets; each set of two declets is placed into the low-order 20 bits of the corresponding word in RA. The high-order 12 bits in each word of RA are set to 0.

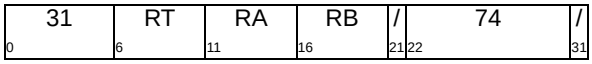
If a 4-bit BCD field has a value greater than 9 the results are undefined.

Special Registers Altered:

None

Add and Generate Sixes *XO-form*

addg6s RT,RA,RB




```
do i = 0 to 15
   dc_i ← carry_out(RA[4xi:63] + RB[4xi:63])
c ← 4(dc_0) || 4(dc_1) || ... || 4(dc_15)
RT ← (¬c) & 0x6666_6666_6666_6666
```

The contents of register RA are added to the contents of register RB. Sixteen carry bits are produced, one for each carry out of decimal position n (bit position 4xn).

A doubleword is composed from the 16 carry bits, and placed into RT. The doubleword consists of a decimal six (0b0110) in every decimal digit position for which the corresponding carry bit is 0, and a zero (0b0000) in every position for which the corresponding carry bit is 1.

Special Registers Altered:

None

Programming Note

addg6s can be used to add or subtract two BCD operands. In these examples it is assumed that r0 contains 0x666...666. (BCD data formats are described in Section 5.3.)

Addition of the unsigned BCD operand in register RA to the unsigned BCD operand in register RB can be accomplished as follows.

```
add       r1,RA,r0
add       r2,r1,RB
addg6s    RT,r1,RB
subf      RT,RT,r2# RT = RA +BCD RB
```

Subtraction of the unsigned BCD operand in register RA from the unsigned BCD operand in register RB can be accomplished as follows. (In this example it is assumed that RB is not register 0.)

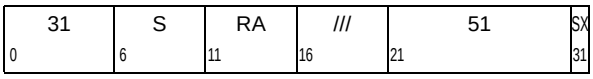
```
addi      r1,RB,1
nor       r2,RA,RA# one's complement of RA
add       r3,r1,r2
addg6s    RT,r1,r2
subf      RT,RT,r3# RT = RB -BCD RA
```

Additional instructions are needed to handle signed BCD operands, and BCD operands that occupy more than one register (e.g., unsigned BCD operands that have more than 16 decimal digits).

3.3.16 Move To/From Vector-Scalar Register Instructions

Move From VSR Doubleword X-form

mfvsrd RA,XS




```
if SX=0 & MSR.FP=0 then FP_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

GPR[RA] ← VSR[32×SX+S].dword[0]
```

Let XS be the value 32×SX + S.

The contents of doubleword element 0 of VSR[XS] are placed into GPR[RA].

For SX=0, ***mfvsrd*** is treated as a *Floating-Point* instruction in terms of resource availability.

For SX=1, ***mfvsrd*** is treated as a *Vector* instruction in terms of resource availability.

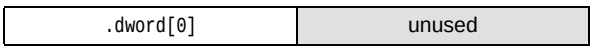
Extended Mnemonics		***Equivalent To***	
mffprd	RA,FRS	mfvsrd	RA,FRS
mfvrd	RA,VRS	mfvsrd	RA,VRS+32

Special Registers Altered

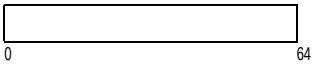
None

Data Layout for mfvsrd

src = VSR[XS]

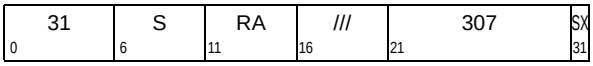


tgt = GPR[RA]



Move From VSR Lower Doubleword X-form

mfvsrld RA,XS




```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

GPR[RA] ← VSR[32×SX+S].dword[1]
```

Let XS be the value 32×SX + S.

The contents of doubleword 1 of VSR[XS] are placed into GPR[RA].

For SX=0, ***mfvsrld*** is treated as a *VSX* instruction in terms of resource availability.

For SX=1, ***mfvsrld*** is treated as a *Vector* instruction in terms of resource availability.

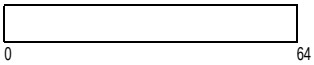
Special Registers Altered:

None

Data Layout for mfvsrld

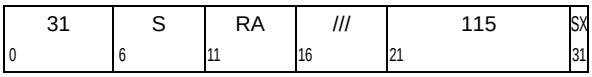
src = VSR[XS]

tgt = GPR[RA]



Move From VSR Word and Zero X-form

mfvsrwz RA,XS




```
if SX=0 & MSR.FP=0 then FP_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

GPR[RA] ← EXTZ64(VSR[32×SX+S].word[1])
```

Let XS be the value 32×SX + S.

The contents of word element 1 of VSR[XS] are placed into bits 32:63 of GPR[RA]. The contents of bits 0:31 of GPR[RA] are set to 0.

For SX=0, ***mfvsrwz*** is treated as a *Floating-Point* instruction in terms of resource availability.

For SX=1, ***mfvsrwz*** is treated as a *Vector* instruction in terms of resource availability.

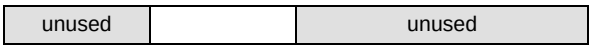
Extended Mnemonics		***Equivalent To***	
mffprwz	RA,FRS	mfvsrwz	RA,FRS
mfvrwz	RA,VRS	mfvsrwz	RA,VRS+32

Special Registers Altered

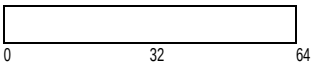
None

Data Layout for mfvsrwz

src = VSR[XS]

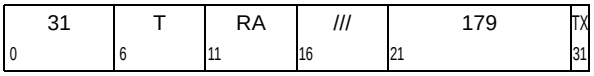


tgt = GPR[RA]



Move To VSR Doubleword X-form

mtvsrd XT,RA




```
if TX=0 & MSR.FP=0 then FP_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

VSR[32×TX+T].dword[0] ← GPR[RA]
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×TX + T.

The contents of GPR[RA] are placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

For TX=0, ***mtvsrd*** is treated as a *Floating-Point* instruction in terms of resource availability.

For TX=1, ***mtvsrd*** is treated as a *Vector* instruction in terms of resource availability.

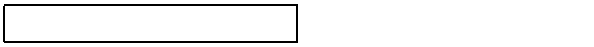
Extended Mnemonics		***Equivalent To***	
mtfprd	FRT,RA	mtvsrd	FRT,RA
mtvrd	VRT,RA	mtvsrd	VRT+32,RA

Special Registers Altered

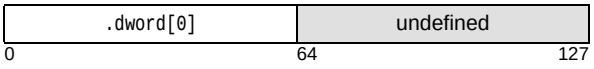
None

Data Layout for mtvsrd

src = GPR[RA]

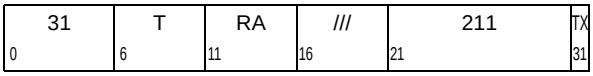


tgt = VSR[XT]



Move To VSR Word Algebraic X-form

mtvsrwa XT,RA




```
if TX=0 & MSR.FP=0 then FP_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

VSR[32×TX+T].dword[0] ← EXTS64(GPR[RA].bit[32:63])
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×TX + T.

The two's-complement integer in bits 32:63 of GPR[RA] is sign-extended to 64 bits and placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

For TX=0, ***mtvsrwa*** is treated as a *Floating-Point* instruction in terms of resource availability.

For TX=1, ***mtvsrwa*** is treated as a *Vector* instruction in terms of resource availability.

Extended Mnemonics		***Equivalent To***	
mtfprwa	FRT,RA	mtvsrwa	FRT,RA
mtvrwa	VRT,RA	mtvsrwa	VRT+32,RA

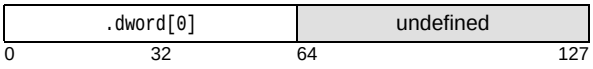
Special Registers Altered

None

Data Layout for mtvsrwa

src = GPR[RA]

tgt = VSR[XT]



Move To VSR Word and Zero X-form

mtvsrwz XT,RA

31	T	RA	///	243	TX
0	6	11	16	21	31

```
if TX=0 & MSR.FP=0 then FP_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

VSR[32×TX+T].dword[0] ← EXTZ64(GPR[RA].word[1])
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×TX + T.

The contents of bits 32:63 of GPR[RA] are placed into word element 1 of VSR[XT]. The contents of word element 0 of VSR[XT] are set to 0.

The contents of doubleword element 1 of VSR[XT] are undefined.

For TX=0, ***mtvsrwz*** is treated as a *Floating-Point* instruction in terms of resource availability.

For TX=1, ***mtvsrwz*** is treated as a *Vector* instruction in terms of resource availability.

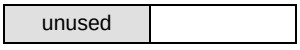
Extended Mnemonics		***Equivalent To***	
mtfprwz	FRT,RA	mtvsrwz	FRT,RA
mtvrwz	VRT,RA	mtvsrwz	VRT+32,RA

Special Registers Altered

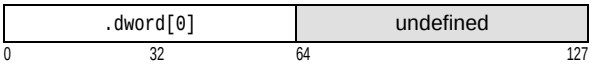
None

Data Layout for mtvsrwz

src = GPR[RA]



tgt = VSR[XT]



Move To VSR Double Doubleword X-form

mtvsrdd XT,RA,RB

31	T	RA	RB	435	TX
0	6	11	16	21	31

```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

VSR[32×TX+T].dword[0] ← (RA=0) ? 0x0000_0000_0000_0000 : GPR[RA]
VSR[32×TX+T].dword[1] ← GPR[RB]
```

Let XT be the value 32×TX + T.

The contents of GPR[RA], or the value 0 if RA=0, are placed into doubleword 0 of VSR[XT].

The contents of GPR[RB] are placed into doubleword 1 of VSR[XT].

For TX=0, ***mtvsrdd*** is treated as a *VSX* instruction in terms of resource availability.

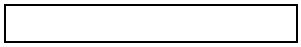
For TX=1, ***mtvsrdd*** is treated as a *Vector* instruction in terms of resource availability.

Special Registers Altered:

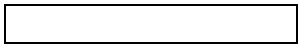
None

Data Layout for mtvsrdd

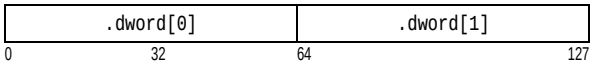
src = GPR[RA]



src = GPR[RB]

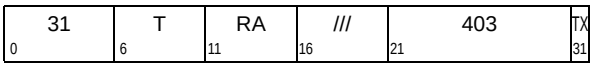


tgt = VSR[XT]



Move To VSR Word & Splat X-form

mtvsrws XT,RA




```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

VSR[32×TX+T].word[0] ← GPR[RA].bit[32:63]
VSR[32×TX+T].word[1] ← GPR[RA].bit[32:63]
VSR[32×TX+T].word[2] ← GPR[RA].bit[32:63]
VSR[32×TX+T].word[3] ← GPR[RA].bit[32:63]
```

Let XT be the value 32×TX + T.

The contents of bits 32:63 of GPR[RA] are placed into each word element of VSR[XT].

For TX=0, ***mtvsrws*** is treated as a *VSX* instruction in terms of resource availability.

For TX=1, ***mtvsrws*** is treated as a *Vector* instruction in terms of resource availability.

Special Registers Altered:

None

3.3.17 Move To/From System Register Instructions

The *Move To Condition Register Fields* instruction has a preferred form; see Section 1.9.1, “Preferred Instruction Forms” on page 23. In the preferred form, the FXM field satisfies the following rule.

- Exactly one bit of the FXM field is set to 1.

Extended mnemonics

Extended mnemonics are provided for the ***mtspr*** and ***mfspr*** instructions so that they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand. An extended mnemonic is provided for the ***mtcrf*** instruction for compatibility with old software (written for a version of the architecture that precedes Version 2.00) that uses it to set the entire Condition Register. Some of these extended mnemonics are shown as examples with the relevant instructions. See Appendix C, “Assembler Extended Mnemonics” on page 791 for additional extended mnemonics.

Move To Special Purpose Register XFX-form

mtspr SPR,RS

31	RS	spr	467	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
switch (n)
  case(13): see Book III
  case(808, 809, 810, 811):
  default:
    if length(SPR(n)) = 64 then
      SPR(n) ← (RS)
    else
      SPR(n) ← (RS)32:63
```

The SPR field denotes a Special Purpose Register, encoded as shown in the table below. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs”. Otherwise, unless the SPR field contains 13 (denoting the AMR), the contents of register RS are placed into the designated Special Purpose Register. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RS are placed into the SPR.

The AMR (Authority Mask Register) is used for “storage protection.” This use, and operation of ***mtspr*** for the AMR, are described in Book III.

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	Register Name
1	00000 00001	XER
3	00000 00011	DSCR
8	00000 01000	LR
9	00000 01001	CTR
13	00000 01101	AMR

[1] Note that the order of the two 5-bit halves of the SPR number is reversed.
[2] See Chapter 5 of Book II.
[3] Accesses to these registers are no-ops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs”

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	Register Name
128	00100 00000	TFHAR[2]
129	00100 00001	TFIAR[2]
130	00100 00010	TEXASR[2]
131	00100 00011	TEXASRU[2]
256	01000 00000	VRSAVE
769	11000 00001	MMCR2
770	11000 00010	MMCRA
771	11000 00011	PMC1
772	11000 00100	PMC2
773	11000 00101	PMC3
774	11000 00110	PMC4
775	11000 00111	PMC5
776	11000 01000	PMC6
779	11000 01011	MMCR0
800	11001 00000	BESCRS
801	11001 00001	BESCRSU
802	11001 00010	BESCRR
803	11001 00011	BESCRRU
804	11001 00100	EBBHR
805	11001 00101	EBBRR
806	11001 00110	BESCR
808	11001 01000	reserved[3]
809	11001 01001	reserved[3]
810	11001 01010	reserved[3]
811	11001 01011	reserved[3]
815	11001 01111	TAR[3]
896	11100 00000	PPR
898	11100 00010	PPR32

[1] Note that the order of the two 5-bit halves of the SPR number is reversed.
[2] See Chapter 5 of Book II.
[3] Accesses to these registers are no-ops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs”

If execution of this instruction is attempted specifying an SPR number that is not shown above, one of the following occurs.

- If spr_0 = 0, the illegal instruction error handler is invoked.
- If spr_0 = 1, the system privileged instruction error handler is invoked.

If an attempt is made to execute ***mtspr*** specifying a TM SPR in other than Non-transactional state, with the exception of TFHAR in suspended state, a TM Bad Thing type Program interrupt is generated.

A complete description of this instruction can be found in Book III.

Special Registers Altered:

See above

Extended Mnemonics:

Examples of extended mnemonics for *Move To Special Purpose Register*:

Extended:		**Equivalent to:**	
mtxer	Rx	mtspr	1,Rx
mtlr	Rx	mtspr	8,Rx
mtctr	Rx	mtspr	9,Rx
mtppr	Rx	mtspr	896,Rx
mtppr32	Rx	mtspr	898,Rx

Programming Note

The AMR is part of the “context” of the program (see Book III). Therefore modification of the AMR requires “synchronization” by software. For this reason, most operating systems provide a system library program that application programs can use to modify the AMR.

Compiler and Assembler Note

For the ***mtspr*** and ***mfspr*** instructions, the SPR number coded in Assembler language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16:20 of the instruction and the low-order 5 bits in bits 11:15.

Move From Special Purpose Register XFX-form

mfspr RT,SPR

31	RT	spr	339	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
switch (n)
  case(129): see Book III
  case(808, 809, 810, 811):
  default:
    if length(SPR(n)) = 64 then
      RT ← SPR(n)
    else
      RT ← 320 || SPR(n)
```

The SPR field denotes a Special Purpose Register, encoded as shown in the table below. If the SPR field contains 129, the instruction references the Transaction Failure Instruction Address Register (TFIAR) and the result is dependent on the privilege with which it is executed. See Book III. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, "Reserved Fields, Reserved Values, and Reserved SPRs". Otherwise, the contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	Register Name
1	00000 00001	XER
3	00000 00011	DSCR
8	00000 01000	LR
9	00000 01001	CTR
13	00000 01101	AMR
128	00100 00000	TFHAR[4]
129	00100 00001	TFIAR[4]
130	00100 00010	TEXASR[4]
131	00100 00011	TEXASRU[4]
136	00100 01000	CTRL
256	01000 00000	VRSAVE
259	01000 00011	SPRG3
268	01000 01100	TB[2]
269	01000 01101	TBU[2]
768	11000 00000	SIER
769	11000 00001	MMCR2
770	11000 00010	MMCRA
771	11000 00011	PMC1
772	11000 00100	PMC2
773	11000 00101	PMC3
774	11000 00110	PMC4
775	11000 00111	PMC5
776	11000 01000	PMC6
779	11000 01011	MMCR0
780	11000 01100	SIAR
781	11000 01101	SDAR
782	11000 01110	MMCR1
800	11001 00000	BESCRS
801	11001 00001	BESCRSU
802	11001 00010	BESCRR
803	11001 00011	BESCRRU
804	11001 00100	EBBHR
805	11001 00101	EBBRR
806	11001 00110	BESCR
808	11001 01000	reserved[3]
809	11001 01001	reserved[3]
810	11001 01010	reserved[3]
811	11001 01011	reserved[3]
815	11001 01111	TAR
896	11100 00000	PPR[10]
898	11100 00010	PPR32

[1] Note that the order of the two 5-bit halves of the SPR number is reversed.
[2] See Chapter 6 of Book II
[3] Accesses to these SPRs are no-ops; see Section 1.3.3, "Reserved Fields, Reserved Values, and Reserved SPRs".
[4] See Chapter 5 of Book II.

If execution of this instruction is attempted specifying an SPR number that is not shown above, one of the following occurs.

- If spr_0 = 0, the illegal instruction error handler is invoked.
- If spr_0 = 1, the system privileged instruction error handler is invoked.

A complete description of this instruction can be found in Book III.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for Move From Special Purpose Register:

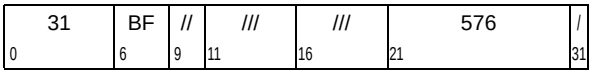
Extended:		Equivalent to:	
mfxer	Rx	mfspr	Rx,1
mflr	Rx	mfspr	Rx,8
mfctr	Rx	mfspr	Rx,9

Note

See the Notes that appear with ***mtspr***.

Move to CR from XER Extended X-form

mcrxrx BF



$CR_{4\times BF+32:4\times BF+35} \leftarrow XER_{OV\ OV32\ CA\ CA32}$

The contents of the OV, OV32, CA, and CA32 are copied to Condition Register field BF.

Special Registers Altered:

CR field BF

Move To One Condition Register Field *XFX-form*

mtocrf FXM,RS

31	RS	1	FXM	/	144	/
0	6	11	12	20	21	31

```
count ← 0
do i = 0 to 7
  if FXM_i = 1 then
    n ← i
    count ← count + 1
if count = 1 then
   CR_(4×n+32:4×n+35) ← (RS)_(4×n+32:4×n+35)
else CR ← undefined
```

If exactly one bit of the FXM field is set to 1, let n be the position of that bit in the field ($0 \le n \le 7$). The contents of bits 4×n+32:4×n+35 of register RS are placed into CR field n (CR bits 4×n+32:4×n+35). Otherwise, the contents of the Condition Register are undefined.

Special Registers Altered:

CR field selected by FXM

Move To Condition Register Fields *XFX-form*

mtcrf FXM,RS

31	RS	0	FXM	/	144	/
0	6	11	12	20	21	31

```
mask ← ⁴(FXM_0) || ⁴(FXM_1) || ... ⁴(FXM_7)
CR ← ((RS)_32:63 & mask) | (CR & ¬mask)
```

The contents of bits 32:63 of register RS are placed into the Condition Register under control of the field mask specified by FXM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0-7. If FXM_i=1 then CR field i (CR bits 4×i+32:4×i+35) is set to the contents of the corresponding field of the low-order 32 bits of RS.

Special Registers Altered:

CR fields selected by mask

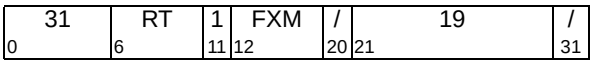
Extended Mnemonics:

Example of extended mnemonics for *Move To Condition Register Fields*:

Extended:	**Equivalent to:**
mtcr Rx	mtcrf 0xFF,Rx

Move From One Condition Register Field XFX-form

mfocrf RT,FXM




```
RT ← undefined
count ← 0
do i = 0 to 7
  if FXM_i = 1 then
    n ← i
    count ← count + 1
if count = 1 then
   RT ← 64 0
   RT 4×n+32:4×n+35 ← CR 4×n+32:4×n+35
```

If exactly one bit of the FXM field is set to 1, let n be the position of that bit in the field ($0 \le n \le 7$). The contents of CR field n (CR bits $4\times n+32:4\times n+35$) are placed into bits $4\times n+32:4\times n+35$ of register RT, and the contents of the remaining bits of register RT are undefined. Otherwise, the contents of register RT are undefined.

If exactly one bit of the FXM field is set to 1, the contents of the remaining bits of register RT are set to 0's instead of being undefined as specified above.

Special Registers Altered:

None

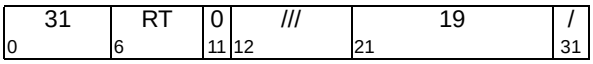
Programming Note

Warning: ***mfocrf*** is not backward compatible with processors that comply with versions of the architecture that precede Version 3.0 B. Such processors may not set to 0 the bits of register RT that do not correspond to the specified CR field. If programs that depend on this clearing behavior are run on such processors, the programs may get incorrect results.

The POWER4, POWER5, POWER7 and POWER8 processors set to 0's all bytes of register RT other than the byte that contains the specified CR field. In the byte that contains the CR field, bits other than those containing the CR field may or may not be set to 0s.

Move From Condition Register XFX-form

mfcr RT



RT ← $^{32}0$ || CR

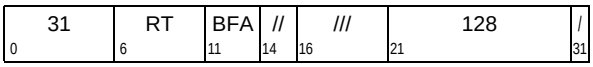
The contents of the Condition Register are placed into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Set Boolean X-form

setb RT,BFA




```
if CR4×BFA+32=1 then
   RT ← 0xFFFF_FFFF_FFFF_FFFF

else if CR4×BFA+33=1 then
   RT ← 0x0000_0000_0000_0001

else
   RT ← 0x0000_0000_0000_0000
```

If the contents of bit 0 of CR field BFA are equal to 0b1, the contents of register RT are set to 0xFFFF_FFFF_FFFF_FFFF.

Otherwise, if the contents of bit 1 of CR field BFA are equal to 0b1, the contents of register RT are set to 0x0000_0000_0000_0001.

Otherwise, the contents of register RT are set to 0x0000_0000_0000_0000.

Special Registers Altered:

None

Chapter 4. Floating-Point Facility

4.1 Floating-Point Facility Overview

This chapter describes the registers and instructions that make up the Floating-Point Facility.

The processor (augmented by appropriate software support, where required) implements a floating-point system compliant with the ANSI/IEEE Standard 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic" (hereafter referred to as "the IEEE standard"). That standard defines certain required "operations" (addition, subtraction, etc.). Herein, the term "floating-point operation" is used to refer to one of these required operations and to additional operations defined (e.g., those performed by *Multiply-Add* or *Reciprocal Estimate* instructions). A Non-IEEE mode is also provided. This mode, which may produce results not in strict compliance with the IEEE standard, allows shorter latency.

Instructions are provided to perform arithmetic, rounding, conversion, comparison, and other operations in floating-point registers; to move floating-point data between storage and these registers; and to manipulate the Floating-Point Status and Control Register explicitly.

These instructions are divided into two categories.

- computational instructions

  The computational instructions are those that perform addition, subtraction, multiplication, division, extracting the square root, rounding, conversion, comparison, and combinations of these operations. These instructions provide the floating-point operations. They place status information into the Floating-Point Status and Control Register. They are the instructions described in Sections 4.6.6 through 4.6.8.

- non-computational instructions

  The non-computational instructions are those that perform loads and stores, move the contents of a floating-point register to another floating-point register possibly altering the sign, manipulate the Floating-Point Status and Control Register explicitly, and select the value from one of two floating-point registers based on the value in a third floating-point register. The operations performed by these instructions are not considered floating-point operations. With the exception of the instructions that manipulate the Floating-Point Status and Control Register explicitly, they do not alter the Floating-Point Status and Control Register. They are the instructions described in Sections 4.6.2 through 4.6.5, and 4.6.10.

A floating-point number consists of a signed exponent and a signed significand. The quantity expressed by this number is the product of the significand and the number $2^{exponent}$. Encodings are provided in the data format to represent finite numeric values, ±Infinity, and values that are "Not a Number" (NaN). Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation. Their encoding permits a variable diagnostic information field. They may be used to indicate such things as uninitialized variables and can be produced by certain invalid operations.

There is one class of exceptional events that occur during instruction execution that is unique to the Floating-Point Facility: the Floating-Point Exception. Floating-point exceptions are signaled with bits set in the Floating-Point Status and Control Register (FPSCR). They can cause the system floating-point enabled exception error handler to be invoked, precisely or imprecisely, if the proper control bits are set.

Floating-Point Exceptions

The following floating-point exceptions are detected by the processor:

- Invalid Operation Exception (VX)
  - SNaN (VXSNAN)
  - Infinity - Infinity (VXISI)
  - Infinity÷Infinity (VXIDI)
  - Zero÷Zero (VXZDZ)
  - Infinity×Zero (VXIMZ)
  - Invalid Compare (VXVC)
  - Software-Defined Condition (VXSOFT)
  - Invalid Square Root (VXSQRT)

Invalid Integer Convert (VXCVI)
- Zero Divide Exception (ZX)
- Overflow Exception (OX)
- Underflow Exception (UX)
- Inexact Exception (XX)

Each floating-point exception, and each category of Invalid Operation Exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. See Section 4.2.2, “Floating-Point Status and Control Register” on page 124 for a description of these exception and enable bits, and Section 4.4, “Floating-Point Exceptions” on page 132 for a detailed discussion of floating-point exceptions, including the effects of the enable bits.

4.2 Floating-Point Facility Registers

4.2.1 Floating-Point Registers

Implementations of this architecture provide 32 floating-point registers (FPRs). The floating-point instruction formats provide 5-bit fields for specifying the FPRs to be used in the execution of the instruction. The FPRs are numbered 0-31. See Figure 45 on page 124.

Each FPR contains 64 bits that support the floating-point double format. Every instruction that interprets the contents of an FPR as a floating-point value uses the floating-point double format for this interpretation.

The computational instructions, and the *Move* and *Select* instructions, operate on data located in FPRs and, with the exception of the *Compare* instructions, place the result value into an FPR and optionally (when Rc=1) place status information into the Condition Register.

Load Double and *Store Double* instructions are provided that transfer 64 bits of data between storage and the FPRs with no conversion. *Load Single* instructions are provided to transfer and convert floating-point values in floating-point single format from storage to the same value in floating-point double format in the FPRs. *Store Single* instructions are provided to transfer and convert floating-point values in floating-point double format from the FPRs to the same value in floating-point single format in storage.

Instructions are provided that manipulate the Floating-Point Status and Control Register and the Condition Register explicitly. Some of these instructions copy data from an FPR to the Floating-Point Status and Control Register or vice versa.

The computational instructions and the *Select* instruction accept values from the FPRs in double format. For single-precision arithmetic instructions, all input values must be representable in single format; if they are not, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the Condition Register (if Rc=1), are undefined.

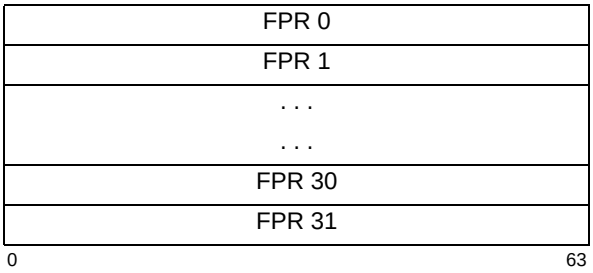


Figure 45. Floating-Point Registers

4.2.2 Floating-Point Status and Control Register

The Floating-Point Status and Control Register (FPSCR) controls the handling of floating-point exceptions and records status resulting from the floating-point operations. Bits 32:55 are status bits. Bits 56:63 are control bits.

The exception bits in the FPSCR (bits 35:44, 53:55) are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an ***mcrfs***, ***mtfsfi***, ***mtfsf***, or ***mtfsb0*** instruction. The exception summary bits in the FPSCR (FX, FEX, and VX, which are bits 32:34) are not considered to be “exception bits”, and only FX is sticky.

FEX and VX are simply the ORs of other FPSCR bits. Therefore these two bits are not listed among the FPSCR bits affected by the various instructions.

Figure 46. Floating-Point Status and Control Register

The bit definitions for the FPSCR are as follows.

Bit(s)	Description
0:31	Reserved
32	***Floating-Point Exception Summary*** (FX) Every floating-point instruction, except ***mtfsfi*** and ***mtfsf***, implicitly sets $FPSCR_{FX}$ to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** can alter $FPSCR_{FX}$ explicitly.

Programming Note

$FPSCR_{FX}$ is defined not to be altered implicitly by ***mtfsfi*** and ***mtfsf*** because permitting these instructions to alter $FPSCR_{FX}$ implicitly could cause a paradox. An example is an ***mtfsfi*** or ***mtfsf*** instruction that supplies 0 for $FPSCR_{FX}$ and 1 for $FPSCR_{OX}$, and is executed when $FPSCR_{OX}$=0. See also the Programming Notes with the definition of these two instructions.

33 ***Floating-Point Enabled Exception Summary*** (FEX)
This bit is the OR of all the floating-point exception bits masked by their respective enable bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter $FPSCR_{FEX}$ explicitly.

34 ***Floating-Point Invalid Operation Exception Summary*** (VX)
This bit is the OR of all the Invalid Operation exception bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter $FPSCR_{VX}$ explicitly.

35 ***Floating-Point Overflow Exception*** (OX)
See Section 4.4.3, “Overflow Exception” on page 135.

36 ***Floating-Point Underflow Exception*** (UX)
See Section 4.4.4, “Underflow Exception” on page 136.

37 ***Floating-Point Zero Divide Exception*** (ZX)
See Section 4.4.2, “Zero Divide Exception” on page 134.

38 ***Floating-Point Inexact Exception*** (XX)
See Section 4.4.5, “Inexact Exception” on page 136.

$FPSCR_{XX}$ is a sticky version of $FPSCR_{FI}$ (see below). Thus the following rules completely describe how $FPSCR_{XX}$ is set by a given instruction.

- If the instruction affects $FPSCR_{FI}$, the new value of $FPSCR_{XX}$ is obtained by ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.
- If the instruction does not affect $FPSCR_{FI}$, the value of $FPSCR_{XX}$ is unchanged.

39 ***Floating-Point Invalid Operation Exception (SNaN)*** (VXSNAN)
See Section 4.4.1, “Invalid Operation Exception” on page 134.

40 ***Floating-Point Invalid Operation Exception (∞ - ∞)*** (VXISI)
See Section 4.4.1.

41 ***Floating-Point Invalid Operation Exception (∞ ÷ ∞)*** (VXIDI)
See Section 4.4.1.

42 ***Floating-Point Invalid Operation Exception (0 ÷0)*** (VXZDZ)
See Section 4.4.1.

43 ***Floating-Point Invalid Operation Exception (∞ ×0)*** (VXIMZ)
See Section 4.4.1.

44 ***Floating-Point Invalid Operation Exception (Invalid Compare)*** (VXVC)
See Section 4.4.1.

45 ***Floating-Point Fraction Rounded*** (FR)
The last *Arithmetic* or *Rounding and Conversion* instruction incremented the fraction during rounding. See Section 4.3.6, “Rounding” on page 131. This bit is not sticky.

46 ***Floating-Point Fraction Inexact*** (FI)
The last *Arithmetic* or *Rounding and Conversion* instruction either produced an inexact result during rounding or caused a disabled Overflow Exception. See Section 4.3.6. This bit is not sticky.

See the definition of $FPSCR_{XX}$, above, regarding the relationship between $FPSCR_{FI}$ and $FPSCR_{XX}$.

47:51 ***Floating-Point Result Flags*** (FPRF)
Arithmetic, rounding, and *Convert From Integer* instructions set this field based on the result placed into the target register and on the target precision, except that if any portion of the result is undefined then the value placed into FPRF is undefined. Floating-point *Compare* instructions set this field based on the relative values of the operands being compared. For *Convert To Integer* instructions, the value placed into FPRF is undefined. Additional details are given below.

Programming Note

A single-precision operation that produces a denormalized result sets FPRF to indicate a denormalized number. When possible, single-precision denormalized numbers are represented in normalized double format in the target register.

47 ***Floating-Point Result Class Descriptor*** (C)
Arithmetic, rounding, and *Convert From Integer* instructions may set this bit with the FPCC bits, to indicate the class of the result as shown in Figure 47 on page 127.

48:51 ***Floating-Point Condition Code*** (FPCC)
Floating-point *Compare* instructions set one of

the FPCC bits to 1 and the other three FPCC bits to 0. Arithmetic, rounding, and *Convert From Integer* instructions may set the FPCC bits with the C bit, to indicate the class of the result as shown in Figure 47 on page 127. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.

48 ***Floating-Point Less Than or Negative*** (FL or <)

49 ***Floating-Point Greater Than or Positive*** (FG or >)

50 ***Floating-Point Equal or Zero*** (FE or =)

51 ***Floating-Point Unordered or NaN*** (FU or ?)

52 Reserved

53 ***Floating-Point Invalid Operation Exception (Software-Defined Condition)*** (VXSOFT)
This bit can be altered only by ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, or ***mtfsb1***. See Section 4.4.1.

Programming Note

$FPSCR_{VXSOFT}$ can be used by software to indicate the occurrence of an arbitrary, software-defined, condition that is to be treated as an Invalid Operation Exception. For example, the bit could be set by a program that computes a base 10 logarithm if the supplied input is negative.

54 ***Floating-Point Invalid Operation Exception (Invalid Square Root)*** (VXSQRT)
See Section 4.4.1.

55 ***Floating-Point Invalid Operation Exception (Invalid Integer Convert)*** (VXCVI)
See Section 4.4.1.

56 ***Floating-Point Invalid Operation Exception Enable*** (VE)
See Section 4.4.1.

57 ***Floating-Point Overflow Exception Enable*** (OE)
See Section 4.4.3, “Overflow Exception” on page 135.

58 ***Floating-Point Underflow Exception Enable*** (UE)
See Section 4.4.4, “Underflow Exception” on page 136.

59 ***Floating-Point Zero Divide Exception Enable*** (ZE)
See Section 4.4.2, “Zero Divide Exception” on page 134.

60 ***Floating-Point Inexact Exception Enable*** (XE)
See Section 4.4.5, “Inexact Exception” on page 136.

61 ***Floating-Point Non-IEEE Mode*** (NI)
Floating-point non-IEEE mode is optional. If floating-point non-IEEE mode is not implemented, this bit is treated as reserved, and the remainder of the definition of this bit does not apply.

If floating-point non-IEEE mode is implemented, this bit has the following meaning.

0 The processor is not in floating-point non-IEEE mode (i.e., all floating-point operations conform to the IEEE standard).
1 The processor is in floating-point non-IEEE mode.

When the processor is in floating-point non-IEEE mode, the remaining FPSCR bits may have meanings different from those given in this document, and floating-point operations need not conform to the IEEE standard. The effects of executing a given floating-point instruction with $FPSCR_{NI}$=1, and any additional requirements for using non-IEEE mode, are implementation-dependent. The results of executing a given instruction in non-IEEE mode may vary between implementations, and between different executions on the same implementation.

Programming Note

When the processor is in floating-point non-IEEE mode, the results of floating-point operations may be approximate, and performance for these operations may be better, more predictable, or less data-dependent than when the processor is not in non-IEEE mode. For example, in non-IEEE mode an implementation may return 0 instead of a denormalized number, and may return a large number instead of an infinity.

62:63 ***Floating-Point Rounding Control*** (RN) See Section 4.3.6, “Rounding” on page 131.

00 Round to Nearest
01 Round toward Zero
10 Round toward +Infinity
11 Round toward -Infinity

Result Flags					Result Value Class
C	<	>	=	?	
1	0	0	0	1	Quiet NaN
0	1	0	0	1	- Infinity
0	1	0	0	0	- Normalized Number
1	1	0	0	0	- Denormalized Number
1	0	0	1	0	- Zero
0	0	0	1	0	+ Zero
1	0	1	0	0	+ Denormalized Number
0	0	1	0	0	+ Normalized Number
0	0	1	0	1	+ Infinity

Figure 47. Floating-Point Result Flags

4.3 Floating-Point Data

4.3.1 Data Format

This architecture defines the representation of a floating-point value in two different binary fixed-length formats. The format may be a 32-bit single format for a single-precision value or a 64-bit double format for a double-precision value. The single format may be used for data in storage. The double format may be used for data in storage and for data in floating-point registers.

The lengths of the exponent and the fraction fields differ between these two formats. The structure of the single and double formats is shown below.

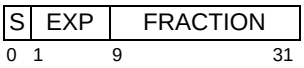


Figure 48. Floating-point single format

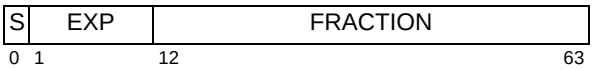


Figure 49. Floating-point double format

Values in floating-point format are composed of three fields:

S	sign bit
EXP	exponent+bias
FRACTION	fraction

Representation of numeric values in the floating-point formats consists of a sign bit (S), a biased exponent (EXP), and the fraction portion (FRACTION) of the significand. The significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit is 1 for normalized numbers and 0 for denormalized numbers and is located in the unit bit position (i.e., the first bit to the left of the binary point). Values representable within the two floating-point formats can be specified by the parameters listed in Figure 50.

	Format	
	Single	Double
Exponent Bias	+127	+1023
Maximum Exponent	+127	+1023
Minimum Exponent	-126	-1022
Widths (bits)		
Format	32	64
Sign	1	1
Exponent	8	11
Fraction	23	52
Significand	24	53

Figure 50. IEEE floating-point fields

The architecture requires that the FPRs of the Floating-Point Facility support the floating-point double format only.

4.3.2 Value Representation

This architecture defines numeric and non-numeric values representable within each of the two supported formats. The numeric values are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The non-numeric values representable are the infinities and the Not a Numbers (NaNs). The infinities are adjoined to the real numbers, but are not numbers themselves, and the standard rules of arithmetic do not hold when they are used in an operation. They are related to the real numbers by order alone. It is possible however to define restricted operations among numbers and infinities as defined below. The relative location on the real number line for each of the defined entities is shown in Figure 51.

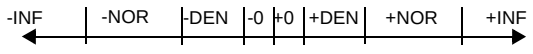


Figure 51. Approximation to real numbers

The NaNs are not related to the numeric values or infinities by order or value but are encodings used to convey diagnostic information such as the representation of uninitialized variables.

The following is a description of the different floating-point values defined in the architecture:

Binary floating-point numbers
Machine representable values used as approximations to real numbers. Three categories of numbers are supported: normalized numbers, denormalized numbers, and zero values.

Normalized numbers (± NOR)
These are values that have a biased exponent value in the range:

1 to 254 in single format
1 to 2046 in double format

They are values in which the implied unit bit is 1. Normalized numbers are interpreted as follows:

$NOR = (-1)^s \times 2^E \times (1.fraction)$

where s is the sign, E is the unbiased exponent, and 1.fraction is the significand, which is composed of a leading unit bit (implied bit) and a fraction part.

The ranges covered by the magnitude (M) of a normalized floating-point number are approximately equal to:

Single Format:

$1.2 \times 10^{-38} \leq M \leq 3.4 \times 10^{38}$

Double Format:

$2.2 \times 10^{-308} \leq M \leq 1.8 \times 10^{308}$

Zero values (± 0)
These are values that have a biased exponent value of zero and a fraction value of zero. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (i.e., comparison regards +0 as equal to -0).

Denormalized numbers (± DEN)
These are values that have a biased exponent value of zero and a nonzero fraction value. They are nonzero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is 0. Denormalized numbers are interpreted as follows:

$DEN = (-1)^s \times 2^{Emin} \times (0.fraction)$

where Emin is the minimum representable exponent value (-126 for single-precision, -1022 for double-precision).

Infinities (± ∞)
These are values that have the maximum biased exponent value:

255 in single format
2047 in double format

and a zero fraction value. They are used to approximate values greater in magnitude than the maximum normalized value.

Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and the real numbers can be related by ordering in the affine sense:

$$-\infty < \text{every finite number} < +\infty$$

Arithmetic on infinities is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in Section 4.4.1, “Invalid Operation Exception” on page 134.

For comparison operations, +Infinity compares equal to +Infinity and -Infinity compares equal to -Infinity.

Not a Numbers (NaNs)
These are values that have the maximum biased exponent value and a nonzero fraction value. The sign bit is ignored (i.e., NaNs are neither positive nor negative). If the high-order bit of the fraction field is 0 then the NaN is a *Signaling NaN*; otherwise it is a *Quiet NaN*.

Signaling NaNs are used to signal exceptions when they appear as operands of computational instructions.

Quiet NaNs are used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when Invalid Operation Exception is disabled ($FPSCR_{VE}$=0). Quiet NaNs propagate through all floating-point operations except ordered comparison, *Floating Round to Single-Precision*, and conversion to integer. Quiet NaNs do not signal exceptions, except for ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of floating-point operations, and used to convey diagnostic information to help identify results from invalid operations.

When a QNaN is the result of a floating-point operation because one of the operands is a NaN or because a QNaN was generated due to a disabled Invalid Operation Exception, then the following rule is applied to determine the NaN with the high-order fraction bit set to 1 that is to be stored as the result.

```
if (FRA) is a NaN
   then FRT ← (FRA)
   else if (FRB) is a NaN
      then if instruction is frsp
           then FRT ← (FRB)0:34 || 29 0
           else FRT ← (FRB)
      else if (FRC) is a NaN
           then FRT ← (FRC)
           else if generated QNaN
                then FRT ← generated QNaN
```

If the operand specified by FRA is a NaN, then that NaN is stored as the result. Otherwise, if the operand specified by FRB is a NaN (if the instruction specifies an FRB operand), then that NaN is stored as the result, with the low-order 29 bits of the result set to 0 if the instruction is ***frsp***. Otherwise, if the operand specified by FRC is a NaN (if the instruction specifies an FRC operand), then that NaN is stored as the result. Otherwise, if a QNaN was generated due to a disabled Invalid Operation Exception, then that QNaN is stored as the result. If a QNaN is to be generated as a result, then the QNaN generated has a sign bit of 0, an exponent field of all 1s, and a high-order fraction bit of 1 with all other fraction bits 0. Any instruction that generates a QNaN as the result of a disabled Invalid Operation

Exception generates this QNaN (i.e., 0x7FF8_0000_0000_0000).

A double-precision NaN is considered to be representable in single format if and only if the low-order 29 bits of the double-precision NaN's fraction are zero.

4.3.3 Sign of Result

The following rules govern the sign of the result of an arithmetic, rounding, or conversion operation, when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the operand having the larger absolute value. If both operands have the same sign, the sign of the result of an add operation is the same as the sign of the operands. The sign of the result of the subtract operation x-y is the same as the sign of the result of the add operation x+(-y).

  When the sum of two operands with opposite sign, or the difference of two operands with the same sign, is exactly zero, the sign of the result is positive in all rounding modes except Round toward -Infinity, in which mode the sign is negative.
- The sign of the result of a multiply or divide operation is the Exclusive OR of the signs of the operands.
- The sign of the result of a *Square Root* or *Reciprocal Square Root Estimate* operation is always positive, except that the square root of -0 is -0 and the reciprocal square root of -0 is -Infinity.
- The sign of the result of a *Round to Single-Precision,* or *Convert From Integer*, or *Round to Integer* operation is the sign of the operand being converted.

For the *Multiply-Add* instructions, the rules given above are applied first to the multiply operation and then to the add or subtract operation (one of the inputs to the add or subtract operation is the result of the multiply operation).

4.3.4 Normalization and Denormalization

The intermediate result of an arithmetic or ***frsp*** instruction may require normalization and/or denormalization as described below. Normalization and denormalization do not affect the sign of the result.

When an arithmetic or rounding instruction produces an intermediate result which carries out of the significand, or in which the significand is nonzero but has a leading zero bit, it is not a normalized number and must be normalized before it is stored. For the carry-out case, the significand is shifted right one bit, with a one shifted into the leading significand bit, and the exponent is incremented by one. For the leading-zero case, the significand is shifted left while decrementing its exponent by one for each bit shifted, until the leading significand bit becomes one. The Guard bit and the Round bit (see Section 4.5.1, "Execution Model for IEEE Operations" on page 137) participate in the shift with zeros shifted into the Round bit. The exponent is regarded as if its range were unlimited.

After normalization, or if normalization was not required, the intermediate result may have a nonzero significand and an exponent value that is less than the minimum value that can be represented in the format specified for the result. In this case, the intermediate result is said to be "Tiny" and the stored result is determined by the rules described in Section 4.4.4, "Underflow Exception". These rules may require denormalization.

A number is denormalized by shifting its significand right while incrementing its exponent by 1 for each bit shifted, until the exponent is equal to the format's minimum value. If any significant bits are lost in this shifting process then "Loss of Accuracy" has occurred (See Section 4.4.4, "Underflow Exception" on page 136) and Underflow Exception is signaled.

4.3.5 Data Handling and Precision

Most of the *Floating-Point Facility Architecture*, including all computational, *Move*, and *Select* instructions, use the floating-point double format to represent data in the FPRs. Single-precision and integer-valued operands may be manipulated using double-precision operations. Instructions are provided to coerce these values from a double format operand. Instructions are also provided for manipulations which do not require double-precision. In addition, instructions are provided to access a true single-precision representation in storage, and a fixed-point integer representation in GPRs.

4.3.5.1 Single-Precision Operands

For single format data, a format conversion from single to double is performed when loading from storage into an FPR and a format conversion from double to single is performed when storing from an FPR to storage. No floating-point exceptions are caused by these instructions. An instruction is provided to explicitly convert a double format operand in an FPR to single-precision. Floating-point single-precision is enabled with four types of instruction.

1. Load Floating-Point Single

   This form of instruction accesses a single-precision operand in single format in storage, converts it to double format, and loads it into an FPR. No floating-point exceptions are caused by these instructions.

2. Round to Floating-Point Single-Precision

   The Floating Round to Single-Precision instruction rounds a double-precision operand to single-precision, checking the exponent for single-precision range and handling any exceptions according to respective enable bits, and places that operand into an FPR in double format. For results produced by single-precision arithmetic instructions, single-precision loads, and other instances of the Floating Round to Single-Precision instruction, this operation does not alter the value.

3. Single-Precision Arithmetic Instructions

   This form of instruction takes operands from the FPRs in double format, performs the operation as if it produced an intermediate result having infinite precision and unbounded exponent range, and then coerces this intermediate result to fit in single format. Status bits, in the FPSCR and optionally in the Condition Register, are set to reflect the single-precision result. The result is then converted to double format and placed into an FPR. The result lies in the range supported by the single format.

   If any input value is not representable in single format and either OE=1 or UE=1, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the Condition Register (if Rc=1), are undefined.

   For ***fres***[**.**] or ***frsqrtes***[**.**], if the input value is finite and has an unbiased exponent greater than +127, the input value is interpreted as an Infinity.

4. Store Floating-Point Single

   This form of instruction converts a double-precision operand to single format and stores that operand into storage. No floating-point exceptions are caused by these instructions. (The value being stored is effectively assumed to be the result of an instruction of one of the preceding three types.)

When the result of a *Load Floating-Point Single*, *Floating Round to Single-Precision*, or single-precision arithmetic instruction is stored in an FPR, the low-order 29 FRACTION bits are zero.

Programming Note

The *Floating Round to Single-Precision* instruction is provided to allow value conversion from double-precision to single-precision with appropriate exception checking and rounding. This instruction should be used to convert double-precision floating-point values (produced by double-precision load and arithmetic instructions and by ***fcfid***) to single-precision values prior to storing them into single format storage elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision load and arithmetic instructions are already single-precision values and can be stored directly into single format storage elements, or used directly as operands for single-precision arithmetic instructions, without preceding the store, or the arithmetic instruction, by a *Floating Round to Single-Precision* instruction.

Programming Note

A single-precision value can be used in double-precision arithmetic operations. The reverse is true only if the double-precision value is representable in single format.

Some implementations may execute single-precision arithmetic instructions faster than double-precision arithmetic instructions. Therefore, if double-precision accuracy is not required, single-precision data and instructions should be used.

4.3.5.2 Integer-Valued Operands

Instructions are provided to round floating-point operands to integer values in floating-point format. To facilitate exchange of data between the floating-point and fixed-Point facilities, instructions are provided to convert between floating-point double format and fixed-point integer format in an FPR. Computation on integer-valued operands may be performed using arithmetic instructions of the required precision. (The results may not be integer values.) The two groups of instructions provided specifically to support integer-valued operands are described below.

1. Floating Round to Integer

   The *Floating Round to Integer* instructions round a double-precision operand to an integer value in floating-point double format. These instructions may cause Invalid Operation (VXSNAN) exceptions. See Sections 4.3.6 and 4.5.1 for more information about rounding.

2. Floating Convert To/From Integer

   The *Floating Convert To Integer* instructions convert a double-precision operand to a 32-bit or 64-bit signed fixed-point integer format. Variants are provided both to perform rounding based on

the value of $FPSCR_{RN}$ and to round toward zero. These instructions may cause Invalid Operation (VXSNaN, VXCVI) and Inexact exceptions. The *Floating Convert From Integer* instruction converts a 64-bit signed fixed-point integer to a double-precision floating-point integer. Because of the limitations of the source format, only an Inexact exception may be generated.

4.3.6 Rounding

The material in this section applies to operations that have numeric operands (i.e., operands that are not infinities or NaNs). Rounding the intermediate result of such an operation may cause an Overflow Exception, an Underflow Exception, or an Inexact Exception. The remainder of this section assumes that the operation causes no exceptions and that the result is numeric. See Section 4.3.2, “Value Representation” and Section 4.4, “Floating-Point Exceptions” for the cases not covered here.

The *Arithmetic* and *Rounding and Conversion* instructions round their intermediate results. With the exception of the *Estimate* instructions, these instructions produce an intermediate result that can be regarded as having infinite precision and unbounded exponent range. All but two groups of these instructions normalize or denormalize the intermediate result prior to rounding and then place the final result into the target FPR in double format. The *Floating Round to Integer* and *Floating Convert To Integer* instructions with biased exponents ranging from 1022 through 1074 are prepared for rounding by repetitively shifting the significand right one position and incrementing the biased exponent until it reaches a value of 1075. (Intermediate results with biased exponents 1075 or larger are already integers, and with biased exponents 1021 or less round to zero.) After rounding, the final result for *Floating Round to Integer* is normalized and put in double format, and for *Floating Convert To Integer* is converted to a signed fixed-point integer.

FPSCR bits FR and FI generally indicate the results of rounding. Each of the instructions which rounds its intermediate result sets these bits. If the fraction is incremented during rounding then FR is set to 1, otherwise FR is set to 0. If the result is inexact then FI is set to 1, otherwise FI is set to zero. The *Round to Integer* instructions are exceptions to this rule, setting FR and FI to 0. The *Estimate* instructions set FR and FI to undefined values. The remaining floating-point instructions do not alter FR and FI.

Four user-selectable rounding modes are provided through the Floating-Point Rounding Control field in the FPSCR. See Section 4.2.2, “Floating-Point Status and Control Register”. These are encoded as follows.

RN	Rounding Mode
00	Round to Nearest
01	Round toward Zero
10	Round toward +Infinity
11	Round toward -Infinity

Let Z be the intermediate arithmetic result or the operand of a convert operation. If Z can be represented exactly in the target format, then the result in all rounding modes is Z as represented in the target format. If Z cannot be represented exactly in the target format, let Z1 and Z2 bound Z as the next larger and next smaller numbers representable in the target format. Then Z1 or Z2 can be used to approximate the result in the target format.

Figure 52 shows the relation of Z, Z1, and Z2 in this case. The following rules specify the rounding in the four modes. “LSB” means “least significant bit”.

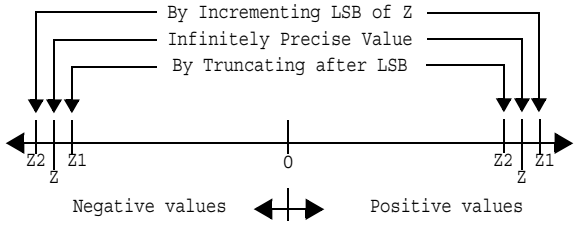


Figure 52. Selection of Z1 and Z2

Round to Nearest
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the one that is even (least significant bit 0).

Round toward Zero
Choose the smaller in magnitude (Z1 or Z2).

Round toward +Infinity
Choose Z1.

Round toward -Infinity
Choose Z2.

See Section 4.5.1, “Execution Model for IEEE Operations” on page 137 for a detailed explanation of rounding.

4.4 Floating-Point Exceptions

This architecture defines the following floating-point exceptions:

- Invalid Operation Exception
  - SNaN
  - Infinity - Infinity
  - Infinity÷Infinity
  - Zero÷Zero
  - Infinity×Zero
  - Invalid Compare
  - Software-Defined Condition
  - Invalid Square Root
  - Invalid Integer Convert
- Zero Divide Exception
- Overflow Exception
- Underflow Exception
- Inexact Exception

These exceptions, other than Invalid Operation Exception due to Software-Defined Condition, may occur during execution of computational instructions. An Invalid Operation Exception due to Software-Defined Condition occurs when a *Move To FPSCR* instruction sets $FPSCR_{VXSOFT}$ to 1.

Each floating-point exception, and each category of Invalid Operation Exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. The exception bit indicates occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the FE0 and FE1 bits (see page 133), whether and how the system floating-point enabled exception error handler is invoked. (In general, the enabling specified by the enable bit is of invoking the system error handler, not of permitting the exception to occur. The occurrence of an exception depends only on the instruction and its inputs, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an Underflow Exception may depend on the setting of the enable bit.)

A single instruction, other than ***mtfsfi*** or ***mtfsf***, may set more than one exception bit only in the following cases:

- Inexact Exception may be set with Overflow Exception.
- Inexact Exception may be set with Underflow Exception.
- Invalid Operation Exception (SNaN) is set with Invalid Operation Exception ($\infty\times0$) for *Multiply-Add* instructions for which the values being multiplied are infinity and zero and the value being added is an SNaN.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Compare) for *Compare Ordered* instructions.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Integer Convert) for *Convert To Integer* instructions.

When an exception occurs the writing of a result to the target register may be suppressed or a result may be delivered, depending on the exception.

The writing of a result to the target register is suppressed for the following kinds of exception, so that there is no possibility that one of the operands is lost:

- Enabled Invalid Operation
- Enabled Zero Divide

For the remaining kinds of exception, a result is generated and written to the destination specified by the instruction causing the exception. The result may be a different value for the enabled and disabled conditions for some of these exceptions. The kinds of exception that deliver a result are the following:

- Disabled Invalid Operation
- Disabled Zero Divide
- Disabled Overflow
- Disabled Underflow
- Disabled Inexact
- Enabled Overflow
- Enabled Underflow
- Enabled Inexact

Subsequent sections define each of the floating-point exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of “traps” and “trap handlers”. In this architecture, an FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the “trap enabled” case; the expectation is that the exception will be detected by software, which will revise the result. An FPSCR exception enable bit of 0 causes generation of the “default result” value specified for the “trap disabled” (or “no trap occurs” or “trap is not implemented”) case; the expectation is that the exception will not be detected by software, which will simply use the default result. The result to be delivered in each case for each exception is described in the sections below.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is desired for all exceptions, all FPSCR exception enable bits should be set to 0 and Ignore Exceptions Mode (see below) should be used. In this case the system floating-point enabled exception error handler is not invoked, even if floating-point exceptions occur: software can inspect the FPSCR exception bits if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to 1 and a mode other than Ignore Exceptions Mode must be used. In this case the system floating-point enabled exception error handler is invoked if an enabled float-

ing-point exception occurs. The system floating-point enabled exception error handler is also invoked if a *Move To FPSCR* instruction causes an exception bit and the corresponding enable bit both to be 1; the *Move To FPSCR* instruction is considered to cause the enabled exception.

The FE0 and FE1 bits control whether and how the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs. The location of these bits and the requirements for altering them are described in Book III. (The system floating-point enabled exception error handler is never invoked because of a disabled floating-point exception.) The effects of the four possible settings of these bits are as follows.

FE0	FE1	Description
0	0	**Ignore Exceptions Mode** Floating-point exceptions do not cause the system floating-point enabled exception error handler to be invoked.
0	1	**Imprecise Nonrecoverable Mode** The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. It may not be possible to identify the excepting instruction or the data that caused the exception. Results produced by the excepting instruction may have been used by or may have affected subsequent instructions that are executed before the error handler is invoked.
1	0	**Imprecise Recoverable Mode** The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. Sufficient information is provided to the error handler that it can identify the excepting instruction and the operands, and correct the result. No results produced by the excepting instruction have been used by or have affected subsequent instructions that are executed before the error handler is invoked.
1	1	**Precise Mode** The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.

In all cases, the question of whether a floating-point result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by the value of the FE0 and FE1 bits.

In all cases in which the system floating-point enabled exception error handler is invoked, all instructions before the instruction at which the system floating-point enabled exception error handler is invoked have completed, and no instruction after the instruction at which the system floating-point enabled exception error handler is invoked has begun execution. The instruction at which the system floating-point enabled exception error handler is invoked has completed if it is the excepting instruction and there is only one such instruction. Otherwise it has not begun execution (or may have been partially executed in some cases, as described in Book III).

Programming Note

In any of the three non-Precise modes, a *Floating-Point Status and Control Register* instruction can be used to force any exceptions, due to instructions initiated before the *Floating-Point Status and Control Register* instruction, to be recorded in the FPSCR. (This forcing is superfluous for Precise Mode.)

In either of the Imprecise modes, a *Floating-Point Status and Control Register* instruction can be used to force any invocations of the system floating-point enabled exception error handler, due to instructions initiated before the *Floating-Point Status and Control Register* instruction, to occur. (This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.)

The last sentence of the paragraph preceding this Programming Note can apply only in the Imprecise modes, or if the mode has just been changed from Ignore Exceptions Mode to some other mode. (It always applies in the latter case.)

In order to obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines.

- If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used with all FPSCR exception enable bits set to 0.
- If the IEEE default results are not acceptable to the application, Imprecise Nonrecoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with FPSCR exception enable bits set to 1 for those exceptions for which the system floating-point enabled exception error handler is to be invoked.
- Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to 1.
- Precise Mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

4.4.1 Invalid Operation Exception

4.4.1.1 Definition

An Invalid Operation Exception occurs when an operand is invalid for the specified operation. The invalid operations are:

- Any floating-point operation on a Signaling NaN (SNaN)
- For add or subtract operations, magnitude subtraction of infinities ($\infty - \infty$)
- Division of infinity by infinity ($\infty \div \infty$)
- Division of zero by zero ($0 \div 0$)
- Multiplication of infinity by zero ($\infty \times 0$)
- Ordered comparison involving a NaN (Invalid Compare)
- Square root or reciprocal square root of a negative (and nonzero) number (Invalid Square Root)
- Integer convert involving a number too large in magnitude to be represented in the target format, or involving an infinity or a NaN (Invalid Integer Convert)

An Invalid Operation Exception also occurs when an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction is executed that sets $FPSCR_{VXSOFT}$ to 1 (Software-Defined Condition).

4.4.1.2 Action

The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the FPSCR.

When Invalid Operation Exception is enabled ($FPSCR_{VE}$=1) and an Invalid Operation Exception occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set
   - $FPSCR_{VXSNAN}$ (if SNaN)
   - $FPSCR_{VXISI}$ (if $\infty - \infty$)
   - $FPSCR_{VXIDI}$ (if $\infty \div \infty$)
   - $FPSCR_{VXZDZ}$ (if $0 \div 0$)
   - $FPSCR_{VXIMZ}$ (if $\infty \times 0$)
   - $FPSCR_{VXVC}$ (if invalid comp)
   - $FPSCR_{VXSOFT}$ (if sfw-def cond)
   - $FPSCR_{VXSQRT}$ (if invalid sqrt)
   - $FPSCR_{VXCVI}$ (if invalid int cvrt)
2. If the operation is an arithmetic, *Floating Round to Single-Precision*, *Floating Round to Integer,* or convert to integer operation,
   - the target FPR is unchanged
   - $FPSCR_{FR\ FI}$ are set to zero
   - $FPSCR_{FPRF}$ is unchanged
3. If the operation is a compare,
   - $FPSCR_{FR\ FI\ C}$ are unchanged
   - $FPSCR_{FPCC}$ is set to reflect unordered
4. If an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction is executed that sets $FPSCR_{VXSOFT}$ to 1,
   - The FPSCR is set as specified in the instruction description.

When Invalid Operation Exception is disabled ($FPSCR_{VE}$=0) and an Invalid Operation Exception occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set
   - $FPSCR_{VXSNAN}$ (if SNaN)
   - $FPSCR_{VXISI}$ (if $\infty - \infty$)
   - $FPSCR_{VXIDI}$ (if $\infty \div \infty$)
   - $FPSCR_{VXZDZ}$ (if $0 \div 0$)
   - $FPSCR_{VXIMZ}$ (if $\infty \times 0$)
   - $FPSCR_{VXVC}$ (if invalid comp)
   - $FPSCR_{VXSOFT}$ (if sfw-def cond)
   - $FPSCR_{VXSQRT}$ (if invalid sqrt)
   - $FPSCR_{VXCVI}$ (if invalid int cvrt)
2. If the operation is an arithmetic or *Floating Round to Single-Precision* operation,
   - the target FPR is set to a Quiet NaN
   - $FPSCR_{FR\ FI}$ are set to zero
   - $FPSCR_{FPRF}$ is set to indicate the class of the result (Quiet NaN)
3. If the operation is a convert to 64-bit integer operation,
   - the target FPR is set as follows:
     - FRT is set to the most positive 64-bit integer if the operand in FRB is a positive number or + ∞, and to the most negative 64-bit integer if the operand in FRB is a negative number, - ∞, or NaN
   - $FPSCR_{FR\ FI}$ are set to zero
   - $FPSCR_{FPRF}$ is undefined
4. If the operation is a convert to 32-bit integer operation,
   - the target FPR is set as follows:
     - $FRT_{0:31} \leftarrow$ undefined
     - $FRT_{32:63}$ are set to the most positive 32-bit integer if the operand in FRB is a positive number or +infinity, and to the most negative 32-bit integer if the operand in FRB is a negative number, -infinity, or NaN
   - $FPSCR_{FR\ FI}$ are set to zero
   - $FPSCR_{FPRF}$ is undefined
5. If the operation is a compare,
   - $FPSCR_{FR\ FI\ C}$ are unchanged
   - $FPSCR_{FPCC}$ is set to reflect unordered
6. If an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction is executed that sets $FPSCR_{VXSOFT}$ to 1,
   - The FPSCR is set as specified in the instruction description.

4.4.2 Zero Divide Exception

4.4.2.1 Definition

A Zero Divide Exception occurs when a *Divide* instruction is executed with a zero divisor value and a finite nonzero dividend value. It also occurs when a *Reciprocal Estimate* instruction (***fre***[***s***] or ***frsqrte***[***s***]) is executed with an operand value of zero.

4.4.2.2 Action

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

When Zero Divide Exception is enabled ($FPSCR_{ZE}$=1) and a Zero Divide Exception occurs, the following actions are taken:

1. Zero Divide Exception is set
   $FPSCR_{ZX} \leftarrow 1$
2. The target FPR is unchanged
3. $FPSCR_{FR\ FI}$ are set to zero
4. $FPSCR_{FPRF}$ is unchanged

When Zero Divide Exception is disabled ($FPSCR_{ZE}$=0) and a Zero Divide Exception occurs, the following actions are taken:

1. Zero Divide Exception is set
   $FPSCR_{ZX} \leftarrow 1$
2. The target FPR is set to ± Infinity, where the sign is determined by the XOR of the signs of the operands
3. $FPSCR_{FR\ FI}$ are set to zero
4. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Infinity)

4.4.3 Overflow Exception

4.4.3.1 Definition

An Overflow Exception occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.

4.4.3.2 Action

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.

When Overflow Exception is enabled ($FPSCR_{OE}$=1) and an Overflow Exception occurs, the following actions are taken:

1. Overflow Exception is set
   $FPSCR_{OX} \leftarrow 1$
2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by subtracting 1536
3. For single-precision arithmetic instructions and the *Floating Round to Single-Precision* instruction, the exponent of the normalized intermediate result is adjusted by subtracting 192
4. The adjusted rounded result is placed into the target FPR
5. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Normal Number)

When Overflow Exception is disabled ($FPSCR_{OE}$=0) and an Overflow Exception occurs, the following actions are taken:

1. Overflow Exception is set
   $FPSCR_{OX} \leftarrow 1$
2. Inexact Exception is set
   $FPSCR_{XX} \leftarrow 1$
3. The result is determined by the rounding mode ($FPSCR_{RN}$) and the sign of the intermediate result as follows:
   - Round to Nearest
     Store ± Infinity, where the sign is the sign of the intermediate result
   - Round toward Zero
     Store the format's largest finite number with the sign of the intermediate result
   - Round toward + Infinity
     For negative overflow, store the format's most negative finite number; for positive overflow, store +Infinity
   - Round toward - Infinity
     For negative overflow, store -Infinity; for positive overflow, store the format's largest finite number
4. The result is placed into the target FPR
5. $FPSCR_{FR}$ is undefined
6. $FPSCR_{FI}$ is set to 1
7. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Infinity or ± Normal Number)

4.4.4 Underflow Exception

4.4.4.1 Definition

Underflow Exception is defined separately for the enabled and disabled states:

- Enabled:
  Underflow occurs when the intermediate result is "Tiny".
- Disabled:
  Underflow occurs when the intermediate result is "Tiny" and there is "Loss of Accuracy".

A "Tiny" result is detected before rounding, when a non-zero intermediate result computed as though both the precision and the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is "Tiny" and Underflow Exception is disabled ($FPSCR_{UE}$=0) then the intermediate result is denormalized (see Section 4.3.4, "Normalization and Denormalization" on page 129) and rounded (see Section 4.3.6, "Rounding" on page 131) before being placed into the target FPR.

"Loss of Accuracy" is detected when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

4.4.4.2 Action

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

When Underflow Exception is enabled ($FPSCR_{UE}$=1) and an Underflow Exception occurs, the following actions are taken:

1. Underflow Exception is set
   $FPSCR_{UX} \leftarrow 1$
2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by adding 1536
3. For single-precision arithmetic instructions and the *Floating Round to Single-Precision* instruction, the exponent of the normalized intermediate result is adjusted by adding 192
4. The adjusted rounded result is placed into the target FPR
5. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Normalized Number)

Programming Note

The FR and FI bits are provided to allow the system floating-point enabled exception error handler, when invoked because of an Underflow Exception, to simulate a "trap disabled" environment. That is, the FR and FI bits allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized.

When Underflow Exception is disabled ($FPSCR_{UE}$=0) and an Underflow Exception occurs, the following actions are taken:

1. Underflow Exception is set
   $FPSCR_{UX} \leftarrow 1$
2. The rounded result is placed into the target FPR
3. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Normalized Number, ± Denormalized Number, or ± Zero)

4.4.5 Inexact Exception

4.4.5.1 Definition

An Inexact Exception occurs when one of two conditions occur during rounding:

1. The rounded result differs from the intermediate result assuming both the precision and the exponent range of the intermediate result to be unbounded. In this case the result is said to be inexact. (If the rounding causes an enabled Overflow Exception or an enabled Underflow Exception, an Inexact Exception also occurs only if the significands of the rounded result and the intermediate result differ.)
2. The rounded result overflows and Overflow Exception is disabled.

4.4.5.2 Action

The action to be taken does not depend on the setting of the Inexact Exception Enable bit of the FPSCR.

When an Inexact Exception occurs, the following actions are taken:

1. Inexact Exception is set
   $FPSCR_{XX} \leftarrow 1$
2. The rounded or overflowed result is placed into the target FPR
3. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result

Programming Note

In some implementations, enabling Inexact Exceptions may degrade performance more than does enabling other types of floating-point exception.

4.5 Floating-Point Execution Models

All implementations of this architecture must provide the equivalent of the following execution models to ensure that identical results are obtained.

Special rules are provided in the definition of the computational instructions for the infinities, denormalized numbers and NaNs. The material in the remainder of this section applies to instructions that have numeric operands and a numeric result (i.e., operands and result that are not infinities or NaNs), and that cause no exceptions. See Section 4.3.2 and Section 4.4 for the cases not covered here.

Although the double format specifies an 11-bit exponent, exponent arithmetic makes use of two additional bits to avoid potential transient overflow conditions. One extra bit is required when denormalized double-precision numbers are prenormalized. The second bit is required to permit the computation of the adjusted exponent value in the following cases when the corresponding exception enable bit is 1:

- Underflow during multiplication using a denormalized operand.
- Overflow during division using a denormalized divisor.

The IEEE standard includes 32-bit and 64-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision floating-point operations to have either (or both) single-precision or double-precision operands, but states that single-precision floating-point operations should not accept double-precision operands. The Power ISA follows these guidelines; double-precision arithmetic instructions can have operands of either or both precisions, while single-precision arithmetic instructions require all operands to be single-precision. Double-precision arithmetic instructions and ***fcfid*** produce double-precision values, while single-precision arithmetic instructions produce single-precision values.

For arithmetic instructions, conversions from double-precision to single-precision must be done explicitly by software, while conversions from single-precision to double-precision are done implicitly.

4.5.1 Execution Model for IEEE Operations

The following description uses 64-bit arithmetic as an example. 32-bit arithmetic is similar except that the FRACTION is a 23-bit field, and the single-precision Guard, Round, and Sticky bits (described in this section) are logically adjacent to the 23-bit FRACTION field.

IEEE-conforming significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:55 comprise the significand of the intermediate result.

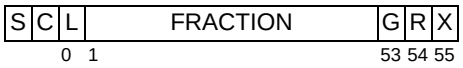


Figure 53. IEEE 64-bit execution model

The S bit is the sign bit.

The C bit is the carry bit, which captures the carry out of the significand.

The L bit is the leading unit bit of the significand, which receives the implicit bit from the operand.

The FRACTION is a 52-bit field that accepts the fraction of the operand.

The Guard (G), Round (R), and Sticky (X) bits are extensions to the low-order bits of the accumulator. The G and R bits are required for postnormalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits that may appear to the low-order side of the R bit, due either to shifting the accumulator right or to other generation of low-order result bits. The G and R bits participate in the left shifts with zeros being shifted into the R bit. Figure 54 shows the significance of the G, R, and X bits with respect to the intermediate result (IR), the representable number next lower in magnitude (NL), and the representable number next higher in magnitude (NH).

G R X	Interpretation
0 0 0	IR is exact
0 0 1	IR closer to NL
0 1 0	
0 1 1	
1 0 0	IR midway between NL and NH
1 0 1	IR closer to NH
1 1 0	
1 1 1	

Figure 54. Interpretation of G, R, and X bits

Figure 55 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers relative to the accumulator illustrated in Figure 53.

Format	Guard	Round	Sticky
Double	G bit	R bit	X bit
Single	24	25	OR of 26:52, G, R, X

Figure 55. Location of the Guard, Round, and Sticky bits in the IEEE execution model

The significand of the intermediate result is prepared for rounding by shifting its contents right, if required, until the least significant bit to be retained is in the low-order bit position of the fraction. Four user-selectable rounding modes are provided through $FPSCR_{RN}$ as described in Section 4.3.6, “Rounding” on page 131. Using Z1 and Z2 as defined on page 131, the rules for rounding in each mode are as follows.

- ***Round to Nearest***

  Guard bit = 0
  The result is truncated. (Result exact (GRX=000) or closest to next lower value in magnitude (GRX=001, 010, or 011))

  Guard bit = 1
  Depends on Round and Sticky bits:

  Case a
  If the Round or Sticky bit is 1 (inclusive), the result is incremented. (Result closest to next higher value in magnitude (GRX=101, 110, or 111))

  Case b
  If the Round and Sticky bits are 0 (result midway between closest representable values), then if the low-order bit of the result is 1 the result is incremented. Otherwise (the low-order bit of the result is 0) the result is truncated (this is the case of a tie rounded to even).

- ***Round toward Zero***
  Choose the smaller in magnitude of Z1 or Z2. If the Guard, Round, or Sticky bit is nonzero, the result is inexact.

- ***Round toward*** + ***Infinity***
  Choose Z1.

- ***Round toward*** - ***Infinity***
  Choose Z2.

If rounding results in a carry into C, the significand is shifted right one position and the exponent is incremented by one. This yields an inexact result, and possibly also exponent overflow. If any of the Guard, Round, or Sticky bits is nonzero, then the result is also inexact. Fraction bits are stored to the target FPR. For *Floating Round to Integer*, *Floating Round to Single-Precision*, and single-precision arithmetic instructions, low-order zeros must be appended as appropriate to fill out the double-precision fraction.

4.5.2 Execution Model for Multiply-Add Type Instructions

The Power ISA provides a special form of instruction that performs up to three operations in one instruction (a multiplication, an addition, and a negation). With this added capability comes the special ability to produce a more exact intermediate result as input to the rounder. 32-bit arithmetic is similar except that the FRACTION field is smaller.

Multiply-add significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:106 comprise the significand of the intermediate result.

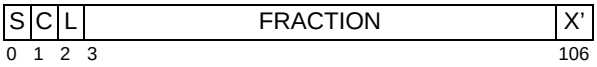


Figure 56. Multiply-add 64-bit execution model

The first part of the operation is a multiplication. The multiplication has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), then the significand is shifted right one position, shifting the L bit (leading unit bit) into the most significant bit of the FRACTION and shifting the C bit (carry out) into the L bit. All 106 bits (L bit, the FRACTION) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount that is added to that exponent to make it equal to the other input's exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the X' bit. The add operation also produces a result conforming to the above model with the X' bit taking part in the add operation.

The result of the addition is then normalized, with all bits of the addition result, except the X' bit, participating in the shift. The normalized result serves as the intermediate result that is input to the rounder.

For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. Figure 57 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers in the multiply-add execution model.

Format	Guard	Round	Sticky
Double	53	54	OR of 55:105, X'
Single	24	25	OR of 26:105, X'

Figure 57. Location of the Guard, Round, and Sticky bits in the multiply-add execution model

The rules for rounding the intermediate result are the same as those given in Section 4.5.1.

If the instruction is *Floating Negative Multiply-Add* or *Floating Negative Multiply-Subtract*, the final result is negated.

4.6 Floating-Point Facility Instructions

4.6.1 Floating-Point Storage Access Instructions

The *Storage Access* instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.11.3, “Effective Address Calculation” on page 27.

Programming Note

The ***la*** extended mnemonic permits computing an effective address as a *Load* or *Store* instruction would, but loads the address itself into a GPR rather than loading the value that is in storage at that address. This extended mnemonic is described in Section C.10, “Miscellaneous Mnemonics” on page 802.

4.6.1.1 Storage Access Exceptions

Storage accesses will cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (*Store* only), or if the program attempts to access storage that is unavailable.

4.6.2 Floating-Point Load Instructions

There are three basic forms of load instruction: single-precision, double-precision, and integer. The integer form is provided by the *Load Floating-Point as Integer Word Algebraic* instruction, described on page 143. Because the FPRs support only floating-point double format, single-precision *Load Floating-Point* instructions convert single-precision data to double format prior to loading the operand into the target FPR. The conversion and loading steps are as follows.

Let $WORD_{0:31}$ be the floating-point single-precision operand accessed from storage.

Normalized Operand

if $WORD_{1:8} > 0$ and $WORD_{1:8} < 255$ then
  $FRT_{0:1} \leftarrow WORD_{0:1}$
  $FRT_2 \leftarrow \neg WORD_1$
  $FRT_3 \leftarrow \neg WORD_1$
  $FRT_4 \leftarrow \neg WORD_1$
  $FRT_{5:63} \leftarrow WORD_{2:31} \,||\, {}^{29}0$

Denormalized Operand

  if $WORD_{1:8} = 0$ and $WORD_{9:31} \neq 0$ then
  sign $\leftarrow WORD_0$
  exp $\leftarrow -126$
  $frac_{0:52} \leftarrow 0b0 \,||\, WORD_{9:31} \,||\, {}^{29}0$
  normalize the operand
    do while $frac_0 = 0$
      $frac_{0:52} \leftarrow frac_{1:52} \,||\, 0b0$
      exp $\leftarrow$ exp - 1
  $FRT_0 \leftarrow$ sign
  $FRT_{1:11} \leftarrow$ exp + 1023
  $FRT_{12:63} \leftarrow frac_{1:52}$

Zero / Infinity / NaN

if $WORD_{1:8} = 255$ or $WORD_{1:31} = 0$ then
  $FRT_{0:1} \leftarrow WORD_{0:1}$
  $FRT_2 \leftarrow WORD_1$
  $FRT_3 \leftarrow WORD_1$
  $FRT_4 \leftarrow WORD_1$
  $FRT_{5:63} \leftarrow WORD_{2:31} \,||\, {}^{29}0$

For double-precision *Load Floating-Point* instructions and for the *Load Floating-Point as Integer Word Algebraic* instruction no conversion is required, as the data from storage are copied directly into the FPR.

Many of the *Load Floating-Point* instructions have an “update” form, in which register RA is updated with the effective address. For these forms, if RA≠0, the effective address is placed into register RA and the storage element (word or doubleword) addressed by EA is loaded into FRT.

Note: Recall that RA and RB denote General Purpose Registers, while FRT denotes a Floating-Point Register.

Load Floating-Point Single D-form

lfs FRT,D(RA)

48	FRT	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
FRT ← DOUBLE(MEM(EA, 4))
```

Let the effective address (EA) be the sum (RA|0)+D.

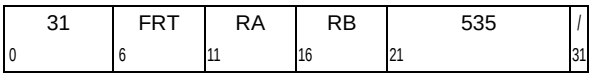
The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 140) and placed into register FRT.

Special Registers Altered:

None

Load Floating-Point Single Indexed X-form

lfsx FRT,RA,RB




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
FRT ← DOUBLE(MEM(EA, 4))
```

Let the effective address (EA) be the sum (RA|0)+(RB).

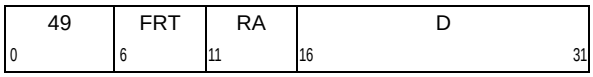
The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 140) and placed into register FRT.

Special Registers Altered:

None

Load Floating-Point Single with Update D-form

lfsu FRT,D(RA)




```
EA ← (RA) + EXTS(D)
FRT ← DOUBLE(MEM(EA, 4))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D.

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 140) and placed into register FRT.

EA is placed into register RA.

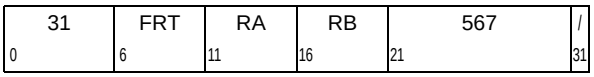
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Load Floating-Point Single with Update Indexed X-form

lfsux FRT,RA,RB




```
EA ← (RA) + (RB)
FRT ← DOUBLE(MEM(EA, 4))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB).

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 140) and placed into register FRT.

EA is placed into register RA.

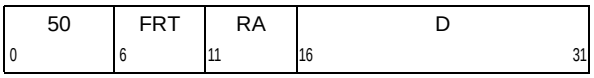
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Load Floating-Point Double D-form

lfd FRT,D(RA)




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
FRT ← MEM(EA, 8)
```

Let the effective address (EA) be the sum (RA|0)+D.

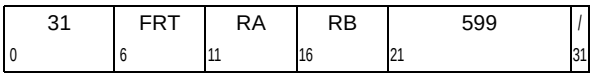
The doubleword in storage addressed by EA is loaded into register FRT.

Special Registers Altered:

None

Load Floating-Point Double Indexed X-form

lfdx FRT,RA,RB




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
FRT ← MEM(EA, 8)
```

Let the effective address (EA) be the sum (RA|0)+(RB).

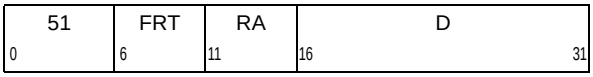
The doubleword in storage addressed by EA is loaded into register FRT.

Special Registers Altered:

None

Load Floating-Point Double with Update D-form

lfdu FRT,D(RA)




```
EA ← (RA) + EXTS(D)
FRT ← MEM(EA, 8)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D.

The doubleword in storage addressed by EA is loaded into register FRT.

EA is placed into register RA.

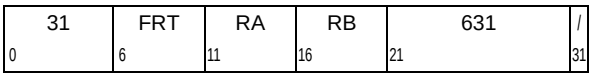
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Load Floating-Point Double with Update Indexed X-form

lfdux FRT,RA,RB




```
EA ← (RA) + (RB)
FRT ← MEM(EA, 8)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB).

The doubleword in storage addressed by EA is loaded into register FRT.

EA is placed into register RA.

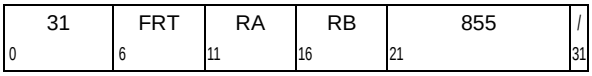
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Load Floating-Point as Integer Word Algebraic Indexed X-form

lfiwax FRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
FRT ← EXTS(MEM(EA, 4))
```

Let the effective address (EA) be the sum (RA|0)+(RB).

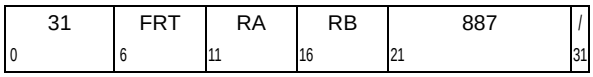
The word in storage addressed by EA is loaded into $FRT_{32:63}$. $FRT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

Special Registers Altered:

None

Load Floating-Point as Integer Word and Zero Indexed X-form

lfiwzx FRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
FRT ← 32 0 || MEM(EA, 4)
```

Let the effective address (EA) be the sum (RA|0)+(RB).

The word in storage addressed by EA is loaded into $FRT_{32:63}$. $FRT_{0:31}$ are set to 0.

Special Registers Altered:

None

4.6.3 Floating-Point Store Instructions

There are three basic forms of store instruction: single-precision, double-precision, and integer. The integer form is provided by the *Store Floating-Point as Integer Word* instruction, described on page 147. Because the FPRs support only floating-point double format for floating-point data, single-precision *Store Floating-Point* instructions convert double-precision data to single format prior to storing the operand into storage. The conversion steps are as follows.

Let $WORD_{0:31}$ be the word in storage written to.

No Denormalization Required (includes Zero / Infinity / NaN)

if $FRS_{1:11} > 896$ or $FRS_{1:63} = 0$ then
    $WORD_{0:1} \leftarrow FRS_{0:1}$
    $WORD_{2:31} \leftarrow FRS_{5:34}$

Denormalization Required

if $874 \leq FRS_{1:11} \leq 896$ then
    sign $\leftarrow FRS_0$
    exp $\leftarrow FRS_{1:11}$ - 1023
    $frac_{0:52} \leftarrow 0b1 \,||\, FRS_{12:63}$
    denormalize operand
        do while exp < -126
            $frac_{0:52} \leftarrow 0b0 \,||\, frac_{0:51}$
            exp $\leftarrow$ exp + 1
    $WORD_0 \leftarrow$ sign
    $WORD_{1:8} \leftarrow 0x00$
    $WORD_{9:31} \leftarrow frac_{1:23}$
else WORD $\leftarrow$ undefined

Notice that if the value to be stored by a single-precision *Store Floating-Point* instruction is larger in magnitude than the maximum number representable in single format, the first case above (No Denormalization Required) applies. The result stored in WORD is then a well-defined value, but is not numerically equal to the value in the source register (i.e., the result of a single-precision *Load Floating-Point* from WORD will not compare equal to the contents of the original source register).

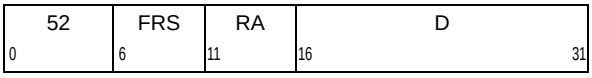
For double-precision *Store Floating-Point* instructions and for the *Store Floating-Point as Integer Word* instruction no conversion is required, as the data from the FPR are copied directly into storage.

Many of the *Store Floating-Point* instructions have an “update” form, in which register RA is updated with the effective address. For these forms, if RA≠0, the effective address is placed into register RA.

Note: Recall that RA and RB denote General Purpose Registers, while FRS denotes a Floating-Point Register.

Store Floating-Point Single D-form

stfs FRS,D(RA)




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 4) ← SINGLE((FRS))
```

Let the effective address (EA) be the sum (RA|0)+D.

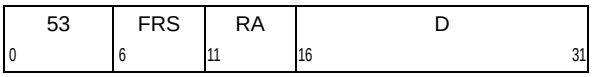
The contents of register FRS are converted to single format (see page 144) and stored into the word in storage addressed by EA.

Special Registers Altered:

None

Store Floating-Point Single with Update D-form

stfsu FRS,D(RA)




```
EA ← (RA) + EXTS(D)
MEM(EA, 4) ← SINGLE((FRS))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D.

The contents of register FRS are converted to single format (see page 144) and stored into the word in storage addressed by EA.

EA is placed into register RA.

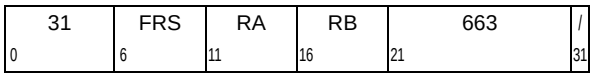
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point Single Indexed X-form

stfsx FRS,RA,RB




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← SINGLE((FRS))
```

Let the effective address (EA) be the sum (RA|0)+(RB).

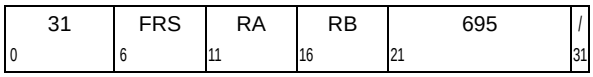
The contents of register FRS are converted to single format (see page 144) and stored into the word in storage addressed by EA.

Special Registers Altered:

None

Store Floating-Point Single with Update Indexed X-form

stfsux FRS,RA,RB




```
EA ← (RA) + (RB)
MEM(EA, 4) ← SINGLE((FRS))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB).

The contents of register FRS are converted to single format (see page 144) and stored into the word in storage addressed by EA.

EA is placed into register RA.

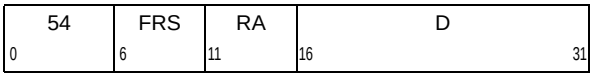
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point Double D-form

stfd FRS,D(RA)




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + EXTS(D)
MEM(EA, 8) ← (FRS)
```

Let the effective address (EA) be the sum (RA|0)+D.

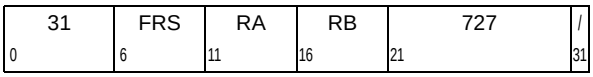
The contents of register FRS are stored into the doubleword in storage addressed by EA.

Special Registers Altered:

None

Store Floating-Point Double Indexed X-form

stfdx FRS,RA,RB




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← (FRS)
```

Let the effective address (EA) be the sum (RA|0)+(RB).

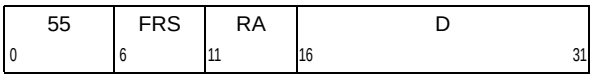
The contents of register FRS are stored into the doubleword in storage addressed by EA.

Special Registers Altered:

None

Store Floating-Point Double with Update D-form

stfdu FRS,D(RA)




```
EA ← (RA) + EXTS(D)
MEM(EA, 8) ← (FRS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+D.

The contents of register FRS are stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

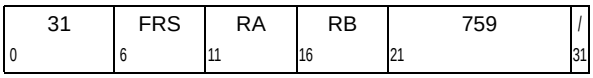
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point Double with Update Indexed X-form

stfdux FRS,RA,RB




```
EA ← (RA) + (RB)
MEM(EA, 8) ← (FRS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+(RB).

The contents of register FRS are stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

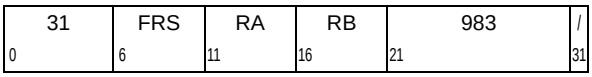
If RA=0, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point as Integer Word Indexed X-form

stfiwx FRS,RA,RB




```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (FRS)32:63
```

Let the effective address (EA) be the sum (RA|0)+(RB).

$(FRS)_{32:63}$ are stored, without conversion, into the word in storage addressed by EA.

If the contents of register FRS were produced, either directly or indirectly, by a *Load Floating-Point Single* instruction, a single-precision *Arithmetic* instruction, or ***frsp***, then the value stored is undefined. (The contents of register FRS are produced directly by such an instruction if FRS is the target register for the instruction. The contents of register FRS are produced indirectly by such an instruction if FRS is the final target register of a sequence of one or more *Floating-Point Move* instructions, with the input to the sequence having been produced directly by such an instruction.)

Special Registers Altered:

None

4.6.4 Floating-Point Load and Store Double Pair Instructions [Phased-Out]

For ***lfdp[x]***, the doubleword-pair in storage addressed by EA is loaded into an even-odd pair of FPRs with the even-numbered FPR being loaded with the leftmost doubleword from storage and the odd-numbered FPR being loaded with the rightmost doubleword.

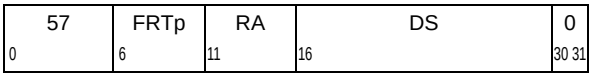
For ***stfdp[x]***, the content of an even-odd pair of FPRs is stored into the doubleword-pair in storage addressed by EA, with the even-numbered FPR being stored into the leftmost doubleword in storage and the odd-numbered FPR being stored into the rightmost doubleword.

Programming Note

The instructions described in this section should not be used to access an operand in DFP Extended format when the processor is in Little-Endian mode.

Load Floating-Point Double Pair DS-form

lfdp FRTp,DS(RA)




```
if RA = 0 then b ← 0
else          b ←(RA)
EA ← b + EXTS(DS||0b00)
FRTp_even ← MEM(EA,8)
FRTp_odd  ← MEM(EA+8, 8)
```

Let the effective address (EA) be the sum (RA|0) + (DS||0b00).

The doubleword in storage addressed by EA is placed into the even-numbered register of FRTp.

The doubleword in storage addressed by EA+8 is placed into the odd-numbered register of FRTp.

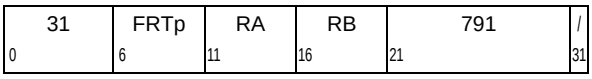
If FRTp is odd, the instruction form is invalid.

Special Registers Altered:

None

Load Floating-Point Double Pair Indexed X-form

lfdpx FRTp,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
FRTp_even ← MEM(EA,8)
FRTp_odd  ← MEM(EA+8, 8)
```

Let the effective address (EA) be the sum (RA|0) + (RB).

The doubleword in storage addressed by EA is placed into the even-numbered register of FRTp.

The doubleword in storage addressed by EA+8 is placed into the odd-numbered register of FRTp.

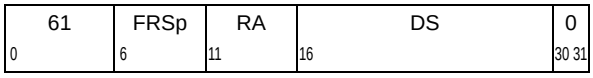
If FRTp is odd, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point Double Pair DS-form

stfdp FRSp,DS(RA)




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(DS||0b00)
MEM(EA, 8)   ← FRSp_even
MEM(EA+8, 8) ← FRSp_odd
```

Let the effective address (EA) be the sum (RA|0) + (DS||0b00).

The contents of the even-numbered register of FRSp are stored into the doubleword in storage addressed by EA.

The contents of the odd-numbered register of FRSp are stored into the doubleword in storage addressed by EA+8.

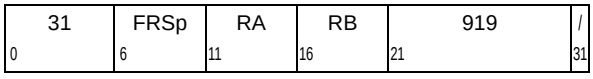
If FRSp is odd, the instruction form is invalid.

Special Registers Altered:

None

Store Floating-Point Double Pair Indexed X-form

stfdpx FRSp,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
MEM(EA, 8)   ← FRSp_even
MEM(EA+8, 8) ← FRSp_odd
```

Let the effective address (EA) be the sum (RA|0) + (DS||0b00).

The contents of the even-numbered register of FRSp are stored into the doubleword in storage addressed by EA.

The contents of the odd-numbered register of FRSp are stored into the doubleword in storage addressed by EA+8.

If FRSp is odd, the instruction form is invalid.

Special Registers Altered:

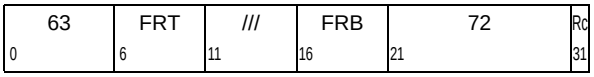
None

4.6.5 Floating-Point Move Instructions

These instructions copy data from one floating-point register to another, altering the sign bit (bit 0) as described below for ***fneg***, ***fabs***, ***fnabs***, and ***fcpsgn***. These instructions treat NaNs just like any other kind of value (e.g., the sign bit of a NaN may be altered by ***fneg***, ***fabs***, ***fnabs***, and ***fcpsgn***). These instructions do not alter the FPSCR.

Floating Move Register X-form

fmr FRT,FRB (Rc=0)
fmr. FRT,FRB (Rc=1)

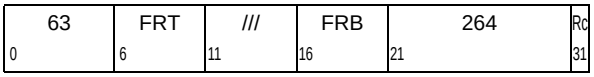


The contents of register FRB are placed into register FRT.

Special Registers Altered:
CR1 (if Rc=1)

Floating Absolute Value X-form

fabs FRT,FRB (Rc=0)
fabs. FRT,FRB (Rc=1)

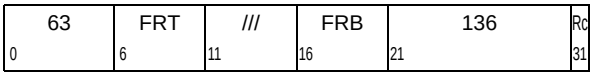


The contents of register FRB with bit 0 set to zero are placed into register FRT.

Special Registers Altered:
CR1 (if Rc=1)

Floating Negative Absolute Value X-form

fnabs FRT,FRB (Rc=0)
fnabs. FRT,FRB (Rc=1)

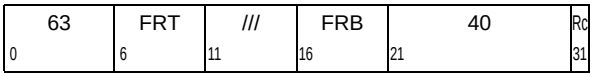


The contents of register FRB with bit 0 set to one are placed into register FRT.

Special Registers Altered:
CR1 (if Rc=1)

Floating Negate X-form

fneg FRT,FRB (Rc=0)
fneg. FRT,FRB (Rc=1)

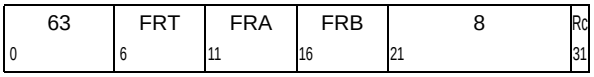


The contents of register FRB with bit 0 inverted are placed into register FRT.

Special Registers Altered:
CR1 (if Rc=1)

Floating Copy Sign X-form

fcpsgn FRT, FRA, FRB (Rc=0)
fcpsgn. FRT, FRA, FRB (Rc=1)

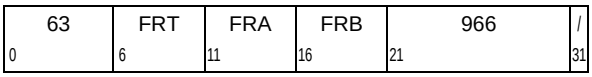


The contents of register FRB with bit 0 set to the value of bit 0 of register FRA are placed into register FRT.

Special Registers Altered:
CR1 (if Rc=1)

Floating Merge Even Word X-form

fmrgew FRT,FRA,FRB




```
if MSR.FP=0 then FP_Unavailable()
FPR[FRT].word[0] ← FPR[FRA].word[0]
FPR[FRT].word[1] ← FPR[FRB].word[0]
```

The contents of word element 0 of FPR[FRA] are placed into word element 0 of FPR[FRT].

The contents of word element 0 of FPR[FRB] are placed into word element 1 of FPR[FRT].

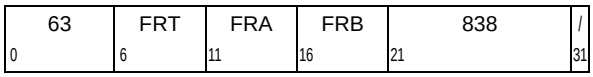
fmrgew is treated as a *Floating-Point* instruction in terms of resource availability.

Special Registers Altered

None

Floating Merge Odd Word X-form

fmrgow FRT,FRA,FRB




```
if MSR.FP=0 then FP_Unavailable()
FPR[FRT].word[0] ← FPR[FRA].word[1]
FPR[FRT].word[1] ← FPR[FRB].word[1]
```

The contents of word element 1 of FPR[FRA] are placed into word element 0 of FPR[FRT].

The contents of word element 1 of FPR[FRB] are placed into word element 1 of FPR[FRT].

fmrgow is treated as a *Floating-Point* instruction in terms of resource availability.

Special Registers Altered

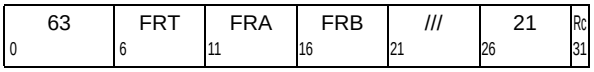
None

4.6.6 Floating-Point Arithmetic Instructions

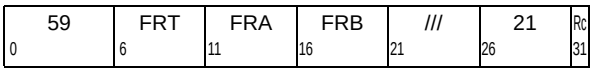
4.6.6.1 Floating-Point Elementary Arithmetic Instructions

Floating Add [Single] A-form

fadd FRT,FRA,FRB (Rc=0)
fadd. FRT,FRA,FRB (Rc=1)



fadds FRT,FRA,FRB (Rc=0)
fadds. FRT,FRA,FRB (Rc=1)



The floating-point operand in register FRA is added to the floating-point operand in register FRB.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one.

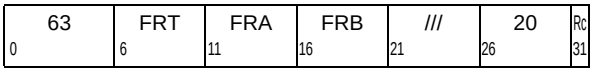
$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

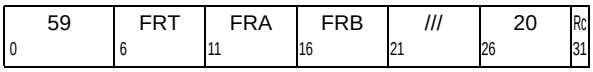
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1 (if Rc=1)

Floating Subtract [Single] A-form

fsub FRT,FRA,FRB (Rc=0)
fsub. FRT,FRA,FRB (Rc=1)



fsubs FRT,FRA,FRB (Rc=0)
fsubs. FRT,FRA,FRB (Rc=1)



The floating-point operand in register FRB is subtracted from the floating-point operand in register FRA.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

The execution of the Floating Subtract instruction is identical to that of Floating Add, except that the contents of FRB participate in the operation with the sign bit (bit 0) inverted.

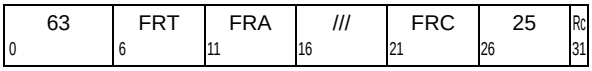
$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

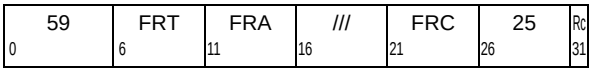
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1 (if Rc=1)

Floating Multiply [Single] A-form

fmul FRT,FRA,FRC (Rc=0)
fmul. FRT,FRA,FRC (Rc=1)



fmuls FRT,FRA,FRC (Rc=0)
fmuls. FRT,FRA,FRC (Rc=1)



The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

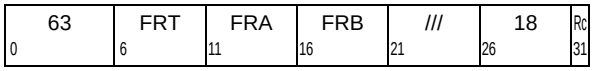
$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

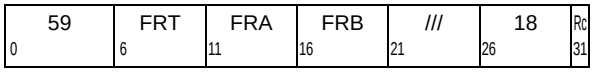
FPRF FR FI
FX OX UX XX
VXSNAN VXIMZ
CR1 (if Rc=1)

Floating Divide [Single] A-form

fdiv FRT,FRA,FRB (Rc=0)
fdiv. FRT,FRA,FRB (Rc=1)



fdivs FRT,FRA,FRB (Rc=0)
fdivs. FRT,FRA,FRB (Rc=1)



The floating-point operand in register FRA is divided by the floating-point operand in register FRB. The remainder is not supplied as a result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point division is based on exponent subtraction and division of the significands.

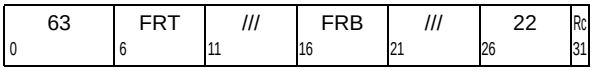
$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1 and Zero Divide Exceptions when $FPSCR_{ZE}$=1.

Special Registers Altered:

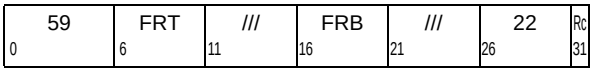
FPRF FR FI
FX OX UX ZX XX
VXSNAN VXIDI VXZDZ
CR1 (if Rc=1)

Floating Square Root [Single] A-form

fsqrt FRT,FRB (Rc=0)
fsqrt. FRT,FRB (Rc=1)



fsqrts FRT,FRB (Rc=0)
fsqrts. FRT,FRB (Rc=1)



The square root of the floating-point operand in register FRB is placed into register FRT.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
-∞	QNaN[1]	VXSQRT
< 0	QNaN[1]	VXSQRT
-0	-0	None
+∞	+∞	None
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

[1] No result if $FPSCR_{VE}$ = 1

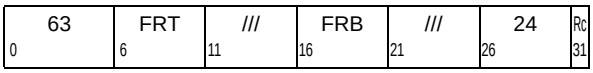
$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

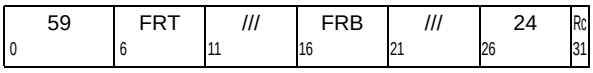
FPRF FR FI FX OX UX XX
VXSNAN VXSQRT
CR1 (if Rc=1)

Floating Reciprocal Estimate [Single] A-form

fre FRT,FRB (Rc=0)
fre. FRT,FRB (Rc=1)



fres FRT,FRB (Rc=0)
fres. FRT,FRB (Rc=1)



An estimate of the reciprocal of the floating-point operand in register FRB is placed into register FRT. Unless the reciprocal would be a zero, an infinity, the result of a trap-disabled Overflow exception, or a QNaN, the estimate is correct to a precision of one part in 256 of the reciprocal of (FRB), i.e.,

$$\text{ABS}\left(\frac{\text{estimate} - 1/x}{1/x}\right) \leq \frac{1}{256}$$

where x is the initial value in FRB.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
-½	-0	None
-0	-½[1]	ZX
+0	+½[1]	ZX
+½	+0	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

[1] No result if $FPSCR_{ZE}$ = 1.
[2] No result if $FPSCR_{VE}$ = 1.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1 and Zero Divide Exceptions when $FPSCR_{ZE}$=1.

The results of executing this instruction may vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

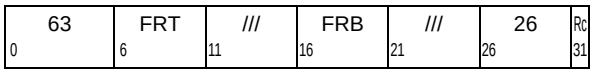
FPRF FR (undefined) FI (undefined)
FX OX UX ZX XX (undefined)
VXSNAN
CR1 (if Rc=1)

Programming Note

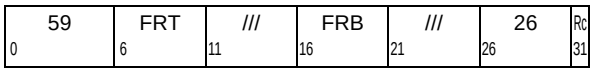
For the *Floating-Point Estimate* instructions, some implementations might implement a precision higher than the minimum architected precision. Thus, a program may take advantage of the higher precision instructions to increase performance by decreasing the iterations needed for software emulation of floating-point instructions. However, there is no guarantee given about the precision which may vary (up or down) between implementations. Only programs targeted at a specific implementation (i.e., the program will not be migrated to another implementation) should take advantage of the higher precision of the instructions. All other programs should rely on the minimum architected precision, which will guarantee the program to run properly across different implementations.

Floating Reciprocal Square Root Estimate [Single] A-form

frsqrte FRT,FRB (Rc=0)
frsqrte. FRT,FRB (Rc=1)



frsqrtes FRT,FRB (Rc=0)
frsqrtes. FRT,FRB (Rc=1)



A estimate of the reciprocal of the square root of the floating-point operand in register FRB is placed into register FRT. The estimate placed into register FRT is correct to a precision of one part in 32 of the reciprocal of the square root of (FRB), i.e.,

$$\text{ABS}\left(\frac{\text{estimate} - 1/(\sqrt{x})}{1/(\sqrt{x})}\right) \leq \frac{1}{32}$$

where x is the initial value in FRB.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
$-\infty$	QNaN[2]	VXSQRT
< 0	QNaN[2]	VXSQRT
-0	$-\infty$[1]	ZX
+0	$+\infty$[1]	ZX
$+\infty$	+0	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

[1] No result if $FPSCR_{ZE}$ = 1.
[2] No result if $FPSCR_{VE}$ = 1.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1 and Zero Divide Exceptions when $FPSCR_{ZE}$=1.

The results of executing this instruction may vary between implementations, and between different executions on the same implementation.

Special Registers Altered:

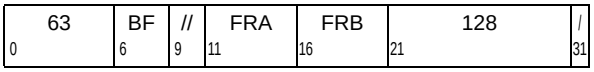
FPRF FR (undefined) FI (undefined)
FX OX UX ZX XX (undefined)
VXSNAN VXSQRT
CR1 (if Rc=1)

Note

See the Notes that appear with ***fre***[***s***].

Floating Test for software Divide X-form

ftdiv BF,FRA,FRB



Let e_a be the unbiased exponent of the double-precision floating-point operand in register FRA.

Let e_b be the unbiased exponent of the double-precision floating-point operand in register FRB.

fe_flag is set to 1 if any of the following conditions occurs.

- The double-precision floating-point operand in register FRA is a NaN or an Infinity.
- The double-precision floating-point operand in register FRB is a Zero, a NaN, or an Infinity.
- e_b is less than or equal to -1022.
- e_b is greater than or equal to 1021.
- The double-precision floating-point operand in register FRA is not a zero and the difference, e_a - e_b, is greater than or equal to 1023.
- The double-precision floating-point operand in register FRA is not a zero and the difference, e_a - e_b, is less than or equal to -1021.
- The double-precision floating-point operand in register FRA is not a zero and e_a is less than or equal to -970

Otherwise fe_flag is set to 0.

fg_flag is set to 1 if either of the following conditions occurs.

- The double-precision floating-point operand in register FRA is an Infinity.
- The double-precision floating-point operand in register FRB is a Zero, an Infinity, or a denormalized value.

Otherwise fg_flag is set to 0.

If the implementation guarantees a relative error of ***fre***[***s***][**.**] of less than or equal to 2^{-14}, then fl_flag is set to 1. Otherwise fl_flag is set to 0.

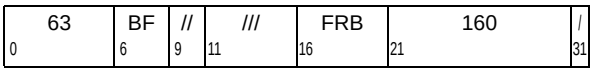
CR field BF is set to the value fl_flag || fg_flag || fe_flag || 0b0.

Special Registers Altered:

CR field BF

Floating Test for software Square Root X-form

ftsqrt BF,FRB



Let e_b be the unbiased exponent of the double-precision floating-point operand in register FRB.

fe_flag is set to 1 if either of the following conditions occurs.

- The double-precision floating-point operand in register FRB is a zero, a NaN, or an infinity, or a negative value.
- e_b is less than or equal to -970.

Otherwise fe_flag is set to 0.

fg_flag is set to 1 if the following condition occurs.

- The double-precision floating-point operand in register FRB is a Zero, an Infinity, or a denormalized value.

Otherwise fg_flag is set to 0.

If the implementation guarantees a relative error of ***frsqrte***[***s***][**.**] of less than or equal to 2^{-14}, then fl_flag is set to 1. Otherwise fl_flag is set to 0.

CR field BF is set to the value fl_flag || fg_flag || fe_flag || 0b0.

Special Registers Altered:

CR field BF

Programming Note

ftdiv and ***ftsqrt*** are provided to accelerate software emulation of divide and square root operations, by performing the requisite special case checking. Software needs only a single branch, on FE=1 (in CR[BF]), to a special case handler. FG and FL may provide further acceleration opportunities.

4.6.6.2 Floating-Point Multiply-Add Instructions

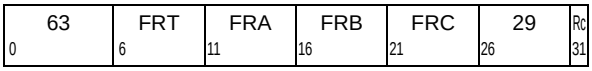
These instructions combine a multiply and an add operation without an intermediate rounding operation. The fraction part of the intermediate product is 106 bits wide (L bit, FRACTION), and all 106 bits take part in the add/subtract portion of the instruction.

Status bits are set as follows.

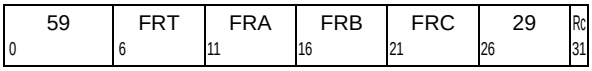
- Overflow, Underflow, and Inexact Exception bits, the FR and FI bits, and the FPRF field are set based on the final result of the operation, and not on the result of the multiplication.
- Invalid Operation Exception bits are set as if the multiplication and the addition were performed using two separate instructions (***fmul*[*s*]**, followed by ***fadd*[*s*]** or ***fsub*[*s*]**). That is, multiplication of infinity by 0 or of anything by an SNaN, and/or addition of an SNaN, cause the corresponding exception bits to be set.

Floating Multiply-Add [Single] A-form

fmadd FRT,FRA,FRC,FRB (Rc=0)
fmadd. FRT,FRA,FRC,FRB (Rc=1)



fmadds FRT,FRA,FRC,FRB (Rc=0)
fmadds. FRT,FRA,FRC,FRB (Rc=1)



The operation

FRT ← [(FRA)×(FRC)] + (FRB)

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

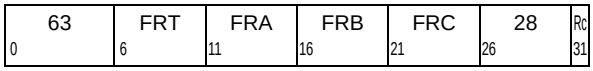
$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

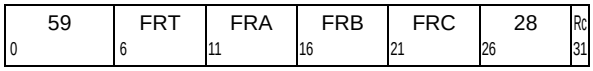
FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

Floating Multiply-Subtract [Single] A-form

fmsub FRT,FRA,FRC,FRB (Rc=0)
fmsub. FRT,FRA,FRC,FRB (Rc=1)



fmsubs FRT,FRA,FRC,FRB (Rc=0)
fmsubs. FRT,FRA,FRC,FRB (Rc=1)



The operation

FRT ← [(FRA)×(FRC)] - (FRB)

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

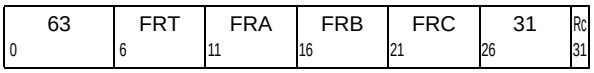
$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

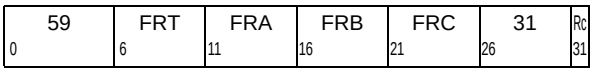
FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

Floating Negative Multiply-Add [Single] A-form

fnmadd FRT,FRA,FRC,FRB (Rc=0)
fnmadd. FRT,FRA,FRC,FRB (Rc=1)



fnmadds FRT,FRA,FRC,FRB (Rc=0)
fnmadds. FRT,FRA,FRC,FRB (Rc=1)



The operation

FRT ← - ([(FRA)×(FRC)] + (FRB))

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the *Floating Multiply-Add* instruction and then negating the result, with the following exceptions.

- QNaNs propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the SNaN.

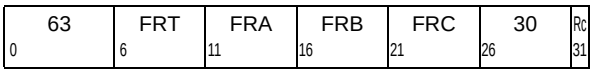
$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

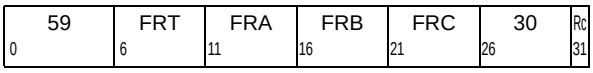
FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

Floating Negative Multiply-Subtract [Single] A-form

fnmsub FRT,FRA,FRC,FRB (Rc=0)
fnmsub. FRT,FRA,FRC,FRB (Rc=1)



fnmsubs FRT,FRA,FRC,FRB (Rc=0)
fnmsubs. FRT,FRA,FRC,FRB (Rc=1)



The operation

FRT ← - ([(FRA)×(FRC)] - (FRB))

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the *Floating Multiply-Subtract* instruction and then negating the result, with the following exceptions.

- QNaNs propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the SNaN.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

4.6.7 Floating-Point Rounding and Conversion Instructions

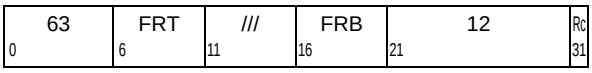
Programming Note

Examples of uses of these instructions to perform various conversions can be found in Section E.2, “Floating-Point Conversions” on page 642.

4.6.7.1 Floating-Point Rounding Instruction

Floating Round to Single-Precision X-form

frsp FRT,FRB (Rc=0)
frsp. FRT,FRB (Rc=1)



The floating-point operand in register FRB is rounded to single-precision, using the rounding mode specified by RN, and placed into register FRT.

The rounding is described fully in Section A.1, “Floating-Point Round to Single-Precision Model” on page 775.

FPRF is set to the class and sign of the result, except for Invalid Operation Exceptions when VE=1.

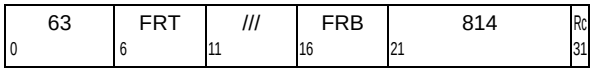
Special Registers Altered:

FPRF FR FI
FX OX UX XX VXSNAN
CR1 (if Rc=1)

4.6.7.2 Floating-Point Convert To/From Integer Instructions

Floating Convert with round Double-Precision To Signed Doubleword format X-form

fctid FRT,FRB (Rc=0)
fctid. FRT,FRB (Rc=1)



Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x8000_0000_0000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode specified by RN.

If the rounded value is greater than $2^{63}-1$, then the result is 0x7FFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{63}, then the result is 0x8000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format, and XX is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into FRT.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 779.

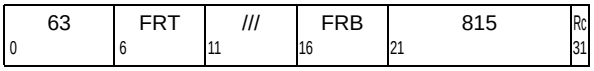
Except for enabled Invalid Operation Exceptions, FPRF is undefined. FR is set if the result is incremented when rounded. FI is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert with truncate Double-Precision To Signed Doubleword format X-form

fctidz FRT,FRB (Rc=0)
fctidz. FRT,FRB (Rc=1)



Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x8000_0000_0000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than $2^{63}-1$, then the result is 0x7FFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{63}, then the result is 0x8000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format, and XX is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into FRT.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 779.

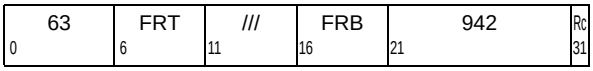
Except for enabled Invalid Operation Exceptions, FPRF is undefined. FR is set if the result is incremented when rounded. FI is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert with round Double-Precision To Unsigned Doubleword format X-form

fctidu FRT,FRB (Rc=0)
fctidu. FRT,FRB (Rc=1)



Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x0000_0000_0000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode specified by RN.

If the rounded value is greater than $2^{64}-1$, then the result is 0xFFFF_FFFF_FFFF_FFFF, and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, then the result is 0x0000_0000_0000_0000, and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format, and XX is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into FRT.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 779.

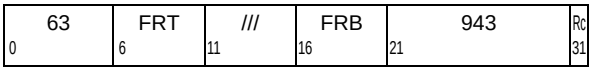
Except for enabled Invalid Operation Exceptions, FPRF is undefined. FR is set if the result is incremented when rounded. FI is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert with truncate Double-Precision To Unsigned Doubleword format X-form

fctiduz FRT,FRB (Rc=0)
fctiduz. FRT,FRB (Rc=1)



Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x0000_0000_0000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than $2^{64}-1$, then the result is 0xFFFF_FFFF_FFFF_FFFF, and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, then the result is 0x0000_0000_0000_0000, and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format, and XX is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into FRT.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 779.

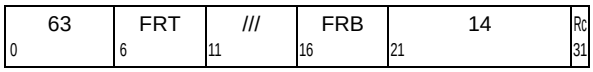
Except for enabled Invalid Operation Exceptions, FPRF is undefined. FR is set if the result is incremented when rounded. FI is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert with round Double-Precision To Signed Word format X-form

fctiw FRT,FRB (Rc=0)
fctiw. FRT,FRB (Rc=1)



Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x8000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode specified by RN.

If the rounded value is greater than $2^{31}-1$, then the result is 0x7FFF_FFFF, and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{31}, then the result is 0x8000_0000, and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit signed-integer format, and XX is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into $FRT_{32:63}$ and $FRT_{0:31}$ is undefined,

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 779.

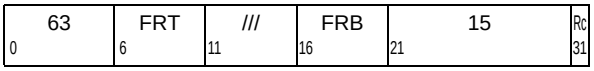
Except for enabled Invalid Operation Exceptions, FPRF is undefined. FR is set if the result is incremented when rounded. FI is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert with truncate Double-Precision To Signed Word fomat X-form

fctiwz FRT,FRB (Rc=0)
fctiwz. FRT,FRB (Rc=1)



Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x8000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than $2^{31}-1$, then the result is 0x7FFF_FFFF, and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{31}, then the result is 0x8000_0000, and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit signed-integer format, and XX is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into $FRT_{32:63}$ and $FRT_{0:31}$ is undefined,

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 779.

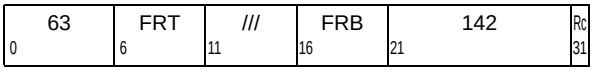
Except for enabled Invalid Operation Exceptions, FPRF is undefined. FR is set if the result is incremented when rounded. FI is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert with round Double-Precision To Unsigned Word format X-form

fctiwu FRT,FRB (Rc=0)
fctiwu. FRT,FRB (Rc=1)



Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x0000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode specified by RN.

If the rounded value is greater than $2^{32}-1$, then the result is 0xFFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, then the result is 0x0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format, and XX is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into $FRT_{32:63}$ and $FRT_{0:31}$ is undefined,

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 779.

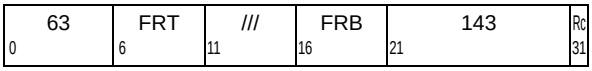
Except for enabled Invalid Operation Exceptions, FPRF is undefined. FR is set if the result is incremented when rounded. FI is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert with truncate Double-Precision To Unsigned Word format X-form

fctiwuz FRT,FRB (Rc=0)
fctiwuz. FRT,FRB (Rc=1)



Let src be the double-precision floating-point value in FRB.

If src is a NaN, then the result is 0x0000_0000, VXCVI is set to 1, and, if src is an SNaN, VXSNAN is set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round toward Zero.

If the rounded value is greater than 2^{32}-1, then the result is 0xFFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0.0, then the result is 0x0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format, and XX is set to 1 if the result is inexact.

If an enabled Invalid Operation Exception does not occur, then the result is placed into $FRT_{32:63}$ and $FRT_{0:31}$ is undefined,

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 779.

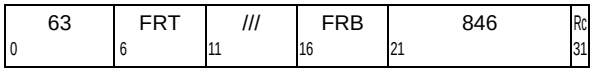
Except for enabled Invalid Operation Exceptions, FPRF is undefined. FR is set if the result is incremented when rounded. FI is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX VXSNAN VXCVI
CR1 (if Rc=1)

Floating Convert with round Signed Doubleword to Double-Precision format X-form

fcfid FRT,FRB (Rc=0)
fcfid. FRT,FRB (Rc=1)



The 64-bit signed fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to double-precision, using the rounding mode specified by RN, and placed into register FRT.

The conversion is described fully in Section A.3, “Floating-Point Convert from Integer Model”.

FPRF is set to the class and sign of the result. FR is set if the result is incremented when rounded. FI is set if the result is inexact.

Special Registers Altered:

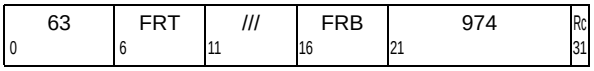
FPRF FR FI FX XX
CR1 (if Rc=1)

Programming Note

Converting a signed integer word to double-precision floating-point can be accomplished by loading the word from storage using *Load Float Word Algebraic Indexed* and then using ***fcfid***.

Floating Convert with round Unsigned Doubleword to Double-Precision format X-form

fcfidu FRT,FRB (Rc=0)
fcfidu. FRT,FRB (Rc=1)



The 64-bit unsigned fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to double-precision, using the rounding mode specified by $FPSCR_{RN}$, and placed into register FRT.

The conversion is described fully in Section A.3, "Floating-Point Convert from Integer Model".

$FPSCR_{FPRF}$ is set to the class and sign of the result. FR is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

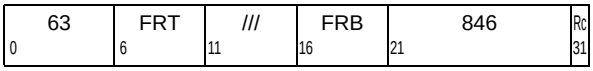
FPRF FR FI
FX XX
CR1 (if Rc=1)

Programming Note

Converting an unsigned integer word to double-precision floating-point can be accomplished by loading the word from storage using *Load Float Word and Zero Indexed* and then using ***fcfidu***.

Floating Convert with round Signed Doubleword to Single-Precision format X-form

fcfids FRT,FRB (Rc=0)
fcfids. FRT,FRB (Rc=1)



The 64-bit signed fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to single-precision, using the rounding mode specified by $FPSCR_{RN}$, and placed into register FRT.

The conversion is described fully in Section A.3, "Floating-Point Convert from Integer Model".

$FPSCR_{FPRF}$ is set to the class and sign of the result. FR is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

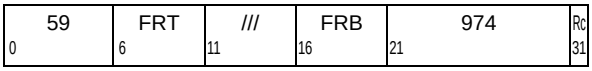
FPRF FR FI
FX XX
CR1 (if Rc=1)

Programming Note

Converting a signed integer word to single-precision floating-point can be accomplished by loading the word from storage using *Load Float Word Algebraic Indexed* and then using ***fcfids***.

Floating Convert with round Unsigned Doubleword to Single-Precision format X-form

fcfidus FRT,FRB (Rc=0)
fcfidus. FRT,FRB (Rc=1)



The 64-bit unsigned fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to single-precision, using the rounding mode specified by $FPSCR_{RN}$, and placed into register FRT.

The conversion is described fully in Section A.3, “Floating-Point Convert from Integer Model”.

$FPSCR_{FPRF}$ is set to the class and sign of the result. FR is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF FR FI
FX XX
CR1 (if Rc=1)

Programming Note

Converting a unsigned integer word to single-precision floating-point can be accomplished by loading the word from storage using *Load Float Word and Zero Indexed* and then using ***fcfidus***.

4.6.7.3 Floating Round to Integer Instructions

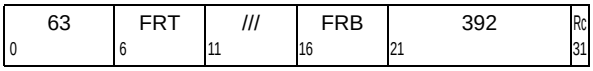
The *Floating Round to Integer* instructions provide direct support for rounding functions found in high level languages. For example, ***frin***, ***friz***, ***frip***, and ***frim*** implement C++ round(), trunc(), ceil(), and floor(), respectively. Note that ***frin*** does not implement the IEEE Round to Nearest function, which is often further described as “ties to even.” The rounding performed by these instructions is described fully in Section A.4, “Floating-Point Round to Integer Model” on page 784.

Programming Note

These instructions set $FPSCR_{FR\ FI}$ to 0b00 regardless of whether the result is inexact or rounded because there is a desire to preserve the value of $FPSCR_{XX}$. Furthermore, it is believed that most programs do not need to know whether these rounding operations produce inexact or rounded results. If it is necessary to determine whether the result is inexact or rounded, software must compare the result with the original source operand.

Floating Round to Integer Nearest X-form

frin FRT,FRB (Rc=0)
frin. FRT,FRB (Rc=1)



The floating-point operand in register FRB is rounded to an integral value as follows, with the result placed into register FRT. If the sign of the operand is positive, (FRB) + 0.5 is truncated to an integral value, otherwise (FRB) - 0.5 is truncated to an integral value.

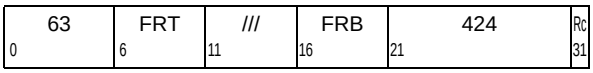
$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$ = 1.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXSNAN
CR1 (if Rc = 1)

Floating Round to Integer Toward Zero X-form

friz FRT,FRB (Rc=0)
friz. FRT,FRB (Rc=1)



The floating-point operand in register FRB is rounded to an integral value using the rounding mode round toward zero, and the result is placed into register FRT.

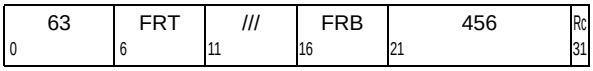
$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$ = 1.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXSNAN
CR1 (if Rc = 1)

Floating Round to Integer Plus X-form

frip FRT,FRB (Rc=0)
frip. FRT,FRB (Rc=1)



The floating-point operand in register FRB is rounded to an integral value using the rounding mode round toward +infinity, and the result is placed into register FRT.

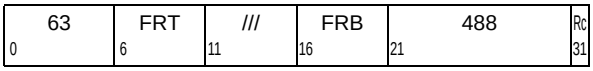
$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$ = 1.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXSNAN
CR1 (if Rc = 1)

Floating Round to Integer Minus X-form

frim FRT,FRB (Rc=0)
frim. FRT,FRB (Rc=1)



The floating-point operand in register FRB is rounded to an integral value using the rounding mode round toward -infinity, and the result is placed into register FRT.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}$ = 1.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXSNAN
CR1 (if Rc = 1)

4.6.8 Floating-Point Compare Instructions

The floating-point *Compare* instructions compare the contents of two floating-point registers. Comparison ignores the sign of zero (i.e., regards +0 as equal to -0). The comparison can be ordered or unordered.

The comparison sets one bit in the designated CR field to 1 and the other three to 0. The FPCC is set in the same way.

The CR field and the FPCC are set as follows.

Bit	Name	Description
0	FL	(FRA) < (FRB)
1	FG	(FRA) > (FRB)
2	FE	(FRA) = (FRB)
3	FU	(FRA) ? (FRB) (unordered)

Floating Compare Unordered X-form

fcmpu BF,FRA,FRB

63	BF	//	FRA	FRB	0	/
0	6	9	11	16	21	31

```
if (FRA) is a NaN or
   (FRB) is a NaN then c ← 0b0001
else if (FRA) < (FRB) then c ← 0b1000
else if (FRA) > (FRB) then c ← 0b0100
else                        c ← 0b0010
FPCC ← c
CR4×BF:4×BF+3 ← c
if (FRA) is an SNaN or
   (FRB) is an SNaN then
     VXSNAN ← 1
```

The floating-point operand in register FRA is compared to the floating-point operand in register FRB. The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, then CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, then VXSNAN is set.

Special Registers Altered:

CR field BF
FPCC
FX
VXSNAN

Floating Compare Ordered X-form

fcmpo BF,FRA,FRB

63	BF	//	FRA	FRB	32	/
0	6	9	11	16	21	31

```
if (FRA) is a NaN or
   (FRB) is a NaN then c ← 0b0001
else if (FRA) < (FRB) then c ← 0b1000
else if (FRA) > (FRB) then c ← 0b0100
else                        c ← 0b0010
FPCC ← c
CR4×BF:4×BF+3 ← c
if (FRA) is an SNaN or
   (FRB) is an SNaN then
     VXSNAN ← 1
     if VE = 0 then VXVC ← 1
else if (FRA) is a QNaN or
   (FRB) is a QNaN then VXVC ← 1
```

The floating-point operand in register FRA is compared to the floating-point operand in register FRB. The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, then CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, then VXSNAN is set and, if Invalid Operation is disabled (VE=0), VXVC is set. If neither operand is a Signaling NaN but at least one operand is a Quiet NaN, then VXVC is set.

Special Registers Altered:

CR field BF
FPCC
FX
VXSNAN VXVC

4.6.9 Floating-Point Select Instruction

Floating Select A-form

fsel FRT,FRA,FRC,FRB (Rc=0)
fsel. FRT,FRA,FRC,FRB (Rc=1)

63	FRT	FRA	FRB	FRC	23	Rc
0	6	11	16	21	26	31

```
if (FRA) ≥ 0.0 then FRT ← (FRC)
else FRT ← (FRB)
```

The floating-point operand in register FRA is compared to the value zero. If the operand is greater than or equal to zero, register FRT is set to the contents of register FRC. If the operand is less than zero or is a NaN, register FRT is set to the contents of register FRB. The comparison ignores the sign of zero (i.e., regards +0 as equal to -0).

Special Registers Altered:

CR1 (if Rc=1)

Programming Note

Examples of uses of this instruction can be found in Sections E.2, “Floating-Point Conversions” on page 642 and E.3, “Floating-Point Selection” on page 646.

Warning: Care must be taken in using ***fsel*** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section E.3.4, “Notes” on page 646.

fsel Usage Notes

This section gives examples of how the *Floating Select* instruction can be used to implement certain simple forms of if-then-else constructions, without branching.

The examples show program fragments in an imaginary, C-like, high-level programming language, and the corresponding program fragment using ***fsel*** and other Power ISA instructions. In the examples, `a`, `b`, `x`, `y`, and `z` are floating-point variables, which are assumed to be in FPRs `fa`, `fb`, `fx`, `fy`, and `fz`. FPR `fs` is assumed to be available for scratch space.

Warning: Care must be taken in using ***fsel*** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section .

Comparison to Zero

High-level language:	Power ISA:	Notes

`if a ≥ 0.0 then`
`  x ← y`
`else`
`  x ← z`
	`fsel fx,fa,fy,fz`	(1)

`if a > 0.0 then`
`  x ← y`
`else`
`  x ← z`
	`fneg fs,fa`
	`fsel fx,fs,fz,fy`
		(1,2)

`if a = 0.0 then`
`  x ← y`
`else`
`  x ← z`
	`fsel fx,fa,fy,fz`
	`fneg fs,fa`
	`fsel fx,fs,fx,fz`
		(1)

Simple if-then-else Constructions

High-level language:	Power ISA:	Notes
`if a ≥ b then x ← y` `else x ← z`	`fsub fs,fa,fb` `fsel fx,fs,fy,fz`	(4,5)
`if a > b then x ← y` `else x ← z`	`fsub fs,fb,fa` `fsel fx,fs,fz,fy`	(3,4,5)

`if a = b then x ← y`
`else x ← z`
	`fsub fs,fa,fb`
	`fsel fx,fs,fy,fz`
	`fneg fs,fs`
	`fsel fx,fs,fx,fz`
		(4,5)

Notes:

The following Notes apply to the preceding examples and to the corresponding cases using the other three arithmetic relations (<, ≤, and ≠). They should also be considered when any other use of ***fsel*** is contemplated.

In these Notes, the “optimized program” is the Power ISA program shown, and the “unoptimized program” (not shown) is the corresponding Power ISA program that uses ***fcmpu*** and *Branch Conditional* instructions instead of ***fsel***.

1. The unoptimized program affects the `VXSNAN` bit of the FPSCR, and therefore may cause the system error handler to be invoked if the corresponding exception is enabled, while the optimized program does not affect this bit. This property of the optimized program is incompatible with the IEEE standard.
2. The optimized program gives the incorrect result if a is a NaN.

3. The optimized program gives the incorrect result if a and/or b is a NaN (except that it may give the correct result in some cases for the minimum and maximum functions, depending on how those functions are defined to operate on NaNs).
4. The optimized program gives the incorrect result if a and b are infinities of the same sign. (Here it is assumed that Invalid Operation Exceptions are disabled, in which case the result of the subtraction is a NaN. The analysis is more complicated if Invalid Operation Exceptions are enabled, because in that case the target register of the subtraction is unchanged.)
5. The optimized program affects the OX, UX, XX, and VXISI bits of the FPSCR, and therefore may cause the system error handler to be invoked if the corresponding exceptions are enabled, while the unoptimized program does not affect these bits. This property of the optimized program is incompatible with the IEEE standard.

4.6.10 Floating-Point Status and Control Register Instructions

Except for ***mffsce***, ***mffscdrn***[*i*], ***mffscrn***[*i*], and ***mffsl***, *Floating-Point Status and Control Register* instructions synchronize the effects of all floating-point instructions executed by a given processor. Executing a *Floating-Point Status and Control Register* instruction ensures that all floating-point instructions previously initiated by the given processor have completed before the *Floating-Point Status and Control Register* instruction is initiated, and that no subsequent floating-point instructions are initiated by the given processor until the *Floating-Point Status and Control Register* instruction has completed. In particular:

- All exceptions that will be caused by the previously initiated instructions are recorded in the FPSCR before the Floating-Point Status and Control Register instruction is initiated.
- All invocations of the system floating-point enabled exception error handler that will be caused by the previously initiated instructions have occurred before the Floating-Point Status and Control Register instruction is initiated.
- No subsequent floating-point instruction that depends on or alters the settings of any FPSCR bits is initiated until the Floating-Point Status and Control Register instruction has completed.

(Floating-point *Storage Access* instructions are not affected.)

The instruction descriptions in this section refer to "FPSCR fields," where FPSCR field k is FPSCR bits `4*k:4*k+3`.

Move From FPSCR [*& Clear Enables* | *Lightweight* | *Control* [*& Set* (*DRN*|*RN*) [*Immediate*]]] *X-form*

mffs FRT (Rc=0)
mffs. FRT (Rc=1)

63	FRT	0	0	///	583	Rc
0	6	11	13	16	21	31

mffsce FRT

63	FRT	0	1	///	583	/
0	6	11	13	16	21	31

mffscdrn FRT,FRB

63	FRT	2	4	FRB	583	/
0	6	11	13	16	21	31

mffscdrni FRT,DRM

63	FRT	2	5	//	DRM	583	/
0	6	11	13	16	18	21	31

mffscrn FRT,FRB

63	FRT	2	6	FRB	583	/
0	6	11	13	16	21	31

mffscrni FRT,RM

63	FRT	2	7	///	RM	583	/
0	6	11	13	16	19	21	31

mffsl FRT

63	FRT	3	0	///	583	/
0	6	11	13	16	21	31

For *Move From FPSCR* (***mffs*[*.*]**), do the following.

The contents of the FPSCR are placed into register FRT.

If `Rc=1`, CR field 1 is set to the value `FX||FEX||VX||0X`.

For *Move From FPSCR & Clear Enables* (***mffsce***), do the following.

The contents of the FPSCR are placed into register FRT.

The contents of bits 56:60 (VE, OE, UE, ZE, XE) of the FPSCR are set to 0.

For *Move From FPSCR Control & set DRN* (***mffscdrn***), do the following.

Let *new_DRN* be the contents of bits 29:31 of register FRB.

The contents of the control bits in the FPSCR, that is, bits 29:31 (DRN) and bits 56:63 (VE, OE, UE, ZE, XE, NI, RN), are placed into the corresponding

bits in register FRT. All other bits in register FRT are set to 0.

new_RN is placed into bits 29:31 of the FPSCR (DRN).

For *Move From FPSCR Control & set DRN Immediate* (***mffscdrni***), do the following.

The contents of the control bits in the FPSCR, that is, bits 29:31 (DRN) and bits 56:63 (VE, OE, UE, ZE, XE, NI, RN), are placed into the corresponding bits in register FRT. All other bits in register FRT are set to 0.

The contents of bits 29:31 of the FPSCR (DRN) are set to the value of DRM.

For *Move From FPSCR Control & set RN* (***mffscrn***), do the following.

Let *new_RN* be the contents of bits 62:63 of register FRB.

The contents of the control bits in the FPSCR, that is, bits 29:31 (DRN) and bits 56:63 (VE, OE, UE, ZE, XE, NI, RN), are placed into the corresponding bits in register FRT. All other bits in register FRT are set to 0.

new_RN is placed into bits 62:63 of the FPSCR (RN).

For *Move From FPSCR Control & set RN Immediate* (***mffscrni***), do the following.

The contents of the control bits in the FPSCR, that is, bits 29:31 (DRN) and bits 56:63 (VE, OE, UE, ZE, XE, NI, RN), are placed into the corresponding bits in register FRT. All other bits in register FRT are set to 0.

The contents of bits 62:63 of the FPSCR (RN) are set to the value of RM.

For *Move From FPSCR Lightweight* (***mffsl***), do the following.

The contents of the control bits in the FPSCR, that is, bits 29:31 (DRN) and bits 56:63 (VE, OE, UE, ZE, XE, NI, RN), and the non-sticky status bits in the FPSCR, that is, bits 45:51 (FR, FI, C, FL, FG, FE, FU), are placed into the corresponding bits in register FRT. All other bits in register FRT are set to 0.

Special Registers Altered:

CR1	(if Rc=1)

Programming Note

mffsl permits software to read the control and non-sticky status bits in the FPSCR without the higher latency typically associated with accessing the sticky status bits.

mffscdrn[***i***] and ***mffscrn***[***i***] permit software to simultaneously read control bits in the FPSCR and set either the DRN or RN fields without the higher latency typically associated with accessing the status bits.

Move to Condition Register from FPSCR X-form

mcrfs BF,BFA

63	BF	//	BFA	//	///	64	/
0	6	9	11	14	16	21	31

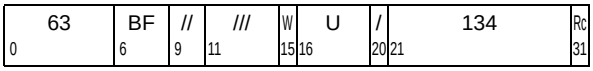
The contents of $FPSCR_{32:63}$ field BFA are copied to Condition Register field BF. All exception bits copied are set to 0 in the FPSCR. If the FX bit is copied, it is set to 0 in the FPSCR.

Special Registers Altered:

CR field BF	
FX OX	(if BFA=0)
UX ZX XX VXSNAN	(if BFA=1)
VXISI VXIDI VXZDZ VXIMZ	(if BFA=2)
VXVC	(if BFA=3)
VXSOFT VXSQRT VXCVI	(if BFA=5)

Move To FPSCR Field Immediate X-form

mtfsfi BF,U,W (Rc=0)
mtfsfi. BF,U,W (Rc=1)



The value of the U field is placed into FPSCR field `BF+8*(1-W)`.

$FPSCR_{FX}$ is altered only if BF = 0 and W = 0.

Special Registers Altered:

FPSCR field BF + 8*(1-W)
CR1 (if Rc=1)

Programming Note

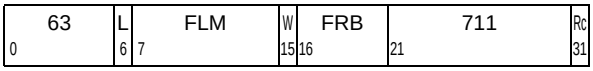
mtfsfi serves as both a basic and an extended mnemonic. The Assembler will recognize a ***mtfsfi*** mnemonic with three operands as the basic form, and a ***mtfsfi*** mnemonic with two operands as the extended form. In the extended form the W operand is omitted and assumed to be 0.

Programming Note

When $FPSCR_{32:35}$ is specified, bits 32 (FX) and 35 (OX) are set to the values of U_0 and U_3 (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from U_0 and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 33 and 34 (FEX and VX) are set according to the usual rule, given on page 125, and not from $U_{1:2}$.

Move To FPSCR Fields XFL-form

mtfsf FLM,FRB,L,W (Rc=0)
mtfsf. FLM,FRB,L,W (Rc=1)



The FPSCR is modified as specified by the FLM, L, and W fields.

L = 0

The contents of register FRB are placed into the FPSCR under control of the W field and the field mask specified by FLM. W and the field mask identify the 4-bit fields affected. Let `i` be an integer in the range 0-7. If FLM_i=1 then FPSCR field `k` is set to the contents of the corresponding field of register FRB, where `k=i+8*(1-W)`.

L = 1

The contents of register FRB are placed into the FPSCR.

$FPSCR_{FX}$ is not altered implicitly by this instruction.

Special Registers Altered:

FPSCR fields selected by mask, L, and W
CR1 (if Rc=1)

Programming Note

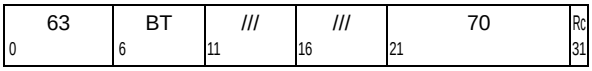
mtfsf serves as both a basic and an extended mnemonic. The Assembler will recognize a ***mtfsf*** mnemonic with four operands as the basic form, and a ***mtfsf*** mnemonic with two operands as the extended form. In the extended form the W and L operands are omitted and both are assumed to be 0.

Programming Note

If L=1 or if L=0 and $FPSCR_{32:35}$ is specified, bits 32 (FX) and 35 (OX) are set to the values of $(FRB)_{32}$ and $(FRB)_{35}$ (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from $(FRB)_{32}$ and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 33 and 34 (FEX and VX) are set according to the usual rule, given on page 125, and not from $(FRB)_{33:34}$.

Move To FPSCR Bit 0 X-form

mtfsb0 BT (Rc=0)
mtfsb0. BT (Rc=1)



Bit BT+32 of the FPSCR is set to 0.

Special Registers Altered:

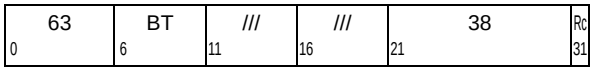
FPSCR bit BT+32
CR1 (if Rc=1)

Programming Note

Bits 33 and 34 (FEX and VX) cannot be explicitly reset.

Move To FPSCR Bit 1 X-form

mtfsb1 BT (Rc=0)
mtfsb1. BT (Rc=1)



Bit BT+32 of the FPSCR is set to 1.

Special Registers Altered:

FPSCR bits BT+32 and FX
CR1 (if Rc=1)

Programming Note

Bits 33 and 34 (FEX and VX) cannot be explicitly set.

Chapter 5. Decimal Floating-Point

5.1 Decimal Floating-Point (DFP) Facility Overview

This chapter describes the behavior of the decimal floating-point facility, the supported data types, formats, and classes, and the usage of registers. Also included are the execution model, exceptions, and instructions supported by the decimal floating-point facility.

The decimal floating-point (DFP) facility shares the 32 floating-point registers (FPRs) and the Floating-Point Status and Control Register (FPSCR) with the floating-point (BFP) facility. However, the interpretation of data formats in the FPRs, and the meaning of some control and status bits in the FPSCR are different between the BFP and DFP facilities.

The DFP facility also shares the Condition Register (CR) with the fixed-Point facility, the BFP faciltiy, and the vector facility.

The DFP facility supports three DFP data formats: DFP Short (single precision), DFP Long (double precision), and DFP Extended (quad precision). Most operations are performed on DFP Long or DFP Extended format directly. Support for DFP Short is limited to conversion to and from DFP Long. Some DFP instructions operate on other data types, including signed or unsigned binary fixed-point data, and signed or unsigned decimal data.

DFP instructions are provided to perform arithmetic, compare, test, quantum-adjustment, conversion, and format operations on operands held in FPRs or FPR pairs.

- Arithmetic instructions

  These instructions perform addition, subtraction, multiplication, and division operations.

- Compare instructions

  These instructions perform a comparison operation on the numerical value of two DFP operands.

- Test instructions

  These instructions test the data class, the data group, the exponent, or the number of significant digits of a DFP operand.

- Quantum-adjustment instructions

  These instructions convert a DFP number to a result in the form that has the designated exponent, which may be explicitly or implicitly specified.

- Conversion instructions

  These instructions perform conversion between different data formats or data types.

- Format instructions

  These instructions facilitate composing or decomposing a DFP operand.

These instructions are described in Section 5.6 “DFP Instruction Descriptions” on page 193.

The three DFP data formats allow finite numbers to be represented with different precision and ranges. Special codes are also provided to represent +Infinity, -Infinity, Quiet NaN (Not-a-Number), and Signaling NaN. Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation. The encoding of NaNs provides a diagnostic information field. This diagnostic field may be used to indicate such things as the source of an uninitialized variable or the reason an invalid result was produced.

The DFP processor recognizes a set of DFP exceptions which are indicated via bits set in the FPSCR. Additionally, the DFP exception actions depend on the setting of the various exception enable bits in the FPSCR.

The following DFP exceptions are detected by the DFP processor. The exception status bits in the FPSCR are indicated in parentheses.

- Invalid Operation Exception (VX)
  - SNaN (VXSNAN)
  - ∞ - ∞ (VXISI)
  - ∞ ÷ ∞ (VXIDI)
  - 0 ÷ 0 (VXZDZ)
  - ∞ % 0 (VXIMZ)
  - Invalid Compare (VXVC)

Invalid conversion (VXCVI)
- Zero Divide Exception (ZX)
- Overflow Exception (OX)
- Underflow Exception (UX)
- Inexact Exception (XX)

Each DFP exception and each category of Invalid Operation Exception has an exception status bit in the FPSCR. In addition, each of the five DFP exceptions has a corresponding enable bit in the FPSCR. These enable bits enable or disable the invocation of the system floating-point enabled exception error handler, and may affect the setting of some exception status bits in the FPSCR.

The usage of these bits by the DFP facility differs from the usage by the BFP facility. Section 5.5.10 "DFP Exceptions" on page 185 provides a detailed discussion of DFP exceptions, including the effects of the enable bits.

5.2 DFP Register Handling

The following sections describe first how the floating-point registers are utilized by the DFP facility. The subsequent section covers the DFP usage of CR and FPSCR.

5.2.1 DFP Usage of Floating-Point Registers

The DFP facility shares the same 32 64-bit FPRs with the BFP facility. Like the FP instructions, DFP instructions also use 5-bit fields for designating the FPRs to hold the source or target operands.

When data in DFP Short format is held in a FPR, it occupies the rightmost 32 bits of the FPR. The *Load Floating-Point as Integer Word Algebraic* instruction is provided to load the rightmost 32 bits of a FPR with a single-word data from storage. The *Store Floating-Point as Integer Word* instruction is available to store the rightmost 32 bits of a FPR to a storage location.

Data in DFP Long format, 64-bit binary fixed-point values, or 64-bit BCD values is held in a FPR using all 64 bits. Data of 64 bits may be loaded from storage via any of the *Load Floating-Point Double* instructions and stored via any of the *Store Floating-Point Double* instructions.

Data in DFP Extended format or 128-bit BCD values is held in an even-odd FPR pair using all 128 bits. Data of 128 bits must be loaded into the desired even-odd pair of floating-point registers using an appropriate sequence of the *Load Floating-Point Double* instructions and stored using an appropriate sequence of the *Store Floating-Point Double* instructions.

Data used as a source operand by any *Decimal Floating-Point* instruction that was produced, either directly or indirectly, by a *Load Floating-Point Single* instruction, a *Floating Round to Single-Precision* instruction, or a binary floating-point single-precision arithmetic instruction is boundedly undefined.

When an even-odd FPR pair is used to hold a 128-bit operand, the even-numbered FPR is used to hold the leftmost doubleword of the operand and the next higher-numbered FPR is used to hold the rightmost doubleword. A DFP instruction designating an odd-numbered FPR for a 128-bit operand is an invalid instruction form.

Programming Note

The *Floating-Point Move* instructions can be used to move operands between FPRs.

The bit definitions for the FPSCR are as follows.

Bit(s)	Description
0:28	Reserved
29:31	***DFP Rounding Control*** (DRN) See Section 5.5.2, "Rounding Mode Specification" on page 183. 000 Round to Nearest, Ties to Even 001 Round toward Zero 010 Round toward +Infinity 011 Round toward -Infinity 100 Round to Nearest, Ties away from 0 101 Round to Nearest, Ties toward 0 110 Round to away from Zero 111 Round to Prepare for Shorter Precision

Programming Note

$FPSCR_{28}$ is reserved for extension of the DRN field, therefore DRN may be set using the ***mtfsfi*** instruction to set the rounding mode.

32	***Floating-Point Exception Summary*** (FX) Every floating-point instruction, except ***mtfsfi*** and ***mtfsf***, implicitly sets $FPSCR_{FX}$ to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** can alter $FPSCR_{FX}$ explicitly.
33	***Floating-Point Enabled Exception Summary*** (FEX) This bit is the OR of all the floating-point exception bits masked by their respective enable bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter $FPSCR_{FEX}$ explicitly.
34	***Floating-Point Invalid Operation Exception Summary*** (VX) This bit is the OR of all the Invalid Operation

exception bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter $FPSCR_{VX}$ explicitly.

35 ***Floating-Point Overflow Exception*** (OX)
See Section 5.5.10.3, “Overflow Exception” on page 189.

36 ***Floating-Point Underflow Exception*** (UX)
See Section 5.5.10.4, “Underflow Exception” on page 189.

37 ***Floating-Point Zero Divide Exception*** (ZX)
See Section 5.5.10.2, “Zero Divide Exception” on page 188.

38 ***Floating-Point Inexact Exception*** (XX)
See Section 5.5.10.5, “Inexact Exception” on page 190.

$FPSCR_{XX}$ is a sticky version of $FPSCR_{FI}$ (see below). Thus the following rules completely describe how $FPSCR_{XX}$ is set by a given instruction.

- If the instruction affects $FPSCR_{FI}$, the new value of $FPSCR_{XX}$ is obtained by ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.
- If the instruction does not affect $FPSCR_{FI}$, the value of $FPSCR_{XX}$ is unchanged.

39 ***Floating-Point Invalid Operation Exception (SNaN)*** (VXSNAN)
See Section 5.5.10.1, “Invalid Operation Exception” on page 187.

40 ***Floating-Point Invalid Operation Exception (½ - ½)*** (VXISI)
See Section 5.5.10.1.

41 ***Floating-Point Invalid Operation Exception (½ + ½)*** (VXIDI)
See Section 5.5.10.1.

~~1~~42 ***Floating-Point Invalid Operation Exception (0 0)*** (VXZDZ)
See Section 5.5.10.1.

43 ***Floating-Point Invalid Operation Exception (½ % 0)*** (VXIMZ)
See Section 5.5.10.1.

44 ***Floating-Point Invalid Operation Exception (Invalid Compare)*** (VXVC)
See Section 5.5.10.1.

45 ***Floating-Point Fraction Rounded*** (FR)
The last *Arithmetic* or *Rounding and Conversion* instruction incremented the fraction during rounding. See Section 5.5.1, “Rounding” on page 182. This bit is not sticky.

46 ***Floating-Point Fraction Inexact*** (FI)
The last *Arithmetic* or *Rounding and Conversion* instruction either produced an inexact result during rounding or caused a disabled Overflow Exception. See Section 5.5.1. This bit is not sticky.

See the definition of $FPSCR_{XX}$, above, regarding the relationship between $FPSCR_{FI}$ and $FPSCR_{XX}$.

47:51 ***Floating-Point Result Flags*** (FPRF)
This field is set as described below. For arithmetic, rounding, and conversion instructions, the field is set based on the result placed into the target register, except that if any portion of the result is undefined then the value placed into FPRF is undefined.

47 ***Floating-Point Result Class Descriptor*** (C)
Arithmetic, rounding, and conversion instructions may set this bit with the FPCC bits, to indicate the class of the result as shown in Figure 58 on page 178.

48:51 ***Floating-Point Condition Code*** (FPCC)
Floating-point *Compare* and *DFP Test* instructions set one of the FPCC bits to 1 and the other three FPCC bits to 0. Arithmetic, rounding, and conversion instructions may set the FPCC bits with the C bit, to indicate the class of the result as shown in Figure 58 on page 178. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.

48 ***Floating-Point Less Than or Negative*** (FL or <)

49 ***Floating-Point Greater Than or Positive*** (FG or >)

50 ***Floating-Point Equal or Zero*** (FE or =)

51 ***Floating-Point Unordered or NaN*** (FU or ?)

52 Reserved

53 ***Floating-Point Invalid Operation Exception (Software Request)*** (VXSOFT)
This bit can be altered only by ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, or ***mtfsb1***. See Section 5.5.10.1, “Invalid Operation Exception” on page 187.

54 Neither used nor changed by DFP.

Programming Note

Although the architecture does not provide a DFP square root instruction, if software simulates such an instruction, it should set bit 54 whenever the source operand of the square root function is invalid.

55 ***Floating-Point Invalid Operation Exception (Invalid Conversion)*** (VXCVI)
See Section 5.5.10.1.

56 ***Floating-Point Invalid Operation Exception Enable*** (VE)
See Section 5.5.10.1.

57 ***Floating-Point Overflow Exception Enable*** (OE)
See Section 5.5.10.3, “Overflow Exception” on page 189.

58 ***Floating-Point Underflow Exception Enable*** (UE)
See Section 5.5.10.4, “Underflow Exception” on page 189.

59 ***Floating-Point Zero Divide Exception Enable*** (ZE)
See Section 5.5.10.2, “Zero Divide Exception” on page 188.

60 ***Floating-Point Inexact Exception Enable*** (XE)
See Section 5.5.10.5, “Inexact Exception” on page 190

61 Reserved (not used by DFP)

62:63 ***Binary Floating-Point Rounding Control*** (RN)
See Section 5.5.1, “Rounding” on page 182.

00 Round to Nearest
01 Round toward Zero
10 Round toward +Infinity
11 Round toward -Infinity

Result Flags					Result Value Class
C	<	>	=	?	
0	0	0	0	1	Signaling NaN (DFP only)
1	0	0	0	1	Quiet NaN
0	1	0	0	1	- Infinity
0	1	0	0	0	- Normal Number
1	1	0	0	0	- Subnormal Number
1	0	0	1	0	- Zero
0	0	0	1	0	+ Zero
1	0	1	0	0	+ Subnormal Number
0	0	1	0	0	+ Normal Number
0	0	1	0	1	+ Infinity

Figure 58. Floating-Point Result Flags

5.3 DFP Support for Non-DFP Data Types

In addition to the DFP data types, the DFP processor provides limited support for the following non-DFP data types: signed or unsigned binary fixed-point data, and signed or unsigned decimal data.

In unsigned binary fixed-point data, all bits are used to express the absolute value of the number. For signed binary fixed-point data, the leftmost bit represents the sign, which is followed by the numeric field. Positive numbers are represented in true binary notation with the sign bit set to zero. When the value is zero, all bits are zeros, including the sign bit. Negative numbers are represented in two’s complement binary notation with a one in the sign-bit position.

For decimal data, each byte contains a pair of four-bit nibbles; each four-bit nibble contains a binary-coded-decimal (BCD) code. There are two kinds of BCD codes: digit code and sign code. For unsigned decimal data, all nibbles contain a digit code (D) as shown in Figure 59

D	D	D	D	. . .	D	D	D	D

Figure 59. Format for Unsigned Decimal Data

For signed decimal data, the rightmost nibble contains a sign code (S) and all other nibbles contain a digit code as shown in Figure 60.

D	D	D	D	. . .	D	D	D	S

Figure 60. Format for Signed Decimal Data

The decimal digits 0-9 have the binary encoding 0000-1001. The preferred plus-sign codes are 1100 and 1111. The preferred minus sign code is 1101. These are the sign codes generated for the results of the *Decode DPD To BCD* instruction. A selection is provided by this instruction to specify which of the two preferred plus sign codes is to be generated. Alternate sign codes are also recognized as valid in the sign position: 1010 and 1110 are alternate sign codes for plus, and 1011 is an alternate sign code for minus. Alternate sign codes are accepted for any source operand, but are not generated as a result by the instruction. When an invalid digit or sign code is detected by the *Encode BCD To DPD* instruction, an invalid-opera-

tion exception occurs. A summary of digit and sign codes are provided in Figure 61.

Binary Code	Recognized As	
	Digit	Sign
0000	0	Invalid
0001	1	Invalid
0010	2	Invalid
0011	3	Invalid
0100	4	Invalid
0101	5	Invalid
0110	6	Invalid
0111	7	Invalid
1000	8	Invalid
1001	9	Invalid
1010	Invalid	Plus
1011	Invalid	Minus
1100	Invalid	Plus (preferred; option 1)
1101	Invalid	Minus (preferred)
1110	Invalid	Plus
1111	Invalid	Plus (preferred; option 2)

Figure 61. Summary of BCD Digit and Sign Codes

5.4 DFP Number Representation

A DFP finite number consists of three components: a sign bit, a signed exponent, and a significand. The signed exponent is a signed binary integer. The *significand* consists of a number of decimal digits, which are to the left of the implied decimal point. The rightmost digit of the significand is called the *units* digit. The numerical value of a DFP finite number is represented as $(-1)^{sign} \times significand \times 10^{exponent}$ and the unit value of this number is (1×10^{exponent}), which is called the *quantum*.

DFP finite numbers are not normalized. This allows leading zeros and trailing zeros to exist in the significand. This unnormalized DFP number representation allows some values to have redundant forms; each form represents the DFP number with a different combination of the significand value and the exponent value. For example, 1000000×10^5 and 10×10^{10} are two different forms of the same numerical value. A *form* of this number representation carries information about both the numerical value and the quantum of a DFP finite number.

The *significant digits* of a DFP finite number are the digits in the significand beginning with the leftmost non-zero digit and ending with the units digit.

5.4.1 DFP Data Format

DFP numbers and NaNs may be represented in FPRs in any of the three data formats: DFP Short, DFP Long, or DFP Extended. The contents of each data format represent encoded information. Special codes are assigned to NaNs and infinities. Different formats support different sizes in both significand and exponent. Arithmetic, compare, test, quantum-adjustment, and format instructions are provided for DFP Long and DFP Extended formats only.

The *sign* is encoded as a one bit binary value. *Significand* is encoded as an unsigned decimal integer in two distinct parts. The leftmost digit (LMD) of the *significand* is encoded as part of the *combination* field*;* the remaining digits of the *significand* are encoded in the *trailing significand* field. The *exponent* is contained in the *combination* field in two parts. However, prior to encoding, the *exponent* is converted to an unsigned binary value called the *biased exponent* by adding a *bias* value which is a constant for each format. The two leftmost bits of the *biased exponent* are encoded with the leftmost digit of the significand in the leftmost bits of the combination field. The rest of the biased exponent occupies the remaining portion of the *combination* field.

5.4.1.1 Fields Within the Data Format

The DFP data representation comprises three fields, as diagrammed below for each of the three formats:

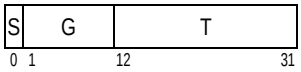


Figure 62. DFP Short format

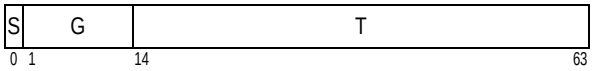


Figure 63. DFP Long format

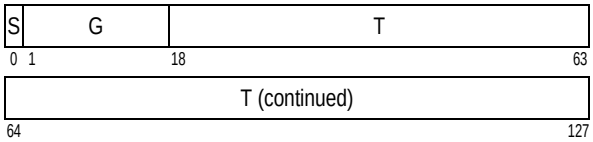


Figure 64. DFP Extended format

The fields are defined as follows:

Sign bit (S)
The sign bit is in bit 0 of each format, and is zero for plus and one for minus.

Combination field (G)
As the name implies, this field provides a combination of the exponent and the left-most digit (LMD) of the significand, for finite numbers, or provides a special code

for denoting the value as either a Not-a-Number or an Infinity.

The first 5 bits of the combination field contain the encoding of NaN or infinity, or the two leftmost bits of the biased exponent and the leftmost digit (LMD) of the significand. The following tables show the encoding:

$G_{0:4}$	Description
11111	NaN
11110	Infinity
All others	Finite Number (see Figure 66)

Figure 65. Encoding of the G field for Special Symbols

LMD	Leftmost 2-bits of biased exponent		
	00	01	10
0	00000	01000	10000
1	00001	01001	10001
2	00010	01010	10010
3	00011	01011	10011
4	00100	01100	10100
5	00101	01101	10101
6	00110	01110	10110
7	00111	01111	10111
8	11000	11010	11100
9	11001	11011	11101

Figure 66. Encoding of bits 0:4 of the G field for Finite Numbers

For DFP finite numbers, the rightmost N-5 bits of the N-bit combination field contain the remaining bits of the *biased exponent*. For NaNs, bit 5 of the combination field is used to distinguish a Quiet NaN from a Signaling NaN; the remaining bits in a source operand are ignored and they are set to zeros in a target operand by most operations. For infinities, the rightmost N-5 bits of the N-bit combination field of a source operand are ignored and they are set to zeros in a target operand by most operations.

Trailing Significand field (T)
For DFP finite numbers, this field contains the remaining *significand* digits. For NaNs, this field may be used to contain diagnostic information. For infinities, contents in this field of a source operand are ignored and they are set to zeros in a target operand by most operations. The trailing significand field is a multiple of 10-bit blocks. The multiple depends on the format. Each 10-bit block is called a declet and represents three decimal digits, using the Densely Packed Decimal (DPD) encoding defined in Appendix B.

5.4.1.2 Summary of DFP Data Formats

The properties of the three DFP formats are summarized in the following table:.

	Format		
	DFP Short	DFP Long	DFP Extended
Widths (bits):			
Format	32	64	128
Sign (S)	1	1	1
Combination (G)	11	13	17
Trailing Significand (T)	20	50	110
Exponent:			
Maximum biased	191	767	12,287
Maximum (X_{max})	90	369	6111
Minimum (X_{min})	-101	-398	-6176
Bias	101	398	6176
Precision (p) (digits)	7	16	34
Magnitude:			
Maximum normal number (N_{max})	$(10^7 - 1) \times 10^{90}$	$(10^{16} - 1) \times 10^{369}$	$(10^{34} - 1) \times 10^{6111}$
Minimum normal number (N_{min})	1×10^{-95}	1×10^{-383}	1×10^{-6143}
Minimum subnormal number (D_{min})	1×10^{-101}	1×10^{-398}	1×10^{-6176}

Figure 67. Summary of DFP Formats

5.4.1.3 Preferred DPD Encoding

Execution of DFP instructions decodes source operands from DFP data formats to an internal format for processing, and encodes the operation result before the final result is returned as the target operand.

As part of the decoding process, declets in the trailing significand field of source operands are decoded to their corresponding BCD digit codes using the DPD-to-BCD decoding algorithm. As part of the encoding process, BCD digit codes to be stored into the trailing significand field of the target operand are encoded into declets using the BCD-to-DPD encoding algorithm. Both the decoding and encoding algorithms are defined in Appendix B.

As explained in Appendix B, there are eight 3-digit decimal values that have redundant DPD codes and one preferred DPD code. All redundant DPD codes are recognized in source operands for the associated 3-digit decimal number. DFP operations will always generate the preferred DPD codes for the trailing significand field of the target operand.

5.4.2 Classes of DFP Data

There are six classes of DFP data, which include numerical and nonnumeric entities. The numerical entities include zero, subnormal number, normal number, and infinity data classes. The nonnumeric entities include quiet and signaling NaNs data classes. The value of a DFP finite number, including zero, subnormal number, and normal number, is a quantization of the real number based on the data format. The *Test Data Class* instruction may be used to determine the class of a DFP operand. In general, an operation that returns a DFP result sets the $FPSCR_{FPRF}$ field to indicate the data class of the result.

The following tables show the value ranges for finite-number data classes, and the codes for NaNs and infinities.

Data Class	Sign	Magnitude
Zero	±	0*
Subnormal	±	$D_{min} \leq \lvert X \rvert < N_{min}$
Normal	±	$N_{min} \leq \lvert Y \rvert \leq N_{max}$
* The significand is zero and the exponent is any representable value		

Figure 68. Value Ranges for Finite Number Data Classes

Data Class	S	G	T
+Infinity	0	11110xxx . . . xxx	xxx . . . xxx
–Infinity	1	11110xxx . . . xxx	xxx . . . xxx
Quiet NaN	x	111110xx . . . xxx	xxx . . . xxx
Signaling NaN	x	111111xx . . . xxx	xxx . . . xxx
x Don’t care			

Figure 69. Encoding of NaN and Infinity Data Classes

Zeros
Zeros have a zero significand and any representable value in the exponent. A +0 is distinct from -0, and zeros with different exponents are distinct, except that comparison treats them as equal.

Subnormal Numbers
Subnormal numbers have values that are smaller than N_{min} and greater than zero in magnitude.

Normal Numbers
Normal numbers are nonzero finite numbers whose magnitude is between N_{min} and N_{max} inclusively.

Infinities
Infinities are represented by 0b11110 in the leftmost 5 bits of the combination field. When an operation is defined to generate an infinity as the result, a default infinity is sometimes supplied. A default infinity has all remaining bits in the combination field and trailing significand field set to zeros.

When infinities are used as source operands, only the leftmost 5 bits of the combination field are interpreted (i.e., 0b11110 indicates the value is an infinity). The trailing significand field of infinities is usually ignored. For generated infinities, the leftmost 5 bits of the combination field are set to 0b11110 and all remaining combination bits are set to zero.

Infinities can participate in most arithmetic operations and give a consistent result. In comparisons, any +Infinity compares greater than any finite number, and any -Infinity compares less than any finite number. All +Infinity are compared equal and all -Infinity are compared equal.

Signaling and Quiet NaNs
There are two types of Not-a-Numbers (NaNs), Signaling (SNaN) and Quiet (QNaN).

0b111110 in the leftmost 6 bits of the combination field indicates a Quiet NaN, whereas 0b111111 indicates a Signaling NaN.

A special QNaN is sometimes supplied as the *default QNaN* for a disabled invalid-operation exception; it has a plus sign, the leftmost 6 bits of the combination field set to 0b111110 and remaining bits in the combination field and the trailing significand field set to zero.

Normally, source QNaNs are *propagated* during operations so that they will remain visible at the end. When a QNaN is propagated, the sign is preserved, the decimal value of the trailing significand field is preserved but reencoded using the preferred DPD codes, and the contents in the rightmost N-6 bits of the combination field set to zero, where N is the width of the combination field for the format.

A source SNaN generally causes an invalid-operation exception. If the exception is disabled, the SNaN is converted to the corresponding QNaN and propagated. The primary encoding difference between an SNaN and a QNaN is that bit 5 of an SNaN is 1 and bit 5 of a QNaN is 0. When an SNaN is propagated as a QNaN, bit 5 is set to 0, and, just as with QNaN proagation, the sign is preserved, the decimal value of the trailing significand field is preserved but reencoded using the preferred DPD codes, and the contents in the rightmost N-6 bits of the combination field set to zero, where N is the width of the combination field for the format. For some format-conversion instructions, a source SNaN does not cause an invalid-operation exception, and an SNaN is returned as the target operand.

For instructions with two source NaNs and a NaN is to be propagated as the result, do the following.

- If there is a QNaN in FRA and an SNaN in FRB, the SNaN in FRB is propagated.
- Otherwise, propagate the NaN is FRA.

5.5 DFP Execution Model

DFP operations are performed as if they first produce an intermediate result correct to infinite precision and with unbounded range. The intermediate result is then rounded to the destination's precision according to one of the eight DFP rounding modes. If the rounded result has only one form, it is delivered as the final result; if the rounded result has redundant forms, then an *ideal exponent* is used to select the form of the final result. The ideal exponent determines the form, not the value, of the final result. (See Section 5.5.3 "Formation of Final Result" on page 183.)

5.5.1 Rounding

Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit the destination's precision. The destination's precision of an operation defines the set of permissible resultant values. For most operations, the destination's precision is the target-format precision and the permissible resultant values are those values representable in the target format. For some special operations, the destination precision is constrained by both the target format and some additional restrictions, and the permissible resultant values are a subset of the values representable in the target format.

Rounding sets FPSCR bits FR and FI. When an inexact exception occurs, FI is set to one; otherwise, FI is set to zero. When an inexact exception occurs and if the rounded result is greater in magnitude than the intermediate result, then FR is set to one; otherwise, FR is set to zero. The exception is the *Round to FP Integer Without Inexact* instruction, which always sets FR and FI to zero. Rounding may cause an overflow exception or underflow exception; it may also cause an inexact exception.

Refer to Figure 70 below for rounding. Let Z be the intermediate result of a DFP operation. Z may or may not fit in the destination's precision. If Z is exactly one of the permissible representable resultant values, then the final result in all rounding modes is Z. Otherwise, either Z1 or Z2 is chosen to approximate the result, where Z1 and Z2 are the next larger and smaller permissible resultant values, respectively.

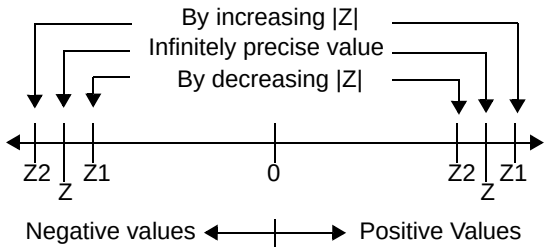


Figure 70. Rounding

Round to Nearest, Ties to Even
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the one whose units digit would have been even in the form with the largest common quantum of the two permissible resultant values. However, an infinitely precise result with magnitude at least (N_{max} + 0.5Q(N_{max})) is rounded to infinity with no change in sign; where Q(N_{max}) is the quantum of N_{max}.

Round toward 0
Choose the smaller in magnitude (Z1 or Z2).

Round toward +∞
Choose Z1.

Round toward -∞
Choose Z2.

Round to Nearest, Ties away from 0
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the larger in magnitude (Z1 or Z2). However, an infinitely precise result with magnitude at least (N_{max} + 0.5Q(N_{max})) is rounded to infinity with no change in sign; where Q(N_{max}) is the quantum of N_{max}.

Round to Nearest, Ties toward 0
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the smaller in magnitude (Z1 or Z2). However, an infinitely precise result with magnitude

greater than (N_{max} + 0.5Q(N_{max})) is rounded to infinity with no change in sign; where Q(N_{max}) is the quantum of N_{max}.

Round away from 0
Choose the larger in magnitude (Z1 or Z2).

Round to prepare for shorter precision
Choose the smaller in magnitude (Z1 or Z2). If the selected value is inexact and the units digit of the selected value is either 0 or 5, then the digit is incremented by one and the incremented result is delivered. In all other cases, the selected value is delivered. When a value has redundant forms, the units digit is determined by using the form that has the smallest exponent.

5.5.2 Rounding Mode Specification

Unless otherwise specified in the instruction definition, the rounding mode used by an operation is specified in the DFP rounding control (DRN) field of the FPSCR. The eight DFP rounding modes are encoded in the DRN field as specified in the table below.

DRN	Rounding Mode
000	Round to Nearest, Ties to Even
001	Round toward 0
010	Round toward +Infinity
011	Round toward -Infinity
100	Round to Nearest, Ties away from 0
101	Round to Nearest, Ties toward 0
110	Round away from 0
111	Round to Prepare for Shorter Precision

Figure 71. Encoding of DFP Rounding-Mode Control (DRN)

For the quantum-adjustment, a 2-bit immediate field, called RMC (*Rounding Mode Control*), in the instruction specifies the rounding mode used. The RMC field may contain a primary encoding or a secondary encoding. For *Quantize*, *Quantize Immediate*, and *Reround*, the RMC field contains the primary encoding. For *Round to FP Integer* the field contains either encoding, depending on the setting of a RMC-encoding-selection bit. The following tables define the primary encoding and the secondary encoding.

Primary RMC	Rounding Mode
00	Round to nearest, ties to even
01	Round toward 0
10	Round to nearest, ties away from 0
11	Round according to $FPSCR_{DRN}$

Figure 72. Primary Encoding of Rounding-Mode Control

Secondary RMC	Rounding Mode
00	Round to + ∞
01	Round to - ∞
10	Round away from 0
11	Round to nearest, ties toward 0

Figure 73. Secondary Encoding of Rounding-Mode Control

5.5.3 Formation of Final Result

An ideal exponent is defined for each DFP instruction that returns a DFP data operand.

5.5.3.1 Use of Ideal Exponent

For all DFP operations,

- if the rounded intermediate result has only one form, then that form is delivered as the final result.
- if the rounded intermediate result has redundant. forms and is exact, then the form with the exponent closest to the ideal exponent is delivered.
- if the rounded intermediate result has redundant forms and is inexact, then the form with the smallest exponent is delivered.

The following table specifies the ideal exponent for each instruction.

Operations	Ideal Exponent
Add	min(E(FRA), E(FRB))
Subtract	min(E(FRA), E(FRB))
Multiply	E(FRA) + E(FRB)
Divide	E(FRA) - E(FRB)
Quantize-Immediate	See Instruction Description
Quantize	E(FRA)
Reround	See Instruction Description
Round to FP Integer	max(0, E(FRA))
Convert to DFP Long	E(FRA)
Convert to DFP Extended	E(FRA)
Round to DFP Short	E(FRA)
Round to DFP Long	E(FRA)
Convert from Fixed	0
Encode BCD to DPD	0
Insert Biased Exponent	E(FRA)
Notes: E(x) - exponent of the DFP operand in register x.	

Figure 74. Summary of Ideal Exponents

5.5.4 Arithmetic Operations

Four arithmetic operations are provided: Add, Subtract, Multiply, and Divide.

5.5.4.1 Sign of Arithmetic Result

The following rules govern the sign of an arithmetic operation when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the source operand having the larger absolute value. If both source operands have the same sign, the sign of the result of an add operation is the same as the sign of the source operands. When the sum of two operands with opposite signs is exactly zero, the sign of the result is positive in all rounding modes except Round toward -∞, in which case the sign is negative.
- The sign of the result of the subtract operation x - y is the same as the sign of the result of the add operation x + (-y).
- The sign of the result of a multiply or divide operation is the exclusive-OR of the signs of the source operands.

5.5.5 Compare Operations

Two sets of instructions are provided for comparing numerical values: *Compare Ordered* and *Compare Unordered*. In the absence of NaNs, these instructions work the same. These instructions work differently when either of the followings is true:

1. At least one source operand of the instruction is an SNaN and the invalid-operation exception is disabled.
2. When there is no SNaN in any source operand, at least one source operand of the instruction is a QNaN

In case 1, *Compare Unordered* recognizes an invalid-operation exception and sets the $FPSCR_{VXSNAN}$ flag, but *Compare Ordered* recognizes the exception and sets both the $FPSCR_{VXSNAN}$ and $FPSCR_{VXVC}$ flags. In case 2, *Compare Unordered* does not recognize an exception, but *Compare Ordered* recognizes an invalid-operation exception and sets the $FPSCR_{VXVC}$ flag.

For finite numbers, comparisons are performed on values, that is, all redundant forms of a DFP number are treated equal.

Comparisons are always exact and cannot cause an inexact exception.

Comparison ignores the sign of zero, that is, +0 equals -0.

Infinities with like sign compare equal, that is, +∞ equals +∞, and -∞ equals -∞.

A NaN compares as unordered with any other operand, whether a finite number, an infinity, or another NaN, including itself.

Execution of a compare instruction always completes, regardless of whether any DFP exception occurs or not, and whether the exception is enabled or not.

5.5.6 Test Operations

Four kinds of test operations are provided: *Test Data Class*, *Test Data Group*, *Test Exponent*, and *Test Significance*.

The *Test Data Class* instruction examines the contents of a source operand and determines if the operand is one of the specified data classes. The test result and the sign of the source operand are indicated in the $FPSCR_{FPCC}$ field and CR field BF.

The *Test Data Group* instruction examines the contents of a source operand and determines if the operand is one of the specified data groups. The test result and the sign of the source operand are indicated in the $FPSCR_{FPCC}$ field and CR field BF.

The *Test Exponent* instruction compares the exponent of the two source operands. The test operation ignores the sign and significand of operands. Infinities compare equal, and NaNs compare equal. The test result is indicated in the $FPSCR_{FPCC}$ field and CR field BF.

The *Test Significance* instruction compares the number of significant digits of one source operand with the referenced number of significant digits in another source operand. The test result is indicated in the $FPSCR_{FPCC}$ field and CR field BF.

Execution of a test instruction does not cause any DFP exception.

5.5.7 Quantum Adjustment Operations

Four kinds of quantum-adjustment operations are provided: *Quantize*, *Quantize Immediate*, *Reround*, and *Round To FP Integer*. Each of them has an immediate field which specifies whether the rounding mode in FPSCR or a different one is to be used.

The *Quantize* instruction is used to adjust a DFP number to the form that has the specified target exponent. The *Quantize Immediate* instruction is similar to the *Quantize* instruction, except that the target exponent is specified in a 5-bit immediate field as a signed binary integer and has a limited range.

The *Reround* instruction is used to simulate a DFP operation of a precision other than that of DFP Long or DFP Extended. For the *Reround* instruction to produce

a result which accurately reflects that which would have resulted from a DFP operation of the desired precision *d* in the range {1: 33} inclusively, the following conditions must be met:

- The precision of the preceding DFP operation must be at least one digit larger than *d*.
- The rounding mode used by the preceding DFP operation must be *round-to-prepare-for-shorter-precision.*

The *Round To FP Integer* instruction is used to round a DFP number to an integer value of the same format. The target exponent is implicitly specified, and is greater than or equal to zero.

5.5.8 Conversion Operations

There are two kinds of conversion operations: data-format conversion and data-type conversion.

5.5.8.1 Data-Format Conversion

The instructions *Convert To DFP Long* and *Convert To DFP Extended* convert DFP operands to wider formats; the instructions *Round To DFP Short* and *Round To DFP Long* convert DFP operands to narrower formats.

When converting a finite number to a wider format, the result is exact. When converting a finite number to a narrower format, the source operand is rounded to the target-format precision, which is specified by the instruction, not by the target register size.

When converting a finite number, the ideal exponent of the result is the source exponent.

Conversion of an infinity or NaN to a different format does not preserve the source combination field. Let N be the width of the target format's combination field.

- When the result is an infinity or a QNaN, the contents of the rightmost N-5 bits of the N-bit target combination field are set to zero.
- When the result is an SNaN, bit 5 of the target format's combination field is set to one and the rightmost N-6 bits of the N-bit target combination field are set to zero.

When converting a NaN to a wider format or when converting an infinity from DFP Short to DFP Long, digits in the source trailing significand field are reencoded using the preferred DPD codes with sufficient zeros appended on the left to form the target trailing significand field. When converting a NaN to a narrower format or when converting an infinity from DFP Long to DFP Short, the appropriate number of leftmost digits of the source trailing significand field are removed and the remaining digits of the field are reencoded using the preferred DPD codes to form the target trailing significand field.

When converting an infinity between DFP Long and DFP Extended, a default infinity with the same sign is produced.

When converting an SNaN between DFP Short and DFP Long, it is converted to an SNaN without causing an invalid-operation exception. When converting an SNaN between DFP Long and DFP Extended, the invalid-operation exception occurs; if the invalid-operation exception is disabled, the result is converted to the corresponding QNaN.

5.5.8.2 Data-Type Conversion

The instructions *Convert From Fixed* and *Convert To Fixed* are provided to convert a number between the DFP data type and the signed 64-bit binary-integer data type.

Conversion of a signed 64-bit binary integer to a DFP Extended number is always exact.

Conversion of a DFP number to a signed 64-bit binary integer results in an invalid-operation exception when the converted value does not fit into the target format, or when the source operand is an infinity or NaN. When the exception is disabled, the most positive integer is returned if the source operand is a positive number or +∞, and the most negative integer is returned if the source operand is a negative number, -∞, or NaN.

5.5.9 Format Operations

The format instructions are provided to facilitate composing or decomposing a DFP number, and consist of *Encode BCD To DPD*, *Decode DPD To BCD*, *Extract Biased Exponent*, *Insert Biased Exponent*, *Shift Significand Left Immediate*, and *Shift Significand Right Immediate*. A source operand of SNaN does not cause an invalid-operation exception, and an SNaN may be produced as the target operand.

5.5.10 DFP Exceptions

This architecture defines the following DFP exceptions:

- Invalid Operation Exception
  - SNaN
  - ∞ - ∞
  - ∞ ÷ ∞
  - 0 ÷ 0
  - ∞ % 0
  - Invalid Compare
  - Invalid Conversion
- Zero Divide Exception
- Overflow Exception
- Underflow Exception
- Inexact Exception

These exceptions may occur during execution of a DFP instruction.

Each DFP exception, and each category of the Invalid Operation Exception, has an exception status bit in the FPSCR. In addition, each DFP exception has a corresponding enable bit in the FPSCR. The exception status bit indicates occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the FE0 and FE1 bits (see the discussion of FE0 and FE1 below), whether and how the system floating-point enabled exception error handler is invoked. (In general, the enabling specified by the enable bit is of invoking the system error handler, not of permitting the exception to occur. The occurrence of an exception depends only on the instruction and its source operands, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an Underflow Exception may depend on the setting of the enable bit.)

A single instruction, other than ***mtfsfi*** or ***mtfsf***, may set more than one exception bit only in the following cases:

- Inexact Exception may be set with Overflow Exception.
- Inexact Exception may be set with Underflow Exception.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Compare) for *Compare Ordered* instructions
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Conversion) for *Convert To Fixed* instructions.

When an exception occurs the instruction execution may be completed or partially completed, depending on the exception and the operation.

For all instructions, except for the Compare and Test instructions, the following exceptions cause the instruction execution to be partially completed. That is, setting of CR field 1(when Rc=1) and exception status flags is performed, but no result is stored into the target FPR or FPR pair. For Compare and Test instructions, instruction execution is always completed, regardless of whether any DFP exception occurs or not, and whether the exception is enabled or not.

- Enabled Invalid Operation
- Enabled Zero Divide

For the remaining kinds of exceptions, instruction execution is completed, a result, if specified by the instruction, is generated and stored into the target FPR or FPR pair, and appropriate status flags are set. The result may be a different value for the enabled and disabled conditions for some of these exceptions. The kinds of exceptions that deliver a result in target FPR are the following:

- Disabled Invalid Operation
- Disabled Zero Divide
- Disabled Overflow
- Disabled Underflow
- Disabled Inexact
- Enabled Overflow
- Enabled Underflow
- Enabled Inexact

Subsequent sections define each of the DFP exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of “traps” and “trap handlers”. In this architecture, a FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the “trap enabled” case: the expectation is that the exception will be detected by software, which will revise the result. A FPSCR exception enable bit of 0 causes generation of the “default result” value specified for the “trap disabled” (or “no trap occurs” or “trap is not implemented”) case: the expectation is that the exception will not be detected by software, which will simply use the default result. The result to be delivered in each case for each exception is described in the sections below.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is desired for all exceptions, all FPSCR exception enable bits should be set to zero and Ignore Exceptions Mode (see below) should be used. In this case the system floating-point enabled exception error handler is not invoked, even if DFP exceptions occur: software can inspect the FPSCR exception bits if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to one and a mode other than Ignore Exceptions Mode must be used. In this case the system floating-point enabled exception error handler is invoked if an enabled DFP exception occurs. The system floating-point enabled exception error handler is also invoked if a *Move To FPSCR* instruction causes an exception bit and the corresponding enable bit both to be 1; the *Move To FPSCR* instruction is considered to cause the enabled exception.

The FE0 and FE1 bits control whether and how the system floating-point enabled exception error handler is invoked if an enabled DFP exception occurs. The location of these bits and the requirements for altering them are described in Book III, *Power ISA Operating Environment Architecture*. (The system floating-point enabled exception error handler is never invoked

because of a disabled DFP exception.) The effects of the four possible settings of these bits are as follows.

FE0	FE1	Description
0	0	**Ignore Exceptions Mode** DFP exceptions do not cause the system floating-point enabled exception error handler to be invoked.
0	1	**Imprecise Nonrecoverable Mode** The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. It may not be possible to identify the excepting instruction or the data that caused the exception. Results produced by the excepting instruction may have been used by or may have affected subsequent instructions that are executed before the error handler is invoked.
1	0	**Imprecise Recoverable Mode** The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. Sufficient information is provided to the error handler that it can identify the excepting instruction and the operands, and correct the result. No results produced by the excepting instruction have been used by or have affected subsequent instructions that are executed before the error handler is invoked.
1	1	**Precise Mode** The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.

In all cases, the question of whether a DFP result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by the value of the FE0 and FE1 bits.

In all cases in which the system floating-point enabled exception error handler is invoked, all instructions before the instruction at which the system floating-point enabled exception error handler is invoked have completed, and no instruction after the instruction at which the system floating-point enabled exception error handler is invoked has begun execution. (Recall that, for the two Imprecise modes, the instruction at which the system floating-point enabled exception error handler is invoked need not be the instruction that caused the exception.) The instruction at which the system floating-point enabled exception error handler is invoked has not been executed unless it is the excepting instruction, in which case it has been executed if the exception is not among those listed on page 185 as suppressed.

Programming Note

In the ignore and both imprecise modes, a *Floating-Point Status and Control Register* instruction can be used to force any exceptions, due to instructions initiated before the *Floating-Point Status and Control Register* instruction, to be recorded in the FPSCR. (This forcing is superfluous for Precise Mode.)

In either of the Imprecise modes, a *Floating-Point Status and Control Register* instruction can be used to force any invocations of the system floating-point enabled exception error handler, due to instructions initiated before the *Floating-Point Status and Control Register* instruction, to occur. (This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.)

In order to obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines.

- If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used with all FPSCR exception enable bits set to zero.
- If the IEEE default results are not acceptable to the application, Imprecise Nonrecoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with FPSCR exception enable bits set to one for those exceptions for which the system floating-point enabled exception error handler is to be invoked.
- Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to one.
- Precise Mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

5.5.10.1 Invalid Operation Exception

Definition

An Invalid Operation Exception occurs when an operand is invalid for the specified DFP operation. The invalid DFP operations are:

- Any DFP operation on a signaling NaN (SNaN), except for *Test*, *Round To DFP Short*, *Convert To DFP Long*, *Decode DPD To BCD*, *Extract Biased Exponent*, *Insert Biased Exponent*, *Shift Significand Left Immediate*, and *Shift Significand Right Immediate*

- For add or subtract operations, magnitude subtraction of infinities (+∞) + (-∞)
- Division of infinity by infinity (∞ ÷ ∞)
- Division of zero by zero (0 ÷ 0)
- Multiplication of infinity by zero (∞ % 0)
- Ordered comparison involving a NaN (Invalid Compare)
- The *Quantize* operation detects that the significand associated with the specified target exponent would have more significant digits than the target-format precision
- For the *Quantize* operation, when one source operand specifies an infinity and the other specifies a finite number
- The *Reround* operation detects that the target exponent associated with the specified target significance would be greater than X_{max}
- The *Encode BCD To DPD* operation detects an invalid BCD digit or sign code
- The *Convert To Fixed* operation involving a number too large in magnitude to be represented in the target format, or involving a NaN.

Programming Note

In addition, an Invalid Operation Exception occurs if software explicitly requests this by executing an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction that sets $FPSCR_{VXSOFT}$ to 1 (Software Request). The purpose of $FPSCR_{VXSOFT}$ is to allow software to cause an Invalid Operation Exception for a condition that is not necessarily associated with the execution of a DFP instruction. For example, it might be set by a program that computes a square root, if the source operand is negative.

Action

The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the FPSCR.

When Invalid Operation Exception is enabled ($FPSCR_{VE}$=1) and Invalid Operation occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set:
   - $FPSCR_{VXSNAN}$ (if SNaN)
   - $FPSCR_{VXISI}$ (if ∞ - ∞)
   - $FPSCR_{VXIDI}$ (if ∞ ÷ ∞)
   - $FPSCR_{VXZDZ}$ (if 0 ÷ 0)
   - $FPSCR_{VXIMZ}$ (if ∞ % 0)
   - $FPSCR_{VXVC}$ (if invalid comp)
   - $FPSCR_{VXCVI}$ (if invalid conversion)
2. If the operation is an arithmetic, quantum-adjustment, conversion, or format,
   the target FPR is unchanged,
   $FPSCR_{FR\ FI}$ are set to zero, and
   $FPSCR_{FPRF}$ is unchanged.
3. If the operation is a compare,
   $FPSCR_{FR\ FI\ C}$ are unchanged, and
   $FPSCR_{FPCC}$ is set to reflect unordered.

When Invalid Operation Exception is disabled ($FPSCR_{VE}$=0) and Invalid Operation occurs, the following actions are taken:

1. One or two Invalid Operation Exceptions are set:
   - $FPSCR_{VXSNAN}$ (if SNaN)
   - $FPSCR_{VXISI}$ (if ∞ - ∞)
   - $FPSCR_{VXIDI}$ (if ∞ ÷ ∞)
   - $FPSCR_{VXZDZ}$ (if 0 ÷ 0)
   - $FPSCR_{VXIMZ}$ (if ∞ x 0)
   - $FPSCR_{VXVC}$ (if invalid comp)
   - $FPSCR_{VXCVI}$ (if invalid conversion)
2. If the operation is an arithmetic, quantum-adjustment, *Round to DFP Long*, *Convert to DFP Extended*, or format
   the target FPR is set to a Quiet NaN
   $FPSCR_{FR\ FI}$ are set to zero
   $FPSCR_{FPRF}$ is set to indicate the class of the result (Quiet NaN)
3. If the operation is a *Convert To Fixed*
   the target FPR is set as follows:
   FRT is set to the most positive 64-bit binary integer if the operand in FRB is a positive or +∞, and to the most negative 64-bit binary integer if the operand in FRB is a negative number, -∞ , or NaN.
   $FPSCR_{FR\ FI}$ are set to zero
   $FPSCR_{FPRF}$ is unchanged
4. If the operation is a compare,
   $FPSCR_{FR\ FI\ C}$ are unchanged
   $FPSCR_{FPCC}$ is set to reflect unordered

5.5.10.2 Zero Divide Exception

Definition

A Zero Divide Exception occurs when a Divide instruction is executed with a zero divisor value and a finite nonzero dividend value.

Action

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

When Zero Divide Exception is enabled ($FPSCR_{ZE}$=1) and Zero Divide occurs, the following actions are taken:

1. Zero Divide Exception is set
   $FPSCR_{ZX} \leftarrow 1$
2. The target FPR is unchanged
3. $FPSCR_{FR\ FI}$ are set to zero
4. $FPSCR_{FPRF}$ is unchanged

When Zero Divide Exception is disabled ($FPSCR_{ZE}$=0) and Zero Divide occurs, the following actions are taken:

1. Zero Divide Exception is set
   $FPSCR_{ZX} \leftarrow 1$
2. The target FPR is set to ±∞, where the sign is determined by the XOR of the signs of the operands

3. $FPSCR_{FR\ FI}$ are set to zero
4. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (±∞)

5.5.10.3 Overflow Exception

Definition

An overflow exception occurs whenever the target format's largest finite number is exceeded in magnitude by what would have been the rounded result if the exponent range were unbounded.

Action

Except for *Reround*, the following describes the handling of the IEEE overflow exception condition. The *Reround* operation does not recognize an overflow exception condition.

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.

When Overflow Exception is enabled ($FPSCR_{OE}$=1) and overflow occurs, the following actions are taken:

1. Overflow Exception is set
   $FPSCR_{OX} \leftarrow 1$
2. The infinitely precise result is divided by 10^{α}. That is, the exponent adjustment α is subtracted from the exponent. This is called the *wrapped result*. The exponent adjustment for all operations, except for *Round To DFP Short* and *Round To DFP Long*, is 576 for DFP Long and 9216 for DFP Extended. For *Round To DFP Short* and *Round To DFP Long*, the exponent adjustment is 192 for the source format of DFP Long and 3072 for the source format of DFP Extended.
3. The wrapped result is rounded to the target-format precision. This is called the *wrapped rounded result*.
4. If the wrapped rounded result has only one form, it is the delivered result. If the wrapped rounded result has redundant forms and is exact, the result of the form that has the exponent closest to the wrapped ideal exponent is returned. If the wrapped rounded result has redundant forms and is inexact, the result of the form that has the smallest exponent is returned. The wrapped ideal exponent is the result of subtracting the exponent adjustment from the ideal exponent.
5. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Normal Number)

When Overflow Exception is disabled ($FPSCR_{OE}$=0) and overflow occurs, the following actions are taken:

1. Overflow Exception is set
   $FPSCR_{OX} \leftarrow 1$
2. Inexact Exception is set
   $FPSCR_{XX} \leftarrow 1$
3. The result is determined by the rounding mode and the sign of the intermediate result as follows.

Rounding Mode	Sign of intermediate result: Plus	Sign of intermediate result: Minus
Round to Nearest, Ties to Even	+∞	-∞
Round toward 0	$+N_{max}$	$-N_{max}$
Round toward +∞	+ ∞	$-N_{max}$
Round toward - ∞	$+N_{max}$	-∞
Round to Nearest, Ties away from 0	+∞	-∞
Round to Nearest, Ties toward 0	+∞	-∞
Round away from 0	+∞	-∞
Round to prepare for shorter precision	$+N_{max}$	$-N_{max}$

Figure 75. Overflow Results When Exception Is Disabled

4. The result is placed into the target FPR
5. $FPSCR_{FR}$ is set to one if the returned result is ± ∞, and is set to zero if the returned result is $\pm N_{max}$
6. $FPSCR_{FI}$ is set to one
7. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± ∞ or ± Normal number)

5.5.10.4 Underflow Exception

Definition

Except for *Reround*, the following describes the handling of the IEEE underflow exception condition. The *Reround* operation does not recognize an underflow exception condition.

The Underflow Exception is defined differently for the enabled and disabled states. However, a tininess condition is recognized in both states when a result computed as though both the precision and exponent range were unbounded would be nonzero and less than the target format's smallest normal number, $N_{min,}$ in magnitude.

Unless otherwise defined in the instruction description, an underflow exception occurs as follows:

- Enabled:
  When the tininess condition is recognized.
- Disabled:
  When the tininess condition is recognized and when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

Action

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

When Underflow Exception is enabled ($FPSCR_{UE}$=1) and underflow occurs, the following actions are taken:

1. Underflow Exception is set
   $FPSCR_{UX} \leftarrow 1$
2. The infinitely precise result is multiplied by 10^{α}. That is, the exponent adjustment α is added to the exponent. This is called the *wrapped result*. The exponent adjustment for all operations, except for *Round To DFP Short* and *Round To DFP Long*, is 576 for DFP Long and 9216 for DFP Extended. For *Round To DFP Short* and *Round To DFP Long*, the exponent adjustment is 192 for the source format of DFP Long and 3072 for the source format of DFP Extended.
3. The wrapped result is rounded to the target-format precision. This is called the *wrapped rounded result*.
4. If the wrapped rounded result has only one form, it is the delivered result. If the wrapped rounded result has redundant forms and is exact, the result of the form that has the exponent closest to the wrapped ideal exponent is returned. If the wrapped rounded result has redundant forms and is inexact, the result of the form that has the smallest exponent is returned. The wrapped ideal exponent is the result of adding the exponent adjustment to the ideal exponent.
5. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Normal number)

When Underflow Exception is disabled ($FPSCR_{UE}$=0) and underflow occurs, the following actions are taken:

1. Underflow Exception is set
   $FPSCR_{UX} \leftarrow 1$
2. The infinitely precise result is rounded to the target-format precision.
3. The rounded result is returned. If this result has redundant forms, the result of the form that is closest to the ideal exponent is returned.
4. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Normal number, ± Subnormal Number, or ± Zero)

5.5.10.5 Inexact Exception

Definition

Except for *Round to FP Integer Without Inexact*, the following describes the handling of the IEEE inexact exception condition. The *Round to FP Integer Without Inexact* does not recognize an inexact exception condition.

An Inexact Exception occurs when either of two conditions occur during rounding:

1. The delivered result differs from what would have been computed were both the precision and exponent range unbounded.
2. The rounded result overflows and Overflow Exception is disabled.

Action

The action to be taken does not depend on the setting of the Inexact Exception Enable bit of the FPSCR.

When Inexact Exception occurs, the following actions are taken:

1. Inexact Exception is set
   $FPSCR_{XX} \leftarrow 1$
2. The rounded or overflowed result is placed into the target FPR
3. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result

Programming Note

In some implementations, enabling Inexact Exceptions may degrade performance more than does enabling other types of floating-point exception.

5.5.11 Summary of Normal Rounding And Range Actions

Figure 76 and Figure 77 summarize rounding and range actions, with the following exceptions:

- The *Reround* operation recognizes neither an underflow nor an overflow exception.
- The *Round to FP Integer Without Inexact* operation does not recognize the inexact operation exception.

Range of v	Case	Result (r) when Rounding Mode Is							
		RNE	RNTZ	RNAZ	RAFZ	RTMI	RFSP	RTPI	RTZ
v < -Nmax, q < -Nmax	Overflow	$-\infty^1$	$-\infty^1$	$-\infty^1$	$-\infty^1$	$-\infty^1$	-Nmax	-Nmax	-Nmax
v < -Nmax, q = -Nmax	Normal	-Nmax	-Nmax	-Nmax	—	—	-Nmax	-Nmax	-Nmax
-Nmax ≤ v ≤ -Nmin	Normal	b	b	b	b	b	b	b	b
-Nmin < v ≤ -Dmin	Tiny	b*	b*	b*	b*	b*	b*	b	b
-Dmin < v < -Dmin/2	Tiny	-Dmin	-Dmin	-Dmin	-Dmin	-Dmin	-Dmin	-0	-0
v = -Dmin/2	Tiny	-0	-0	-Dmin	-Dmin	-Dmin	-Dmin	-0	-0
-Dmin/2 < v < 0	Tiny	-0	-0	-0	-Dmin	-Dmin	-Dmin	-0	-0
v = 0	EZD	+0	+0	+0	+0	-0	+0	+0	+0
0 < v < +Dmin/2	Tiny	+0	+0	+0	+Dmin	+0	+Dmin	+Dmin	+0
v = +Dmin/2	Tiny	+0	+0	+Dmin	+Dmin	+0	+Dmin	+Dmin	+0
+Dmin/2 < v < +Dmin	Tiny	+Dmin	+Dmin	+Dmin	+Dmin	+0	+Dmin	+Dmin	+0
+Dmin ≤ v < +Nmin	Tiny	b*	b*	b*	b*	b	b*	b*	b
+Nmin ≤ v ≤ +Nmax	Normal	b	b	b	b	b	b	b	b
+Nmax < v, q = +Nmax	Normal	+Nmax	+Nmax	+Nmax	—	+Nmax	+Nmax	—	+Nmax
+Nmax < v, q > +Nmax	Overflow	$+\infty^1$	$+\infty^1$	$+\infty^1$	$+\infty^1$	+Nmax	+Nmax	$+\infty^1$	+Nmax

Explanation:

—	This situation cannot occur.
1	The normal result r is considered to have been incremented.
*	The rounded value, in the extreme case, may be Nmin. In this case, the exception conditions are underflow, inexact, and incremented.
b	The value derived when the precise result v is rounded to the destination's precision, including both bounded precision and bounded exponent range.
q	The value derived when the precise result v is rounded to the destination's precision, but assuming an unbounded exponent range.
r	This is the returned value when neither overflow nor underflow is enabled.
v	Precise result before rounding, assuming unbounded precision and an unbounded exponent range. For data-format conversion operations, v is the source value.
Dmin	Smallest (in magnitude) representable subnormal number in the target format.
EZD	The result r of the exact-zero-difference case applies only to ADD and SUBTRACT with both source operands having opposite signs. (For ADD and SUBTRACT, when both source operands have the same sign, the sign of the zero result is the same sign as the sign of the source operands.)
Nmax	Largest (in magnitude) representable finite number in the target format.
Nmin	Smallest (in magnitude) representable normalized number in the target format.
RAFZ	Round away from 0.
RFSP	Round to Prepare for Shorter Precision.
RNAZ	Round to Nearest, Ties away from 0.
RNE	Round to Nearest, Ties to even.
RNTZ	Round to Nearest, Ties toward 0.
RTPI	Round toward +∞.
RTMI	Round toward -∞.
RTZ	Round toward 0.

Figure 76. Rounding and Range Actions (Part 1)

Case	Is r inexact (r≠v)	OE=1	UE=1	XE=1	Is r Incre-mented (|r|>|v|)	Is q inexact (q≠v)	Is q Incre-mented (|q|>|v|)	Returned Results and Status Setting*
Overflow	Yes[1]	No	—	No	No	—	—	T(r), OX← 1, FI← 1, FR← 0, XX ← 1
Overflow	Yes[1]	No	—	No	Yes	—	—	T(r), OX← 1, FI← 1, FR← 1, XX ← 1
Overflow	Yes[1]	No	—	Yes	No	—	—	T(r), OX← 1, FI← 1, FR← 0, XX ← 1, TX
Overflow	Yes[1]	No	—	Yes	Yes	—	—	T(r), OX← 1, FI← 1, FR← 1, XX ← 1, TX
Overflow	Yes[1]	Yes	—	—	—	No	No[1]	Tw(q÷β), OX← 1, FI← 0, FR← 0, TO
Overflow	Yes[1]	Yes	—	—	—	Yes	No	Tw(q÷β), OX← 1, FI← 1, FR← 0, XX← 1,TO
Overflow	Yes[1]	Yes	—	—	—	Yes	Yes	Tw(q÷β), OX← 1, FI← 1, FR← 1, XX← 1,TO
Normal	No	—	—	—	—	—	—	T(r), FI← 0, FR← 0
Normal	Yes	—	—	No	No	—	—	T(r), FI← 1, FR← 0, XX ← 1
Normal	Yes	—	—	No	Yes	—	—	T(r), FI← 1, FR← 1, XX ← 1
Normal	Yes	—	—	Yes	No	—	—	T(r), FI← 1, FR← 0, XX ← 1, TX
Normal	Yes	—	—	Yes	Yes	—	—	T(r), FI← 1, FR← 1, XX ← 1, TX
Tiny	No	—	No	—	—	—	—	T(r), FI← 0, FR← 0
Tiny	No	—	Yes	—	—	No[1]	No[1]	Tw(q•β), UX← 1, FI← 0, FR← 0, TU
Tiny	Yes	—	No	No	No	—	—	T(r), UX← 1, FI← 1, FR← 0, XX ← 1
Tiny	Yes	—	No	No	Yes	—	—	T(r), UX← 1, FI← 1, FR← 1, XX ← 1
Tiny	Yes	—	No	Yes	No	—	—	T(r), UX← 1, FI← 1, FR← 0, XX ← 1, TX
Tiny	Yes	—	No	Yes	Yes	—	—	T(r), UX← 1, FI← 1, FR← 1, XX ← 1, TX
Tiny	Yes	—	Yes	—	—	No	No[1]	Tw(q•β), UX← 1, FI← 0, FR← 0, TU
Tiny	Yes	—	Yes	—	—	Yes	No	Tw(q•β), UX← 1, FI← 1, FR← 0, XX ← 1,TU
Tiny	Yes	—	Yes	—	—	Yes	Yes	Tw(q•β), UX← 1, FI← 1, FR← 1, XX ← 1,TU

Explanation:

—	The results do not depend on this condition.
1	This condition is true by virtue of the state of some condition to the left of this column.
*	Rounding sets only the FI and FR status flags. Setting of the OX, XX, or UX flag is part of the exception actions. They are listed here for reference.
β	Wrap adjust, which depends on the type of operation and operand format. For all operations except *Round to DFP Short* and *Round to DFP Long*, the wrap adjust depends on the target format: $\beta = 10^{\alpha}$, where α is 576 for DFP Long, and 9216 for DFP Extended. For *Round to DFP Short* and *Round to DFP Long*, the wrap adjust depends on the source format: $\beta = 10^{\kappa}$ where κ is 192 for DFP Long and 3072 for DFP Extended.
q	The value derived when the precise result v is rounded to destination's precision, but assuming an unbounded exponent range.
r	The result as defined in Part 1 of this figure.
v	Precise result before rounding, assuming unbounded precision and unbounded exponent range.
FI	Floating-Point-Fraction-Inexact status flag, $FPSCR_{FI}$. This status flag is non-sticky.
FR	Floating-Point-Fraction-Rounded status flag, $FPSCR_{FR}$.
OX	Floating-Point Overflow Exception status flag, $FPSCR_{oX}$.
TO	The system floating-point enabled exception error handler is invoked for the overflow exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
TU	The system floating-point enabled exception error handler is invoked for the underflow exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
TX	The system floating-point enabled exception error handler is invoked for the inexact exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
T(x)	The value x is placed at the target operand location.
Tw(x)	The wrapped rounded result x is placed at the target operand location. For all operations except data format conversions, the wrapped rounded result is in the same format and length as normal results at the target location. For data format conversions, the wrapped rounded result is in the same format and length as the source, but rounded to the target-format precision.
UX	Floating-Point-Underflow-Exception status flag, $FPSCR_{UX}$
XX	Float-Point-Inexact-Exception Status flag, $FPSCR_{XX}$. The flag is a sticky version of $FPSCR_{FI}$. When $FPSCR_{FI}$ is set to a new value, the new value of $FPSCR_{XX}$ is set to the result of ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.

Figure 77. Rounding and Range Actions (Part 2)

5.6 DFP Instruction Descriptions

The following sections describe the DFP instructions. When a 128-bit operand is used, it is held in a FPR pair and the instruction mnemonic uses a letter “q” to mean the quad-precision operation. Note that in the following descriptions, FPXp denotes a FPR pair and must address an even-odd pair. If the FPXp field specifies an odd-numbered register, then the instruction form is invalid. The notation FPX[p] means either a FPR, FPX, or a FPR pair, FPXp.

For DFP instructions, if a DFP operand is returned, the trailing significand field of the target operand is encoded using preferred DPD codes.

5.6.1 DFP Arithmetic Instructions

All DFP arithmetic instructions are X-form instructions. They all set the FI and FR status flags, and also set the $FPSCR_{FPRF}$ field. Furthermore, they all have an ideal exponent assigned and employ the record bit (Rc).

The arithmetic instructions consist of Add, Divide, Multiply, and Subtract.

DFP Add [Quad] *X-form*

dadd FRT,FRA,FRB (Rc=0)
dadd. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	2	Rc
0	6	11	16	21	31

daddq FRTp,FRAp,FRBp (Rc=0)
daddq. FRTp,FRAp,FRBp (Rc=1)

63	FRTp	FRAp	FRBp	2	Rc
0	6	11	16	21	31

The DFP operand in FRA[p] is added to the DFP operand in FRB[p].

The result is rounded to the target-format precision under control of the DRN (bits 29:31) of the FPSCR. An appropriate form of the rounded result is selected based on the ideal exponent and is placed in FRT[p]. The ideal exponent is the smaller exponent of the two source operands.

Figure 78 summarizes the actions for Add. Figure 78 does not include the setting of the $FPSCR_{FPRF}$ field. The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1 (if Rc=1)

DFP Subtract [Quad] *X-form*

dsub FRT,FRA,FRB (Rc=0)
dsub. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	514	Rc
0	6	11	16	21	31

dsubq FRTp,FRAp,FRBp (Rc=0)
dsubq. FRTp,FRAp,FRBp (Rc=1)

63	FRTp	FRAp	FRBp	514	Rc
0	6	11	16	21	31

The DFP operand in FRB[p] is subtracted from the DFP operand in FRA[p].

The result is rounded to the target-format precision under control of the DRN (bits 29:31) of the FPSCR. An appropriate form of the rounded result is selected based on the ideal exponent and is placed in FRT[p]. The ideal exponent is the smaller exponent of the two source operands.

The execution of Subtract is identical to that of Add, except that the operand in FRB participates in the operation with its sign bit inverted. See Figure 78. The table does not include the setting of the $FPSCR_{FPRF}$ field. The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1 (if Rc=1)

Operand a in FRA[p] is	Actions for Add (a + b) when operand b in FRB[p] is				
	-∞	**F**	**+∞**	**QNaN**	**SNaN**
-∞	T(-dINF)	T(-dINF)	V_{XISI}: T(dNaN)	P(b)	V_{XSNAN}: U(b)
F	T(-dINF)	S(a + b)	T(+dINF)	P(b)	V_{XSNAN}: U(b)
+∞	V_{XISI}: T(dNaN)	T(+dINF)	T(+dINF)	P(b)	V_{XSNAN}: U(b)
QNaN	P(a)	P(a)	P(a)	P(a)	V_{XSNAN}: U(b)
SNaN	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)

Explanation:

a + b	The value a added to b, rounded to the target-format precision and returned in the appropriate form. (See Section 5.5.11 on page 191)
+dINF	Default plus infinity.
- dINF	Default minus infinity.
dNaN	Default quiet NaN.
F	All finite numbers, including zeros.
P(x)	The QNaN of operand x is propagated and placed in FRT[p].
S(x)	The value x is placed in FRT[p] with the sign set by the rules of algebra. When the source operands have the same sign, the sign of the result is the same as the sign of the operands, including the case when the result is zero. When the operands have opposite signs, the sign of a zero result is positive in all rounding modes, except round toward -∞, in which case, the sign is minus.
T(x)	The value x is placed in FRT[p].
U(x)	The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
V_{XISI}	The Invalid-Operation Exception (VXISI) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 187 for the exception actions.)
V_{XSNAN}	The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 187 for the exception actions.)

Figure 78. Actions: Add

DFP Multiply [Quad] *X-form*

dmul FRT,FRA,FRB (Rc=0)
dmul. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	34	Rc
0	6	11	16	21	31

dmulq FRTp,FRAp,FRBp (Rc=0)
dmulq. FRTp,FRAp,FRBp (Rc=1)

63	FRTp	FRAp	FRBp	34	Rc
0	6	11	16	21	31

The DFP operand in FRA[p] is multiplied by the DFP operand in FRB[p].

The result is rounded to the target-format precision under control of the DRN (bits 29:31) of the FPSCR. An appropriate form of the rounded result is selected based on the ideal exponent and is placed in FRT[p]. The ideal exponent is the sum of the two exponents of the source operands.

Figure 79 summarizes the actions for Multiply. Figure 79 does not include the setting of the $FPSCR_{FPRF}$ field. The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

Special Registers Altered:

FPRF FR FI
FX OX UX XX VXSNAN VXIMZ
CR1 (if Rc=1)

Operand a in FRA[p] is	Actions for Multiply (a*b) when operand b in FRB[p] is				
	0	**Fn**	**∞**	**QNaN**	**SNaN**
0	S(a * b)	S(a * b)	V_{XIMZ}: T(dNaN)	P(b)	V_{XSNAN}: U(b)
Fn	S(a * b)	S(a * b)	S(dINF)	P(b)	V_{XSNAN}: U(b)
∞	V_{XIMZ}: T(dNaN)	S(dINF)	S(dINF)	P(b)	V_{XSNAN}: U(b)
QNaN	P(a)	P(a)	P(a)	P(a)	V_{XSNAN}: U(b)
SNaN	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)

Explanation:

a * b	The value a multiplied by b, rounded to the target-format precision and returned in the appropriate form. (See Section 5.5.11 on page 191)
dINF	Default infinity.
dNaN	Default quiet NaN.
Fn	Finite nonzero number (includes both normal and subnormal numbers).
P(x)	The QNaN of operand x is propagated and placed in FRT[p].
S(x)	The value x is placed in FRT[p] with the sign set to the exclusive-OR of the source-operand signs.
T(x)	The value x is placed in FRT[p].
U(x)	The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
V_{XIMZ}:	The Invalid-Operation Exception (VXIMZ) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 187 for the exception actions.)
V_{XSNAN}:	The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 187 for the exception actions.)

Figure 79. Actions: Multiply

DFP Divide [Quad] *X-form*

ddiv FRT,FRA,FRB (Rc=0)
ddiv. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	546	Rc
0	6	11	16	21	31

ddivq FRTp,FRAp,FRBp (Rc=0)
ddivq. FRTp,FRAp,FRBp (Rc=1)

63	FRTp	FRAp	FRBp	546	Rc
0	6	11	16	21	31

The DFP operand in FRA[p] is divided by the DFP operand in FRB[p].

The result is rounded to the target-format precision under control of the DRN (bits 29:31) of the FPSCR. An appropriate form of the rounded result is selected based on the ideal exponent and is placed in FRT[p]. The ideal exponent is the difference of subtracting the exponent of the divisor from the exponent of the dividend.

Figure 80 summarizes the actions for Divide. Figure 80 does not include the setting of the $FPSCR_{FPRF}$ field. The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation and enabled zero-divide exceptions, in which cases the field remains unchanged.

Special Registers Altered:

FPRF FR FI
FX OX UX ZX XX
VXSNAN VXIDI VXZDZ
CR1 (if Rc=1)

Operand a in FRA[p] is	Actions for Divide (a ÷ b) when operand b in FRB[p] is				
	0	**Fn**	**∞**	**QNaN**	**SNaN**
0	V_{XZDZ}: T(dNaN)	S(a ÷ b)	S(zt)	P(b)	V_{XSNAN}: U(b)
Fn	Zx: S(dINF)	S(a ÷ b)	S(zt)	P(b)	V_{XSNAN}: U(b)
∞	S(dINF)	S(dINF)	V_{XIDI}: T(dNaN)	P(b)	V_{XSNAN}: U(b)
QNaN	P(a)	P(a)	P(a)	P(a)	V_{XSNAN}: U(b)
SNaN	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)

Explanation:

a ÷ b	The value a divided by b, rounded to the target-format precision and returned in the appropriate form. (See Section 5.5.11 on page 191.)
dINF	Default infinity.
dNaN	Default quiet NaN.
Fn	Finite nonzero number (includes both normal and subnormal numbers).
P(x)	The QNaN of operand x is propagated and placed in FRT[p].
S(x)	The value x is placed in FRT[p] with the sign set to the exclusive-OR of the source-operand signs.
T(x)	The value x is placed in FRT[p].
U(x)	The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
V_{XIDI}:	The Invalid-Operation Exception (VXIDI) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 187 for the exception actions.)
$V_{XSNAN:}$	The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 187 for the exception actions.)
$V_{XZDZ:}$	The Invalid-Operation Exception (VXZDZ) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 “Invalid Operation Exception” on page 187 for the exception actions.)
zt	True zero (zero significand and most negative exponent).
Zx	The Zero-Divide Exception occurs. The result is produced only when the exception is disabled (See Section 5.5.10.2 “Zero Divide Exception” on page 188 for the exception actions.)

Figure 80. Actions: Divide

5.6.2 DFP Compare Instructions

The DFP compare instructions consist of the *Compare Ordered* and *Compare Unordered* instructions. The compare instructions do not provide the record bit.

The comparison sets the designated CR field to indicate the result. The $FPSCR_{FPCC}$ is set in the same way.

The codes in the CR field BF and $FPSCR_{FPCC}$ are defined for the DFP compare operations as follows.

Bit	Name	Description
0	FL	(FRA[p]) < (FRB[p])
1	FG	(FRA[p]) > (FRB[p])
2	FE	(FRA[p]) = (FRB[p])
3	FU	(FRA[p]) ? (FRB[p])

DFP Compare Unordered [Quad] X-form

dcmpu BF,FRA,FRB

59	BF	//	FRA	FRB	642	/
0	6	9	11	16	21	31

dcmpuq BF,FRAp,FRBp

63	BF	//	FRAp	FRBp	642	/
0	6	9	11	16	21	31

The DFP operand in FRA[p] is compared to the DFP operand in FRB[p]. The result of the compare is placed into CR field BF and the $FPSCR_{FPCC}$.

Special Registers Altered:

CR field BF
FPCC
FX VXSNAN

Operand a in FRA[p] is	Actions for Compare Unordered (a:b) when operand b in FRB[p] is				
	-∞	**F**	**+∞**	**QNaN**	**SNaN**
-∞	AeqB	AltB	AltB	AuoB	Fu, V_{XSNAN}
F	AgtB	C(a:b)	AltB	AuoB	Fu, V_{XSNAN}
+∞	AgtB	AgtB	AeqB	AuoB	Fu, V_{XSNAN}
QNaN	AuoB	AuoB	AuoB	AuoB	Fu, V_{XSNAN}
SNaN	Fu, V_{XSNAN}	Fu, V_{XSNAN}	Fu, V_{XSNAN}	Fu, V_{XSNAN}	Fu, V_{XSNAN}

Explanation:

C(a:b)	Algebraic comparison. See the table below.
F	All finite numbers, including zeros.
AeqB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0010.
AgtB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0100.
AltB	CR field BF and $FPSCR_{FPCC}$ are set to 0b1000.
AuoB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0001.
V_{XSNAN}	The invalid-operation exception (VXSNAN) occurs. See Section 5.5.10.1 for actions.

Relation of Value a to Value b	Action for C(a:b)
a = b	AeqB
a < b	AltB
a > b	AgtB

Figure 81. Actions: Compare Unordered

DFP Compare Ordered [Quad] X-form

dcmpo BF,FRA,FRB

59	BF	//	FRA	FRB	130	/
0	6	9	11	16	21	31

dcmpoq BF,FRAp,FRBp

63	BF	//	FRAp	FRBp	130	/
0	6	9	11	16	21	31

The DFP operand in FRA[p] is compared to the DFP operand in FRB[p]. The result of the compare is placed into CR field BF and the $FPSCR_{FPCC}$.

Special Registers Altered:

CR field BF
FPCC
FX VXSNAN VXVC

Operand a in FRA[p] is	**Actions for Compare ordered (a:b) when operand b in FRB[p] is**				
	-∞	**F**	**+∞**	**QNaN**	**SNaN**
-∞	AeqB	AltB	AltB	AuoB, V_{XVC}	AuoB, V_{XSV}
F	AgtB	C(a:b)	AltB	AuoB, V_{XVC}	AuoB, V_{XSV}
+∞	AgtB	AgtB	AeqB	AuoB, V_{XVC}	AuoB, V_{XSV}
QNaN	AuoB, V_{XVC}	AuoB, V_{XVC}	AuoB, V_{XVC}	AuoB, V_{XVC}	AuoB, V_{XSV}
SNaN	AuoB, V_{XSV}	AuoB, V_{XSV}	AuoB, V_{XSV}	AuoB, V_{XSV}	AuoB, V_{XSV}

Explanation:

C(a:b)	Algebraic comparison. See the table below
F	All finite numbers, including zeros
AeqB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0010.
AgtB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0100.
AltB	CR field BF and $FPSCR_{FPCC}$ are set to 0b1000.
AuoB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0001.
V_{XSV}	The invalid-operation exception (VXSNAN) occurs. Additionally, if the exception is disabled ($FPSCR_{VE}$=0), then $FPSCR_{VXVC}$ is also set to one. See Section 5.5.10.1 for actions.
V_{XVC}	The invalid-operation exception (VXVC) occurs. See Section 5.5.10.1 for actions.

Relation of Value a to Value b	**Action for C(a:b)**
a = b	AeqB
a < b	AltB
a > b	AgtB

Figure 82. Actions: Compare Ordered

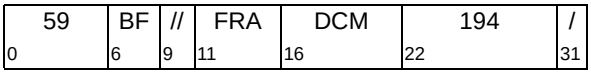
5.6.3 DFP Test Instructions

The DFP test instructions consist of the *Test Data Class*, *Test Data Group*, *Test Exponent*, and *Test Significance* instructions, and they do not provide the record bit.

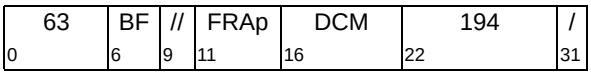
The test instructions set the designated CR field to indicate the result. The $FPSCR_{FPCC}$ is set in the same way.

DFP Test Data Class [Quad] Z22-form

dtstdc BF,FRA,DCM



dtstdcq BF,FRAp,DCM



Let the DCM (Data Class Mask) field specify one or more of the 6 possible data classes, where each bit corresponds to a specific data class.

DCM Bit	Data Class
0	Zero
1	Subnormal
2	Normal
3	Infinity
4	Quiet NaN
5	Signaling NaN

CR field BF and $FPSCR_{FPCC}$ are set to indicate the sign of the DFP operand in FRA[p] and whether the data class of the DFP operand in FRA[p] matches any of the data classes specified by DCM.

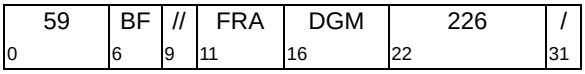
Field	Meaning
0000	Operand positive with no match
0010	Operand positive with match
1000	Operand negative with no match
1010	Operand negative with match

Special Registers Altered:

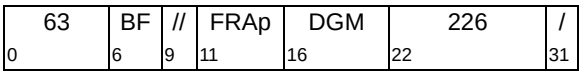
CR field BF
FPCC

DFP Test Data Group [Quad] Z22-form

dtstdg BF,FRA,DGM



dtstdgq BF,FRAp,DGM



Let the DGM (Data Group Mask) field specify one or more of the 6 possible data groups, where each bit corresponds to a specific data group.

The term extreme exponent means either the maximum exponent, X_{max}, or the minimum exponent, X_{min}.

DGM Bit	Data Group
0	Zero with non-extreme exponent
1	Zero with extreme exponent
2	Subnormal or (Normal with extreme exponent)
3	Normal with non-extreme exponent and leftmost zero digit in significand
4	Normal with non-extreme exponent and leftmost nonzero digit in significand
5	Special symbol (Infinity, QNaN, or SNaN)

CR field BF and $FPSCR_{FPCC}$ are set to indicate the sign of the DFP operand in FRA[p] and whether the data group of the DFP operand in FRA[p] matches any of the data groups specified by DGM.

Field	Meaning
0000	Operand positive with no match
0010	Operand positive with match
1000	Operand negative with no match
1010	Operand negative with match

Special Registers Altered:

CR field BF
FPCC

DFP Test Exponent [Quad] *X-form*

dtstex BF,FRA,FRB

59	BF	//	FRA	FRB	162	/
0	6	9	11	16	21	31

dtstexq BF,FRAp,FRBp

63	BF	//	FRAp	FRBp	162	/
0	6	9	11	16	21	31

The exponent value (Ea) of the DFP operand in FRA[p] is compared to the exponent value (Eb) of the DFP operand in FRB [p]. The result of the compare is placed into CR field BF and the $FPSCR_{FPCC.}$

The codes in the CR field BF and $FPSCR_{FPCC}$ are defined for the *DFP Test Exponent* operations as follows.

Bit	Description
0	Ea < Eb
1	Ea > Eb
2	Ea = Eb
3	Ea ? Eb

Special Registers Altered:

CR field BF
FPCC

Operand a in FRA[p] is	Actions for Test Exponent (Ea:Eb) when operand b in FRB[p] is			
	F	**∞**	**QNaN**	**SNaN**
F	C(Ea:Eb)	AuoB	AuoB	AuoB
∞	AuoB	AeqB	AuoB	AuoB
QNaN	AuoB	AuoB	AeqB	AeqB
SNaN	AuoB	AuoB	AeqB	AeqB

Explanation:

C(Ea:Eb)	Algebraic comparison. See the table below.
F	All finite numbers, including zeros
AeqB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0010.
AgtB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0100.
AltB	CR field BF and $FPSCR_{FPCC}$ are set to 0b1000.
AuoB	CR field BF and $FPSCR_{FPCC}$ are set to 0b0001.

Relation of Value Ea to Value Eb	Action for C(Ea:Eb)
Ea = Eb	AeqB
Ea < Eb	AltB
Ea > Eb	AgtB

Figure 83. Actions: Test Exponent

DFP Test Significance [Quad] *X-form*

dtstsf BF,FRA,FRB

59	BF	/	FRA	FRB	674	/
0	6	9	10	16	21	31

dtstsfq BF,FRA,FRBp

63	BF	/	FRA	FRBp	674	/
0	6	9	10	16	21	31

Let k be the contents of bits 58:63 of FPR[FRA] that specifies the reference significance.

For ***dtstsf***, let the value NSDb be the number of significant digits of the DFP value in FPR[FRB].

For ***dtstsfq***, let the value NSDb be the number of significant digits of the DFP value in FPR[FRBp:FRBp+1].

For this instruction, the number of significant digits of the value 0 is considered to be zero.

NSDb is compared to k. The result of the compare is placed into CR field BF and the FPCC as follows.

Bit	Description
0	k g 0 and k < NSDb
1	k g 0 and k > NSDb, or k = 0
2	k g 0 and k = NSDb
3	k ? NSDb

Special Registers Altered:

CR field BF
FPCC

Actions for Test Significance when the operand in VSR[FRB] or VSR[FRBp:FRBp+1] is			
F	**∞**	**QNaN**	**SNaN**
C(UIM:NSDb)	AuoB	AuoB	AuoB

Explanation:

C(k:NSDb)	Algebraic comparison. See the table below.
F	All finite numbers, including zeros.
AeqB	CR field BF and FPCC are set to 0b0010.
AgtB	CR field BF and FPCC are set to 0b0100.
AltB	CR field BF and FPCC are set to 0b1000.
AuoB	CR field BF and FPCC are set to 0b0001.

Relation of Value NSDb to Value k	Action for C(k:NSDb)
k g 0 and k = NSDb	AeqB
k g 0 and k < NSDb	AltB
k g 0 and k > NSDb, or k = 0	AgtB

Figure 84. Actions: Test Significance

Programming Note

The reference significance can be loaded into a FPR using a *Load Float as Integer Word Algebraic* instruction

DFP Test Significance Immediate [*Quad*] *X-form*

dtstsfi BF,UIM,FRB

59	BF	/	UIM	FRB	675	/
0	6	9	10	16	21	31

dtstsfiq BF,UIM,FRBp

63	BF	/	UIM	FRBp	675	/
0	6	9	10	16	21	31

Let the value UIM specify the reference significance.

For ***dtstsfi***, let the value NSDb be the number of significant digits of the DFP value in FPR[FRB].

For ***dtstsfiq***, let the value NSDb be the number of significant digits of the DFP value in FPR[FRBp:FRBp+1].

For this instruction, the number of significant digits of the value 0 is considered to be zero.

NSDb is compared to UIM. The result of the compare is placed into CR field BF and the FPCC as follows.

Bit	Description
0	UIM ≠ 0 and UIM < NSDb
1	UIM ≠ 0 and UIM > NSDb, or UIM = 0
2	UIM ≠ 0 and UIM = NSDb
3	UIM ? NSDb

Special Registers Altered:

CR field BF
FPCC

Actions for Test Significance when the operand in VSR[FRB] or VSR[FRBp:FRBp+1] is			
F	**∞**	**QNaN**	**SNaN**
C(UIM:NSDb)	AuoB	AuoB	AuoB

Explanation:

C(UIM:NSDb)	Algebraic comparison. See the table below.
F	All finite numbers, including zeros.
AeqB	CR field BF and FPCC are set to 0b0010.
AgtB	CR field BF and FPCC are set to 0b0100.
AltB	CR field BF and FPCC are set to 0b1000.
AuoB	CR field BF and FPCC are set to 0b0001.

Relation of Value NSDb to Value UIM	Action for C(UIM:NSDb)
UIM≠0 and UIM = NSDb	AeqB
UIM≠0 and UIM < NSDb	AltB
UIM≠0 and UIM > NSDb, or UIM = 0	AgtB

Figure 85. Actions: Test Significance

5.6.4 DFP Quantum Adjustment Instructions

The *Quantum Adjustment* operations consist of the *Quantize*, *Quantize Immediate*, *Reround*, and *Round To FP Integer* operations.

The *Quantum Adjustment* instructions are Z23-form instructions and have an immediate RMC (Rounding-Mode-Control) field, which specifies the rounding mode used. For *Quantize*, *Quantize Immediate*, and *Reround*, the RMC field contains the primary encoding. For *Round to FP Integer*, the field contains either primary or secondary encoding, depending on the setting of a RMC-encoding-selection bit. See Section 5.5.2 “Rounding Mode Specification” on page 183 for the definition of RMC encoding.

All *Quantum Adjustment* instructions set the FI and FR status flags, and also set the $FPSCR_{FPRF}$ field. The record bit is provided to each of these instructions. They return the target operand in a form with the ideal exponent.

DFP Quantize Immediate [Quad] Z23-form

dquai TE,FRT,FRB,RMC (Rc=0)
dquai. TE,FRT,FRB,RMC (Rc=1)

59	FRT	TE	FRB	RMC	67	Rc
0	6	11	16	21	23	31

dquaiq TE,FRTp,FRBp,RMC (Rc=0)
dquaiq. TE,FRTp,FRBp,RMC (Rc=1)

63	FRTp	TE	FRBp	RMC	67	Rc
0	6	11	16	21	23	31

The DFP operand in FRB[p] is converted and rounded to the form with the exponent specified by TE based on the rounding mode specified in the RMC field. TE is a 5-bit signed binary integer. The result of that form is placed in FRT[p]. The sign of the result is the same as the sign of the operand in FRB[p]. The ideal exponent is the exponent specified by TE.

When the value of the operand in FRB[p] is greater than $(10^{p}\text{-}1) \% 10^{TE}$, where p is the format precision, an invalid operation exception is recognized.

When the delivered result differs in value from the operand in FRB[p], an inexact exception is recognized. No underflow exception is recognized by this operation, regardless of the value of the operand in FRB[p].

The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

Special Registers Altered:

FPRF FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Programming Note

DFP Quantize Immediate can be used to adjust values to a form having the specified exponent in the range -16 to 15. If the adjustment requires the significand to be shifted left, then:

- if the result would cause overflow from the most significant digit, the result is a default QNaN.;
- otherwise the result is the adjusted value (left shifted with matching exponent).

If the adjustment requires the significand to be shifted right, the result is rounded based on the value of the RMC field.

DFP Quantize Immediate can round a value to a specific number of fractional digits. Consider the computation of sales tax. Values expressed in U.S. dollars have 2 fractional digits, and sales tax rates typically have 3 fractional digits. The product of value and rate will yield 5 fractional digits. For example:

39.95 * 0.075 = 2.99625

This result needs to be rounded to the penny to compute the correct tax of $3.00.

The following sequence computes the sales tax assuming the pre-tax total is in FRA and the tax rate is in FRB. The *DFP Quantize Immediate* instruction rounds the product (FRA * FRB) to 2 fractional digits (TE field = -2) using Round to nearest, ties away from 0 (RMC field = 2). The quantized and rounded result is placed in FRT.

```
dmul  f0,FRA,FRB
dquai -2,FRT,f0,2
```

DFP Quantize [Quad] *Z23-form*

dqua FRT,FRA,FRB,RMC (Rc=0)
dqua. FRT,FRA,FRB,RMC (Rc=1)

59	FRT	FRA	FRB	RMC	3	Rc
0	6	11	16	21	23	31

dquaq FRTp,FRAp,FRBp,RMC (Rc=0)
dquaq. FRTp,FRAp,FRBp,RMC (Rc=1)

63	FRTp	FRAp	FRBp	RMC	3	Rc
0	6	11	16	21	23	31

The DFP operand in register FRB[p] is converted and rounded to the form with the same exponent as that of the DFP operand in FRA[p] based on the rounding mode specified in the RMC field. The result of that form is placed in FRT[p]. The sign of the result is the same as the sign of the operand in FRB[p]. The ideal exponent is the exponent specified in FRA[p].

When the value of the operand in FRB[p] is greater than $(10^{p}-1) \% 10^{Ea}$, where p is the format precision and Ea is the exponent of the operand in FRA[p], an invalid operation exception is recognized.

When the delivered result differs in value from the operand in FRB[p], an inexact exception is recognized. No underflow exception is recognized by this operation, regardless of the value of the operand in FRB[p].

Figure 87 and Figure 88 summarize the actions. The tables do not include the setting of the $FPSCR_{FPRF}$ field. The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

Special Register Altered:

FPRF FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Programming Note

DFP Quantize can be used to adjust one DFP value (FRB[p]) to a form having the same exponent as a second DFP value (FRA[p]). If the adjustment requires the significand to be shifted left, then:

- if the result would cause overflow from the most significant digit, the result is a default QNaN.;
- otherwise the result is the adjusted value (left shifted with matching exponent).

If the adjustment requires the significand to be shifted right, the result is rounded based on the value of the RMC field. Figure 86 shows examples of these adjustments.

FRA	FRB	FRT when RMC=1	FRT when RMC=2
1 (1×10^{0})	9. (9×10^{0})	9 (9×10^{0})	9 (9×10^{0})
1.00 (100×10^{-2})	9. (9×10^{0})	9.00 (900×10^{-2})	9.00 (900×10^{-2})
1 (1×10^{0})	49.1234 (491234×10^{-4})	49 (49×10^{0})	49 (49×10^{0})
1.00 (100×10^{-2})	49.1234 (491234×10^{-4})	49.12 (4912×10^{-2})	49.12 (4912×10^{-2})
1 (1×10^{0})	49.9876 (499876×10^{-4})	49 (49×10^{0})	50 (50×10^{0})
1.00 (100×10^{-2})	49.9876 (499876×10^{-4})	49.98 (4998×10^{-2})	49.99 (4999×10^{-2})
0.01 (1×10^{-2})	49.9876 (499876×10^{-4})	49.98 (4998×10^{-2})	49.99 (4999×10^{-2})
1 (1×10^{0})	9999999999999999 ($9999999999999999 \times 10^{0}$)	9999999999999999 ($9999999999999999 \times 10^{0}$)	9999999999999999 ($9999999999999999 \times 10^{0}$)
1.0 (10×10^{-1})	9999999999999999 ($9999999999999999 \times 10^{0}$)	QNaN	QNaN

Figure 86. DFP Quantize examples

Operand a in FRA[p] is	Actions for Quantize when operand b in FRB[p] is				
	0	**Fn**	∞	**QNaN**	**SNaN**
0	*	*	V_{XCVI}: T(dNaN)	P(b)	V_{XSNAN}: U(b)
Fn	*	*	V_{XCVI}: T(dNaN)	P(b)	V_{XSNAN}: U(b)
•	V_{XCVI}: T(dNaN)	V_{XCVI}: T(dNaN)	T(dINF)	P(b)	V_{XSNAN}: U(b)
QNaN	P(a)	P(a)	P(a)	P(a)	V_{XSNAN}: U(b)
SNaN	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)	V_{XSNAN}: U(a)

Explanation:

*	See next table.
dINF	Default infinity
dNaN	Default quiet NaN
Fn	Finite nonzero numbers (includes both subnormal and normal numbers)
P(x)	The QNaN of operand x is propagated and placed in FRT[p]
T(x)	The value x is placed in FRT[p]
U(x)	The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
V_{XCVI}	The Invalid-Operation Exception (VXCVI) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 for actions)
V_{XSNAN}	The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 for actions)

Figure 87. Actions (part 1) Quantize

	Actions for Quantize when operand b in FRB[p] is		
		0	**Fn**
Te < Se	$V_b > (10^p - 1) \% 10^{Te}$	E(0)	V_{XCVI}: T(dNaN)
	$V_b [(10^p - 1) \% 10^{Te}$	E(0)	L(b)
Te = Se		E(0)	W(b)
Te > Se		E(0)	QR(b)

Explanation:

dNaN	Default quiet NaN
E(0)	The value of zero with the exponent value Te is placed in FRT[p].
L(x)	The operand x is converted to the form with the exponent value Te.
p	The precision of the format.
QR(x)	The operand x is rounded to the result of the form with the exponent value Te based on the specified rounding mode. The result of that form is placed in FRT[p].
Se	The exponent of the operand in FRB[p].
Te	The target exponent; FRA[p] for ***dqua***[***q***], or TE, a 5-bit signed binary integer for ***dquai***[***q***].
T(x)	The value x is placed in FRT[p].
V_b	The value of the operand in FRB[p].
W(x)	The value and the form of operand x is placed in FRT[p].
V_{XCVI}:	The Invalid-Operation Exception (VXCVI) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 for actions.)

Figure 88. Actions (part2) Quantize

DFP Reround [Quad] Z23-form

drrnd FRT,FRA,FRB,RMC (Rc=0)
drrnd. FRT,FRA,FRB,RMC (Rc=1

59	FRT	FRA	FRB	RMC	35	Rc
0	6	11	16	21	23	31

drrndq FRTp,FRA,FRBp,RMC (Rc=0)
drrndq. FRTp,FRA,FRBp,RMC (Rc=1)

63	FRTp	FRA	FRBp	RMC	35	Rc
0	6	11	16	21	23	31

Let k be the contents of bits 58:63 of FRA that specifies the reference significance.

When the DFP operand in FRB[p] is a finite number, and if the reference significance is zero, or if the reference significance is nonzero and the number of significant digits of the source operand is less than or equal to the reference significance, then the value and the form of the source operand is placed in FRT[p]. If the reference significance is nonzero and the number of significant digits of the source operand is greater than the reference significance, then the source operand is converted and rounded to the number of significant digits specified in the reference significance based on the rounding mode specified in the RMC field. The result of the form with the specified number of significant digits is placed in FRT[p]. The sign of the result is the same as the sign of the operand in FRB[p].

For this instruction, the number of significant digits of the value 0 is considered to be zero. The ideal exponent is the greater value of the exponent of the operand in FRB[p] and the referenced exponent. The referenced exponent is the resultant exponent if the operand in FRB[p] would have been converted and rounded to the number of significant digits specified in the reference significance based on the rounding mode specified in the RMC field.

If the exponent of the rounded result of the form that has the specified number of significant digits would be greater than $X_{max,}$ an invalid operation exception (VXCVI) occurs. When the invalid-operation exception occurs, and if the exception is disabled, a default QNaN is returned. When an invalid-operation exception occurs, no inexact exception is recognized.

In the absence of an invalid-operation exception, if the result differs in value from the operand in FRB[p], an inexact exception is recognized.

This operation causes neither an overflow nor an underflow exception.

Figure 90 summarizes the actions for *Reround*. The table does not include the setting of the $FPSCR_{FPRF}$ field. The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation exception, in which case the field remains unchanged.

Special Registers Altered:

FPRF FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Programming Note

DFP Reround can be used to adjust a DFP value (FRB[p]) to have no more than a specified number (FRA[p]58:63) of significant digits. The result (FRT[p]) is right-justified leaving the specified number of digits and rounded as specified by the RMC field. If rounding increases the number of significant digits, the result is adjusted again (the significand is shifted right 1 digit and the exponent is incremented by 1). Figure 89 has example results from *DFP Reround* for 1, 2, and 10 significant digits.

Programming Note

DFP Reround is primarily used to round a DFP value to a specific number of digits before conversion to string format for printing or display. Another use for *DFP Reround* is to obtain the effective exponent of the most significant digit by specifying a reference significance of 1. The exponent can be extracted and used to compute the number of significant digits or to left-justify a value.

For example, the following sequence computes the number of significant digits and returns it as an integer. FRB is the DFP value for which we want the number of significant digits; f13 contains the reference significance value 0x0000000000000001; and r1 is the stack pointer, with free space for doublewords at offsets -8 and -16. These doublewords are used to transfer the biased exponents from the FPRs to GPRs for integer computation. R3 contains the result of E(reround(1,FRA)) - E(FRA) + 1, where E(x) represents the biased exponent of x.

```
dxex   f0,FRB
stfd   f0,-16(r1)
drrnd  f1,f13,FRB,1 # reround 1 digit toward 0
dxex   f1,f1
stfd   f1,-8(r1)
lfd    r11,-16(r1)
lfd    r3,-8(r1)
subf   r3,r11,r3
addi   r3,r3,1
```

Given the value 412.34 the result is E(4 x 102) - E(41234 x 10-2) + 1 = (398+2) - (398-2) + 1 = 400 - 396 + 1 = 5. Additional code is required to detect and handle special values like Subnormal, Infinity, and NAN.

$FRA_{58:63}$ (binary)	FRB	FRT when RMC=1	FRT when RMC=2
1	0.41234 (41234 % 10^{-5})	0.4 (4 % 10^{-1})	0.4 (4 % 10^{-1})
1	4.1234 (41234 % 10^{-4})	4 (4 % 10^{0})	4 (4 % 10^{0})
1	41.234 (41234 % 10^{-3})	4 (4 % 10^{1})	4 (4 % 10^{1})
1	412.34 (41234 % 10^{-2})	4 (4 % 10^{2})	4 (4 % 10^{2})
2	0.491234 (491234 % 10^{-6})	0.49 (49 % 10^{-2})	0.49 (49 % 10^{-2})
2	0.499876 (499876 % 10^{-6})	0.49 (49 % 10^{-2})	0.50 (50 % 10^{-2})
2	0.999876 (999876 % 10^{-6})	0.99 (99 % 10^{-2})	1.0 (10 % 10^{-1})
10	0.491234 (491234 % 10^{-6})	0.491234 (491234 % 10^{-6})	0.491234 (491234 % 10^{-6})
10	999.999 (999999 % 10^{-3})	999.999 (999999 % 10^{-3})	999.999 (999999 % 10^{-3})
10	9999999999999999 (9999999999999999 % 10^{0})	9.999999999E+14 (9999999999 % 10^{5})	1.000000000E+15 (1000000000 % 10^{6})

Figure 89. DFP Reround examples

Programming Note

DFP Reround combined with *DFP Quantize* can be used to left justify a value (as needed by the frexp function). FRB is the DFP value for which we want to left justify; f13 contains the reference significance value 0x0000000000000001; and r1 is the stack pointer, with free space for a doubleword at offset -8. This doubleword is used to transfer the biased exponents from the FPR to a GPR, for integer computation. The adjusted biased exponent (+ format precision - 1) is transferred back into an FPR so it can be inserted into the rerounded value. The adjusted rerounded value becomes the quantize reference value. The quantize instruction returns the left justified result in FRT.

```
drrnd  f1,f13,FRB,1 # reround 1 digit toward 0
dxex   f0,f1
stfd   f0,-8(r1)
lfd    r11,-8(r1)
addi   r11,r11,15  # biased exp + precision - 1
lfd    r11,-8(r1)
stfd   f0,-8(r1)
diex   f1,f0,f1    # adjust exponent
dqua   FRT,f1,f0,1 # quantize to adjusted
                     exponent
```

	Actions for Reround when operand b in FRB[p] is				
	0*	**Fn**	**∞**	**QNaN**	**SNaN**
k g 0, k < m	-	RR(b) or V_{XCVI}: T(dNaN)	T(dINF)	P(b)	V_{XSNAN}: U(b)
k g 0, k = m	-	W(b)	T(dINF)	P(b)	V_{XSNAN}: U(b)
k g 0 and k > m, or k = 0	W(b)	W(b)	T(dINF)	P(b)	V_{XSNAN}: U(b)

Explanation:

*	The number of significant digits of the value 0 is considered to be zero for this instruction.
-	Not applicable.
dINF	Default infinity.
Fn	Finite nonzero numbers (includes both subnormal and normal numbers).
k	Reference significance, which specifies the number of significant digits in the target operand.
m	Number of significant digits in the operand in FRB[p].
P(x)	The QNaN of operand x is propagated and placed in FRT[p].
RR(x)	The value x is rounded to the form that has the specified number of significant digits. If RR(x) [$(10^k\text{-}1) \% 10^{Xmax}$, then RR(x) is returned; otherwise an invalid-operation exception is recognized.
T(x)	The value x is placed in FRT[p].
U(x)	The SNaN of operand x is converted to the corresponding QNaN and placed in FRT[p].
V_{XCVI}	The Invalid-Operation Exception (VXCVI) occurs. The result is produced only when the exception is disabled. (See Section 5.5.10.1 for actions.)
V_{XSNAN}:	The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. See Section 5.5.10.1 for actions.
W(x)	The value and the form of x is placed in FRT[p].

Figure 90. Actions: Reround

DFP Round To FP Integer With Inexact [Quad] *Z23-form*

drintx R,FRT,FRB,RMC (Rc=0)
drintx. R,FRT,FRB,RMC (Rc=1)

59	FRT	///	R	FRB	RMC	99	Rc
0	6	11	15	16	21	23	31

drintxq R,FRTp,FRBp,RMC (Rc=0)
drintxq. R,FRTp,FRBp,RMC (Rc=1)

63	FRTp	///	R	FRBp	RMC	99	Rc
0	6	11	15	16	21	23	31

The DFP operand in FRB[p] is rounded to a floating-point integer and placed into FRT[p]. The sign of the result is the same as the sign of the operand in FRB[p]. The ideal exponent is the larger value of zero and the exponent of the operand in FRB[p].

The rounding mode used is specified in the RMC field. When the RMC-encoding-selection (R) bit is zero, the RMC field contains the primary encoding; when the bit is one, the field contains the secondary encoding.

In addition to coercion of the converted value to fit the target format, the special rounding used by *Round To FP Integer* also coerces the target exponent to the ideal exponent.

When the operand in FRB[p] is a finite number and the exponent is less than zero, the operand is rounded to the result with an exponent of zero. When the exponent is greater than or equal to zero, the result is set to the numerical value and the form of the operand in FRB[p].

When the result differs in value from the operand in FRB[p], an inexact exception is recognized. No underflow exception is recognized by this operation, regardless of the value of the operand in FRB[p].

Figure 91 summarizes the actions for *Round To FP Integer With Inexact*. The table does not include the setting of the $FPSCR_{FPRF}$ field. The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation, in which case the field remains unchanged.

Special Registers Altered:

FPRF FR FI
FX XX
VXSNAN
CR1 (if Rc=1)

Programming Note

The *DFP Round To FP Integer With Inexact* and *DFP Round To FP Integer With Inexact Quad* instructions can be used to implement the decimal equivalent of the C99 rint function by specifying the primary RMC encoding for round according to $FPSCR_{DRN}$ (R=0, RMC=11). The specification for rint requires the inexact exception be raised if detected.

Operand b in FRB is	Is n not precise (n ≠ b)	Inv.-Op. Exception Enabled	Inexact Exception Enabled	Is n Incremented (|n| > |b|)	Actions*
-∞	No[1]	-	-	-	T(-dINF), FI ← 0, FR ← 0
F	No	-	-	-	W(n), FI ← 0, FR ← 0
F	Yes	-	No	No	W(n), FI ← 1, FR ← 0, XX ← 1
F	Yes	-	No	Yes	W(n), FI ← 1, FR ← 1, XX ← 1
F	Yes	-	Yes	No	W(n), FI ← 1, FR ← 0, XX ← 1, TX
F	Yes	-	Yes	Yes	W(n), FI ← 1, FR ← 1, XX ← 1, TX
+∞	No[1]	-	-	-	T(+dINF), FI ← 0, FR ← 0
QNaN	No[1]	-	-	-	P(b), FI ← 0, FR ← 0
SNaN	No[1]	No	-	-	U(b), FI ← 0, FR ← 0, VXSNAN ← 1
SNaN	No[1]	Yes	-	-	VXSNAN ← 1, TV

Explanation:

*	Setting of XX and VXSNAN is part of the corresponding exception actions. Also, when an invalid-operation exception occurs, setting of FI and FR is part of the exception actions.(See the sections, “Inexact Exception” and “Invalid Operation Exception” for more details.)
-	The actions do not depend on this condition.
1	This condition is true by virtue of the state of some condition to the left of this column.
dINF	Default infinity.
F	All finite numbers, including zeros.
FI	Floating-Point-Fraction-Inexact status flag, $FPSCR_{FI}$.
FR	Floating-Point-Fraction-Rounded status flag, $FPSCR_{FR}$.
n	The value derived when the source operand, b, is rounded to an integer using the special rounding for *Round To FP Integer*.
P(x)	The QNaN of operand x is propagated and placed in FRT[p].
T(x)	The value x is placed in FRT[p].
TV	The system floating-point enabled exception error handler is invoked for the invalid-operation exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
TX	The system floating-point enabled exception error handler is invoked for the inexact exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
U(x)	The SNaN of operand x is converted to the corresponding QNaN and placed in FPT[p].
W(x)	The value x in the form of zero exponent or the source exponent is placed in FRT[p].
XX	Floating-Point-Inexact-Exception status flag, $FPSCR_{XX}$.

Figure 91. Actions: Round to FP Integer With Inexact

DFP Round To FP Integer Without Inexact [Quad] Z23-form

drintn R,FRT,FRB,RMC (Rc=0)
drintn. R,FRT,FRB,RMC (Rc=1)

59	FRT	///	R	FRB	RMC	227	Rc
0	6	11	15	16	21	23	31

drintnq R,FRTp,FRBp,RMC (Rc=0)
drintnq. R,FRTp,FRBp,RMC (Rc=1)

63	FRTp	///	R	FRBp	RMC	227	Rc
0	6	11	15	16	21	23	31

This operation is the same as the *Round To FP Integer With Inexact* operation, except that this operation does not recognize an inexact exception.

Figure 92 summarizes the actions for *Round To FP Integer Without Inexact*. The table does not include the setting of the $FPSCR_{FPRF}$ field. The $FPSCR_{FPRF}$ field is always set to the class and sign of the result, except for an enabled invalid-operation, in which case the field remains unchanged.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXSNAN
CR1 (if Rc=1)

Programming Note

The *DFP Round To FP Integer Without Inexact* and *DFP Round To FP Integer Without Inexact Quad* instructions can be used to implement decimal equivalents of several C99 rounding functions by specifying the appropriate R and RMC field values.

Function	R	RMC
Ceil	1	0b00
Floor	1	0b01
Nearbyint	0	0b11
Round	0	0b10
Trunc	0	0b01

Note that nearbyint is similar to the rint function but without raising the inexact exception. Similarly ceil, floor, round, and trunc do not require the inexact exception.

Operand b in FRB is	Inv.-Op. Exception Enabled	Actions*
-∞	-	T(-dINF), FI ← 0, FR ← 0
F	-	W(n), FI ← 0, FR ← 0
+∞	-	T(+dINF), FI ← 0, FR ← 0
QNaN	-	P(b), FI ← 0, FR ← 0
SNaN	No	U(b), FI ← 0, FR ← 0, VXSNAN←1
SNaN	Yes	VXSNAN ← 1, TV

Explanation:

*	Setting of VXSNAN is part of the corresponding exception actions. Also, when an invalid-operation exception occurs, setting of FI and FR bits is part of the exception actions. (See the sections, “Invalid Operation Exception” for more details.)
-	The actions do not depend on this condition.
dINF	Default infinity.
F	All finite numbers, including zeros.
FI	Floating-Point-Fraction-Inexact status flag, $FPSCR_{FI}$.
FR	Floating-Point-Fraction-Rounded status flag, $FPSCR_{FR}$.
n	The value derived when the source operand, b, is rounded to an integer using the special rounding for Round-To-FP-Integer.
P(x)	The QNaN of operand x is propagated and placed in FRT[p].
T(x)	The value x is placed in FRT[p].
TV	The system floating-point enabled exception error handler is invoked for the invalid-operation exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
U(x)	The SNaN of operand x is converted to the corresponding QNaN and placed in FPT[p].
W(x)	The value x in the form of zero exponent or the source exponent is placed in FRT[p].

Figure 92. Actions: Round to FP Integer Without Inexact

5.6.5 DFP Conversion Instructions

The DFP conversion instructions consist of data-format conversion instructions and data-type conversion instructions. They are all X-form instructions and employ the record bit (Rc).

5.6.5.1 DFP Data-Format Conversion Instructions

The data-format conversion instructions consist of *Convert To DFP Long*, *Convert To DFP Extended*, *Round To DFP Short*, and *Round To DFP Long*. Figure 93 summarizes the actions for these instructions.

Programming Note

DFP does not provide operations on short operands, so they must be converted to long format, and then converted back to be stored. Preserving correct signaling NaN semantics requires that signaling NaNs be propagated from the source to the result without recognizing an exception during widening from short to long or narrowing from long to short. Because DFP does not provide equivalents to the FP *Load Floating-Point Single* and *Store Floating-Point Single* functions, the widening is performed by loading the DFP short value with a *Load Floating as Integer Word Indexed* followed by a *DFP Convert to DFP Long*, and narrowing is performed by a *DFP Round to DFP Short* followed by a *Store Floating-Point as Integer Word Indexed*. If the SNaN or infinity in DFP short format uses the preferred DPD encoding, then converting this operand to DFP long format and back to DFP short will result in the original bit pattern.

Instruction	Actions when operand b in FRB[p] is			
	F	**∞**	**QNaN**	**SNaN**
Convert To DFP Long	T(b)[1]	P(b)[2,4]	P(b)[2,4]	P(b)[3,4]
Convert To DFP Extended	T(b)[1]	T(dINF)	P(b)[2,4]	V_{XSNAN}: U(b)[2,4]
Round To DFP Short	R(b)[1]	P(b)[2,5]	P(b)[2,5]	P(b)[3,5]
Round To DFP Long	R(b)[1]	T(dINF)	P(b)[2,5]	V_{XSNAN}: U(b)[2,5]

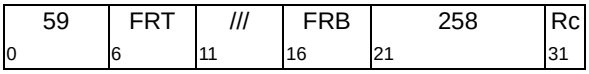
Explanation:

1The ideal exponent is the exponent of the source operand.
2Bits 5:N-1 of the N-bit combination field are set to zero.
3Bit 5 of the N-bit combination field is set to one. Bits 6:N-1 of the combination field are set to zero.
4The trailing significand field is padded on the left with zeros.
5Leftmost digits in the trailing significand field are removed.
dINFDefault infinity.
FAll finite numbers, including zeros.
P(x)The special symbol in operand x is propagated into FRT[p].
R(x)The value x is rounded to the target-format precision; see Section 5.5.11
T(x)The value x is placed in FRT[p].
U(x)The SNaN of operand x is converted to the corresponding QNaN.
V_{XSNAN}The Invalid-Operation Exception (VXSNAN) occurs. The result is produced only when the exception is disabled. See Section 5.5.10.1 for actions.

Figure 93. Actions: Data-Format Conversion Instructions

DFP Convert To DFP Long X-form

dctdp FRT,FRB (Rc=0)
dctdp. FRT,FRB (Rc=1)



59	FRT	///	FRB	258	Rc
0	6	11	16	21	31

The DFP short operand in bits 32:63 of FRB is converted to DFP long format and the converted result is placed into FRT. The sign of the result is the same as the sign of the source operand. The ideal exponent is the exponent of the source operand.

If the operand in FRB is an SNaN, it is converted to an SNaN in DFP long format and does not cause an invalid-operation exception.

Special Registers Altered:

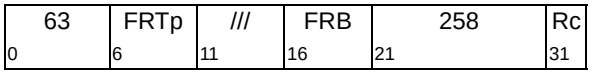
FPRF FR (undefined) FI (undefined)
CR1 (if Rc=1)

Programming Note

Note that DFP short format is a storage-only format, Therefore, conversion of a short SNaN to long format will not cause an exception and the SNaN is preserved. Subsequent operation on that SNaN in long format will cause an exception.

DFP Convert To DFP Extended X-form

dctqpq FRTp,FRB (Rc=0)
dctqpq. FRTp,FRB (Rc=1)



63	FRTp	///	FRB	258	Rc
0	6	11	16	21	31

The DFP long operand in the FRB is converted to DFP extended format and placed into FRTp. The sign of the result is the same as the sign of the operand in FRB. The ideal exponent is the exponent of the operand in FRB.

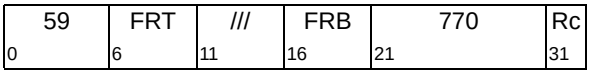
If the operand in FRB is an SNaN, an invalid-operation exception is recognized. If the exception is disabled, the SNaN is converted to the corresponding QNaN in DFP extended format.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXSNAN
CR1 (if Rc=1)

DFP Round To DFP Short X-form

drsp FRT,FRB (Rc=0)
drsp. FRT,FRB (Rc=1)



The DFP long operand in FRB is converted and rounded to DFP short format. The DFP short value is extended on the left with zeros to form a 64-bit entity and placed into FRT. The sign of the result is the same as the sign of the source operand. The ideal exponent is the exponent of the source operand.

If the operand in FRB is an SNaN, it is converted to an SNaN in DFP short format and does not cause an invalid-operation exception.

Normally, the result is in the format and length of the target. However, when an overflow or underflow exception occurs and if the exception is enabled, the operation is completed by producing a wrapped rounded result in the same format and length as the source but rounded to the target-format precision.

Special Registers Altered:

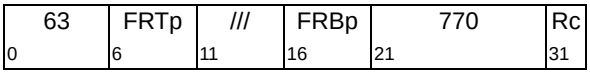
FPRF FR FI
FX OX UX XX
CR1 (if Rc=1)

Programming Note

Note that DFP short format is a storage-only format, Therefore, conversion of a long SNaN to short format will not cause an exception. Converting a long format SNaN to short format is an implied move operation.

DFP Round To DFP Long X-form

drdpq FRTp,FRBp (Rc=0)
drdpq. FRTp,FRBp (Rc=1)



The DFP extended operand in FRBp is converted and rounded to DFP long format. The result concatenated with 64 0s is placed in FRTp. The sign of the result is the same as the sign of the source operand. The ideal exponent is the exponent of the operand in FRBp.

If the operand in FRBp is an SNaN, an invalid-operation exception is recognized. If the exception is disabled, the SNaN is converted to the corresponding QNaN in DFP long format.

Normally, the result is in the format and length of the target. However, when an overflow or underflow exception occurs and if the exception is enabled, the operation is completed by producing a wrapped rounded result in the same format and length as the source but rounded to the target-format precision.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN
CR1 (if Rc=1)

Programming Note

Note that DFP Round to DFP Long, while producing a result in DFP long format, actually targets a register pair, writing 64 0s in FRTp+1.

5.6.5.2 DFP Data-Type Conversion Instructions

The DFP data-type conversion instructions are used to convert data type between DFP and fixed.

The data-type conversion instructions consist of *Convert From Fixed* and *Convert To Fixed*.

DFP Convert From Fixed *X-form*

dcffix FRT,FRB (Rc=0)
dcffix. FRT,FRB (Rc=1)

59	FRT	///	FRB	802	Rc
0	6	11	16	21	31

The 64-bit signed binary integer in FRB is converted and rounded to a DFP Long value and placed into FRT. The sign of the result is the same as the sign of the source operand. The ideal exponent is zero.

If the source operand is a zero, then a plus zero with a zero exponent is returned.

The $FPSCR_{FPRF}$ field is set to the class and sign of the result.

Special Registers Altered:

FPRF FR FI
FX XX
CR1 (if Rc=1)

DFP Convert From Fixed Quad *X-form*

dcffixq FRTp,FRB (Rc=0)
dcffixq. FRTp,FRB (Rc=1)

63	FRTp	///	FRB	802	Rc
0	6	11	16	21	31

The 64-bit signed binary integer in FRB is converted and rounded to a DFP Extended value and placed into FRTp. The sign of the result is the same as the sign of the source operand. The ideal exponent is zero.

If the source operand is a zero, then a plus zero with a zero exponent is returned.

The $FPSCR_{FPRF}$ field is set to the class and sign of the result.

Special Registers Altered:

FPRF FR (undefined) FI (undefined)
CR1 (if Rc=1)

DFP Convert To Fixed [Quad] *X-form*

dctfix FRT,FRB (Rc=0)
dctfix. FRT,FRB (Rc=1)

59	FRT	///	FRB	290	Rc
0	6	11	16	21	31

dctfixq FRT,FRBp (Rc=0)
dctfixq. FRT,FRBp (Rc=1)

63	FRT	///	FRBp	290	Rc
0	6	11	16	21	31

The DFP operand in FRB[p] is rounded to an integer value and is placed into FRT in the 64-bit signed binary integer format. The sign of the result is the same as the sign of the source operand, except when the source operand is a NaN or a zero.

Figure 94 summarizes the actions for *Convert To Fixed*.

Special Registers Altered:

FPRF (undefined) FR FI
FX XX
VXSNAN VXCVI
CR1 (if Rc=1)

Programming Note

It is recommended that software pre-round the operand to a floating-point integral using ***drintx***[***q***] or ***drintn***[***q***] is a rounding mode other than the current rounding mode specified by $FPSCR_{DRN}$ is needed. Saving, modifying and restoring the FPSCR just to temporarily change the rounding mode is less efficient than just employing drintx[p] or drint[p] which override the current rounding mode using an immediate control field.

For example if the desired function rounding is Round to Nearest, Ties away from 0 but the default rounding (from $FPSCR_{DRN}$) is Round to Nearest, Ties to Even then following is preferred.

```
drintn    0,f1,f1,2
dctfix    f1,f1
```

Operand b in FRB[p] is	q is	Is n not precise (n ≠ b)	Inv.-Op. Except. Enabled	Inexact Except. Enabled	Is n Incremented (|n| > |b|)	Actions *
-∞ ≤ b < MN	< MN	-	No	-	-	T(MN), FI ← 0, FR ← 0, VXCVI ← 1
-∞ ≤ b < MN	< MN	-	Yes	-	-	VXCVI ← 1, TV
-∞ < b < MN	= MN	-	-	No	-	T(MN), FI ← 1, FR ← 0, XX ← 1
-∞ < b < MN	= MN	-	-	Yes	-	T(MN), FI ← 1, FR ← 0, XX ← 1,TX
MN ≤ b < 0	-	No	-	-	-	T(n), FI ← 0, FR ← 0
MN ≤ b < 0	-	Yes	-	No	No	T(n), FI ← 1, FR ← 0, XX ← 1
MN ≤ b < 0	-	Yes	-	No	Yes	T(n), FI ← 1, FR ← 1, XX ← 1
MN ≤ b < 0	-	Yes	-	Yes	No	T(n), FI ← 1, FR ← 0, XX ← 1, TX
MN ≤ b < 0	-	Yes	-	Yes	Yes	T(n), FI ← 1, FR ← 1, XX ← 1, TX
±0	-	No	-	-	-	T(0), FI ← 0, FR ← 0
0 < b ≤ MP	-	No	-	-	-	T(n), FI ← 0, FR ← 0
0 < b ≤ MP	-	Yes	-	No	No	T(n), FI ← 1, FR ← 0, XX ← 1
0 < b ≤ MP	-	Yes	-	No	Yes	T(n), FI ← 1, FR ← 1, XX ← 1
0 < b ≤ MP	-	Yes	-	Yes	No	T(n), FI ← 1, FR ← 0, XX ← 1, TX
0 < b ≤ MP	-	Yes	-	Yes	Yes	T(n), FI ← 1, FR ← 1, XX ← 1, TX
MP < b < +∞	= MP	-	-	No	-	T(MP), FI ← 1, FR ← 0, XX ← 1
MP < b < +∞	= MP	-	-	Yes	-	T(MP), FI ← 1, FR ← 0, XX ← 1, TX
MP < b ≤ +∞	> MP	-	No	-	-	T(MP), FI ← 0, FR ← 0, VXCVI ← 1
MP < b ≤ +∞	> MP	-	Yes	-	-	VXCVI ← 1, TV
QNaN	-	-	No	-	-	T(MN), FI←0, FR←0, VXCVI←1
QNaN	-	-	Yes	-	-	VXCVI←1, TV
SNaN	-	-	No	-	-	T(MN),FI←0, FR←0, VXCVI←1,VXSNAN ←1
SNaN	-	-	Yes	-	-	VXCVI←1,VXSNAN ← 1, TV

Explanation:

*	Setting of XX, VXCVI, and VXSNAN is part of the corresponding exception actions. Also, when an invalid-operation exception occurs, setting of FI and FR bits is part of the exception actions. (See the sections, “Inexact Exception” and “Invalid Operation Exception” for more details.)
-	The actions do not depend on this condition.
FI	Floating-Point-Fraction-Inexact status flag, $FPSCR_{FI}$.
FR	Floating-Point-Fraction-Rounded status flag, $FPSCR_{FR}$.
MN	Maximum negative number representable by the 64-bit binary integer format
MP	Maximum positive number representable by the 64-bit binary integer format.
n	The value q converted to a fixed-point result.
q	The value derived when the source value b is rounded to an integer using the specified rounding mode
T(x)	The value x is placed in FRT[p].
TV	The system floating-point enabled exception error handler is invoked for the invalid-operation exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
TX	The system floating-point enabled exception error handler is invoked for the inexact exception if the FE0 and FE1 bits in the machine-state register are set to any mode other than the ignore-exception mode.
VXCVI	The $FPSCR_{VXCVI}$ invalid operation exception status bit.
VXSNAN	The $FPSCR_{VXSNAN}$ invalid operation exception status bit.
XX	Floating-Point-Inexact-Exception status flag, $FPSCR_{XX.}$

Figure 94. Actions: Convert To Fixed

5.6.6 DFP Format Instructions

The DFP format instructions are used to compose or decompose a DFP operand. A source operand of SNaN does not cause an invalid-operation exception. All format instructions employ the record bit (Rc).

The format instructions consist of *Decode DPD To BCD*, *Encode BCD To DPD*, *Extract Biased Exponent*, *Insert Biased Exponent*, *Shift Significand Left Immediate*, and *Shift Significand Right Immediate*.

DFP Decode DPD To BCD [Quad] X-form

ddedpd SP,FRT,FRB (Rc=0)
ddedpd. SP,FRT,FRB (Rc=1)

59	FRT	SP	///	FRB	322	Rc
0	6	11	13	16	21	31

ddedpdq SP,FRTp,FRBp (Rc=0)
ddedpdq. SP,FRTp,FRBp (Rc=1)

63	FRTp	SP	///	FRBp	322	Rc
0	6	11	13	16	21	31

A portion of the significand of the DFP operand in FRB[p] is converted to a signed or unsigned BCD number depending on the SP field. For infinity and NaN, the significand is considered to be the contents in the trailing significand field padded on the left by a zero digit.

SP_0 = 0 (unsigned conversion)

The rightmost 16 digits of the significand (32 digits for ***ddedpdq***) is converted to an unsigned BCD number and the result is placed into FRT[p].

SP_0 = 1 (signed conversion)

The rightmost 15 digits of the significand (31 digits for ***ddedpdq***) is converted to a signed BCD number with the same sign as the DFP operand, and the result is placed into FRT[p]. If the DFP operand is negative, the sign is encoded as 0b1101. If the DFP operand is positive, SP_1 indicates which preferred plus sign encoding is used. If SP_1 = 0, the plus sign is encoded as 0b1100 (the option-1 preferred sign code), otherwise the plus sign is encoded as 0b1111(the option-2 preferred sign code).

Special Registers Altered:

CR1 (if Rc=1)

DFP Encode BCD To DPD [Quad] X-form

denbcd S,FRT,FRB (Rc=0)
denbcd. S,FRT,FRB (Rc=1)

59	FRT	S	///	FRB	834	Rc
0	6	11	12	16	21	31

denbcdq S,FRTp,FRBp (Rc=0)
denbcdq. S,FRTp,FRBp (Rc=1)

63	FRTp	S	///	FRBp	834	Rc
0	6	11	12	16	21	31

The signed or unsigned BCD operand, depending on the S field, in FRB[p] is converted to a DFP number. The ideal exponent is zero.

S = 0 (unsigned BCD operand)

The unsigned BCD operand in FRB[p] is converted to a positive DFP number of the same magnitude and the result is placed into FRT[p].

S = 1 (signed BCD operand)

The signed BCD operand in FRB[p] is converted to the corresponding DFP number and the result is placed into FRT[p].

If an invalid BCD digit or sign code is detected in the source operand, an invalid-operation exception (VXCVI) occurs.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exception when $FPSCR_{VE}$=1.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX
VXCVI
CR1 (if Rc=1)

DFP Extract Biased Exponent [Quad]
X-form

dxex FRT,FRB (Rc=0)
dxex. FRT,FRB (Rc=1)

59	FRT	///	FRB	354	Rc
0	6	11	16	21	31

dxexq FRT,FRBp (Rc=0)
dxexq. FRT,FRBp (Rc=1)

63	FRT	///	FRBp	354	Rc
0	6	11	16	21	31

The biased exponent of the operand in FRB[p] is extracted and placed into FRT in the 64-bit signed binary integer format. When the operand in FRB is an infinity, QNaN, or SNaN, a special code is returned.

Operand	Result
Finite Number	biased exponent value
Infinity	-1
QNaN	-2
SNaN	-3

Special Registers Altered:

CR1 (if Rc=1)

Programming Note

The exponent bias value is 101 for DFP Short, 398 for DFP Long, and 6176 for DFP Extended.

DFP Insert Biased Exponent [Quad]
X-form

diex FRT,FRA,FRB (Rc=0)
diex. FRT,FRA,FRB (Rc=1)

59	FRT	FRA	FRB	866	Rc
0	6	11	16	21	31

diexq FRTp,FRA,FRBp (Rc=0)
diexq. FRTp,FRA,FRBp (Rc=1)

63	FRTp	FRA	FRBp	866	Rc
0	6	11	16	21	31

Let *a* be the value of the 64-bit signed binary integer in FRA.

a	**Result**
a > MBE[1]	QNaN
MBE m a m 0	Finite number with biased exponent a
a = -1	Infinity
a = -2	QNaN
a = -3	SNaN
a < -3	QNaN

[1] Maximum biased exponent for the target format

When 0 [*a* [MBE, *a* is the biased target exponent that is combined with the sign bit and the significand value of the DFP operand in FRB[p] to form the DFP result in FRT[p]. The ideal exponent is the specified target exponent.

When *a* specifies a special code (a < 0 or a > MBE), an infinity, QNaN, or SNaN is formed in FRT[p] with the trailing significand field containing the value from the trailing significand field of the source operand in FRB[p], and with an N-bit combination field set as follows.

- For an Infinity result,
  - the leftmost 5 bits are set to 0b11110, and
  - the rightmost N-5 bits are set to zero.
- For a QNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to zero, and
  - the rightmost N-5 bits are set to zero.
- For an SNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to one, and
  - the rightmost N-5 bits are set to zero.

Special Registers Altered:

CR1 (if Rc=1)

Programming Note

The exponent bias value is 101 for DFP Short, 398 for DFP Long, and 6176 for DFP Extended.

Operand a in FRA[p] specifies	Actions for Insert Biased Exponent when operand b in FRB[p] specifies			
	F	∞	**QNaN**	**SNaN**
F	N, Rb	Z, Rb	Z, Rb	Z, Rb
∞	I, Rb	I, Rb	I, Rb	I, Rb
QNaN	Q, Rb	Q, Rb	Q, Rb	Q, Rb
SNaN	S, Rb	S, Rb	S, Rb	S, Rb

Explanation:

F	All finite numbers, including zeros
I	The combination field in FRT[p] is set to indicate a default Infinity.
N	The combination field in FRT[p] is set to the specified biased exponent in FRA and the leftmost significand digit in FRB[p].
Q	The combination field in FRT[p] is set to indicate a default QNaN.
S	The combination field in FRT[p] is set to indicate a default SNaN.
Z	The combination field in FRT[p] is set to indicate the specific biased exponent in FRA and a leftmost coefficient digit of zero.
Rb	The contents of the trailing significand field in FRB[p] are reencoded using preferred DPD encodings and the reencoded result is placed in the same field in FRT[p]. The sign bit of FRB[p] is copied into the sign bit in FRT[p].

Figure 95. Actions: Insert Biased Exponent

DFP Shift Significand Left Immediate [Quad] Z22-form

dscli FRT,FRA,SH (Rc=0)
dscli. FRT,FRA,SH (Rc=1)

59	FRT	FRA	SH	66	Rc
0	6	11	16	22	31

dscliq FRTp,FRAp,SH (Rc=0)
dscliq. FRTp,FRAp,SH (Rc=1)

63	FRTp	FRAp	SH	66	Rc
0	6	11	16	22	31

The significand of the DFP operand in FRA[p] is shifted left SH digits. For a NaN or infinity, all significand digits are in the trailing significand field. SH is a 6-bit unsigned binary integer. Digits shifted out of the leftmost digit are lost. Zeros are supplied to the vacated positions on the right. The result is placed into FRT[p]. The sign of the result is the same as the sign of the source operand in FRA[p].

If the source operand in FRA[p] is a finite number, the exponent of the result is the same as the exponent of the source operand.

For an Infinity, QNaN or SNaN result, the target format's N-bit combination field is set as follows.

- For an Infinity result,
  - the leftmost 5 bits are set to 0b11110, and
  - the rightmost N-5 bits are set to zero.
- For a QNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to zero, and
  - the rightmost N-6 bits are set to zero.
- For an SNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to one, and
  - the rightmost N-6 bits are set to zero.

Special Registers Altered:

CR1 (if Rc=1)

DFP Shift Significand Right Immediate [Quad] Z22-form

dscri FRT,FRA,SH (Rc=0)
dscri. FRT,FRA,SH (Rc=1)

59	FRT	FRA	SH	98	Rc
0	6	11	16	22	31

dscriq FRTp,FRAp,SH (Rc=0)
dscriq. FRTp,FRAp,SH (Rc=1)

63	FRTp	FRAp	SH	98	Rc
0	6	11	16	22	31

The significand of the DFP operand in FRA[p] is shifted right SH digits. For a NaN or infinity, all significand digits are in the trailing significand field. SH is a 6-bit unsigned binary integer. Digits shifted out of the units digit are lost. Zeros are supplied to the vacated positions on the left. The result is placed into FRT[p]. The sign of the result is the same as the sign of the source operand in FRA[p].

If the source operand in FRA[p] is a finite number, the exponent of the result is the same as the exponent of the source operand.

For an Infinity, QNaN or SNaN result, the target format's N-bit combination field is set as follows.

- For an Infinity result,
  - the leftmost 5 bits are set to 0b11110, and
  - the rightmost N-5 bits are set to zero.
- For a QNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to zero, and
  - the rightmost N-6 bits are set to zero.
- For an SNaN result,
  - the leftmost 5 bits are set to 0b11111,
  - bit 5 is set to one, and
  - the rightmost N-6 bits are set to zero.

Special Registers Altered:

CR1 (if Rc=1)

5.6.7 DFP Instruction Summary

Mnemonic	Full Name	FORM	Operands	SNaN Vs G	Encoding	FPRF C	FPRF FPCC	FP Exception V Z O U X	FRIFI	IE	Rc
dadd	DFP Add	X	FRT, FRA, FRB	Y N	RE	Y	Y	V O U X	Y	Y	Y
daddq	DFP Add Quad	X	FRTp, FRAp, FRBp	Y N	RE	Y	Y	V O U X	Y	Y	Y
dsub	DFP Subtract	X	FRT, FRA, FRB	Y N	RE	Y	Y	V O U X	Y	Y	Y
dsubq	DFP Subtract Quad	X	FRTp, FRAp, FRBp	Y N	RE	Y	Y	V O U X	Y	Y	Y
dmul	DFP Multiply	X	FRT, FRA, FRB	Y N	RE	Y	Y	V O U X	Y	Y	Y
dmulq	DFP Multiply Quad	X	FRTp, FRAp, FRBp	Y N	RE	Y	Y	V O U X	Y	Y	Y
ddiv	DFP Divide	X	FRT, FRA, FRB	Y N	RE	Y	Y	V Z O U X	Y	Y	Y
ddivq	DFP Divide Quad	X	FRTp, FRAp, FRBp	Y N	RE	Y	Y	V Z O U X	Y	Y	Y
dcmpo	DFP Compare Ordered	X	BF, FRA, FRB	Y -	-	N	Y	V	-	-	N
dcmpoq	DFP Compare Ordered Quad	X	BF, FRAp, FRBp	Y -	-	N	Y	V	-	-	N
dcmpu	DFP Compare Unordered	X	BF, FRA, FRB	Y -	-	N	Y	V	-	-	N
dcmpuq	DFP Compare Unordered Quad	X	BF, FRAp, FRBp	Y -	-	N	Y	V	-	-	N
dtstdc	DFP Test Data Class	Z22	BF, FRA, DCM	N -	-	N	Y[1]		-	-	N
dtstdcq	DFP Test Data Class Quad	Z22	BF, FRAp, DCM	N -	-	N	Y[1]		-	-	N
dtstdg	DFP Test Data Group	Z22	BF, FRA,DGM	N -	-	N	Y[1]		-	-	N
dtstdgq	DFP Test Data Group Quad	Z22	BF, FRAp, DGM	N -	-	N	Y[1]		-	-	N
dtstex	DFP Test Exponent	X	BF, FRA, FRB	N -	-	N	Y		-	-	N
dtstexq	DFP Test Exponent Quad	X	BF, FRAp, FRBp	N -	-	N	Y		-	-	N
dtstsf	DFP Test Significance	X	BF, FRA(FIX), FRB	N -	-	N	Y		-	-	N
dtstsfq	DFP Test Significance Quad	X	BF, FRA(FIX), FRBp	N -	-	N	Y		-	-	N
dquai	DFP Quantize Immediate	Z23	TE, FRT, FRB, RMC	Y N	RE	Y	Y	V X	Y	Y	Y
dquaiq	DFP Quantize Immediate Quad	Z23	TE, FRTp, FRBp, RMC	Y N	RE	Y	Y	V X	Y	Y	Y
dqua	DFP Quantize	Z23	FRT,FRA,FRB,RMC	Y N	RE	Y	Y	V X	Y	Y	Y
dquaq	DFP Quantize Quad	Z23	FRTp,FRAp,FRBp, RMC	Y N	RE	Y	Y	V X	Y	Y	Y
drrnd	DFP Reround	Z23	FRT,FRA(FIX),FRB,RMC	Y N	RE	Y	Y	V X	Y	Y	Y
drrndq	DFP Reround Quad	Z23	FRTp, FRA(FIX), FRBp, RMC	Y N	RE	Y	Y	V X	Y	Y	Y
drintx	DFP Round To FP Integer With Inexact	Z23	R,FRT, FRB,RMC	Y N	RE	Y	Y	V X	Y	Y	Y
drintxq	DFP Round To FP Integer With Inexact Quad	Z23	R,FRTp,FRBp,RMC	Y N	RE	Y	Y	V X	Y	Y	Y
drintn	DFP Round To FP Integer With-out Inexact	Z23	R,FRT, FRB,RMC	Y N	RE	Y	Y	V	Y[#]	Y	Y
drintnq	DFP Round To FP Integer With-out Inexact Quad	Z23	R,FRTp, FRBp,RMC	Y N	RE	Y	Y	V	Y[#]	Y	Y
dctdp	DFP Convert To DFP Long	X	FRT, FRB (DFP Short)	N Y	RE	Y	Y[2]		U	Y	Y
dctqpq	DFP Convert To DFP Extended	X	FRTp, FRB	Y N	RE	Y	Y	V	Y[#]	Y	Y
drsp	DFP Round To DFP Short	X	FRT (DFP Short), FRB	N Y	RE	Y	Y[2]	O U X	Y	Y	Y
drdpq	DFP Round To DFP Long	X	FRTp, FRBp	Y N	RE	Y	Y	V O U X	Y	Y	Y
dcffixq	DFP Convert From Fixed Quad	X	FRTp, FRB (FIX)	- N	RE	Y	Y		U	Y	Y
dctfix	DFP Convert To Fixed	X	FRT (FIX), FRB	Y N	-	U	U	V X	Y	-	Y
dctfixq	DFP Convert To Fixed Quad	X	FRT (FIX), FRBp	Y N	-	U	U	V X	Y	-	Y
ddedpd	DFP Decode DPD To BCD	X	SP, FRT(BCD), FRB	N -	-	N	N		-	-	Y

Figure 96. Decimal Floating-Point Instructions Summary

Mnemonic	Full Name	FORM	Operands	SNaN Vs	SNaN G	Encoding	FPRF C	FPRF FPCC	FP Exception V Z O U X	FR|FI	IE	Rc
ddedpdq	DFP Decode DPD To BCD Quad	X	SP, FRTp(BCD), FRBp	N	-	-	N	N		-	-	Y
denbcd	DFP Encode BCD To DPD	X	S, FRT, FRB (BCD)	-	N	RE	Y	Y	V	Y#	Y	Y
denbcdq	DFP Encode BCD To DPD Quad	X	S, FRTp, FRBp (BCD)	-	N	RE	Y	Y	V	Y#	Y	Y
dxex	DFP Extract Biased Exponent	X	FRT (FIX), FRB	N	N	-	N	N		-	-	Y
dxexq	DFP Extract Biased Exponent Quad	X	FRT (FIX), FRBp	N	N	-	N	N		-	-	Y
diex	DFP Insert Biased Exponent	X	FRT, FRA(FIX), FRB	N	Y	RE	N	N		-	Y	Y
diexq	DFP Insert Biased Exponent Quad	X	FRTp, FRA(FIX), FRBp	N	Y	RE	N	N		-	Y	Y
dscli	DFP Shift Significand Left Immediate	Z22	FRT,FRA,SH	N	Y	RE	N	N		-	-	Y
dscliq	DFP Shift Significand Left Immediate Quad	Z22	FRTp,FRAp,SH	N	Y	RE	N	N		-	-	Y
dscri	DFP Shift Significand Right Immediate	Z22	FRT,FRA,SH	N	Y	RE	N	N		-	-	Y
dscriq	DFP Shift Significand Right Immediate Quad	Z22	FRTp,FRAp,SH	N	Y	RE	N	N		-	-	Y

Explanation:

#	FI and FR are set to zeros for these instructions.
-	Not applicable.
1	A unique definition of the $FPSCR_{FPCC}$ field is provided for the instruction.
2	These are the only instructions that may generate an SNaN and also set the $FPSC_{FPRF}$ field. Since the BFP $FPSCR_{FPRF}$ field does not include a code for SNaN, these instructions cause the need for redefining the $FPSCR_{FPRF}$ field for DFP.
DCM	A 6-bit immediate operand specifying the data-class mask.
DGM	A 6-bit immediate operand specifying the data-group mask.
G	An SNaN can be generated as the target operand.
IE	An ideal exponent is defined for the instruction.
FI	Setting of the $FPSCR_{FI}$ flag.
FR	Setting of the $FPSCR_{FR}$ flag.
N	No.
O	An overflow exception may be recognized.
Rc	The record bit, Rc, is provided to record $FPSCR_{32:35}$ in CR field 1.
RE	The trailing significand field is reencoded using preferred DPD encodings.The preferred DPD encoding are also used for propagated NaNs, or converted NaNs and infinities.
RMC	A 2-bit immediate operand specifying the rounding-mode control.
S	An one-bit immediate operand specifying if the operation is signed or unsigned.
SP	A two-bit immediate operand: one bit specifies if the operation is signed or unsigned and, for signed operations, another bit specifies which preferred plus sign code is generated.
U	An underflow exception may be recognized.
V	An invalid-operation exception may be recognized.
Vs	An input operand of SNaN causes an invalid-operation exception.
X	An inexact exception may be recognized.
Y	Yes.
U	Undefined
Z	A zero-divide exception may be recognized.

Figure 96. Decimal Floating-Point Instructions Summary (Continued)

Chapter 6. Vector Facility

6.1 Vector Facility Overview

This chapter describes the registers and instructions that make up the Vector Facility.

6.2 Chapter Conventions

6.2.1 Description of Instruction Operation

The following notation, in addition to that described in Section 1.3.2, is used in this chapter.

x.bit[y]
Return the contents of bit y of x.

x.bit[y:z]
Return the contents of bits y:z of x.

x.nibble[y]
Return the contents of the 4-bit nibble element y of x.

x.nibble[y:z]
Return the contents of the nibble elements y:z of x.

x.byte[y]
Return the contents of byte element y of x.

x.byte[y:z]
Return the contents of byte elements y:z of x.

x.hword[y]
Return the contents of halfword element y of x.

x.hword[y:z]
Return the contents of halfword elements y:z of x.

x.word[y]
Return the contents of word element y of x.

x.word[y:z]
Return the contents of word element y:z of x.

x.dword[y]
Return the contents of doubleword element y of x.

x.dword[y:z]
Return the contents of doubleword elements y:z of x.

x ? y : z
if the value of x is true, then the value of y, otherwise the value z.

$+_{int}$
Integer addition.

$+_{fp}$
Floating-point addition.

$-_{fp}$
Floating-point subtraction.

$\times_{sui}$
Multiplication of a signed-integer (first operand) by an unsigned-integer (second operand).

$\times_{fp}$
Floating-point multiplication.

$=_{int}$
Integer equals relation.

$=_{fp}$
Floating-point equals relation.

$<_{ui}$, $\leq_{ui}$, $>_{ui}$, $\geq_{ui}$
Unsigned-integer comparison relations.

$<_{si}$, $\leq_{si}$, $>_{si}$, $\geq_{si}$
Signed-integer comparison relations.

$<_{fp}$, $\leq_{fp}$, $>_{fp}$, $\geq_{fp}$
Floating-point comparison relations.

LENGTH(x)

Length of x, in bits. If x is the word “element”, LENGTH(x) is the length, in bits, of the element implied by the instruction mnemonic.

x $+_{bcd}$ 1

Increments the magnitude of the packed decimal value x by 1.

x << y

Result of shifting x left by y bits, filling vacated bits with zeros.

```
b ← LENGTH(x)
result ← (y < b) ? (x[y:b-1] || [y]0) : [b]0
```

x $>>_{ui}$ y

Result of shifting x right by y bits, filling vacated bits with zeros.

```
b ← LENGTH(x)
result ← (y < b) ? ([y]0 || x[0:(b-y)-1]) : [b]0
```

x >> y

Result of shifting x right by y bits, filling vacated bits with copies of bit 0 (sign bit) of x.

```
b ← LENGTH(x)
result ← (y<b) ? ([y]x[0] || x[0:(b-y)-1]) : [b]x[0]
```

x <<< y

Result of rotating x left by y bits.

```
b ← LENGTH(x)
result ← x[y:b-1] || x[0:y-1]
```

x >>> y

Returns the contents of x rotated right by y bits.

Chop(x, y)

Result of extending the right-most y bits of x on the left with zeros.

```
result ← x & ((1<<y)-1)
```

Clamp(x, y, z)

x is interpreted as a signed integer. If the value of x is less than y, then the value y is returned, else if the value of x is greater than z, the value z is returned, else the value x is returned.

```
if (x < y) then
   result ← y
   VSCR[SAT] ← 1
else if (x > z) then
   result ← z
   VSCR[SAT] ← 1
else result ← x
```

ConvertSItoBCD(x,y)

Let x be a signed integer quadword.
Let y indicate the preferred sign code.

Return the signed integer value x in packed decimal format.

```
if (x<0) then do
   x    ← ¬x + 1
   sign ← 0x000D
end
else
   sign ← (y=0) ? 0x000C : 0x000F

result ← 0
shcnt  ← 4

do while (x > 0)
   digit  ← x % 10
   result ← result | (digit<<shcnt)
   x      ← x ÷ 10
   shcnt  ← shcnt + 4
end

return (result | sign)
```

ConvertBCDtoSI(x)

Let x be a packed decimal value.

Return the value x in signed integer format.

```
result ← 0
scale  ← 1
sign   ← x.bit[124:127]
x      ← x >> 4
do while (x > 0)
   digit  ← x & 0x000F
   result ← result + (digit × scale)
   x      ← x >> 4
   scale  ← scale × 10
end

if (sign==0x000B) | (sign==0x000D) then
   result ← ¬result + 1

return result
```

ConvertSPtoSXWsaturate(x, y)

Let x be a single-precision floating-point value.
Let y be an unsigned integer value.

```
sign            ← x.bit[0]
exp             ← x.bit[1:8]
frac.bit[0:22]  ← x.bit[9:31]
frac.bit[23:30] ← 0b0000_0000
if (exp==255) & (frac!=0) then return (0x0000_0000)    // NaN operand
if (exp==255) & (frac==0) then do                      // infinity operand
   VSCR.SAT ← 1
   return ((sign==1) ? 0x8000_0000 : 0x7FFF_FFFF)
end
if ((exp+Y-127)>30) then do                            // large operand
   VSCR.SAT ← 1
   return ((sign==1) ? 0x8000_0000 : 0x7FFF_FFFF)
end
if ((exp+y-127)<0) then return (0x0000_0000)           // -1.0 < value < 1.0 (value rounds to 0)
significand.bit[0]    ← 0b1
significand.bit[1:31] ← frac
do i = 1 to 31-(exp+Y-127)
   significand ← significand >>ui 1
end
return ((sign==0) ? significand : (¬significand + 1))
```

ConvertSPtoUXWsaturate(x, y)

Let x be a single-precision floating-point value.
Let y be an unsigned integer value.

```
sign            ← x.bit[0]
exp             ← x.bit[1:8]
frac.bit[0:22]  ← x.bit[9:31]
frac.bit[23:30] ← 0b0000_0000
if (exp==255) & (frac!=0) then return (0x0000_0000)    // NaN operand
if (exp==255) & (frac==0) then do                      // infinity operand
   VSCR.SAT ← 1
   return ((sign==1) ? 0x0000_0000 : 0xFFFF_FFFF)
end
if ((exp+Y-127)>31) then do                            // large operand
   VSCR.SAT ← 1
   return ((sign==1) ? 0x0000_0000 : 0xFFFF_FFFF)
end
if ((exp+Y-127)<0) then return (0x0000_0000)           // -1.0 < value < 1.0
                                                       //   value rounds to 0
if( sign==1 ) then do                                  // negative operand
   VSCR.SAT ← 1
   return (0x0000_0000)
end
significand.bit[0]    ← 0b1
significand.bit[1:31] ← frac
do i = 1 to 31-(exp+Y-127)
   significand = significand >>ui 1
end
return (significand)
```

ConvertSXWtoSP(x)

Let x be a 32-bit signed integer value.

```
sign           ← X.bit[0]
exp            ← 32 + 127
frac.bit[0]    ← x.bit[0]
frac.bit[1:32] ← x.bit[0:31]
if (frac==0) return (0x0000_0000)     // Zero Operand
if (sign==1) then frac = ¬frac + 1
do while (frac.bit[0]=0)
   frac ← frac << 1
   exp  ← exp - 1
end
lsb  ← frac.bit[23]
gbit ← frac.bit[24]
xbit ← frac.bit[25:32]!=0
inc  ← (lsb & gbit) | (gbit & xbit)
frac.bit[0:23] ← frac.bit[0:23] + inc
if (carry_out=1) then exp ← exp + 1
result.bit[0]    ← sign
result.bit[1:8]  ← exp
result.bit[9:31] ← frac.bit[1:23]
return (result)
```

ConvertUXWtoSP(x)

Let x be a 32-bit unsigned integer value.

```
exp  ← 31 + 127
frac ← x.bit[0:31]
if (frac==0) return (0x0000_0000)     // Zero Operand
do while( frac0==0 )
   frac ← frac << 1
   exp  ← exp - 1
end
lsb  ← frac.bit[23]
gbit ← frac.bit[24]
xbit ← frac.bit[25:31]!=0
inc  ← (lsb & gbit) | (gbit & xbit)
frac.bit[0:23] ← frac.bit[0:23] + inc
if (carry_out=1) then exp ← exp + 1
result.bit[0]    ← 0b0
result.bit[1:8]  ← exp
result.bit[9:31] ← frac.bit[1:23]
return (result)
```

DUP(x,y)

Return the concatenation of y copies x.

```
DUP(0b01,4)  = 0b01010101
DUP(0b001,3) = 0b001001001
```

EXTZ(x)

Result of extending x on the left with zeros.

```
b ← LENGTH(x)
result ← x & ((1<<b)-1)
```

InvMixColumns(x)

```
do c = 0 to 3
   result.word[c].byte[0] = 0x0E•x.word[c].byte[0] ^ 0x0B•x.word[c].byte[1] ^ 0x0D•x.word[c].byte[2] ^ 0x09•x.word[c].byte[3]
   result.word[c].byte[1] = 0x09•x.word[c].byte[0] ^ 0x0E•x.word[c].byte[1] ^ 0x0B•x.word[c].byte[2] ^ 0x0D•x.word[c].byte[3]
   result.word[c].byte[2] = 0x0D•x.word[c].byte[0] ^ 0x09•x.word[c].byte[1] ^ 0x0E•x.word[c].byte[2] ^ 0x0B•x.word[c].byte[3]
   result.word[c].byte[3] = 0x0B•x.word[c].byte[0] ^ 0x0D•x.word[c].byte[1] ^ 0x09•x.word[c].byte[2] ^ 0x0E•x.word[c].byte[3]
end
return(result);
```

where "•" is a GF(2^8) multiply, a binary polynomial multiplication reduced by modulo `0x11B`.

The GF(2^8) multiply of `0x09`•x can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[0] ^ x.bit[3]
product.bit[1] = x.bit[1] ^ x.bit[4] ^ x.bit[0]
product.bit[2] = x.bit[2] ^ x.bit[5] ^ x.bit[0] ^ x.bit[1]
product.bit[3] = x.bit[3] ^ x.bit[6] ^ x.bit[1] ^ x.bit[2]
product.bit[4] = x.bit[4] ^ x.bit[7] ^ x.bit[0] ^ x.bit[2]
product.bit[5] = x.bit[5] ^ x.bit[0] ^ x.bit[1]
product.bit[6] = x.bit[6] ^ x.bit[1] ^ x.bit[2]
product.bit[7] = x.bit[7] ^ x.bit[2]
```

The GF(2^8) multiply of `0x0B`•x can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[0] ^ x.bit[1] ^ x.bit[3]
product.bit[1] = x.bit[1] ^ x.bit[2] ^ x.bit[4] ^ x.bit[0]
product.bit[2] = x.bit[2] ^ x.bit[3] ^ x.bit[5] ^ x.bit[0] ^ x.bit[1]
product.bit[3] = x.bit[3] ^ x.bit[4] ^ x.bit[6] ^ x.bit[0] ^ x.bit[1] ^ x.bit[2]
product.bit[4] = x.bit[4] ^ x.bit[5] ^ x.bit[7] ^ x.bit[2]
product.bit[5] = x.bit[5] ^ x.bit[6] ^ x.bit[0] ^ x.bit[1]
product.bit[6] = x.bit[6] ^ x.bit[7] ^ x.bit[0] ^ x.bit[1] ^ x.bit[2]
product.bit[7] = x.bit[7] ^ x.bit[0] ^ x.bit[2]
```

The GF(2^8) multiply of `0x0D`•x can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[0] ^ x.bit[2] ^ x.bit[3]
product.bit[1] = x.bit[1] ^ x.bit[3] ^ x.bit[4] ^ x.bit[0]
product.bit[2] = x.bit[2] ^ x.bit[4] ^ x.bit[5] ^ x.bit[1]
product.bit[3] = x.bit[3] ^ x.bit[5] ^ x.bit[6] ^ x.bit[0] ^ x.bit[2]
product.bit[4] = x.bit[4] ^ x.bit[6] ^ x.bit[7] ^ x.bit[0] ^ x.bit[1] ^ x.bit[2]
product.bit[5] = x.bit[5] ^ x.bit[7] ^ x.bit[1]
product.bit[6] = x.bit[6] ^ x.bit[0] ^ x.bit[2]
product.bit[7] = x.bit[7] ^ x.bit[1] ^ x.bit[2]
```

The GF(2^8) multiply of `0x0E`•x can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[1] ^ x.bit[2] ^ x.bit[3]
product.bit[1] = x.bit[2] ^ x.bit[3] ^ x.bit[4] ^ x.bit[0]
product.bit[2] = x.bit[3] ^ x.bit[4] ^ x.bit[5] ^ x.bit[1]
product.bit[3] = x.bit[4] ^ x.bit[5] ^ x.bit[6] ^ x.bit[2]
product.bit[4] = x.bit[5] ^ x.bit[6] ^ x.bit[7] ^ x.bit[1] ^ x.bit[2]
product.bit[5] = x.bit[6] ^ x.bit[7] ^ x.bit[1]
product.bit[6] = x.bit[7] ^ x.bit[2]
product.bit[7] = x.bit[0] ^ x.bit[1] ^ x.bit[2]
```

InvShiftRows(x)

```
result.word[0].byte[0] = x.word[0].byte[0]
result.word[1].byte[0] = x.word[1].byte[0]
result.word[2].byte[0] = x.word[2].byte[0]
result.word[3].byte[0] = x.word[3].byte[0]

result.word[0].byte[1] = x.word[3].byte[1]
result.word[1].byte[1] = x.word[0].byte[1]
result.word[2].byte[1] = x.word[1].byte[1]
result.word[3].byte[1] = x.word[2].byte[1]

result.word[0].byte[2] = x.word[2].byte[2]
result.word[1].byte[2] = x.word[3].byte[2]
result.word[2].byte[2] = x.word[0].byte[2]
result.word[3].byte[2] = x.word[1].byte[2]

result.word[0].byte[3] = x.word[1].byte[3]
result.word[1].byte[3] = x.word[2].byte[3]
result.word[2].byte[3] = x.word[3].byte[3]
result.word[3].byte[3] = x.word[0].byte[3]

return(result)
```

InvSubBytes(x)

```
InvSBOX.byte[256] = { 0x52,0x09,0x6A,0xD5,0x30,0x36,0xA5,0x38,0xBF,0x40,0xA3,0x9E,0x81,0xF3,0xD7,0xFB,
                      0x7C,0xE3,0x39,0x82,0x9B,0x2F,0xFF,0x87,0x34,0x8E,0x43,0x44,0xC4,0xDE,0xE9,0xCB,
                      0x54,0x7B,0x94,0x32,0xA6,0xC2,0x23,0x3D,0xEE,0x4C,0x95,0x0B,0x42,0xFA,0xC3,0x4E,
                      0x08,0x2E,0xA1,0x66,0x28,0xD9,0x24,0xB2,0x76,0x5B,0xA2,0x49,0x6D,0x8B,0xD1,0x25,
                      0x72,0xF8,0xF6,0x64,0x86,0x68,0x98,0x16,0xD4,0xA4,0x5C,0xCC,0x5D,0x65,0xB6,0x92,
                      0x6C,0x70,0x48,0x50,0xFD,0xED,0xB9,0xDA,0x5E,0x15,0x46,0x57,0xA7,0x8D,0x9D,0x84,
                      0x90,0xD8,0xAB,0x00,0x8C,0xBC,0xD3,0x0A,0xF7,0xE4,0x58,0x05,0xB8,0xB3,0x45,0x06,
                      0xD0,0x2C,0x1E,0x8F,0xCA,0x3F,0x0F,0x02,0xC1,0xAF,0xBD,0x03,0x01,0x13,0x8A,0x6B,
                      0x3A,0x91,0x11,0x41,0x4F,0x67,0xDC,0xEA,0x97,0xF2,0xCF,0xCE,0xF0,0xB4,0xE6,0x73,
                      0x96,0xAC,0x74,0x22,0xE7,0xAD,0x35,0x85,0xE2,0xF9,0x37,0xE8,0x1C,0x75,0xDF,0x6E,
                      0x47,0xF1,0x1A,0x71,0x1D,0x29,0xC5,0x89,0x6F,0xB7,0x62,0x0E,0xAA,0x18,0xBE,0x1B,
                      0xFC,0x56,0x3E,0x4B,0xC6,0xD2,0x79,0x20,0x9A,0xDB,0xC0,0xFE,0x78,0xCD,0x5A,0xF4,
                      0x1F,0xDD,0xA8,0x33,0x88,0x07,0xC7,0x31,0xB1,0x12,0x10,0x59,0x27,0x80,0xEC,0x5F,
                      0x60,0x51,0x7F,0xA9,0x19,0xB5,0x4A,0x0D,0x2D,0xE5,0x7A,0x9F,0x93,0xC9,0x9C,0xEF,
                      0xA0,0xE0,0x3B,0x4D,0xAE,0x2A,0xF5,0xB0,0xC8,0xEB,0xBB,0x3C,0x83,0x53,0x99,0x61,
                      0x17,0x2B,0x04,0x7E,0xBA,0x77,0xD6,0x26,0xE1,0x69,0x14,0x63,0x55,0x21,0x0C,0x7D }

do i = 0 to 15
   result.byte[i] = InvSBOX.byte[x.byte[i]]
end
return(result)
```

MixColumns(x)

```
do c = 0 to 3
   result.word[c].byte[0] = 0x02•x.word[c].byte[0] ^ 0x03•x.word[c].byte[1] ^      x.word[c].byte[2] ^      x.word[c].byte[3]
   result.word[c].byte[1] =      x.word[c].byte[0] ^ 0x02•x.word[c].byte[1] ^ 0x03•x.word[c].byte[2] ^      x.word[c].byte[3]
   result.word[c].byte[2] =      x.word[c].byte[0] ^      x.word[c].byte[1] ^ 0x02•x.word[c].byte[2] ^ 0x03•x.word[c].byte[3]
   result.word[c].byte[3] = 0x03•x.word[c].byte[0] ^      x.word[c].byte[1] ^      x.word[c].byte[2] ^ 0x02•x.word[c].byte[3]
end
return(result)
```

The GF(2^8) multiply of `0x02•x` can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[1]
product.bit[1] = x.bit[2]
product.bit[2] = x.bit[3]
product.bit[3] = x.bit[4] ^ x.bit[0]
product.bit[4] = x.bit[5] ^ x.bit[0]
product.bit[5] = x.bit[6]
product.bit[6] = x.bit[7] ^ x.bit[0]
product.bit[7] = x.bit[0]
```

The GF(2^8) multiply of 0x03•x can be expressed in minimized terms as the following.

```
product.bit[0] = x.bit[0] ^ x.bit[1]
product.bit[1] = x.bit[1] ^ x.bit[2]
product.bit[2] = x.bit[2] ^ x.bit[3]
product.bit[3] = x.bit[3] ^ x.bit[4] ^ x.bit[0]
product.bit[4] = x.bit[4] ^ x.bit[5] ^ x.bit[0]
product.bit[5] = x.bit[5] ^ x.bit[6]
product.bit[6] = x.bit[6] ^ x.bit[7] ^ x.bit[0]
product.bit[7] = x.bit[7] ^ x.bit[0]
```

ShiftRows(x)

```
result.word[0].byte[0] = x.word[0].byte[0]
result.word[1].byte[0] = x.word[1].byte[0]
result.word[2].byte[0] = x.word[2].byte[0]
result.word[3].byte[0] = x.word[3].byte[0]

result.word[0].byte[1] = x.word[1].byte[1]
result.word[1].byte[1] = x.word[2].byte[1]
result.word[2].byte[1] = x.word[3].byte[1]
result.word[3].byte[1] = x.word[0].byte[1]

result.word[0].byte[2] = x.word[2].byte[2]
result.word[1].byte[2] = x.word[3].byte[2]
result.word[2].byte[2] = x.word[0].byte[2]
result.word[3].byte[2] = x.word[1].byte[2]

result.word[0].byte[3] = x.word[3].byte[3]
result.word[1].byte[3] = x.word[0].byte[3]
result.word[2].byte[3] = x.word[1].byte[3]
result.word[3].byte[3] = x.word[2].byte[3]

return(result)
```

Signed_BCD_Add(x,y,z)

Let x and y be 31-digit signed decimal values.

Performs a signed decimal addition of x and y.

If the unbounded result is equal to zero, eq_flag is set to 1. Otherwise, eq_flag is set to 0.
If the unbounded result is greater than zero, gt_flag is set to 1. Otherwise, gt_flag is set to 0.
If the unbounded result is less than zero, lt_flag is set to 1. Otherwise, lt_flag is set to 0.

If the magnitude of the unbounded result is greater than 10^{31}-1, ox_flag is set to 1. Otherwise, ox_flag is set to 0.

If the unbounded result is greater than or equal to zero, the sign code of the result is set to 0b1100 if z=0.
If the unbounded result is greater than or equal to zero, the sign code of the result is set to 0b1111 if z=1.
If the unbounded result is less than zero, the sign code of the result is set to 0b1101.

The low-order 31 digits of the unbounded result magnitude concatented with the sign code are returned.

If either operand is an invalid encoding of a signed decimal value, the result returned is undefined and inv_flag is set to 1 and lt_flag, gt_flag and eq_flag are set to 0. Otherwise, inv_flag is set to 0.

Signed_BCD_Subtract(x,y,z)

Let x and y be 31-digit signed decimal values.

Performs a signed decimal subtract of y from x.

If the unbounded result is equal to zero, eq_flag is set to 1. Otherwise, eq_flag is set to 0.
If the unbounded result is greater than zero, gt_flag is set to 1. Otherwise, gt_flag is set to 0.
If the unbounded result is less than zero, lt_flag is set to 1. Otherwise, lt_flag is set to 0.

If the magnitude of the unbounded result is greater than 10^{31}-1, ox_flag is set to 1. Otherwise, ox_flag is set to 0.

If the unbounded result is greater than or equal to zero, the sign code of the result is set to 0b1100 if z=0.
If the unbounded result is greater than or equal to zero, the sign code of the result is set to 0b1111 if z=1.
If the unbounded result is less than zero, the sign code of the result is set to 0b1101.

The low-order 31 digits of the unbounded result magnitude concatented with the sign code are returned.

If either operand is an invalid encoding of a signed decimal value, the result returned is undefined and inv_flag is set to 1 and lt_flag, gt_flag and eq_flag are set to 0. Otherwise, inv_flag is set to 0.

SubBytes(x)

```
SBOX.byte[0:255] = { 0x63,0x7C,0x77,0x7B,0xF2,0x6B,0x6F,0xC5,0x30,0x01,0x67,0x2B,0xFE,0xD7,0xAB,0x76,
                     0xCA,0x82,0xC9,0x7D,0xFA,0x59,0x47,0xF0,0xAD,0xD4,0xA2,0xAF,0x9C,0xA4,0x72,0xC0,
                     0xB7,0xFD,0x93,0x26,0x36,0x3F,0xF7,0xCC,0x34,0xA5,0xE5,0xF1,0x71,0xD8,0x31,0x15,
                     0x04,0xC7,0x23,0xC3,0x18,0x96,0x05,0x9A,0x07,0x12,0x80,0xE2,0xEB,0x27,0xB2,0x75,
                     0x09,0x83,0x2C,0x1A,0x1B,0x6E,0x5A,0xA0,0x52,0x3B,0xD6,0xB3,0x29,0xE3,0x2F,0x84,
                     0x53,0xD1,0x00,0xED,0x20,0xFC,0xB1,0x5B,0x6A,0xCB,0xBE,0x39,0x4A,0x4C,0x58,0xCF,
                     0xD0,0xEF,0xAA,0xFB,0x43,0x4D,0x33,0x85,0x45,0xF9,0x02,0x7F,0x50,0x3C,0x9F,0xA8,
                     0x51,0xA3,0x40,0x8F,0x92,0x9D,0x38,0xF5,0xBC,0xB6,0xDA,0x21,0x10,0xFF,0xF3,0xD2,
                     0xCD,0x0C,0x13,0xEC,0x5F,0x97,0x44,0x17,0xC4,0xA7,0x7E,0x3D,0x64,0x5D,0x19,0x73,
                     0x60,0x81,0x4F,0xDC,0x22,0x2A,0x90,0x88,0x46,0xEE,0xB8,0x14,0xDE,0x5E,0x0B,0xDB,
                     0xE0,0x32,0x3A,0x0A,0x49,0x06,0x24,0x5C,0xC2,0xD3,0xAC,0x62,0x91,0x95,0xE4,0x79,
                     0xE7,0xC8,0x37,0x6D,0x8D,0xD5,0x4E,0xA9,0x6C,0x56,0xF4,0xEA,0x65,0x7A,0xAE,0x08,
                     0xBA,0x78,0x25,0x2E,0x1C,0xA6,0xB4,0xC6,0xE8,0xDD,0x74,0x1F,0x4B,0xBD,0x8B,0x8A,
                     0x70,0x3E,0xB5,0x66,0x48,0x03,0xF6,0x0E,0x61,0x35,0x57,0xB9,0x86,0xC1,0x1D,0x9E,
                     0xE1,0xF8,0x98,0x11,0x69,0xD9,0x8E,0x94,0x9B,0x1E,0x87,0xE9,0xCE,0x55,0x28,0xDF,
                     0x8C,0xA1,0x89,0x0D,0xBF,0xE6,0x42,0x68,0x41,0x99,0x2D,0x0F,0xB0,0x54,0xBB,0x16 }

do i = 0 to 15
   result.byte[i] = SBOX.byte[x.byte[i]]
end
return(result)
```

RoundToSPIntCeil(x)

The value x if x is a single-precision floating-point integer; otherwise the smallest single-precision floating-point integer that is greater than x.

RoundToSPIntFloor(x)

The value x if x is a single-precision floating-point integer; otherwise the largest single-precision floating-point integer that is less than x.

RoundToSPIntNear(x)

The value x if x is a single-precision floating-point integer; otherwise the single-precision floating-point integer that is nearest in value to x (in case of a tie, the even single-precision floating-point integer is used).

RoundToSPIntTrunc(x)

The value x if x is a single-precision floating-point integer; otherwise the largest single-precision floating-point integer that is less than x if x>0, or the smallest single-precision floating-point integer that is greater than x if x<0.

RoundToNearSP(x)

The single-precision floating-point number that is nearest in value to the infinitely-precise floating-point intermediate result x (in case of a tie, the single-precision floating-point value with the least-significant bit equal to 0 is used).

ReciprocalEstimateSP(x)
A single-precision floating-point estimate of the reciprocal of the single-precision floating-point number x.

ReciprocalSquareRootEstimateSP(x)
A single-precision floating-point estimate of the reciprocal of the square root of the single-precision floating-point number x.

LogBase2EstimateSP(x)
A single-precision floating-point estimate of the base 2 logarithm of the single-precision floating-point number x.

Power2EstimateSP(x)
A single-precision floating-point estimate of the 2 raised to the power of the single-precision floating-point number x.

6.3 Vector Facility Registers

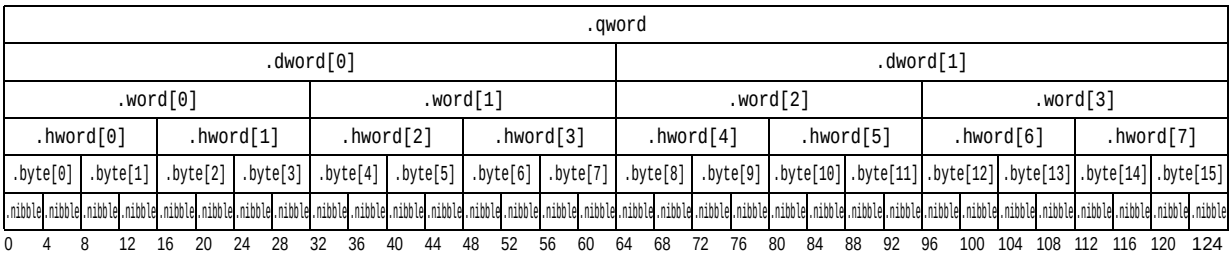


Figure 97. Vector Register elements

6.3.1 Vector Registers

There are 32 Vector Registers (VRs), each containing 128 bits. See Figure 98. All computations and other data manipulation are performed on data residing in Vector Registers, and results are placed into a VR.

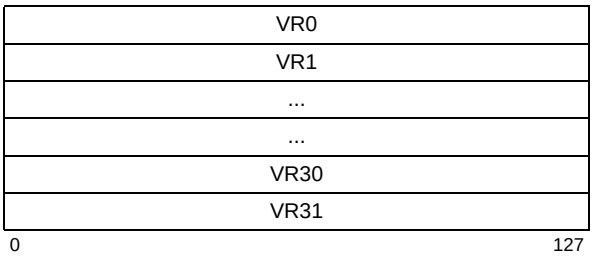


Figure 98. Vector Registers

Depending on the instruction, the contents of a Vector Register are interpreted as a sequence of equal-length elements (bytes, halfwords, or words) or as a quadword. Each of the elements is aligned within the Vector Register, as shown in Figure 97. Many instructions perform a given operation in parallel on all elements in a Vector Register. Depending on the instruction, a byte, halfword, or word element can be interpreted as a signed-integer, an unsigned-integer, or a logical value; a word element can also be interpreted as a single-precision floating-point value. In the instruction descriptions, phrases like "signed-integer word element" are used as shorthand for "word element, interpreted as a signed-integer".

Load and *Store* instructions are provided that transfer a byte, halfword, word, or quadword between storage and a Vector Register.

6.3.2 Vector Status and Control Register

The Vector Status and Control Register (VSCR) is a special 32-bit register (not an SPR) that is read and written in a manner similar to the FPSCR in the Power ISA scalar floating-point unit. Special instructions (***mfvscr*** and ***mtvscr***) are provided to move the VSCR from and to a vector register. When moved to or from a vector register, the 32-bit VSCR is right justified in the 128-bit vector register. When moved to a vector register, bits 0:95 of the vector register are cleared (set to 0).

Figure 99. Vector Status and Control Register

The bit definitions for the VSCR are as follows.

Bit(s)	Description
96:110	Reserved
111	***Vector Non-Java Mode*** (NJ) This bit controls how denormalized values are handled by *Vector Floating-Point* instructions. 0 Denormalized values are handled as specified by Java and the IEEE standard; see Section 6.6.1. 1 If an element in a source VR contains a denormalized value, the value 0 is used instead. If an instruction causes an Underflow Exception, the corresponding element in the target VR is set to 0. In both cases the 0 has the same sign as the denormalized or underflowing value.
112:126	Reserved
127	***Vector Saturation*** (SAT)

Every vector instruction having “Saturate” in its name implicitly sets this bit to 1 if any result of that instruction “saturates”; see Section 6.8. ***mtvscr*** can alter this bit explicitly. This bit is sticky; that is, once set to 1 it remains set to 1 until it is set to 0 by an ***mtvscr*** instruction.

After the ***mfvscr*** instruction executes, the result in the target vector register will be architecturally precise. That is, it will reflect all updates to the SAT bit that could have been made by vector instructions logically preceding it in the program flow, and further, it will not reflect any SAT updates that may be made to it by vector instructions logically following it in the program flow. To implement this, processors may choose to make the ***mfvscr*** instruction execution serializing within the vector unit, meaning that it will stall vector instruction execution until all preceding vector instructions are complete and have updated the architectural machine state. This is permitted in order to simplify implementation of the sticky status bit (SAT) which would otherwise be difficult to implement in an out-of-order execution machine. The implication of this is that reading the VSCR can be much slower than typical Vector instructions, and therefore care must be taken in reading it, as advised in Section 6.5.1, to avoid performance problems.

The ***mtvscr*** is context synchronizing. This implies that all Vector instructions logically preceding an ***mtvscr*** in the program flow will execute in the architectural context (NJ mode) that existed prior to completion of the ***mtvscr***, and that all instructions logically following the ***mtvscr*** will execute in the new context (NJ mode) established by the ***mtvscr***.

6.3.3 VR Save Register

The VR Save Register (VRSAVE) is a 32-bit register in the fixed-point processor provided for application and operating system use; see Section 3.2.3.

Programming Note

The VRSAVE register can be used to indicate which VRs are currently being used by a program. If this is done, the operating system could save only those VRs when an “interrupt” occurs (see Book III), and could restore only those VRs when resuming the interrupted program.

If this approach is taken it must be applied rigorously; if a program fails to indicate that a given VR is in use, software errors may occur that will be difficult to detect and correct because they are timing-dependent.

Some operating systems save and restore VRSAVE only for programs that also use other vector registers.

6.4 Vector Storage Access Operations

The *Vector Storage Access* instructions provide the means by which data can be copied from storage to a Vector Register or from a Vector Register to storage. Instructions are provided that access byte, halfword, word, and quadword storage operands. These instructions differ from the fixed-point and floating-point *Storage Access* instructions in that vector storage operands are assumed to be aligned, and vector storage accesses are performed as if the appropriate number of low-order bits of the specified effective address (EA) were zero. For example, the low-order bit of EA is ignored for halfword *Vector Storage Access* instructions, and the low-order four bits of EA are ignored for quadword *Vector Storage Access* instructions. The effect is to load or store the storage operand of the specified length that contains the byte addressed by EA.

If a storage operand is unaligned, additional instructions must be used to ensure that the operand is correctly placed in a Vector Register or in storage. Instructions are provided that shift and merge the contents of two Vector Registers, such that an unaligned quadword storage operand can be copied between storage and the Vector Registers in a relatively efficient manner.

As shown in Figure 97, the elements in Vector Registers are numbered; the high-order (or most significant) byte element is numbered 0 and the low-order (or least significant) byte element is numbered 15. The numbering affects the values that must be placed into the permute control vector for the *Vector Permute* instruction in order for that instruction to achieve the desired effects, as illustrated by the examples in the following subsections.

A vector quadword *Load* instruction for which the effective address (EA) is quadword-aligned places the byte in storage addressed by EA into byte element 0 of the target Vector Register, the byte in storage addressed by EA+1 into byte element 1 of the target Vector Register, etc. Similarly, a vector quadword *Store* instruction for which the EA is quadword-aligned places the contents of byte element 0 of the source Vector Register into the byte in storage addressed by EA, the contents of byte element 1 of the source Vector Register into the byte in storage addressed by EA+1, etc.

Figure 100 shows an aligned quadword in storage. Figure 101 shows the result of loading that quadword into a Vector Register or, equivalently, shows the contents that must be in a Vector Register if storing that Vector Register is to produce the storage contents shown in Figure 100.

When an aligned byte, halfword, or word storage operand is loaded into a Vector Register, the element (byte, halfword, or word respectively) that receives the data is the element that would have received the data had the entire aligned quadword containing the storage operand addressed by EA been loaded. Similarly, when a byte, halfword, or word element in a Vector Register is stored into an aligned storage operand (byte, halfword, or word respectively), the element selected to be stored is the element that would have been stored into the storage operand addressed by EA had the entire Vector Register been stored to the aligned quadword containing the storage operand addressed by EA. (Byte storage operands are always aligned.)

For aligned byte, halfword, and word storage operands, if the corresponding element number is known when the program is written, the appropriate *Vector Splat* and *Vector Permute* instructions can be used to copy or replicate the data contained in the storage operand after loading the operand into a Vector Register. An example of this is given in the Programming Note for *Vector Splat*; see page 257. Another example is to replicate the element across an entire Vector Register before storing it into an arbitrary aligned storage operand of the same length; the replication ensures that the correct data are stored regardless of the offset of the storage operand in its aligned quadword in storage.

00	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
10																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Figure 100.Aligned quadword storage operand

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Figure 101.Vector Register contents for aligned quadword Load or Store

00												00	01	02	03	04
10	05	06	07	08	09	0A	0B	0C	0D	0E	0F					
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Figure 102.Unaligned quadword storage operand

Vhi												00	01	02	03	04
Vlo	05	06	07	08	09	0A	0B	0C	0D	0E	0F					

Vt,Vs	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
	0															15

Figure 103.Vector Register contents

6.4.1 Accessing Unaligned Storage Operands

Figure 102 shows an unaligned quadword storage operand that spans two aligned quadwords. In the remainder of this section, the aligned quadword that contains the most significant bytes of the unaligned quadword is called the most significant quadword (MSQ) and the aligned quadword that contains the least significant bytes of the unaligned quadword is called the least significant quadword (LSQ). Because the V*ector Storage Access* instructions ignore the low-order bits of the effective address, the unaligned quadword cannot be transferred between storage and a Vector Register using a single instruction. The remainder of this section gives examples of accessing unaligned quadword storage operands. Similar sequences can be used to access unaligned halfword and word storage operands.

Programming Note

The sequence of instructions given below is one approach that can be used to load the unaligned quadword shown in Figure 102 into a Vector Register. In Figure 103 Vhi and Vlo are the Vector Registers that will receive the most significant quadword and least significant quadword respectively. VRT is the target Vector Register.

After the two quadwords have been loaded into Vhi and Vlo, using *Load Vector Indexed* instructions, the alignment is performed by shifting the 32-byte quantity Vhi || Vlo left by an amount determined by the address of the first byte of the desired data. The shifting is done using a *Vector Permute* instruction for which the permute control vector is generated by a *Load Vector for Shift Left* instruction. The *Load Vector* for *Shift Left* instruction uses the same address specification as the *Load Vector Indexed* instruction that loads the Vhi register; this is the address of the desired unaligned quadword.

The following sequence of instructions copies the unaligned quadword storage operand into register Vt.

```
# Assumptions:
# Rb != 0 and contents of Rb = 0xB
lvx    Vhi,0,Rb       # load MSQ
lvsl   Vp,0,Rb        # set permute control vector
addi   Rb,Rb,16       # address of LSQ
lvx    Vlo,0,Rb       # load LSQ
vperm  Vt,Vhi,Vlo,Vp  # align the data
```

The procedure for storing an unaligned quadword is essentially the reverse of the procedure for loading one. However, a read-modify-write sequence is required that inserts the source quadword into two aligned quadwords in storage. The quadword to be stored is assumed to be in Vs; see Figure 103 The contents of Vs are shifted right and split into two parts, each of which is merged (using a Vector Select instruction) with the current contents of the two aligned quadwords (MSQ and LSQ) that will contain the most significant bytes and least significant bytes, respectively, of the unaligned quadword. The resulting two quadwords are stored using *Store Vector Indexed* instructions. A *Load Vector for Shift Right* instruction is used to generate the permute control vector that is used for the shifting. A single register is used for the "shifted" contents; this is possible because the "shifting" is done by means of a right rotation. The rotation is accomplished by specifying Vs for both components of the *Vector Permute* instruction. In addition, the same permute control vector is used on a sequence of 1s and 0s to generate the mask used by the *Vector Select* instructions that do the merging.

The following sequence of instructions copies the contents of Vs into an unaligned quadword in storage.

```
# Assumptions:
# Rb != 0 and contents of Rb = 0xB
lvx       Vhi,0,Rb     # load current MSQ
lvsr      Vp,0,Rb      # set permute control vector
addi      Rb,Rb,16     # address of LSQ
lvx       Vlo,0,Rb     # load current LSQ
vspltisb  V1s,-1       # generate the select mask bits
vspltisb  V0s,0
vperm     Vmask,V0s,V1s,Vp # generate the select mask
vperm     Vs,Vs,Vs,Vp      # right rotate the data
vsel      Vlo,Vs,Vlo,Vmask # insert LSQ component
vsel      Vhi,Vhi,Vs,Vmask # insert MSQ component
stvx      Vlo,0,Rb         # store LSQ
addi      Rb,Rb,-16        # address of MSQ
stvx      Vhi,0,Rb         # store MSQ
```

6.5 Vector Integer Operations

Many of the instructions that produce fixed-point integer results have the potential to compute a result value that cannot be represented in the target format. When this occurs, this unrepresentable intermediate value is converted to a representable result value using one of the following methods.

1. The high-order bits of the intermediate result that do not fit in the target format are discarded. This method is used by instructions having names that include the word "Modulo".

2. The intermediate result is converted to the nearest value that is representable in the target format (i.e., to the minimum or maximum representable value, as appropriate). This method is used by instructions having names that include the word "Saturate". An intermediate result that is forced to the minimum or maximum representable value as just described is said to "saturate".

   An instruction for which an intermediate result saturates causes $VSCR_{SAT}$ to be set to 1; see Section 6.3.2.

3. If the intermediate result includes non-zero fraction bits it is rounded up to the nearest fixed-point integer value. This method is used by the six *Vector Average Integer* instructions and by the *Vector Multiply-High-Round-Add Signed Halfword Saturate* instruction. The latter instruction then uses method 2, if necessary.

Programming Note

Because $VSCR_{SAT}$ is sticky, it can be used to detect whether any instruction in a sequence of "Saturate"-type instructions produced an inexact result due to saturation. For example, the contents of the VSCR can be copied to a VR (***mfvscr***), bits other than the SAT bit can be cleared in the VR (***vand*** with a constant), the result can be compared to zero setting CR6 (***vcmpequb***.), and a branch can be taken according to whether $VSCR_{SAT}$ was set to 1 (*Branch Conditional* that tests CR field 6).

Testing $VSCR_{SAT}$ after each "Saturate"-type instruction would degrade performance considerably. Alternative techniques include the following:

– Retain sufficient information at "checkpoints" that the sequence of computations performed between one checkpoint and the next can be redone (more slowly) in a manner that detects exactly when saturation occurs. Test $VSCR_{SAT}$ only at checkpoints, or when redoing a sequence of computations that saturated.

– Perform intermediate computations using an element length sufficient to prevent saturation, and then use a *Vector Pack Integer Saturate* instruction to pack the final result to the desired length. (Vector Pack Integer Saturate causes results to saturate if necessary, and sets $VSCR_{SAT}$ to 1 if any result saturates.)

6.5.1 Integer Saturation

Saturation occurs whenever the result of a saturating instruction does not fit in the result field. Unsigned saturation clamps results to zero (0) on underflow and to the maximum positive integer value (2^n-1, e.g. 255 for byte fields) on overflow. Signed saturation clamps results to the smallest representable negative number (-2^{n-1}, e.g. -128 for byte fields) on underflow, and to the largest representable positive number (2^{n-1}-1, e.g. +127 for byte fields) on overflow.

In most cases, the simple maximum/minimum saturation performed by the vector instructions is adequate. However, sometimes, e.g. in the creation of very high quality images, more complex saturation functions must be applied. To support this, the Vector facility provides a mechanism for detecting that saturation has occurred. The VSCR has a bit, the SAT bit, which is set to a one (1) anytime any field in a saturating instruction saturates. The SAT bit can only be cleared by explicitly writing zero to it. Thus SAT accumulates a summary result of any integer overflow or underflow that occurs on a saturating instruction.

Borderline cases that generate results equal to saturation values, for example unsigned 0+0=0 and unsigned byte 1+254=255, are not considered saturation conditions and do not cause SAT to be set.

The SAT bit can be set by the following types of instructions:

- Move To VSCR
- Vector Add Integer with Saturation
- Vector Subtract Integer with Saturation
- Vector Multiply-Add Integer with Saturation
- Vector Multiply-Sum with Saturation
- Vector Sum-Across with Saturation
- Vector Pack with Saturation
- Vector Convert to Fixed-point with Saturation

Note that only instructions that explicitly call for “saturation” can set SAT. “Modulo” integer instructions and floating-point arithmetic instructions never set SAT.

Programming Note

The SAT state can be tested and used to alter program flow by moving the VSCR to a vector register (with ***mfvscr***), then masking out bits 0:126 (to clear undefined and reserved bits) and performing a vector compare equal-to unsigned byte w/record (***vcmpequb***.) with zero to get a testable value into the condition register for consumption by a subsequent branch.

Since ***mfvscr*** will be slow compared to other Vector instructions, reading and testing SAT after each instruction would be prohibitively expensive. Therefore, software is advised to employ strategies that minimize checking SAT. For example: checking SAT periodically and backtracking to the last checkpoint to identify exactly which field in which instruction saturated; or, working in an element size sufficient to prevent any overflow or underflow during intermediate calculations, then packing down to the desired element size as the final operation (the vector pack instruction saturates the results and updates SAT when a loss of significance is detected).

6.6 Vector Floating-Point Operations

6.6.1 Floating-Point Overview

Unless $VSCR_{NJ}$=1 (see Section 6.3.2), the floating-point model provided by the Vector Facility conforms to The Java Language Specification (hereafter referred to as “Java”), which is a subset of the default environment specified by the IEEE standard (i.e., by ANSI/IEEE Standard 754-1985, “IEEE Standard for Binary Floating-Point Arithmetic”). For aspects of floating-point behavior that are not defined by Java but are defined by the IEEE standard, vector floating-point conforms to the IEEE standard. For aspects of floating-point behavior that are defined neither by Java nor by the IEEE standard but are defined by the “C9X Floating-Point Proposal” (hereafter referred to as “C9X”), vector floating-point conforms to C9X.

The single-precision floating-point data format, value representations, and computational models defined in Chapter 4. “Floating-Point Facility” on page 123 apply to vector floating-point except as follows.

– In general, no status bits are set to reflect the results of floating-point operations. The only exception is that $VSCR_{SAT}$ may be set by the *Vector Convert To Fixed-Point Word* instructions.

– With the exception of the two *Vector Convert To Fixed-Point Word* instructions and three of the four *Vector Round to Floating-Point Integer* instructions, all vector floating-point instructions that round use the rounding mode Round to Nearest.

– Floating-point exceptions (see Section 6.6.2) cannot cause the system error handler to be invoked.

Programming Note

If a function is required that is specified by the IEEE standard, is not supported by the Vector Facility, and cannot be emulated satisfactorily using the functions that are supported by the Vector Facility, the functions provided by the Floating-Point Facility should be used; see Chapter 4.

6.6.2 Floating-Point Exceptions

The following floating-point exceptions may occur during execution of vector floating-point instructions.

– NaN Operand Exception
– Invalid Operation Exception
– Zero Divide Exception
– Log of Zero Exception
– Overflow Exception
– Underflow Exception

If an exception occurs, a result is placed into the corresponding target element as described in the following subsections. This result is the default result specified by Java, the IEEE standard, or C9X, as applicable.

Recall that denormalized source values are treated as if they were zero when $VSCR_{NJ}$=1. This has the following consequences regarding exceptions.

– Exceptions that can be caused by a zero source value can be caused by a denormalized source value when $VSCR_{NJ}$=1.

– Exceptions that can be caused by a nonzero source value cannot be caused by a denormalized source value when $VSCR_{NJ}$=1.

6.6.2.1 NaN Operand Exception

A NaN Operand Exception occurs when a source value for any of the following instructions is a NaN.

– A vector instruction that would normally produce floating-point results

– Either of the two *Vector Convert To Fixed-Point Word* instructions

– Any of the four *Vector Floating-Point Compare* instructions

The following actions are taken:

If the vector instruction would normally produce floating-point results, the corresponding result is a source NaN selected as follows. In all cases, if the selected source NaN is a Signaling NaN it is converted to the corresponding Quiet NaN (by setting the high-order bit of the fraction field to 1) before being placed into the target element.

```
if the element in VRA is a NaN
  then the result is that NaN
  else if the element in VRB is a NaN
       then the result is that NaN
       else if the element in VRC is a NaN
```

then the result is that NaN
else if Invalid Operation exception
(Section 6.6.2.2)
then the result is the QNaN 0x7FC0_0000

If the instruction is either of the two *Vector Convert To Fixed-Point Word* instructions, the corresponding result is 0x0000_0000. $VSCR_{SAT}$ is not affected.

If the instruction is *Vector Compare Bounds Floating-Point*, the corresponding result is 0xC000_0000.

If the instruction is one of the other *Vector Floating-Point Compare* instructions, the corresponding result is 0x0000_0000.

6.6.2.2 Invalid Operation Exception

An Invalid Operation Exception occurs when a source value or set of source values is invalid for the specified operation. The invalid operations are:

- Magnitude subtraction of infinities
- Multiplication of infinity by zero
- Reciprocal square root estimate of a negative, nonzero number or -infinity.
- Log base 2 estimate of a negative, nonzero number or -infinity.

The corresponding result is the QNaN 0x7FC0_0000.

6.6.2.3 Zero Divide Exception

A Zero Divide Exception occurs when a V*ector Reciprocal Estimate Floating-Point* or *Vector Reciprocal Square Root Estimate Floating-Point* instruction is executed with a source value of zero.

The corresponding result is an infinity, where the sign is the sign of the source value.

6.6.2.4 Log of Zero Exception

A Log of Zero Exception occurs when a *Vector Log Base 2 Estimate Floating-Point* instruction is executed with a source value of zero.

The corresponding result is -Infinity.

6.6.2.5 Overflow Exception

An Overflow Exception occurs under either of the following conditions.

- For a vector instruction that would normally produce floating-point results, the magnitude of what would have been the result if the exponent range were unbounded exceeds that of the largest finite floating-point number for the target floating-point format.
- For either of the two *Vector Convert To Fixed-Point Word* instructions, either a source value is an infinity or the product of a source value and 2^{UIM} is a number too large in magnitude to be represented in the target fixed-point format.

The following actions are taken:

1. If the vector instruction would normally produce floating-point results, the corresponding result is an infinity, where the sign is the sign of the intermediate result.
2. If the instruction is *Vector Convert To Unsigned Fixed-Point Word Saturate*, the corresponding result is 0xFFFF_FFFF if the source value is a positive number or +infinity, and is 0x0000_0000 if the source value is a negative number or -infinity. $VSCR_{SAT}$ is set to 1.
3. If the instruction is *Vector Convert To Signed Fixed-Point Word Saturate*, the corresponding result is 0x7FFF_FFFF if the source value is a positive number or +infinity., and is 0x8000_0000 if the source value is a negative number or -infinity. $VSCR_{SAT}$ is set to 1.

6.6.2.6 Underflow Exception

An Underflow Exception can occur only for vector instructions that would normally produce floating-point results. It is detected before rounding. It occurs when a nonzero intermediate result computed as though both the precision and the exponent range were unbounded is less in magnitude than the smallest normalized floating-point number for the target floating-point format.

The following actions are taken:

1. If $VSCR_{NJ}$=0, the corresponding result is the value produced by denormalizing and rounding the intermediate result.
2. If $VSCR_{NJ}$=1, the corresponding result is a zero, where the sign is the sign of the intermediate result.

6.7 Vector Storage Access Instructions

The *Vector Storage Access* instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.11.3, “Effective Address Calculation” on page 27. The low-order bits of the EA that would correspond to an unaligned storage operand are ignored.

The *Load Vector Element Indexed* and *Store Vector Element Indexed* instructions transfer a byte, halfword, or word element between storage and a Vector Register. The *Load Vector Indexed* and *Store Vector Indexed* instructions transfer an aligned quadword between storage and a Vector Register.

6.7.1 Storage Access Exceptions

Storage accesses will cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (*Store* only), or if the program attempts to access storage that is unavailable.

6.7.2 Vector Load Instructions

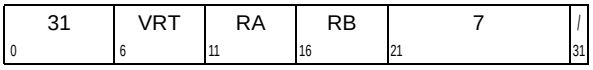
The aligned byte, halfword, word, or quadword in storage addressed by EA is loaded into register VRT.

Programming Note

The *Load Vector Element* instructions load the specified element into the same location in the target register as the location into which it would be loaded using the *Load Vector* instruction.

Load Vector Element Byte Indexed X-form

lvebx VRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← b + (RB)
eb  ← EA60:63

VRT ← undefined
if Big-Endian byte ordering then
   VRT8×eb:8×eb+7 ← MEM(EA,1)
else
   VRT120-(8×eb):127-(8×eb) ← MEM(EA,1)
```

Let the effective address (EA) be the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access, the contents of the byte in storage at address EA are placed into byte eb of register VRT. The remaining bytes in register VRT are set to undefined values.

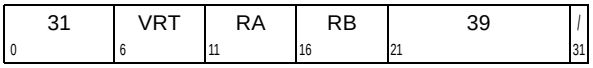
If Little-Endian byte ordering is used for the storage access, the contents of the byte in storage at address EA are placed into byte 15-eb of register VRT. The remaining bytes in register VRT are set to undefined values.

Special Registers Altered:

None

Load Vector Element Halfword Indexed X-form

lvehx VRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← (b + (RB)) & 0xFFFF_FFFF_FFFF_FFFE
eb ← EA60:63

VRT ← undefined
if Big-Endian byte ordering then
   VRT8×eb:8×eb+15 ← MEM(EA,2)
else
   VRT112-(8×eb):127-(8×eb) ← MEM(EA,2)
```

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFE with the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

- the contents of the byte in storage at address EA are placed into byte eb of register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte eb+1 of register VRT, and
- the remaining bytes in register VRT are set to undefined values.

If Little-Endian byte ordering is used for the storage access,

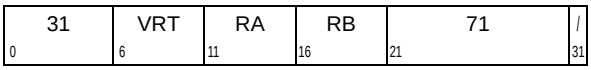
- the contents of the byte in storage at address EA are placed into byte 15-eb of register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte 14-eb of register VRT, and
- the remaining bytes in register VRT are set to undefined values.

Special Registers Altered:

None

Load Vector Element Word Indexed X-form

lvewx VRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← (b + (RB)) & 0xFFFF_FFFF_FFFF_FFFC

eb  ← EA60:63
VRT ← undefined
if Big-Endian byte ordering then
   VR T8×eb:8×eb+31 ← MEM(EA,4)
else
   VRT96-(8×eb):127-(8×eb) ← MEM(EA,4)
```

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFC with the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

- the contents of the byte in storage at address EA are placed into byte eb of register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte eb+1 of register VRT,
- the contents of the byte in storage at address EA+2 are placed into byte eb+2 of register VRT,
- the contents of the byte in storage at address EA+3 are placed into byte eb+3 of register VRT, and
- the remaining bytes in register VRT are set to undefined values.

If if Little-Endian byte ordering is used for the storage access,

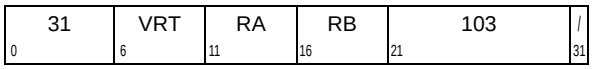
- the contents of the byte in storage at address EA are placed into byte 15-eb of register VRT,
- the contents of the byte in storage at address EA+1 are placed into byte 14-eb of register VRT,
- the contents of the byte in storage at address EA+2 are placed into byte 13-eb of register VRT,
- the contents of the byte in storage at address EA+3 are placed into byte 12-eb of register VRT, and
- the remaining bytes in register VRT are set to undefined values.

Special Registers Altered:

None

Load Vector Indexed X-form

lvx VRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← b + (RB)
VRT ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16)
```

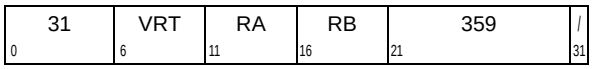
Let the effective address (EA) be the sum (RA|0)+(RB). The quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0 is loaded into VRT.

Special Registers Altered:

None

Load Vector Indexed Last X-form

lvxl VRT,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← b + (RB)
VRT ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16)
mark_as_not_likely_to_be_needed_again_anytime_soon(EA)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0 is loaded into VRT.

lvxl provides a hint that the quadword in storage addressed by EA will probably not be needed again by the program in the near future.

Special Registers Altered:

None

Programming Note

On some implementations, the hint provided by the ***lvxl*** instruction and the corresponding hint provided by the ***stvxl***, ***lvepxl***, and ***stvepxl*** instructions are applied to the entire cache block containing the specified quadword. On such implementations, the effect of the hint may be to cause that cache block to be considered a likely candidate for replacement when space is needed in the cache for a new block. Thus, on such implementations, the hint should be used with caution if the cache block containing the quadword also contains data that may be needed by the program in the near future. Also, the hint may be used before the last reference in a sequence of references to the quadword if the subsequent references are likely to occur sufficiently soon that the cache block containing the quadword is not likely to be displaced from the cache before the last reference.

6.7.3 Vector Store Instructions

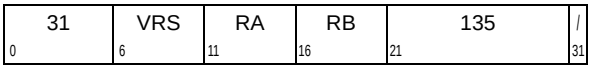
Some portion or all of the contents of VRS are stored into the aligned byte, halfword, word, or quadword in storage addressed by EA.

Programming Note

The *Store Vector Element* instructions store the specified element into the same storage location as the location into which it would be stored using the *Store Vector* instruction.

Store Vector Element Byte Indexed X-form

stvebx VRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← b + (RB)
eb  ← EA60:63
if Big-Endian byte ordering then
   MEM(EA,1) ← VRS8×eb:8×eb+7
else
   MEM(EA,1) ← VRS120-(8×eb):127-(8×eb)
```

Let the effective address (EA) be the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access, the contents of byte eb of register VRS are placed in the byte in storage at address EA.

If Little-Endian byte ordering is used for the storage access, the contents of byte 15-eb of register VRS are placed in the byte in storage at address EA.

Special Registers Altered:

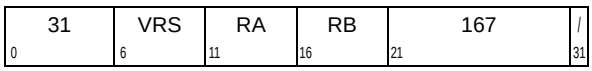
None

Programming Note

Unless bits 60:63 of the address are known to match the byte offset of the subject byte element in register VRS, software should use *Vector Splat* to splat the subject byte element before performing the store.

Store Vector Element Halfword Indexed X-form

stvehx VRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← (b + (RB)) & 0xFFFF_FFFF_FFFF_FFFE
eb  ← EA60:63
if Big-Endian byte ordering then
   MEM(EA,2) ← VRS8×eb:8×eb+15
else
   MEM(EA,2) ← VRS112-(8×eb):127-(8×eb)
```

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFE with the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

- the contents of byte eb of register VRS are placed in the byte in storage at address EA, and
- the contents of byte eb+1 of register VRS are placed in the byte in storage at address EA+1.

If Little-Endian byte ordering is used for the storage access,

- the contents of byte 15-eb of register VRS are placed in the byte in storage at address EA, and
- the contents of byte 14-eb of register VRS are placed in the byte in storage at address EA+1.

Special Registers Altered:

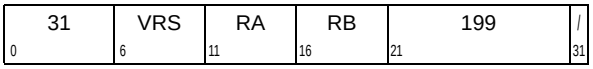
None

Programming Note

Unless bits 60:62 of the address are known to match the halfword offset of the subject halfword element in register VRS software should use *Vector Splat* to splat the subject halfword element before performing the store.

Store Vector Element Word Indexed X-form

stvewx VRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← (b + (RB)) & 0xFFFF_FFFF_FFFF_FFFC
eb  ← EA60:63
if Big-Endian byte ordering then
   MEM(EA,4) ← VRS8×eb:8×eb+31
else
   MEM(EA,4) ← VRS96-(8×eb):127-(8×eb)
```

Let the effective address (EA) be the result of ANDing 0xFFFF_FFFF_FFFF_FFFC with the sum (RA|0)+(RB).

Let eb be bits 60:63 of EA.

If Big-Endian byte ordering is used for the storage access,

- the contents of byte eb of register VRS are placed in the byte in storage at address EA,
- the contents of byte eb+1 of register VRS are placed in the byte in storage at address EA+1,
- the contents of byte eb+2 of register VRS are placed in the byte in storage at address EA+2, and
- the contents of byte eb+3 of register VRS are placed in the byte in storage at address EA+3.

If Little-Endian byte ordering is used for the storage access,

- the contents of byte 15-eb of register VRS are placed in the byte in storage at address EA,
- the contents of byte 14-eb of register VRS are placed in the byte in storage at address EA+1,
- the contents of byte 13-eb of register VRS are placed in the byte in storage at address EA+2, and
- the contents of byte 12-eb of register VRS are placed in the byte in storage at address EA+3.

Special Registers Altered:

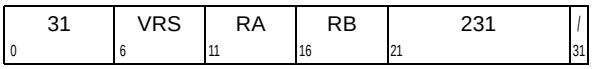
None

Programming Note

Unless bits 60:61 of the address are known to match the word offset of the subject word element in register VRS, software should use *Vector Splat* to splat the subject word element before performing the store.

Store Vector Indexed X-form

stvx VRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← b + (RB)
MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16) ← (VRS)
```

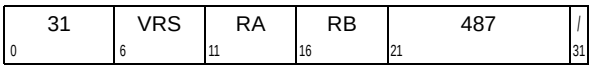
Let the effective address (EA) be the sum (RA|0)+(RB). The contents of VRS are stored into the quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0.

Special Registers Altered:

None

Store Vector Indexed Last X-form

stvxl VRS,RA,RB




```
if RA = 0 then b ← 0
else          b ← (RA)
EA  ← b + (RB)
MEM(EA & 0xFFFF_FFFF_FFFF_FFF0, 16) ← (VRS)
mark_as_not_likely_to_be_needed_again_anytime_soon(EA)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The contents of VRS are stored into the quadword in storage addressed by the result of EA ANDed with 0xFFFF_FFFF_FFFF_FFF0.

stvxl provides a hint that the quadword in storage addressed by EA will probably not be needed again by the program in the near future.

Special Registers Altered:

None

Programming Note

See the Programming Note for the ***lvxl*** instruction on page 243.

6.7.4 Vector Alignment Support Instructions

Programming Note

The ***lvsl*** and ***lvsr*** instructions can be used to create the permute control vector to be used by a subsequent ***vperm*** instruction (see page 260). Let X and Y be the contents of register VRA and VRB specified by the ***vperm***. The control vector created by ***lvsl*** causes the ***vperm*** to select the high-order 16 bytes of the result of shifting the 32-byte value X || Y left by sh bytes. The control vector created by ***lvsr*** causes the ***vperm*** to select the low-order 16 bytes of the result of shifting X || Y right by sh bytes.

Programming Note

Examples of uses of ***lvsl***, ***lvsr***, and ***vperm*** to load and store unaligned data are given in Section 6.4.1.

These instructions can also be used to rotate or shift the contents of a Vector Register left (***lvsl***) or right (***lvsr***) by sh bytes. For rotating, the Vector Register to be rotated should be specified as both register VRA and VRB for ***vperm***. For shifting left, VRB for ***vperm*** should be a register containing all zeros and VRA should contain the value to be shifted, and vice versa for shifting right.

Load Vector for Shift Left Indexed X-form

lvsl VRT,RA,RB

31	VRT	RA	RB	6	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
sh  ← (b + (RB))60:63
switch(sh)
   case(0x0): VRT←0x000102030405060708090A0B0C0D0E0F
   case(0x1): VRT←0x0102030405060708090A0B0C0D0E0F10
   case(0x2): VRT←0x020304050607080 90A0B0C0D0E0F1011
   case(0x3): VRT←0x030405060708090A0B0C0D0E0F101112
   case(0x4): VRT←0x0405060708090A0B0C0D0E0F10111213
   case(0x5): VRT←0x05060708090A0B0C0D0E0F1011121314
   case(0x6): VRT←0x060708090A0B0C0D0E0F101112131415
   case(0x7): VRT←0x0708090A0B0C0D0E0F10111213141516
   case(0x8): VRT←0x08090A0B0C0D0E0F1011121314151617
   case(0x9): VRT←0x090A0B0C0D0E0F101112131415161718
   case(0xA): VRT←0x0A0B0C0D0E0F10111213141516171819
   case(0xB): VRT←0x0B0C0D0E0F101112131415161718191A
   case(0xC): VRT←0x0C0D0E0F10111213141516171819 1A1B
   case(0xD): VRT←0x0D0E0F101112131415161718191A1B1C
   case(0xE): VRT←0x0E0F10111213141516171819 1A1B1C1D
   case(0xF): VRT←0x0F101112131415161718191A1B1C1D1E
```

Let sh be bits 60:63 of the sum (RA|0)+(RB). Let X be the 32 byte value `0x00` || `0x01` || `0x02` || ... || `0x1E` || `0x1F`.

Bytes sh to sh+15 of X are placed into VRT.

Special Registers Altered:

None

Load Vector for Shift Right Indexed X-form

lvsr VRT,RA,RB

31	VRT	RA	RB	38	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
sh  ← (b + (RB))60:63
switch(sh)
   case(0x0): VRT←0x101112131415161718191A1B1C1D1E1F
   case(0x1): VRT←0x0F101112131415161718191A1B1C1D1E
   case(0x2): VRT←0x0E0F101112131415161718191A1B1C1D
   case(0x3): VRT←0x0D0E0F101112131415161718191A1B1C
   case(0x4): VRT←0x0C0D0E0F101112131415161718191A1B
   case(0x5): VRT←0x0B0C0D0E0F1011121314151617 18191A
   case(0x6): VRT←0x0A0B0C0D0E0F10111213141516171819
   case(0x7): VRT←0x090A0B0C0D0E0F101112131415161718
   case(0x8): VRT←0x08090A0B0C0D0E0F1011121314151617
   case(0x9): VRT←0x0708090A0B0C0D0E0F10111213141516
   case(0xA): VRT←0x060708090A0B0C0D0E0F101112131415
   case(0xB): VRT←0x05060708090A0B0C0D0E0F1011121314
   case(0xC): VRT←0x0405060708090A0B0C0D0E0F10111213
   case(0xD): VRT←0x030405060708090A0B0C0D0E0F101112
   case(0xE): VRT←0x020304050607080 90A0B0C0D0E0F1011
   case(0xF): VRT←0x0102030405060708090A0B0C0D0E0F10
```

Let sh be bits 60:63 of the sum (RA|0)+(RB). Let X be the 32-byte value `0x00` || `0x01` || `0x02` || ... || `0x1E` || `0x1F`.

Bytes 16-sh to 31-sh of X are placed into VRT.

Special Registers Altered:

None

6.8 Vector Permute and Formatting Instructions

6.8.1 Vector Pack and Unpack Instructions

Vector Pack Pixel VX-form

vpkpx VRT,VRA,VRB

4	VRT	VRA	VRB	782
0	6	11	16	21 31

```
do i = 0 to 63 by 16
   VR[VRT]i          ← VR[VRA]i×2+7
   VR[VRT]i+1:i+5    ← VR[VRA]i×2+8:i×2+12
   VR[VRT]i+6:i+10   ← VR[VRA]i×2+16:i×2+20
   VR[VRT]i+11:i+15  ← VR[VRA]i×2+24:i×2+28
   VR[VRT]i+64       ← VR[VRB]i×2+7
   VR[VRT]i+65:i+69  ← VR[VRB]i×2+8:i×2+12
   VR[VRT]i+70:i+74  ← VR[VRB]i×2+16:i×2+20
   VR[VRT]i+75:i+79  ← VR[VRB]i×2+24:i×2+28
end
```

Let the source vector be the concatenation of the contents of VR[VRA] followed by the contents of VR[VRB].

For each integer value i from 0 to 7, do the following.

Word element i in the source vector is packed to produce a 16-bit value as described below.

- bit 7 of the first byte (bit 7 of the word)
- bits 0:4 of the second byte (bits 8:12 of the word)
- bits 0:4 of the third byte (bits 16:20 of the word)
- bits 0:4 of the fourth byte (bits 24:28 of the word)

The result is placed into halfword element i of VR[VRT].

Special Registers Altered:

None

Programming Note

Each source word can be considered to be a 32-bit "pixel", consisting of four 8-bit "channels". Each target halfword can be considered to be a 16-bit pixel, consisting of one 1-bit channel and three 5-bit channels. A channel can be used to specify the intensity of a particular color, such as red, green, or blue, or to provide other information needed by the application.

Vector Pack Signed Doubleword Signed Saturate VX-form

vpksdss VRT,VRA,VRB

4	VRT	VRA	VRB	1486
0	6	11	16	21 31

```
src.qword[0] ← VR[VRA]
src.qword[1] ← VR[VRB]
do i = 0 to 3
   VR[VRT].word[i] ← Chop( Clamp( EXTS( src.dword[i]), -2^31,
2^31-1 ), 32 )
end
```

Let doubleword elements 0 and 1 of src be the contents of VR[VRA].

Let doubleword elements 2 and 3 of src be the contents of VR[VRB].

For each integer value i from 0 to 3, do the following.

The signed integer value in doubleword element i of src is placed into word element i of VR[VRT] in signed integer format.

- If the value is greater than $2^{31}-1$ the result saturates to $2^{31}-1$.
- If the value is less than -2^{31} the result saturates to -2^{31}.

Special Registers Altered:

SAT

Vector Pack Signed Doubleword Unsigned Saturate VX-form

vpksdus VRT,VRA,VRB

4	VRT	VRA	VRB	1358
0	6	11	16	21 31

```
src.qword[0] ← VR[VRA]
src.qword[1] ← VR[VRB]
do i = 0 to 3
   VR[VRT].word[i] ← Chop( Clamp( EXTS(src.dword[i]), 0, 2^32-1
), 32 )
end
```

Let doubleword elements 0 and 1 of src be the contents of VR[VRA].

Let doubleword elements 2 and 3 of src be the contents of VR[VRB].

For each integer value i from 0 to 3, do the following.

The signed integer value in doubleword element i of src is placed into word element i of VR[VRT] in unsigned integer format.

- If the value is greater than 2^{32}-1 the result saturates to 2^{32}-1.
- If the value is less than 0 the result saturates to 0.

Special Registers Altered:

SAT

Vector Pack Signed Halfword Signed Saturate VX-form

vpkshss VRT,VRA,VRB

4	VRT	VRA	VRB	398
0	6	11	16	21 31

```
do i=0 to 63 by 8
   src1 ← EXTS((VRA)i×2:i×2+15)
   src2 ← EXTS((VRB)i×2:i×2+15)
   VRTi:i+7     ← Clamp(src1, -128, 127)24:31
   VRTi+64:i+71 ← Clamp(src2, -128, 127)24:31
end
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 15, do the following.

Signed-integer halfword element i in the source vector is converted to an signed-integer byte.

- If the value of the element is greater than 127 the result saturates to 127
- If the value of the element is less than -128 the result saturates to -128.

The low-order 8 bits of the result is placed into byte element *i* of VRT.

Special Registers Altered:

SAT

Vector Pack Signed Halfword Unsigned Saturate VX-form

vpkshus VRT,VRA,VRB

4	VRT	VRA	VRB	270
0	6	11	16	21 31

```
do i=0 to 63 by 8
   src1 ← EXTS((VRA)i×2:i×2+15)
   src2 ← EXTS((VRB)i×2:i×2+15)
   VRTi:i+7     ← Clamp(src1, 0, 255)24:31
   VRTi+64:i+71 ← Clamp(src2, 0, 255)24:31
end
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 15, do the following.
Signed-integer halfword element i in the source vector is converted to an unsigned-integer byte.

– If the value of the element is greater than 255 the result saturates to 255

– If the value of the element is less than 0 the result saturates to 0.

The low-order 8 bits of the result is placed into byte element *i* of VRT.

Special Registers Altered:
SAT

Vector Pack Signed Word Signed Saturate VX-form

vpkswss VRT,VRA,VRB

4	VRT	VRA	VRB	462
0	6	11	16	21 31

```
do i=0 to 63 by 16
   src1 ← EXTS((VRA)i×2:i×2+31)
   src2 ← EXTS((VRB)i×2:i×2+31)
   VRTi:i+15    ← Clamp(src1, -2^15, 2^15-1)16:31
   VRTi+64:i+79 ← Clamp(src2, -2^15, 2^15-1)16:31
end
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 7, do the following.
Signed-integer word element i in the source vector is converted to an signed-integer halfword.

– If the value of the element is greater than 2^{15}-1 the result saturates to 2^{15}-1

– If the value of the element is less than -2^{15} the result saturates to -2^{15}.

The low-order 16 bits of the result is placed into halfword element i of VRT.

Special Registers Altered:
SAT

Vector Pack Signed Word Unsigned Saturate VX-form

vpkswus VRT,VRA,VRB

4	VRT	VRA	VRB	334
0	6	11	16	21 31

```
do i=0 to 63 by 16
   src1 ← EXTS((VRA)ix2:ix2+31)
   src2 ← EXTS((VRB)ix2:ix2+31)
   VRTi:i+15    ← Clamp(src1, 0, 2^16-1)16:31
   VRTi+64:i+79 ← Clamp(src2, 0, 2^16-1)16:31
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 7, do the following.

Signed-integer word element i in the source vector is converted to an unsigned-integer halfword.

– If the value of the element is greater than 2^{16}-1 the result saturates to 2^{16}-1

– If the value of the element is less than 0 the result saturates to 0.

The low-order 16 bits of the result is placed into halfword element i of VRT.

Special Registers Altered:

SAT

Vector Pack Unsigned Doubleword Unsigned Modulo VX-form

vpkudum VRT,VRA,VRB

4	VRT	VRA	VRB	1102
0	6	11	16	21 31

```
if MSR.VEC then Vector_Unavailable()

src.qword[0] ← VR[VRA]
src.qword[1] ← VR[VRB]
do i = 0 to 3
   VR[VRT].word[i] ← Chop( EXTZ(src.dword[i]), 32 )
end
```

Let doubleword elements 0 and 1 of src be the contents of VR[VRA].

Let doubleword elements 2 and 3 of src be the contents of VR[VRB].

For each integer value i from 0 to 3, do the following.

The contents of bits 32:63 of doubleword element i of src is placed into word element i of VR[VRT].

Special Registers Altered:

None

Vector Pack Unsigned Doubleword Unsigned Saturate VX-form

vpkudus VRT,VRA,VRB

4	VRT	VRA	VRB	1230
0	6	11	16	21 31

```
if MSR.VEC then Vector_Unavailable()

src.qword[0] ← VR[VRA]
src.qword[1] ← VR[VRB]
do i = 0 to 3
   VR[VRT].word[i] ← Chop( Clamp( EXTZ(src.dword[i]), 0, 2^32-1
), 32 )
end
```

Let doubleword elements 0 and 1 of src be the contents of VR[VRA].

Let doubleword elements 2 and 3 of src be the contents of VR[VRB].

For each integer value i from 0 to 3, do the following.

The unsigned integer value in doubleword element i of src is placed into word element i of VR[VRT] in unsigned integer format.

– If the value of the element is greater than 2^{32}-1 the result saturates to 2^{32}-1

Special Registers Altered:

SAT

Vector Pack Unsigned Halfword Unsigned Modulo VX-form

vpkuhum VRT,VRA,VRB

4	VRT	VRA	VRB	14
0	6	11	16	21 31

```
do i=0 to 63 by 8
   VRTi:i+7    ← (VRA)ix2+8:ix2+15
   VRTi+64:i+71 ← (VRB)ix2+8:ix2+15
end
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 15, do the following.

The contents of bits 8:15 of halfword element i in the source vector is placed into byte element i of VRT.

Special Registers Altered:

None

Vector Pack Unsigned Halfword Unsigned Saturate VX-form

vpkuhus VRT,VRA,VRB

4	VRT	VRA	VRB	142
0	6	11	16	21 31

```
do i=0 to 63 by 8
   src1 ← EXTZ((VRA)ix2:ix2+15)
   src2 ← EXTZ((VRB)ix2:ix2+15)
   VRTi:i+7     ← Clamp( src1, 0, 255 )24:31
   VRTi+64:i+71 ← Clamp( src2, 0, 255 )24:31
end
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 15, do the following.
Unsigned-integer halfword element i in the source vector is converted to an unsigned-integer byte.

– If the value of the element is greater than 255 the result saturates to 255.

The low-order 8 bits of the result is placed into byte element i of VRT.

Special Registers Altered:
SAT

Vector Pack Unsigned Word Unsigned Modulo VX-form

vpkuwum VRT,VRA,VRB

4	VRT	VRA	VRB	78
0	6	11	16	21 31

```
do i=0 to 63 by 16
   VRTi:i+15    ← (VRA)ix2+16:ix2+31
   VRTi+64:i+79 ← (VRB)ix2+16:ix2+31
end
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 7, do the following.
The contents of bits 16:31 of word element i in the source vector is placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Pack Unsigned Word Unsigned Saturate VX-form

vpkuwus VRT,VRA,VRB

4	VRT	VRA	VRB	206
0	6	11	16	21 31

```
do i=0 to 63 by 16
   src1 ← EXTZ((VRA)ix2:ix2+31 )
   src2 ← EXTZ((VRB)ix2:ix2+31 )
   VRTi:i+15 ← Clamp( src1, 0, 2^16-1 )16:31
   VRTi+64:i+79 ← Clamp( src2, 0, 2^16-1 )16:31
end
```

Let the source vector be the concatenation of the contents of VRA followed by the contents of VRB.

For each integer value i from 0 to 7, do the following.
Unsigned-integer word element i in the source vector is converted to an unsigned-integer halfword.

– If the value of the element is greater than 2^{16}-1 the result saturates to 2^{16}-1.

The low-order 16 bits of the result is placed into halfword element i of VRT.

Special Registers Altered:
SAT

Vector Unpack High Pixel VX-form

vupkhpx VRT,VRB

4	VRT	///	VRB	846
0	6	11	16	21 31

```
do i=0 to 63 by 16
   VRT[ix2:ix2+7]     ← EXTS((VB)[i       ])
   VRT[ix2+8:ix2+15]  ← EXTZ((VB)[i+1:i+5 ])
   VRT[ix2+16:ix2+23] ← EXTZ((VB)[i+6:i+10])
   VRT[ix2+24:ix2+31] ← EXTZ((VB)[i+11:i+15])
end
```

For each vector element *i* from 0 to 3, do the following.
Halfword element *i* in VRB is unpacked as follows.

- sign-extend bit 0 of the halfword to 8 bits
- zero-extend bits 1:5 of the halfword to 8 bits
- zero-extend bits 6:10 of the halfword to 8 bits
- zero-extend bits 11:15 of the halfword to 8 bits

The result is placed in word element *i* of VRT.

Special Registers Altered:
None

Programming Note

The source and target elements can be considered to be 16-bit and 32-bit “pixels” respectively, having the formats described in the Programming Note for the *Vector Pack Pixel* instruction on page 248.

Programming Note

Notice that the unpacking done by the *Vector Unpack Pixel* instructions does not reverse the packing done by the *Vector Pack Pixel* instruction. Specifically, if a 16-bit pixel is unpacked to a 32-bit pixel which is then packed to a 16-bit pixel, the resulting 16-bit pixel will not, in general, be equal to the original 16-bit pixel (because, for each channel except the first, *Vector Unpack Pixel* inserts high-order bits while *Vector Pack Pixel* discards low-order bits).

Vector Unpack Low Pixel VX-form

vupklpx VRT,VRB

4	VRT	///	VRB	974
0	6	11	16	21 31

```
do i=0 to 63 by 16
   VRT[ix2:ix2+7]     ← EXTS((VB)[i+64    ])
   VRT[ix2+8:ix2+15]  ← EXTZ((VB)[i+65:i+69])
   VRT[ix2+16:ix2+23] ← EXTZ((VB)[i+70:i+74])
   VRT[ix2+24:ix2+31] ← EXTZ((VB)[i+75:i+79])
end
```

For each vector element *i* from 0 to 3, do the following.
Halfword element *i*+4 in VRB is unpacked as follows.

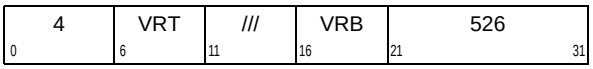
- sign-extend bit 0 of the halfword to 8 bits
- zero-extend bits 1:5 of the halfword to 8 bits
- zero-extend bits 6:10 of the halfword to 8 bits
- zero-extend bits 11:15 of the halfword to 8 bits

The result is placed in word element *i* of VRT.

Special Registers Altered:
None

Vector Unpack High Signed Byte VX-form

vupkhsb VRT,VRB




```
do i=0 to 63 by 8
   VRT[ix2:ix2+15] ← EXTS((VRB)i:i+7)
end
```

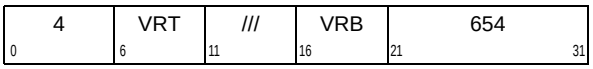
For each vector element i from 0 to 7, do the following.
Signed-integer byte element i in VRB is sign-extended to produce a signed-integer halfword and placed into halfword element *i* in VRT.

Special Registers Altered:

None

Vector Unpack Low Signed Byte VX-form

vupklsb VRT,VRB




```
do i=0 to 63 by 8
   VRT[ix2:ix2+15] ← EXTS((VRB)i+64:i+71)
end
```

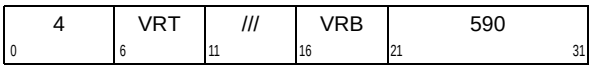
For each vector element *i* from 0 to 7, do the following.
Signed-integer byte element i+8 in VRB is sign-extended to produce a signed-integer halfword and placed into halfword element *i* in VRT.

Special Registers Altered:

None

Vector Unpack High Signed Halfword VX-form

vupkhsh VRT,VRB




```
do i=0 to 63 by 16
   VRT[ix2:ix2+31] ← EXTS((VRB)i:i+15)
end
```

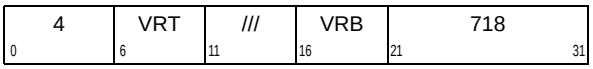
For each vector element *i* from 0 to 3, do the following.
Signed-integer halfword element *i* in VRB is sign-extended to produce a signed-integer word and placed into word element *i* in VRT.

Special Registers Altered:

None

Vector Unpack Low Signed Halfword VX-form

vupklsh VRT,VRB




```
do i=0 to 63 by 16
   VRT[ix2:ix2+31] ← EXTS((VRB)i+64:i+79)
end
```

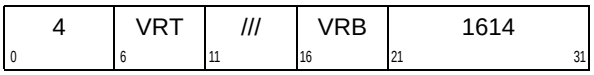
For each vector element *i* from 0 to 3, do the following.
Signed-integer halfword element *i*+4 in VRB is sign-extended to produce a signed-integer word and placed into word element *i* in VRT.

Special Registers Altered:

None

Vector Unpack High Signed Word VX-form

vupkhsw VRT,VRB




```
VR[VRT].dword[0] ← Chop( EXTS(VR[VRB].word[0]), 64 )
VR[VRT].dword[1] ← Chop( EXTS(VR[VRB].word[1]), 64 )
```

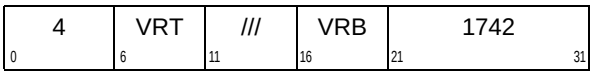
For each integer value i from 0 to 1, do the following.
The signed integer value in word element i of VR[VRB] is sign-extended and placed into doubleword element i of VR[VRT].

Special Registers Altered:

None

Vector Unpack Low Signed Word VX-form

vupklsw VRT,VRB




```
VR[VRT].dword[0] ← Chop( EXTS(VR[VRB].word[2]), 64 )
VR[VRT].dword[1] ← Chop( EXTS(VR[VRB].word[3]), 64 )
```

For each integer value i from 0 to 1, do the following.
The signed integer value in word element i+2 of VR[VRB] is sign-extended and placed into doubleword element i of VR[VRT].

Special Registers Altered:

None

6.8.2 Vector Merge Instructions

Vector Merge High Byte VX-form

vmrghb VRT,VRA,VRB

4	VRT	VRA	VRB	12
0	6	11	16	21 31

```
do i=0 to 63 by 8
   VRT[ix2:ix2+7]    ← (VRA)[i:i+7]
   VRT[ix2+8:ix2+15] ← (VRB)[i:i+7]
end
```

For each vector element *i* from 0 to 7, do the following.

Byte element *i* in VRA is placed into byte element 2×i in VRT.

Byte element *i* in VRB is placed into byte element 2×i+1 in VRT.

Special Registers Altered:

None

Vector Merge Low Byte VX-form

vmrglb VRT,VRA,VRB

4	VRT	VRA	VRB	268
0	6	11	16	21 31

```
do i=0 to 63 by 8
   VRT[ix2:ix2+7]    ← (VRA)[i+64:i+71]
   VRT[ix2+8:ix2+15] ← (VRB)[i+64:i+71]
end
```

For each vector element *i* from 0 to 7, do the following.

Byte element i+8 in VRA is placed into byte element 2×i in VRT.

Byte element i+8 in VRB is placed into byte element 2×i+1 in VRT.

Special Registers Altered:

None

Vector Merge High Halfword VX-form

vmrghh VRT,VRA,VRB

4	VRT	VRA	VRB	76
0	6	11	16	21 31

```
do i=0 to 63 by 16
   VRT[ix2:ix2+15]    ← (VRA)[i:i+15]
   VRT[ix2+16:ix2+31] ← (VRB)[i:i+15]
end
```

For each vector element *i* from 0 to 3, do the following.

Halfword element *i* in VRA is placed into halfword element 2×i in VRT.

Halfword element *i* in VRB is placed into halfword element 2×i+1 in VRT.

Special Registers Altered:

None

Vector Merge Low Halfword VX-form

vmrglh VRT,VRA,VRB

4	VRT	VRA	VRB	332
0	6	11	16	21 31

```
do i=0 to 63 by 16
   VRT[ix2:ix2+15]    ← (VRA)[i+64:i+79]
   VRT[ix2+16:ix2+31] ← (VRB)[i+64:i+79]
end
```

For each vector element *i* from 0 to 3, do the following.

Halfword element i+4 in VRA is placed into halfword element 2×i in VRT.

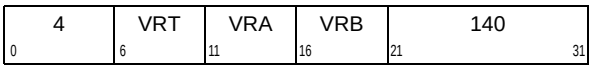
Halfword element i+4 in VRB is placed into halfword element 2×i+1 in VRT.

Special Registers Altered:

None

Vector Merge High Word VX-form

vmrghw VRT,VRA,VRB




```
do i=0 to 63 by 32
   VRT[ix2:ix2+31]    ← (VRA)[i:i+31]
   VRT[ix2+32:ix2+63] ← (VRB)[i:i+31]
end
```

For each vector element *i* from 0 to 1, do the following.

Word element *i* in VRA is placed into word element 2×i in VRT.

Word element *i* in VRB is placed into word element 2×i+1 in VRT.

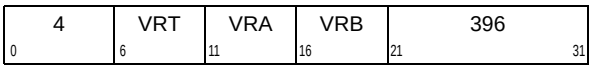
The word elements in the high-order half of VRA are placed, in the same order, into the even-numbered word elements of VRT. The word elements in the high-order half of VRB are placed, in the same order, into the odd-numbered word elements of VRT.

Special Registers Altered:

None

Vector Merge Low Word VX-form

vmrglw VRT,VRA,VRB




```
do i=0 to 63 by 32
   VRT[ix2:ix2+31]    ← (VRA)[i+64:i+95]
   VRT[ix2+32:ix2+63] ← (VRB)[i+64:i+95]
end
```

For each vector element *i* from 0 to 1, do the following.

Word element i+2 in VRA is placed into word element 2×i in VRT.

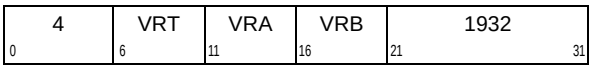
Word element i+2 in VRB is placed into word element 2×i+1 in VRT.

Special Registers Altered:

None

Vector Merge Even Word VX-form

vmrgew VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()
VR[VRT].word[0] ← VR[VRA].word[0]
VR[VRT].word[1] ← VR[VRB].word[0]
VR[VRT].word[2] ← VR[VRA].word[2]
VR[VRT].word[3] ← VR[VRB].word[2]
```

The contents of word element 0 of VR[VRA] are placed into word element 0 of VR[VRT].

The contents of word element 0 of VR[VRB] are placed into word element 1 of VR[VRT].

The contents of word element 2 of VR[VRA] are placed into word element 2 of VR[VRT].

The contents of word element 2 of VR[VRB] are placed into word element 3 of VR[VRT].

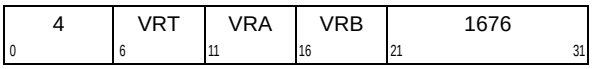
vmrgew is treated as a *Vector* instruction in terms of resource availability.

Special Registers Altered

None

Vector Merge Odd Word VX-form

vmrgow VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()
VR[VRT].word[0] ← VR[VRA].word[1]
VR[VRT].word[1] ← VR[VRB].word[1]
VR[VRT].word[2] ← VR[VRA].word[3]
VR[VRT].word[3] ← VR[VRB].word[3]
```

The contents of word element 1 of VR[VRA] are placed into word element 0 of VR[VRT].

The contents of word element 1 of VR[VRB] are placed into word element 1 of VR[VRT].

The contents of word element 3 of VR[VRA] are placed into word element 2 of VR[VRT].

The contents of word element 3 of VR[VRB] are placed into word element 3 of VR[VRT].

vmrgow is treated as a *Vector* instruction in terms of resource availability.

Special Registers Altered

None

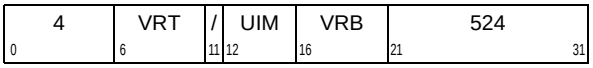
6.8.3 Vector Splat Instructions

Programming Note

The *Vector Splat* instructions can be used in preparation for performing arithmetic for which one source vector is to consist of elements that all have the same value (e.g., multiplying all elements of a Vector Register by a constant).

Vector Splat Byte VX-form

vspltb VRT,VRB,UIM



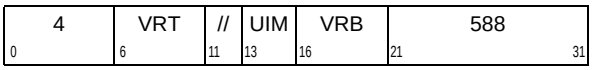

```
b ← UIM || 0b000
do i=0 to 127 by 8
   VRT_{i:i+7} ← (VRB)_{b:b+7}
end
```

For each integer value i from 0 to 15, do the following.
The contents of byte element UIM in VRB are placed into byte element i of VRT.

Special Registers Altered:
None

Vector Splat Halfword VX-form

vsplth VRT,VRB,UIM



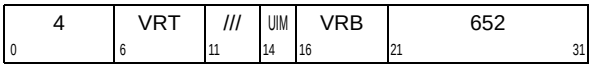

```
b ← UIM || 0b0000
do i=0 to 127 by 16
   VRT_{i:i+15} ← (VRB)_{b:b+15}
end
```

For each integer value i from 0 to 7, do the following.
The contents of halfword element UIM in VRB are placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Splat Word VX-form

vspltw VRT,VRB,UIM



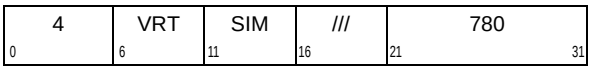

```
b ← UIM || 0b00000
do i=0 to 127 by 32
   VRT_{i:i+31} ← (VRB)_{b:b+31}
end
```

For each integer value i from 0 to 3, do the following.
The contents of word element UIM in VRB are placed into word element i of VRT.

Special Registers Altered:
None

Vector Splat Immediate Signed Byte VX-form

vspltisb VRT,SIM



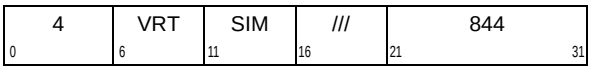

```
do i=0 to 127 by 8
   VRT[i:i+7] ← EXTS(SIM, 8)
end
```

For each integer value i from 0 to 15, do the following.
The value of the SIM field, sign-extended to 8 bits, is placed into byte element i of VRT.

Special Registers Altered:
None

Vector Splat Immediate Signed Halfword VX-form

vspltish VRT,SIM



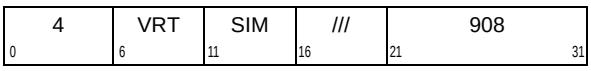

```
do i=0 to 127 by 16
   VRT[i:i+15] ← EXTS(SIM, 16)
end
```

For each integer value i from 0 to 7, do the following.
The value of the SIM field, sign-extended to 16 bits, is placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Splat Immediate Signed Word VX-form

vspltisw VRT,SIM




```
do i=0 to 127 by 32
   VRT[i:i+31] ← EXTS(SIM, 32)
end
```

For each vector element *i* from 0 to 3, do the following.
The value of the SIM field, sign-extended to 32 bits, is placed into word element *i* of VRT.

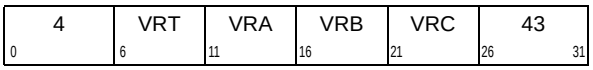
Special Registers Altered:
None

6.8.4 Vector Permute Instruction

The *Vector Permute* instruction allows any byte in two source Vector Registers to be copied to any byte in the target Vector Register. The bytes in a third source Vector Register specify from which byte in the first two source Vector Registers the corresponding target byte is to be copied. The contents of the third source Vector Register are sometimes referred to as the “permute control vector”.

Vector Permute VA-form

vperm VRT,VRA,VRB,VRC




```
if MSR.VEC=0 then Vector_Unavailable()

src.qword[0] ← VR[VRA]
src.qword[1] ← VR[VRB]

do i = 0 to 15
   index ← VR[VRC].byte[i].bit[3:7]
   VR[VRT].byte[i] ← src.byte[index]
end
```

Let the source vector be the concatenation of the contents of VR[VRA] followed by the contents of VR[VRB].

For each integer value i from 0 to 15, do the following.

Let index be the value specified by bits 3:7 of byte element i of VR[VRC].

The contents of byte element index of src are placed into byte element i of VR[VRT].

Special Registers Altered:

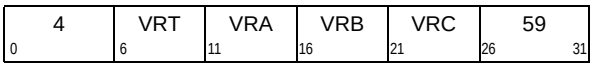
None

Programming Note

See the Programming Notes with the *Load Vector for Shift Left* and *Load Vector for Shift Right* instructions on page 247 for examples of uses of ***vperm***.

Vector Permute Right-indexed VA-form

vpermr VRT,VRA,VRB,VRC




```
if MSR.VEC=0 then Vector_Unavailable()

src.qword[0] ← VR[VRA]
src.qword[1] ← VR[VRB]

do i = 0 to 15
   index ← VR[VRC].byte[i].bit[3:7]
   VR[VRT].byte[i] ← src.byte[31-index]
end
```

Let the source vector be the concatenation of the contents of VR[VRA] followed by the contents of VR[VRB].

For each integer value i from 0 to 15, do the following.

Let index be the value specified by bits 3:7 of byte element i of VR[VRC].

The contents of byte element 31-index of src are placed into byte element i of VR[VRT].

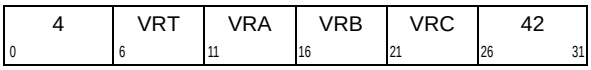
Special Registers Altered:

None

6.8.5 Vector Select Instruction

Vector Select VA-form

vsel VRT,VRA,VRB,VRC




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 127
   mask ← VR[VRC].bit[i]
   VR[VRT].bit[i] ← (mask=0) ? VR[VRA].bit[i] : VR[VRB].bit[i]
end
```

For each bit in VR[VRC] that contains the value 0, the corresponding bit in VR[VRA] is placed into the corresponding bit of VR[VRT]. Otherwise, the corresponding bit in VR[VRB] is placed into the corresponding bit of VR[VRT].

Special Registers Altered:

None

6.8.6 Vector Shift Instructions

The *Vector Shift* instructions rotate or shift the contents of a Vector Register or a pair of Vector Registers left or right by a specified number of bytes (***vslo***, ***vsro***, ***vsldoi***) or bits (***vsl***, ***vsr***). Depending on the instruction, this "shift count" is specified either by the contents of a Vector Register or by an immediate field in the instruction. In the former case, 7 bits of the shift count register give the shift count in bits ($0 \le$ count ≤ 127). Of these 7 bits, the high-order 4 bits give the number of complete bytes by which to shift and are used by ***vslo*** and ***vsro***; the low-order 3 bits give the number of remaining bits by which to shift and are used by ***vsl*** and ***vsr***.

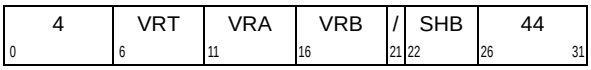
Programming Note

A pair of these instructions, specifying the same shift count register, can be used to shift the contents of a Vector Register left or right by the number of bits (0-127) specified in the shift count register. The following example shifts the contents of register Vx left by the number of bits specified in register Vy and places the result into register Vz.

```
vslo      Vz,Vx,Vy
vspltb    Vy,Vy,15
vsl       Vz,Vz,Vy
```

Vector Shift Left Double by Octet Immediate VA-form

vsldoi VRT,VRA,VRB,SHB




```
if MSR.VEC=0 then Vector_Unavailable()

src.qword[0] ← VR[VRA]
src.qword[1] ← VR[VRB]

VR[VRT] ← src.byte[SHB:SHB+15]
```

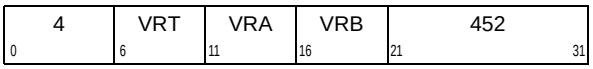
Let the source vector be the concatenation of the contents of VR[VRA] followed by the contents of VR[VRB]. Bytes SHB:SHB+15 of the source vector are placed into VR[VRT].

Special Registers Altered:

None

Vector Shift Left VX-form

vsl VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

shb ← VR[VRB].bit[125:127]

t ← 1
do i = 0 to 15
   t ← t & (VR[VRB].byte[i].bit[5:7] = sh)
end
if t=1 then
   VR[VRT] ← VR[VRA] << sh
else
   VR[VRT] ← undefined
```

The contents of VR[VRA] are shifted left by the number of bits specified in bits 125:127 of VR[VRB].

– Bits shifted out of bit 0 are lost.
– Zeros are supplied to the vacated bits on the right.

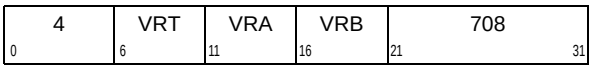
The result is place into VR[VRT], except if, for any byte element in register VR[VRB], the low-order 3 bits are not equal to the shift amount, then VR[VRT] is undefined.

Special Registers Altered:

None

Vector Shift Right VX-form

vsr VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

sh ← VR[VRB].bit[125:127]

t ← 1
do i = 0 to 15
   t ← t & (VR[VRB].byte[i].bit[5:7]=sh)
end
if t=1 then
   VR[VRT] ← VR[VRA] >> sh
else
   VR[VRT] ← undefined
```

The contents of VR[VRA] are shifted right by the number of bits specified in bits 125:127 of VR[VRB].

– Bits shifted out of bit 127 are lost.
– Zeros are supplied to the vacated bits on the left.

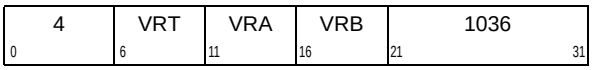
The result is place into VR[VRT], except if, for any byte element in register VR[VRB], the low-order 3 bits are not equal to the shift amount, then VR[VRT] is undefined.

Special Registers Altered:

None

Vector Shift Left by Octet VX-form

vslo VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

shb ← VR[VRB].bit[121:124] << 3

VR[VRT] ← VR[VRA] << shb
```

The contents of VR[VRA] are shifted left by the number of bytes specified in bits 121:124 of VR[VRB].

– Bytes shifted out of byte 0 are lost.
– Zeros are supplied to the vacated bytes on the right.

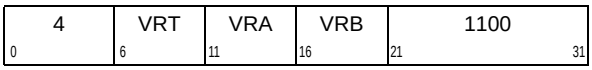
The result is placed into VR[VRT].

Special Registers Altered:

None

Vector Shift Right by Octet VX-form

vsro VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

shb ← VR[VRB].bit[121:124] << 3

VR[VRT] ← VR[VRA] >> shb
```

The contents of VR[VRA] are shifted right by the number of bytes specified in bits 121:124 of VR[VRB].

– Bytes shifted out of byte 15 are lost.
– Zeros are supplied to the vacated bytes on the left.

The result is placed into VR[VRT].

Special Registers Altered:

None

Programming Note

A double-register shift by a dynamically specified number of bits (0-127) can be performed in six instructions. The following example shifts Vw || Vx left by the number of bits specified in Vy and places the high-order 128 bits of the result into Vz.

```
vslo      Vt1,Vw,Vy     # shift high-order reg left
vspltb    Vy,Vy,15
vsl       Vt1,Vt1,Vy
vsububm   Vt3,V0,Vy     # adjust shift count ((V0)=0)
vsro      Vt2,Vx,Vt3    # shift low-order reg right
vspltb    Vt3,Vt3,15
vsr       Vt2,Vt2,Vt3
vor       Vz,Vt1,Vt2    # merge to get final result
```

Vector Shift Left Variable VX-form

vslv VRT,VRA,VRB

4	VRT	VRA	VRB	1860
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable_Interrupt()

src.byte[0:15] ← VR[VRA]
src.byte[16]   ← 0x00

do i = 0 to 15
   sh ← VR[VRB].byte[i].bit[5:7]
   VR[VRT].byte[i] ← src.byte[i:i+1].bit[sh:sh+7]
end
```

Let bytes 0:15 of src be the contents of VR[VRA].
Let byte 16 of src be the value 0x00.

For each integer value i from 0 to 15, do the following.

Let sh be the value in bits 5:7 of byte element i of VR[VRB].

The contents of bits sh:sh+7 of the halfword in byte elements i:i+1 of src are placed into byte element i of VR[VRT].

Special Registers Altered:

None

Vector Shift Right Variable VX-form

vsrv VRT,VRA,VRB

4	VRT	VRA	VRB	1796
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable_Interrupt()

src.byte[0]   ← 0x00
src.byte[1:16] ← VR[VRA]

do i = 0 to 15
   sh ← VR[VRB].byte[i].bit[5:7]
   VR[VRT].byte[i] ← src.byte[i:i+1].bit[8-sh:15-sh]
end
```

Let bytes 1:16 of src be the contents of VR[VRA].
Let byte 0 of src be the value 0x00.

For each integer value i from 0 to 15, do the following.

Let sh be the value in bits 5:7 of byte element i of VR[VRB].

The contents of bits 8-sh:15-sh of the halfword in byte elements i:i+1 of src are placed into byte element i of VR[VRT].

Special Registers Altered:

None

Programming Note

Assume vSRC contains a vector of packed 7-bit values, A located in bits 0:6, B located in bits 7:13, C located in bits 14:20, etc..

```
                                # vSRC    = { 0bAAAAAAAB, 0bBBBBBBCC, 0bCCCCCDDD, 0bDDDDEEEE,
                                #             0bEEEFFFFF, 0bFFGGGGGG, 0bGHHHHHHH, 0bIIIIIIIJ,
                                #             0bJJJJJJKK, 0bKKKKKLLL, 0bLLLLMMMM, 0bMMMNNNNN,
                                #             0bNNOOOOOO, 0bOPPPPPPP, 0bQQQQQQQR, 0bRRRRRRSS };
```

Assume the following registers are pre-loaded as follows,

```
                                # vSHCNT1 = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x07,
                                #             0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07 };
                                # vSHCNT2 = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
                                #             0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x07, 0x07 };
                                # vSHCNT3 = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                #             0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x02 };
                                # vMASK   = { 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F,
                                #             0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F };
```

The leftmost seven packed 7-bit values can be unpacked into byte elements 0 to 6 using ***vsrv*** with vSHCNT1.

```
vsrv     vTMP1, vSRC, vSHCT1    # vTMP1   = { 0b0AAAAAAA, 0bABBBBBBB, 0bBCCCCCCC, 0bCDDDDDDD,
                                #             0bDEEEEEEE, 0bEFFFFFFF, 0bFGGGGGGG, 0bHHHHHHHI,
                                #             0bIIIIIIIJ, 0bJJJJJKKK, 0bKKKKLLLL, 0bLLLMMMMM,
                                #             0bMMNNNNNN, 0bNOOOOOOO, 0bPPPPPPPQ, 0bQQQQQQRR };
```

The next seven packed 7-bit values can then be unpacked into byte elements 7 to 13 using ***vsrv*** with vSHCNT2.

```
vsrv     vTMP2, vTMP1, vSHCT2    # vTMP2   = { 0b0AAAAAAA, 0bABBBBBBB, 0bBCCCCCCC, 0bCDDDDDDD,
                                #             0bDEEEEEEE, 0bEFFFFFFF, 0bFGGGGGGG, 0bGHHHHHHH,
                                #             0bHIIIIIII, 0bIJJJJJJJ, 0bJKKKKKKK, 0bKLLLLLLL,
                                #             0bLMMMMMMM, 0bMNNNNNNN, 0bOOOOOOOP, 0bPPPPPPQQ };
```

The next two packed 7-bit values can then be unpacked into byte elements 14 to 15 using ***vsrv*** with vSHCNT3.

```
vsrv     vTMP3, vTMP2, vSHCT3    # vTMP3   = { 0b0AAAAAAA, 0bABBBBBBB, 0bBCCCCCCC, 0bCDDDDDDD,
                                #             0bDEEEEEEE, 0bEFFFFFFF, 0bFGGGGGGG, 0bGHHHHHHH,
                                #             0bHIIIIIII, 0bIJJJJJJJ, 0bJKKKKKKK, 0bKLLLLLLL,
                                #             0bLMMMMMMM, 0bMNNNNNNN, 0bNOOOOOOO, 0bOPPPPPPP };
```

The most-significant bit in each byte element is masked off to produce a vector of sixteen unsigned byte elements.

```
vand     vTMP4, vTMP3, vMASK    # vTMP4   = { 0b0AAAAAAA, 0b0BBBBBBB, 0b0CCCCCCC, 0b0DDDDDDD,
                                #             0b0EEEEEEE, 0b0FFFFFFF, 0b0GGGGGGG, 0b0HHHHHHH,
                                #             0b0IIIIIII, 0b0JJJJJJJ, 0b0KKKKKKK, 0b0LLLLLLL,
                                #             0b0MMMMMMM, 0b0NNNNNNN, 0b0OOOOOOO, 0b0PPPPPPP };
```

The vector of sixteen unsigned byte elements can be further unpacked to two vectors of eight unsigned halfword elements using a ***vupkhsb*** and a ***vupklsb***.

```
vupkhsb  vTMP5, vTMP4           # vTMP5 = { 0b00000000_0AAAAAAA, 0b00000000_0BBBBBBB, ... };
vupklsb  vTMP6, vTMP4           # vTMP6 = { 0b00000000_0IIIIIII, 0b00000000_0JJJJJJJ, ... };
```

The resultant two vectors of eight unsigned halfword elements can then be further unpacked to four vectors of four unsigned word elements using two ***vupkhsh*** and two ***vupklsh*** instructions.

```
vupkhsh  vRESULT0, vTMP5        # vRESULT0 = { 0b00000000_00000000_00000000_0AAAAAAA, ... };
vupklsh  vRESULT1, vTMP5        # vRESULT1 = { 0b00000000_00000000_00000000_0EEEEEEE, ... };
vupkhsh  vRESULT2, vTMP6        # vRESULT2 = { 0b00000000_00000000_00000000_0IIIIIII, ... };
vupklsh  vRESULT3, vTMP6        # vRESULT3 = { 0b00000000_00000000_00000000_0MMMMMMM, ... };
```

6.8.7 Vector Extract Element Instructions

Vector Extract Unsigned Byte VX-form

vextractub VRT,VRB,UIM

4	VRT	/	UIM	VRB	525
0	6	11	12	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src ← VR[VRB].byte[UIM]

VR[VRT].dword[0] ← EXTZ64(src)
VR[VRT].dword[1] ← 0x0000_0000_0000_0000
```

The contents of byte element UIM of VR[VRB] are placed into bits 56:63 of VR[VRT]. The contents of the remaining byte elements of VR[VRT] are set to 0.

Special Registers Altered:

None

Vector Extract Unsigned Halfword VX-form

vextractuh VRT,VRB,UIM

4	VRT	/	UIM	VRB	589
0	6	11	12	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src ← VR[VRB].byte[UIM:UIM+1]

VR[VRT].dword[0] ← EXTZ64(src)
VR[VRT].dword[1] ← 0x0000_0000_0000_0000
```

The contents of byte elements UIM:UIM+1 of VR[VRB] are placed into halfword element 3 of VR[VRT]. The contents of the remaining halfword elements of VR[VRT] are set to 0.

If the value of UIM is greater than 14, the results are undefined.

Special Registers Altered:

None

Vector Extract Unsigned Word VX-form

vextractuw VRT,VRB,UIM

4	VRT	/	UIM	VRB	653
0	6	11	12	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src ← VR[VRB].byte[UIM:UIM+3]

VR[VRT].dword[0] ← EXTZ64(src)
VR[VRT].dword[1] ← 0x0000_0000_0000_0000
```

The contents of byte elements UIM:UIM+3 of VR[VRB] are placed into word element 1 of VR[VRT]. The contents of the remaining word elements of VR[VRT] are set to 0.

If the value of UIM is greater than 12, the results are undefined.

Special Registers Altered:

None

Vector Extract Doubleword VX-form

vextractd VRT,VRB,UIM

4	VRT	/	UIM	VRB	717
0	6	11	12	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src ← VR[VRB].byte[UIM:UIM+7]

VR[VRT].dword[0] ← src
VR[VRT].dword[1] ← 0x0000_0000_0000_0000
```

The contents of byte elements UIM:UIM+7 of VR[VRB] are placed into VR[VRT]. The contents of doubleword element 1 of VR[VRT] are set to 0.

If the value of UIM is greater than 8, the results are undefined.

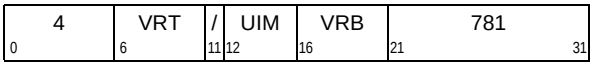
Special Registers Altered:

None

6.8.8 Vector Insert Element Instructions

Vector Insert Byte VX-form

vinsertb VRT,VRB,UIM




```
if MSR.VEC=0 then Vector_Unavailable()

VR[VRT].byte[UIM] ← VR[VRB].byte[7]
```

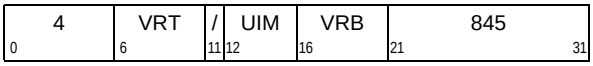
The contents of byte element 7 of VR[VRB] are placed into byte element UIM of VR[VRT]. The contents of the remaining byte elements of VR[VRT] are not modified.

Special Registers Altered:

None

Vector Insert Halfword VX-form

vinserth VRT,VRB,UIM




```
if MSR.VEC=0 then Vector_Unavailable()

VR[VRT].byte[UIM:UIM+1] ← VR[VRB].hword[3]
```

The contents of halfword element 3 of VR[VRB] are placed into byte elements UIM:UIM+1 of VR[VRT]. The contents of the remaining byte elements of VR[VRT] are not modified.

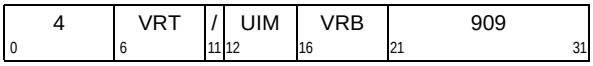
If the value of UIM is greater than 14, the results are undefined.

Special Registers Altered:

None

Vector Insert Word VX-form

vinsertw VRT,VRB,UIM




```
if MSR.VEC=0 then Vector_Unavailable()

VR[VRT].byte[UIM:UIM+3] ← VR[VRB].word[1]
```

The contents of word element 1 of VR[VRB] are placed into byte elements UIM:UIM+3 of VR[VRT]. The contents of the remaining byte elements of VR[VRT] are not modified.

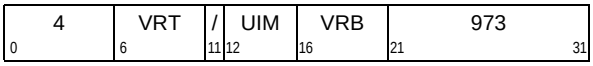
If the value of UIM is greater than 12, the results are undefined.

Special Registers Altered:

None

Vector Insert Doubleword VX-form

vinsertd VRT,VRB,UIM




```
if MSR.VEC=0 then Vector_Unavailable()

VR[VRT].byte[UIM:UIM+7] ← VR[VRB].dword[0]
```

The contents of doubleword element 0 of VR[VRB] are placed into byte elements UIM:UIM+7 of VR[VRT]. The contents of the remaining byte elements of VR[VRT] are not modified.

If the value of UIM is greater than 8, the results are undefined.

Special Registers Altered:

None

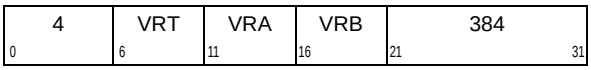
6.9 Vector Integer Instructions

6.9.1 Vector Integer Arithmetic Instructions

6.9.1.1 Vector Integer Add Instructions

Vector Add and Write Carry-Out Unsigned Word VX-form

vaddcuw VRT,VRA,VRB



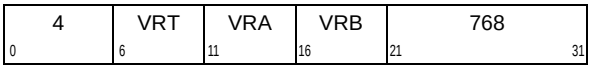

```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← Chop( ( aop +int bop ) >>ui 32,1)
end
```

For each integer value i from 0 to 3, do the following.
Unsigned-integer word element i in VRA is added to unsigned-integer word element i in VRB. The carry out of the 32-bit sum is zero-extended to 32 bits and placed into word element i of VRT.

Special Registers Altered:
None

Vector Add Signed Byte Saturate VX-form

vaddsbs VRT,VRA,VRB




```
do i=0 to 127 by 8
   aop ← EXTS(VRAi:i+7)
   bop ← EXTS(VRBi:i+7)
   VRTi:i+7 ← Clamp( aop +int bop, -128, 127 )24:31
end
```

For each integer value i from 0 to 15, do the following.
Signed-integer byte element i in VRA is added to signed-integer byte element i in VRB.

– If the sum is greater than 127 the result saturates to 127.

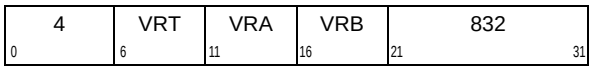
– If the sum is less than -128 the result saturates to -128.

The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:
SAT

Vector Add Signed Halfword Saturate VX-form

vaddshs VRT,VRA,VRB




```
do i=0 to 127 by 16
   aop ← EXTS((VRA)i:i+15)
   bop ← EXTS((VRB)i:i+15)
   VRTi:i+15 ← Clamp(aop +int bop, -2^15, 2^15-1)16:31
end
```

For each integer value i from 0 to 7, do the following.
Signed-integer halfword element i in VRA is added to signed-integer halfword element i in VRB.

– If the sum is greater than 2^{15}-1 the result saturates to 2^{15}-1

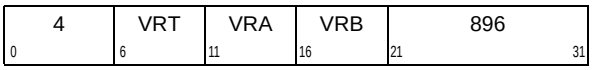
– If the sum is less than -2^{15} the result saturates to -2^{15}.

The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:
SAT

Vector Add Signed Word Saturate VX-form

vaddsws VRT,VRA,VRB




```
do i=0 to 127 by 32
   aop ← EXTS((VRA)i:i+31)
   bop ← EXTS((VRB)i:i+31)
   VRTi:i+31 ← Clamp(aop +int bop, -2^31, 2^31-1)
end
```

For each integer value i from 0 to 3, do the following.

Signed-integer word element i in VRA is added to signed-integer word element i in VRB.

– If the sum is greater than 2^{31}-1 the result saturates to 2^{31}-1.

– If the sum is less than -2^{31} the result saturates to -2^{31}.

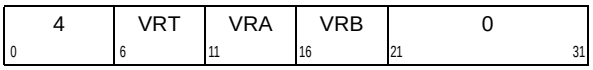
The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:

SAT

Vector Add Unsigned Byte Modulo VX-form

vaddubm VRT,VRA,VRB




```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← Chop( aop +int bop, 8 )
end
```

For each integer value i from 0 to 15, do the following.

Unsigned-integer byte element i in VRA is added to unsigned-integer byte element i in VRB.

The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:

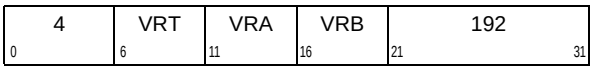
None

Programming Note

vaddubm can be used for unsigned or signed-integers.

Vector Add Unsigned Doubleword Modulo VX-form

vaddudm VRT,VRA,VRB




```
do i = 0 to 1
   aop ← VR[VRA].dword[i]
   bop ← VR[VRB].dword[i]
   VR[VRT].dword[i] ← Chop( aop +int bop, 64 )
end
```

For each integer value i from 0 to 1, do the following.

The integer value in doubleword element i of VR[VRB] is added to the integer value in doubleword element i of VR[VRA].

The low-order 64 bits of the result are placed into doubleword element i of VR[VRT].

Special Registers Altered:

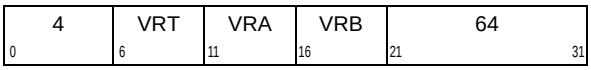
None

Programming Note

vaddudm can be used for signed or unsigned integers.

Vector Add Unsigned Halfword Modulo VX-form

vadduhm VRT,VRA,VRB




```
do i=0 to 127 by 16
   aop  ← EXTZ((VRA)i:i+15)
   bop  ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← Chop( aop +int bop, 16 )
end
```

For each integer value i from 0 to 7, do the following.

Unsigned-integer halfword element i in VRA is added to unsigned-integer halfword element i in VRB.

The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:

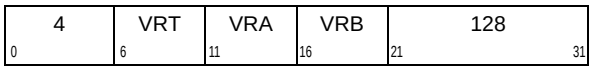
None

Programming Note

vadduhm can be used for unsigned or signed-integers.

Vector Add Unsigned Word Modulo VX-form

vadduwm VRT,VRA,VRB




```
do i=0 to 127 by 32
   aop  ← EXTZ((VRA)i:i+31)
   bop  ← EXTZ((VRB)i:i+31)
   temp ← aop +int bop
   VRTi:i+31 ← Chop( aop +int bop, 32 )
end
```

For each integer value i from 0 to 3, do the following.

Unsigned-integer word element i in VRA is added to unsigned-integer word element i in VRB.

The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:

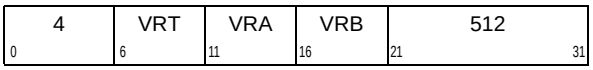
None

Programming Note

vadduwm can be used for unsigned or signed-integers.

Vector Add Unsigned Byte Saturate VX-form

vaddubs VRT,VRA,VRB




```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← Clamp( aop +int bop, 0, 255 )24:31
end
```

For each integer value i from 0 to 15, do the following.
Unsigned-integer byte element i in VRA is added to unsigned-integer byte element i in VRB.

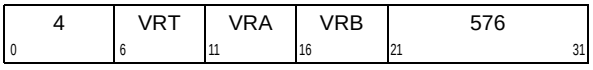
– If the sum is greater than 255 the result saturates to 255.

The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:
SAT

Vector Add Unsigned Halfword Saturate VX-form

vadduhs VRT,VRA,VRB




```
do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← Clamp(aop +int bop, 0, 2^16-1)16:31
end
```

For each integer value i from 0 to 7, do the following.
Unsigned-integer halfword element i in VRA is added to unsigned-integer halfword element i in VRB.

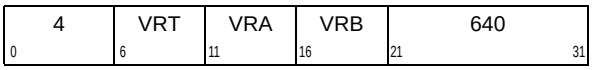
– If the sum is greater than 2^{16}-1 the result saturates to 2^{16}-1.

The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:
SAT

Vector Add Unsigned Word Saturate VX-form

vadduws VRT,VRA,VRB




```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← Clamp(aop +int bop, 0, 2^32-1)
end
```

For each integer value i from 0 to 3, do the following.
Unsigned-integer word element i in VRA is added to unsigned-integer word element i in VRB.

– If the sum is greater than 2^{32}-1 the result saturates to 2^{32}-1.

The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:
SAT

Vector Add Unsigned Quadword Modulo VX-form

vadduqm VRT,VRA,VRB

4	VRT	VRA	VRB	256
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← VR[VRA]
src2 ← VR[VRB]
sum  ← EXTZ(src1) + EXTZ(src2)

VR[VRT] ← Chop(sum, 128)
```

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].

src1 and src2 can be signed or unsigned integers.

The rightmost 128 bits of the sum of src1 and src2 are placed into VR[VRT].

Special Registers Altered:

None

Vector Add & write Carry Unsigned Quadword VX-form

vaddcuq VRT,VRA,VRB

4	VRT	VRA	VRB	320
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← VR[VRA]
src2 ← VR[VRB]
sum  ← EXTZ(src1) + EXTZ(src2)

VR[VRT] ← Chop( EXTZ( Chop(sum>>128, 1) ), 128 )
```

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].

src1 and src2 can be signed or unsigned integers.

The carry out of the sum of src1 and src2 is placed into VR[VRT].

Special Registers Altered:

None

Vector Add Extended Unsigned Quadword Modulo VA-form

vaddeuqm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	60
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← VR[VRA]
src2 ← VR[VRB]
cin  ← VR[VRC].bit[127]
sum  ← EXTZ(src1) + EXTZ(src2) + EXTZ(cin)

VR[VRT] ← Chop(sum, 128)
```

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].
Let cin be the integer value in bit 127 of VR[VRC].

src1 and src2 can be signed or unsigned integers.

The rightmost 128 bits of the sum of src1, src2, and cin are placed into VR[VRT].

Special Registers Altered:

None

Vector Add Extended & write Carry Unsigned Quadword VA-form

vaddecuq VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	61
0	6	11	16	21	26 31

```
if MSR.VEC=0 then Vector_Unavailable()

src1 ← VR[VRA]
src2 ← VR[VRB]
cin  ← VR[VRC].bit[127]
sum  ← EXTZ(src1) + EXTZ(src2) + EXTZ(cin)

VR[VRT] ← Chop( EXTZ( Chop(sum >> 128, 1) ), 128 )
```

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].
Let cin be the integer value in bit 127 of VR[VRC].

src1 and src2 can be signed or unsigned integers.

The carry out of the sum of src1, src2, and cin are placed into VR[VRT].

Special Registers Altered:

None

Programming Note

The *Vector Add Unsigned Quadword* instructions support efficient wide-integer addition. The following code sequence can be used to implement a 512-bit signed or unsigned add operation.

```
vadduqm         vS3,vA3,vB3        # bits 384:511 of sum
vaddcuq         vC3,vA3,vB3        # carry out of bit 384 of sum
vaddeuqm        vS2,vA2,vB2,vC3    # bits 256:383 of sum
vaddecuq        vC2,vA2,vB2,vC3    # carry out of bit 256 of sum
vaddeuqm        vS1,vA1,vB1,vC2    # bits 128:255 of sum
vaddecuq        vC1,vA1,vB1,vC2    # carry out of bit 128 of sum
vaddeuqm        vS0,vA0,vB0,vC1    # bits 0:127 of sum
```

6.9.1.2 Vector Integer Subtract Instructions

Vector Subtract and Write Carry-Out Unsigned Word VX-form

vsubcuw VRT,VRA,VRB

4	VRT	VRA	VRB	1408
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop  ← EXTZ((VRA)i:i+31)
   bop  ← EXTZ((VRB)i:i+31)
   temp ← (aop +int ¬bop +int 1) >> 32
   VRTi:i+31 ← temp & 0x0000_0001
end
```

For each integer value i from 0 to 3, do the following.

Unsigned-integer word element i in VRB is subtracted from unsigned-integer word element i in VRA. The complement of the borrow out of bit 0 of the 32-bit difference is zero-extended to 32 bits and placed into word element i of VRT.

Special Registers Altered:

None

Vector Subtract Signed Byte Saturate VX-form

vsubsbs VRT,VRA,VRB

4	VRT	VRA	VRB	1792
0	6	11	16	21 31

```
do i=0 to 127 by 8
   aop ← EXTS((VRA)i:i+7)
   bop ← EXTS((VRB)i:i+7)
   VRTi:i+7 ← Clamp(aop +int ¬bop +int 1, -128, 127)24:31
end
```

For each integer value i from 0 to 15, do the following.

Signed-integer byte element i in VRB is subtracted from signed-integer byte element i in VRA.

- If the intermediate result is greater than 127 the result saturates to 127.
- If the intermediate result is less than -128 the result saturates to -128.

The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:

SAT

Vector Subtract Signed Halfword Saturate VX-form

vsubshs VRT,VRA,VRB

4	VRT	VRA	VRB	1856
0	6	11	16	21 31

```
do i=0 to 127 by 16
   aop ← EXTS((VRA)i:i+15)
   bop ← EXTS((VRB)i:i+15)
   temp ← aop +int ¬bop +int 1
   VRTi:i+15 ← Clamp(temp, -2^15, 2^15-1)16:31
end
```

For each integer value i from 0 to 7, do the following.

Signed-integer halfword element i in VRB is subtracted from signed-integer halfword element i in VRA.

- If the intermediate result is greater than 2^{15}-1 the result saturates to 2^{15}-1.
- If the intermediate result is less than -2^{15} the result saturates to -2^{15}.

The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:

SAT

Vector Subtract Signed Word Saturate VX-form

vsubsws VRT,VRA,VRB

4	VRT	VRA	VRB	1920
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop ← EXTS((VRA)i:i+31)
   bop ← EXTS((VRB)i:i+31)
   VRTi:i+31 ← Clamp(aop +int ¬bop +int 1,-2^31,2^31-1)
end
```

For each integer value i from 0 to 3, do the following.

Signed-integer word element i in VRB is subtracted from signed-integer word element i in VRA.

- If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.
- If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

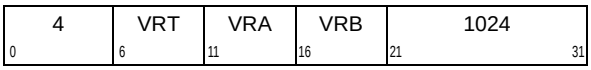
The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:

SAT

Vector Subtract Unsigned Byte Modulo VX-form

vsububm VRT,VRA,VRB




```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← Chop( aop +int ¬bop +int 1, 8 )
end
```

For each integer value i from 0 to 15, do the following.

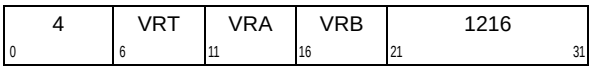
Unsigned-integer byte element i in VRB is subtracted from unsigned-integer byte element i in VRA. The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:

None

Vector Subtract Unsigned Doubleword Modulo VX-form

vsubudm VRT,VRA,VRB




```
do i = 0 to 1
   aop ← VR[VRA].dword[i]
   bop ← VR[VRB].dword[i]
   VR[VRT].dword[i] ← Chop( aop +int ~bop +int 1, 64 )
end
```

For each integer value `i` from 0 to 1, do the following.

The integer value in doubleword element `i` of `VR[VRB]` is subtracted from the integer value in doubleword element `i` of `VR[VRA]`.

The low-order 64 bits of the result are placed into doubleword element `i` of `VR[VRT]`.

Special Registers Altered:

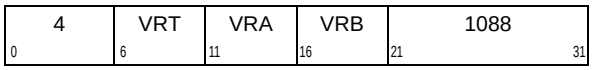
None

Programming Note

vsubudm can be used for signed or unsigned integers.

Vector Subtract Unsigned Halfword Modulo VX-form

vsubuhm VRT,VRA,VRB




```
do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+16 ← Chop( aop +int ¬bop +int 1, 16 )
end
```

For each integer value i from 0 to 7, do the following.

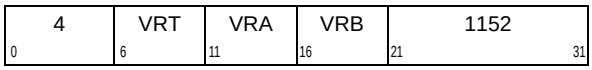
Unsigned-integer halfword element i in VRB is subtracted from unsigned-integer halfword element i in VRA. The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:

None

Vector Subtract Unsigned Word Modulo VX-form

vsubuwm VRT,VRA,VRB




```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← Chop( aop +int ¬bop +int 1, 32 )
end
```

For each integer value i from 0 to 3, do the following.

Unsigned-integer word element i in VRB is subtracted from unsigned-integer word element i in VRA. The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:

None

Vector Subtract Unsigned Byte Saturate VX-form

vsububs VRT,VRA,VRB

4	VRT	VRA	VRB	1536
0	6	11	16	21 31

```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← Clamp(aop +int ¬bop +int 1, 0, 255)24:31
end
```

For each integer value i from 0 to 15, do the following.

Unsigned-integer byte element i in VRB is subtracted from unsigned-integer byte element i in VRA. If the intermediate result is less than 0 the result saturates to 0. The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:

SAT

Vector Subtract Unsigned Halfword Saturate VX-form

vsubuhs VRT,VRA,VRB

4	VRT	VRA	VRB	1600
0	6	11	16	21 31

```
do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← Clamp(aop +int ¬bop +int 1,0,2^16-1)16:31
end
```

For each integer value i from 0 to 7, do the following.

Unsigned-integer halfword element i in VRB is subtracted from unsigned-integer halfword element i in VRA. If the intermediate result is less than 0 the result saturates to 0. The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:

SAT

Vector Subtract Unsigned Word Saturate VX-form

vsubuws VRT,VRA,VRB

4	VRT	VRA	VRB	1664
0	6	11	16	21 31

```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← Clamp(aop +int ¬bop +int 1, 0, 2^32-1)
end
```

For each integer value i from 0 to 7, do the following.

Unsigned-integer word element i in VRB is subtracted from unsigned-integer word element i in VRA.

– If the intermediate result is less than 0 the result saturates to 0.

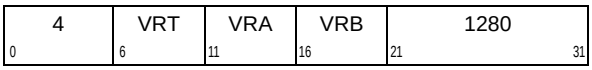
The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:

SAT

Vector Subtract Unsigned Quadword Modulo VX-form

vsubuqm VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()
src1 ← VR[VRA]
src2 ← VR[VRB]
sum  ← EXTZ(src1) + EXTZ(¬src2) + EXTZ(1)
VR[VRT] ← Chop(sum, 128)
```

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].

src1 and src2 can be signed or unsigned integers.

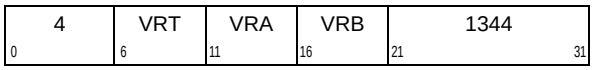
The rightmost 128 bits of the sum of src1, the one's complement of src2, and the value 1 are placed into VR[VRT].

Special Registers Altered:

None

Vector Subtract & write Carry Unsigned Quadword VX-form

vsubcuq VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()
src1 ←  VR[VRA]
src2 ← VR[VRB]
sum  ← EXTZ(src1) + EXTZ(¬src2) + EXTZ(1)
VR[VRT] ← Chop( EXTZ( Chop(sum >> 128, 1) ), 128 )
```

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].

src1 and src2 can be signed or unsigned integers.

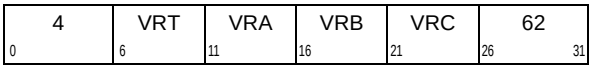
The carry out of the sum of src1, the one's complement of src2, and the value 1 is placed into VR[VRT].

Special Registers Altered:

None

Vector Subtract Extended Unsigned Quadword Modulo VA-form

vsubeuqm VRT,VRA,VRB,VRC




```
if MSR.VEC=0 then Vector_Unavailable()
src1 ← VR[VRA]
src2 ← VR[VRB]
cin  ← VR[VRC].bit[127]
sum  ← EXTZ(src1) + EXTZ(¬src2) + EXTZ(cin)
VR[VRT] ← Chop(sum, 128)
```

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].
Let cin be the integer value in bit 127 of VR[VRC].

src1 and src2 can be signed or unsigned integers.

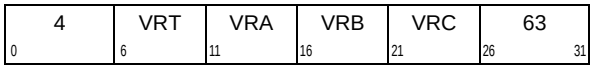
The rightmost 128 bits of the sum of src1, the one's complement of src2, and cin are placed into VR[VRT].

Special Registers Altered:

None

Vector Subtract Extended & write Carry Unsigned Quadword VA-form

vsubecuq VRT,VRA,VRB,VRC




```
if MSR.VEC=0 then Vector_Unavailable()
src1 ← VR[VRA]
src2 ← VR[VRB]
cin  ← VR[VRC].bit[127]
sum  ← EXTZ(src1) + EXTZ(¬src2) + EXTZ(cin)
VR[VRT] ← Chop( EXTZ( Chop(sum >> 128, 1) ), 128 )
```

Let src1 be the integer value in VR[VRA].
Let src2 be the integer value in VR[VRB].
Let cin be the integer value in bit 127 of VR[VRC].

src1 and src2 can be signed or unsigned integers.

The carry out of the sum of src1, the one's complement of src2, and cin are placed into VR[VRT].

Special Registers Altered:

None

Programming Note

The *Vector Subtract Unsigned Quadword* instructions support efficient wide-integer subtraction. The following code sequence can be used to implement a 512-bit signed or unsigned subtract operation.

```
vsubuqm     vS3,vA3,vB3          # bits 384:511 of difference
vsubcuq     vC3,vA3,vB3          # carry out of bit 384 of difference
vsubeuqm    vS2,vA2,vB2,vC3      # bits 256:383 of difference
vsubecuq    vC2,vA2,vB2,vC3      # carry out of bit 256 of difference
vsubeuqm    vS1,vA1,vB1,vC2      # bits 128:255 of difference
vsubecuq    vC1,vA1,vB1,vC2      # carry out of bit 128 of difference
vsubeuqm    vS0,vA0,vB0,vC1      # bits 0:127 of difference
```

6.9.1.3 Vector Integer Multiply Instructions

Vector Multiply Even Signed Byte VX-form

vmulesb VRT,VRA,VRB

4	VRT	VRA	VRB	776
0	6	11	16	21 31

```
do i=0 to 127 by 16
   prod ← EXTS((VRA)i:i+7) xsi EXTS((VRB)i:i+7)
   VRTi:i+15 ← Chop( prod, 16 )
end
```

For each integer value i from 0 to 7, do the following.
Signed-integer byte element i×2 in VRA is multiplied by signed-integer byte element i×2 in VRB. The low-order 16 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

Vector Multiply Odd Signed Byte VX-form

vmulosb VRT,VRA,VRB

4	VRT	VRA	VRB	264
0	6	11	16	21 31

```
do i=0 to 127 by 16
   prod ← EXTS((VRA)i+8:i+15) xsi EXTS((VRB)i+8:i+15)
   VRTi:i+15 ← Chop( prod, 16 )
end
```

For each integer value i from 0 to 7, do the following.
Signed-integer byte element i×2+1 in VRA is multiplied by signed-integer byte element i×2+1 in VRB. The low-order 16 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

Vector Multiply Even Unsigned Byte VX-form

vmuleub VRT,VRA,VRB

4	VRT	VRA	VRB	520
0	6	11	16	21 31

```
do i=0 to 127 by 16
   prod ← EXTZ((VRA)i:i+7) xui EXTZ((VRB)i:i+7)
   VRTi:i+15 ← Chop(prod, 16)
end
```

For each integer value i from 0 to 7, do the following.
Unsigned-integer byte element i×2 in VRA is multiplied by unsigned-integer byte element i×2 in VRB. The low-order 16 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

Vector Multiply Odd Unsigned Byte VX-form

vmuloub VRT,VRA,VRB

4	VRT	VRA	VRB	8
0	6	11	16	21 31

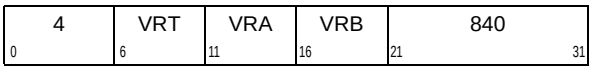
```
do i=0 to 127 by 16
   prod ← EXTZ((VRA)i+8:i+15) xui EXTZ((VRB)i+8:i+15)
   VRTi:i+15 ← Chop( prod, 16 )
end
```

For each integer value i from 0 to 7, do the following.
Unsigned-integer byte element i×2+1 in VRA is multiplied by unsigned-integer byte element i×2+1 in VRB. The low-order 16 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

Vector Multiply Even Signed Halfword VX-form

vmulesh VRT,VRA,VRB



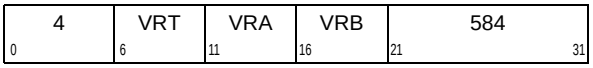

```
do i=0 to 127 by 32
   prod ← EXTS((VRA)i:i+15) xsi EXTS((VRB)i:i+15)
   VRTi:i+31 ← Chop( prod, 32 )
end
```

For each integer value i from 0 to 3, do the following.
Signed-integer halfword element i×2 in VRA is multiplied by signed-integer halfword element i×2 in VRB. The low-order 32 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

Vector Multiply Even Unsigned Halfword VX-form

vmuleuh VRT,VRA,VRB



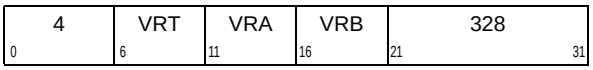

```
do i=0 to 127 by 32
   prod ← EXTZ((VRA)i:i+15) xui EXTZ((VRB)i:i+15)
   VRTi:i+31 ← Chop(prod, 32)
end
```

For each integer value i from 0 to 3, do the following.
Unsigned-integer halfword element i×2 in VRA is multiplied by unsigned-integer halfword element i×2 in VRB. The low-order 32 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

Vector Multiply Odd Signed Halfword VX-form

vmulosh VRT,VRA,VRB



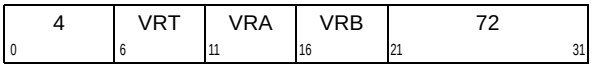

```
do i=0 to 127 by 32
   prod ← EXTS((VRA)i+16:i+31) xsi EXTS((VRB)i+16:i+31)
   VRTi:i+31 ← Chop( prod, 32 )
end
```

For each integer value i from 0 to 3, do the following.
Signed-integer halfword element i×2+1 in VRA is multiplied by signed-integer halfword element i×2+1 in VRB. The low-order 32 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

Vector Multiply Odd Unsigned Halfword VX-form

vmulouh VRT,VRA,VRB



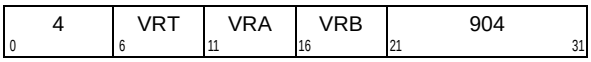

```
do i=0 to 127 by 32
   prod ← EXTZ((VRA)i+16:i+31)xui EXTZ((VRB)i+16:i+31)
   VRTi:i+31 ← Chop( prod, 32 )
end
```

For each integer value i from 0 to 3, do the following.
Unsigned-integer halfword element i×2+1 in VRA is multiplied by unsigned-integer halfword element i×2+1 in VRB. The low-order 32 bits of the product are placed into halfword element i VRT.

Special Registers Altered:
None

Vector Multiply Even Signed Word VX-form

vmulesw VRT,VRA,VRB




```
do i = 0 to 1
   src1 ← VR[VRA].word[2×i]
   src2 ← VR[VRB].word[2×i]
   VR[VRT].dword[i] ← src1 ×si src2
end
```

For each integer value i from 0 to 1, do the following.

The signed integer in word element 2×i of VR[VRA] is multiplied by the signed integer in word element 2×i of VR[VRB].

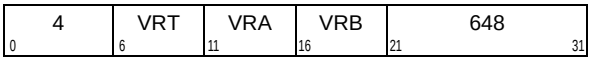
The 64-bit product is placed into doubleword element i of VR[VRT].

Special Registers Altered:

None

Vector Multiply Even Unsigned Word VX-form

vmuleuw VRT,VRA,VRB




```
do i = 0 to 1
   src1 ← VR[VRA].word[2×i]
   src2 ← VR[VRB].word[2×i]
   VR[VRT].dword[i] ← src1 ×ui src2
end
```

For each integer value i from 0 to 1, do the following.

The unsigned integer in word element 2×i of VR[VRA] is multiplied by the unsigned integer in word element 2×i of VR[VRB].

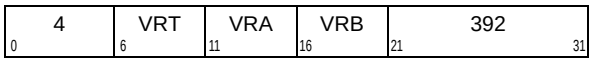
The 64-bit product is placed into doubleword element i of VR[VRT].

Special Registers Altered:

None

Vector Multiply Odd Signed Word VX-form

vmulosw VRT,VRA,VRB




```
do i = 0 to 1
   src1 ← VR[VRA].word[2×i+1]
   src2 ← VR[VRB].word[2×i+1]
   VR[VRT].dword[i] ← src1 ×si src2
end
```

For each integer value i from 0 to 1, do the following.

The signed integer in word element 2×i+1 of VR[VRA] is multiplied by the signed integer in word element 2×i+1 of VR[VRB].

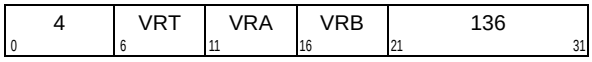
The 64-bit product is placed into doubleword element i of VR[VRT].

Special Registers Altered:

None

Vector Multiply Odd Unsigned Word VX-form

vmulouw VRT,VRA,VRB




```
do i = 0 to 1
   src1 ← VR[VRA].word[2×i+1]
   src2 ← VR[VRB].word[2×i+1]
   VR[VRT].dword[i] ← src1 ×ui src2
end
```

For each integer value i from 0 to 1, do the following.

The unsigned integer in word element 2×i+1 of VR[VRA] is multiplied by the unsigned integer in word element 2×i+1 of VR[VRB].

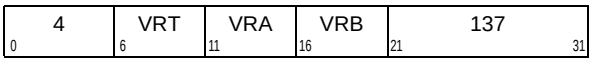
The 64-bit product is placed into doubleword element i of VR[VRT].

Special Registers Altered:

None

Vector Multiply Unsigned Word Modulo VX-form

vmuluwm VRT,VRA,VRB




```
do i = 0 to 3
   src1 ← VR[VRA].word[i]
   src2 ← VR[VRB].word[i]
   VR[VRT].word[i] ← Chop( src1 ×ui src2, 32 )
end
```

The integer in word element i of VR[VRA] is multiplied by the integer in word element i of VR[VRB].

The least-significant 32 bits of the product are placed into word element i of VR[VRT].

Special Registers Altered:

None

Programming Note

vmuluwm can be used for unsigned or signed integers.

6.9.1.4 Vector Integer Multiply-Add/Sum Instructions

Vector Multiply-High-Add Signed Halfword Saturate VA-form

vmhaddshs VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	32
0	6	11	16	21	26 31

```
do i=0 to 127 by 16
   prod ← EXTS((VRA)i:i+15) xsi EXTS((VRB)i:i+15)
   sum  ← (prod >>si 15) +int EXTS((VRC)i:i+15)
   VRTi:i+15 ← Clamp(sum, -2^15, 2^15-1)16:31
end
```

For each vector element i from 0 to 7, do the following.

Signed-integer halfword element i in VRA is multiplied by signed-integer halfword element i in VRB, producing a 32-bit signed-integer product. Bits 0:16 of the product are added to signed-integer halfword element i in VRC.

- If the intermediate result is greater than 2^{15}-1 the result saturates to 2^{15}-1.
- If the intermediate result is less than -2^{15} the result saturates to -2^{15}.

The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:

SAT

Vector Multiply-High-Round-Add Signed Halfword Saturate VA-form

vmhraddshs VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	33
0	6	11	16	21	26 31

```
do i=0 to 127 by 16
   temp ← EXTS((VRC)i:i+15)
   prod ← EXTS((VRA)i:i+15) xsi EXTS((VRB)i:i+15)
   sum  ← ((prod +int 0x0000_4000) >>si 15) +int temp
   VRTi:i+15 ← Clamp(sum, -2^15, 2^15-1)16:31
end
```

For each vector element i from 0 to 7, do the following.

Signed-integer halfword element i in VRA is multiplied by signed-integer halfword element i in VRB, producing a 32-bit signed-integer product. The value `0x0000_4000` is added to the product, producing a 32-bit signed-integer sum. Bits 0:16 of the sum are added to signed-integer halfword element i in VRC.

- If the intermediate result is greater than 2^{15}-1 the result saturates to 2^{15}-1.
- If the intermediate result is less than -2^{15} the result saturates to -2^{15}.

The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:

SAT

Vector Multiply-Low-Add Unsigned Halfword Modulo VA-form

vmladduhm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	34
0	6	11	16	21	26 31

```
do i=0 to 127 by 16
   prod ← EXTZ((VRA)i:i+15) xui EXTZ((VRB)i:i+15)
   sum  ← Chop( prod, 16 ) +int (VRC)i:i+15
   VRTi:i+15 ← Chop( sum, 16 )
end
```

For each integer value i from 0 to 3, do the following.

Unsigned-integer halfword element i in VRA is multiplied by unsigned-integer halfword element i in VRB, producing a 32-bit unsigned-integer product. The low-order 16 bits of the product are added to unsigned-integer halfword element i in VRC.

The low-order 16 bits of the sum are placed into halfword element i of VRT.

Special Registers Altered:

None

Programming Note

vmladduhm can be used for unsigned or signed-integers.

Vector Multiply-Sum Unsigned Byte Modulo VA-form

vmsumubm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	36
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   temp ← EXTZ((VRC)i:i+31)
   do j=0 to 31 by 8
      prod ← EXTZ((VRA)i+j:i+j+7) xui EXTZ((VRB)i+j:i+j+7)
      temp ← temp +int prod
   end
   VRTi:i+31 ← Chop( temp, 32 )
end
```

For each word element in VRT the following operations are performed, in the order shown.

– Each of the four unsigned-integer byte elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer byte element in VRB, producing an unsigned-integer halfword product.

– The sum of these four unsigned-integer halfword products is added to the unsigned-integer word element in VRC.

– The unsigned-integer word result is placed into the corresponding word element of VRT.

Special Registers Altered:

None

Vector Multiply-Sum Mixed Byte Modulo VA-form

vmsummbm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	37
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   temp ← (VRC)i:i+31
   do j=0 to 31 by 8
      prod0:15 ← (VRA)i+j:i+j+7 xsui (VRB)i+j:i+j+7
      temp     ← temp +int EXTS(prod)
   end
   VRTi:i+31 ← temp
end
```

For each word element in VRT the following operations are performed, in the order shown.

– Each of the four signed-integer byte elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer byte element in VRB, producing a signed-integer product.

– The sum of these four signed-integer halfword products is added to the signed-integer word element in VRC.

– The signed-integer result is placed into the corresponding word element of VRT.

Special Registers Altered:

None

Vector Multiply-Sum Signed Halfword Modulo VA-form

vmsumshm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	40
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   temp ← (VRC)i:i+31
   do j=0 to 31 by 16
      prod0:31 ← (VRA)i+j:i+j+15 xsi (VRB)i+j:i+j+15
      temp     ← temp +int prod
   end
   VRTi:i+31 ← temp
end
```

For each word element in VRT the following operations are performed, in the order shown.

– Each of the two signed-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding signed-integer halfword element in VRB, producing a signed-integer product.

– The sum of these two signed-integer word products is added to the signed-integer word element in VRC.

– The signed-integer word result is placed into the corresponding word element of VRT.

Special Registers Altered:

None

Vector Multiply-Sum Signed Halfword Saturate VA-form

vmsumshs VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	41
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   temp ← EXTS((VRC)i:i+31)
   do j=0 to 31 by 16
      srcA ← EXTS((VRA)i+j:i+j+15)
      srcB ← EXTS((VRB)i+j:i+j+15)
      prod ← srcA xsi srcB
      temp ← temp +int prod
   end
   VRTi:i+31 ← Clamp(temp, -2^31, 2^31-1)
end
```

For each word element in VRT the following operations are performed, in the order shown.

– Each of the two signed-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding signed-integer halfword element in VRB, producing a signed-integer product.

– The sum of these two signed-integer word products is added to the signed-integer word element in VRC.

– If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1 and if it is less than -2^{31} it saturates to -2^{31}.

– The result is placed into the corresponding word element of VRT.

Special Registers Altered:

SAT

Vector Multiply-Sum Unsigned Halfword Modulo VA-form

vmsumuhm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	38
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   temp ← EXTZ((VRC)i:i+31)
   do j=0 to 31 by 16
      srcA ← EXTZ((VRA)i+j:i+j+15)
      srcB ← EXTZ((VRB)i+j:i+j+15)
      prod ← srcA xui srcB
      temp ← temp +int prod
   end
   VRTi:i+31 ← Chop( temp, 32 )
end
```

For each word element in VRT the following operations are performed, in the order shown.

– Each of the two unsigned-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer halfword element in VRB, producing an unsigned-integer word product.

– The sum of these two unsigned-integer word products is added to the unsigned-integer word element in VRC.

– The unsigned-integer result is placed into the corresponding word element of VRT.

Special Registers Altered:

None

Vector Multiply-Sum Unsigned Halfword Saturate VA-form

vmsumuhs VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	39
0	6	11	16	21	26 31

```
do i=0 to 127 by 32
   temp ← EXTZ((VRC)i:i+31)
   do j=0 to 31 by 16
      src1 ← EXTZ((VRA)i+j:i+j+15)
      src2 ← EXTZ((VRB)i+j:i+j+15)
      prod ← src1 xui src2
   end
   temp ← temp +int prod
   VRTi:i+31 ← Clamp(temp, 0, 2^32-1)
end
```

For each word element in VRT the following operations are performed, in the order shown.

– Each of the two unsigned-integer halfword elements contained in the corresponding word element of VRA is multiplied by the corresponding unsigned-integer halfword element in VRB, producing an unsigned-integer product.

– The sum of these two unsigned-integer word products is added to the unsigned-integer word element in VRC.

– If the intermediate result is greater than 2^{32}-1 the result saturates to 2^{32}-1.

– The result is placed into the corresponding word element of VRT.

Special Registers Altered:

SAT

Vector Multiply-Sum Unsigned Doubleword Modulo VA-form

vmsumudm VRT,VRA,VRB,VRC

4	VRT	VRA	VRB	VRC	35
0	6	11	16	21	26 31

```
temp ← EXTZ(VR[VRC])
do i = 0 to 1
   prod ← EXTZ(VR[VRA].dword[i]) × EXTZ(VR[VRB].dword[i])
   temp ← temp + prod
end
VR[VRT] ← Chop(temp, 128)
```

The unsigned integer value in doubleword element 0 of VR[VRA] is multiplied by the unsigned integer value in doubleword element 0 of VR[VRB] to produce a 128-bit product.

The unsigned integer value in doubleword element 1 of VR[VRA] is multiplied by the unsigned integer value in doubleword element 1 of VR[VRB] to produce a 128-bit product.

The two 128-bit unsigned integer products and the 128-bit unsigned integer in VR[VRC] are summed.

The low-order 128 bits of the sum are placed into VR[VRT]. Any carry out or overflow status is discarded.

Special Registers Altered:

None

Programming Note

A horizontal add of the doubleword elements in VR[VRA] can be performed using ***vmsumudm*** when VR[VRB] contains the doubleword integer values {1,1} and VR[VRC] contains the quadword integer value 0.

A horizontal subtract of the doubleword elements in VR[VRA] can be performed using ***vmsumudm*** when VR[VRB] contains the doubleword integer values {1,-1} and VR[VRC] contains the quadword integer value 0.

A multiply even unsigned doubleword operation can be performed using ***vmsumudm*** when the contents of doubleword element 1 of VR[VRA] or VR[VRB] are 0 and the contents of VR[VRC] to 0.

A multiply odd unsigned doubleword operation can be performed using ***vmsumudm*** when the contents of doubleword element 0 of VR[VRA] or VR[VRB] are 0 and the contents of VR[VRC] to 0.

6.9.1.5 Vector Integer Sum-Across Instructions

Vector Sum across Signed Word Saturate VX-form

vsumsws VRT,VRA,VRB

4	VRT	VRA	VRB	1928
0	6	11	16	21 31

```
temp ← EXTS((VRB)96:127)
do i=0 to 127 by 32
   temp ← temp +int EXTS((VRA)i:i+31)
end
VRT0:31   ← 0x0000_0000
VRT32:63  ← 0x0000_0000
VRT64:95  ← 0x0000_0000
VRT96:127 ← Clamp(temp, -2^31, 2^31-1)
```

The sum of the four signed-integer word elements in VRA is added to signed-integer word element 3 of VRB.

– If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.

– If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

The low-end 32 bits of the result are placed into word element 3 of VRT.

Word elements 0 to 2 of VRT are set to 0.

Special Registers Altered:

SAT

Vector Sum across Half Signed Word Saturate VX-form

vsum2sws VRT,VRA,VRB

4	VRT	VRA	VRB	1672
0	6	11	16	21 31

```
do i=0 to 127 by 64
   temp ← EXTS((VRB)i+32:i+63)
   do j=0 to 63 by 32
      temp ← temp +int EXTS((VRA)i+j:i+j+31)
   end
   VRTi:i+63 ← 0x0000_0000 || Clamp(temp, -2^31, 2^31-1)
end
```

Word elements 0 and 2 of VRT are set to 0.

The sum of the signed-integer word elements 0 and 1 in VRA is added to the signed-integer word element in bits 32:63 of VRB.

– If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.

– If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

The low-order 32 bits of the result are placed into word element 1 of VRT.

The sum of signed-integer word elements 2 and 3 in VRA is added to the signed-integer word element in bits 96:127 of VRB.

– If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.

– If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

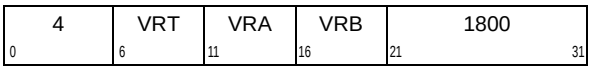
The low-order 32 bits of the result are placed into word element 3 of VRT.

Special Registers Altered:

SAT

Vector Sum across Quarter Signed Byte Saturate VX-form

vsum4sbs VRT,VRA,VRB




```
do i=0 to 127 by 32
   temp ← EXTS((VRB)i:i+31)
   do j=0 to 31 by 8
      temp ← temp +int EXTS((VRA)i+j:i+j+7)
   end
   VRTi:i+31 ← Clamp(temp, -2^31, 2^31-1)
end
```

For each integer value i from 0 to 3, do the following.

The sum of the four signed-integer byte elements contained in word element i of VRA is added to signed-integer word element i in VRB.

- If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.
- If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

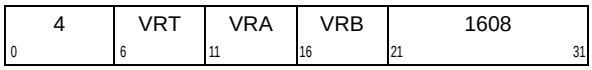
The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:

SAT

Vector Sum across Quarter Signed Halfword Saturate VX-form

vsum4shs VRT,VRA,VRB




```
do i=0 to 127 by 32
   temp ← EXTS((VRB)i:i+31)
   do j=0 to 31 by 16
      temp ← temp +int EXTS((VRA)i+j:i+j+15)
   end
   VRTi:i+31 ← Clamp(temp, -2^31, 2^31-1)
end
```

For each integer value i from 0 to 3, do the following.

The sum of the two signed-integer halfword elements contained in word element i of VRA is added to signed-integer word element i in VRB.

- If the intermediate result is greater than 2^{31}-1 the result saturates to 2^{31}-1.
- If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

The low-order 32 bits of the result are placed into the corresponding word element of VRT.

Special Registers Altered:

SAT

Vector Sum across Quarter Unsigned Byte Saturate VX-form

vsum4ubs VRT,VRA,VRB

4	VRT	VRA	VRB	1544
0	6	11	16	21 31

```
do i=0 to 127 by 32
   temp ← EXTZ((VB)i:i+31)
   do j=0 to 31 by 8
      temp ← temp +int EXTZ((VA)i+j:i+j+7)
   end
   VRTi:i+31 ← Clamp( temp, 0, 2^32-1 )
end
```

For each integer value i from 0 to 3, do the following.

The sum of the four unsigned-integer byte elements contained in word element i of VRA is added to unsigned-integer word element i in VRB.

– If the intermediate result is greater than 2^{32}-1 it saturates to 2^{32}-1.

The low-order 32 bits of the result are placed into word element i of VRT.

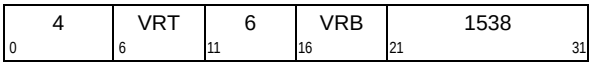
Special Registers Altered:

SAT

6.9.1.6 Vector Integer Negate Instructions

Vector Negate Word VX-form

vnegw VRT,VRB



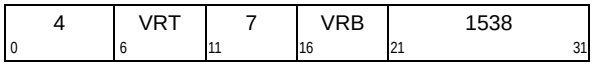

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src ← EXTS(VR[VRB].word[i])
   VR[VRT].word[i] ← Chop((¬src + 1), 32)
end
```

For each integer value i from 0 to 3, do the following.
The sum of the one's-complement of the signed integer in word element i of VR[VRB] and 1 is placed into word element i of VR[VRT].

Special Registers Altered:
None

Vector Negate Doubleword VX-form

vnegd VRT,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src ← EXTS(VR[VRB].dword[i])
   VR[VRT]dword[i] ← Chop((¬src + 1), 64)
end
```

For each integer value i from 0 to 1, do the following.
The sum of the one's-complement of the signed integer in doubleword element i of VR[VRB] and 1 is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

6.9.2 Vector Extend Sign Instructions

Vector Extend Sign Byte To Word VX-form

vextsb2w VRT,VRB

4	VRT	16	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   VR[VRT].word[i] ← EXTS32(VR[VRB].word[i].byte[3])
end
```

For each integer value i from 0 to 3, do the following.
The rightmost byte of word element i of VR[VRB] is sign-extended and placed into word element i of VR[VRT].

Special Registers Altered:
None

Vector Extend Sign Halfword To Word VX-form

vextsh2w VRT,VRB

4	VRT	17	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   VR[VRT].word[i] ← EXTS32(VR[VRB].word[i].hword[1])
end
```

For each integer value i from 0 to 3, do the following.
The rightmost halfword of word element i of VR[VRB] is sign-extended and placed into word element i of VR[VRT].

Special Registers Altered:
None

Vector Extend Sign Word To Doubleword VX-form

vextsw2d VRT,VRB

4	VRT	26	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   VR[VRT].dword[i] ← EXTS64(VR[VRB].dword[i].word[1])
end
```

For each integer value i from 0 to 1, do the following.
The rightmost word of doubleword element i of VR[VRB] is sign-extended and placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Extend Sign Byte To Doubleword VX-form

vextsb2d VRT,VRB

4	VRT	24	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   VR[VRT].dword[i] ← EXTS64(VR[VRB].dword[i].byte[7])
end
```

For each integer value i from 0 to 1, do the following.
The rightmost byte of doubleword element i of VR[VRB] is sign-extended and placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Extend Sign Halfword To Doubleword VX-form

vextsh2d VRT,VRB

4	VRT	25	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

if “vextsh2d” then do i = 0 to 1
   VR[VRT].dword[i] ← EXTS64(VR[VRB].dword[i].hword[3])
end
```

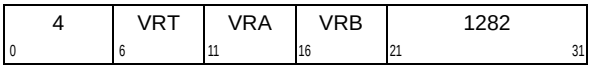
For each integer value i from 0 to 1, do the following.
The rightmost halfword of doubleword element i of VR[VRB] is sign-extended and placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

6.9.2.1 Vector Integer Average Instructions

Vector Average Signed Byte VX-form

vavgsb VRT,VRA,VRB




```
do i=0 to 127 by 8
        aop ← EXTS((VRA)i:i+7)
        bop ← EXTS((VRB)i:i+7)
        VRTi:i+7 ← Chop(( aop +int bop +int 1 ) >> 1, 8)
end
```

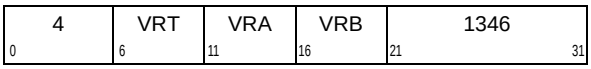
For each integer value i from 0 to 15, do the following.
Signed-integer byte element i in VRA is added to signed-integer byte element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:
None

Vector Average Signed Halfword VX-form

vavgsh VRT,VRA,VRB




```
do i=0 to 127 by 16
   aop ← EXTS((VRA)i:i+15)
   bop ← EXTS((VRB)i:i+15)
   VRTi:i+15 ← Chop(( aop +int bop +int 1 ) >> 1, 16)
end
```

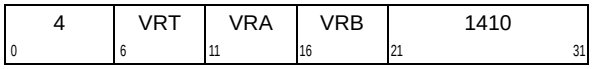
For each integer value i from 0 to 7, do the following.
Signed-integer halfword element i in VRA is added to signed-integer halfword element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Average Signed Word VX-form

vavgsw VRT,VRA,VRB




```
do i=0 to 127 by 32
   aop ← EXTS((VRA)i:i+31)
   bop ← EXTS((VRB)i:i+31)
   VRTi:i+31 ← Chop(( aop +int bop +int 1 ) >> 1, 32)
end
```

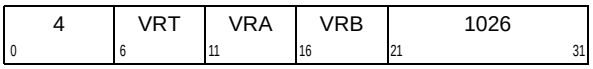
For each integer value i from 0 to 3, do the following.
Signed-integer word element i in VRA is added to signed-integer word element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:
None

Vector Average Unsigned Byte VX-form

vavgub VRT,VRA,VRB




```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7
   VRTi:i+7 ← Chop((aop +int bop +int 1) >>ui 1, 8)
end
```

For each integer value i from 0 to 15, do the following.

Unsigned-integer byte element i in VRA is added to unsigned-integer byte element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

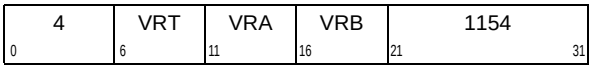
The low-order 8 bits of the result are placed into byte element i of VRT.

Special Registers Altered:

None

Vector Average Unsigned Word VX-form

vavguw VRT,VRA,VRB




```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← Chop((aop +int bop +int 1) >>ui 1, 32)
end
```

For each integer value i from 0 to 3, do the following.

Unsigned-integer word element i in VRA is added to unsigned-integer word element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

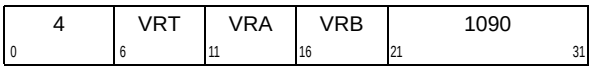
The low-order 32 bits of the result are placed into word element i of VRT.

Special Registers Altered:

None

Vector Average Unsigned Halfword VX-form

vavguh VRT,VRA,VRB




```
do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← Chop((aop +int bop +int 1) >>ui 1, 16)
end
```

For each integer value i from 0 to 7, do the following.

Unsigned-integer halfword element i in VRA is added to unsigned-integer halfword element i in VRB. The sum is incremented by 1 and then shifted right 1 bit.

The low-order 16 bits of the result are placed into halfword element i of VRT.

Special Registers Altered:

None

6.9.2.2 Vector Integer Absolute Difference Instructions

This section describes a set of instructions that return the absolute value of the difference of integer values.

Vector Absolute Difference Unsigned Byte VX-form

vabsdub VRT,VRA,VRB

4	VRT	VRA	VRB	1027
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

for i = 0 to 15
   src1 ← EXTZ(VR[VRA].byte[i])
   src2 ← EXTZ(VR[VRB].byte[i])
   if (src1>src2) then
      VR[VRT].byte[i] ← Chop(src1 + ¬src2 + 1, 8)
   else
      VR[VRT].byte[i] ← Chop(src2 + ¬src1 + 1, 8)
end
```

For each integer value i from 0 to 15, do the following.
The unsigned integer value in byte element i of VR[VRA] is subtracted by the unsigned integer value in byte element i of VR[VRB]. The absolute value of the difference is placed into byte element i of VR[VRT].

Special Registers Altered:
None

Vector Absolute Difference Unsigned Halfword VX-form

vabsduh VRT,VRA,VRB

4	VRT	VRA	VRB	1091
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

for i = 0 to 7
   src1 ← EXTZ(VR[VRA].hword[i])
   src2 ← EXTZ(VR[VRB].hword[i])
   if (src1>src2) then
      VR[VRT].hword[i] ← Chop(src1 + ¬src2 + 1, 16)
   else
      VR[VRT].hword[i] ← Chop(src2 + ¬src1 + 1, 16)
end
```

For each integer value i from 0 to 7, do the following.
The unsigned integer value in halfword element i of VR[VRA] is subtracted by the unsigned integer value in halfword element i of VR[VRB]. The absolute value of the difference is placed into halfword element i of VR[VRT].

Special Registers Altered:
None

Vector Absolute Difference Unsigned Word VX-form

vabsduw VRT,VRA,VRB

4	VRT	VRA	VRB	1155
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

for i = 0 to 3
   src1 ← EXTZ(VR[VRA].word[i])
   src2 ← EXTZ(VR[VRB].word[i])
   if (src1>src2) then
      VR[VRT].word[i] ← Chop(src1 + ¬src2 + 1, 32)
   else
      VR[VRT].word[i] ← Chop(src2 + ¬src1 + 1, 32)
end
```

For each integer value i from 0 to 3, do the following.

The unsigned integer value in word element i of VR[VRA] is subtracted by the unsigned integer value in word element i of VR[VRB]. The absolute value of the difference is placed into word element i of VR[VRT].

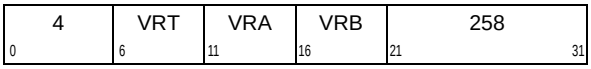
Special Registers Altered:

None

6.9.2.3 Vector Integer Maximum and Minimum Instructions

Vector Maximum Signed Byte VX-form

vmaxsb VRT,VRA,VRB



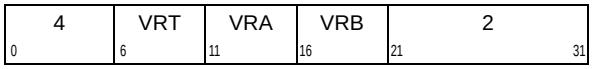

```
do i=0 to 127 by 8
   aop ← EXTS((VRA)i:i+7)
   bop ← EXTS((VRB)i:i+7)
   VRTi:i+7 ← ( aop >si bop ) ? (VRA)i:i+7 : (VRB)i:i+7
end
```

For each integer value i from 0 to 15, do the following.
Signed-integer byte element i in VRA is compared to signed-integer byte element i in VRB. The larger of the two values is placed into byte element i of VRT.

Special Registers Altered:
None

Vector Maximum Unsigned Byte VX-form

vmaxub VRT,VRA,VRB



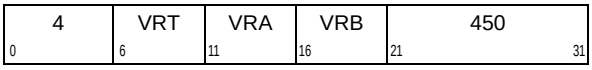

```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7)
   VRTi:i+7 ← (aop >ui bop) ? (VRA)i:i+7 : (VRB)i:i+7
end
```

For each integer value i from 0 to 15, do the following.
Unsigned-integer byte element i in VRA is compared to unsigned-integer byte element i in VRB. The larger of the two values is placed into byte element i of VRT.

Special Registers Altered:
None

Vector Maximum Signed Doubleword VX-form

vmaxsd VRT,VRA,VRB



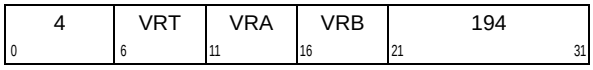

```
do i = 0 to 1
   aop ← VR[VRA].dword[i]
   bop ← VR[VRB].dword[i]
   VR[VRT].dword[i] ← (aop >si bop) ? aop : bop
end
```

For each integer value i from 0 to 1, do the following.
The signed integer value in doubleword element i of VR[VRA] is compared to the signed integer value in doubleword element i of VR[VRB]. The larger of the two values is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Maximum Unsigned Doubleword VX-form

vmaxud VRT,VRA,VRB



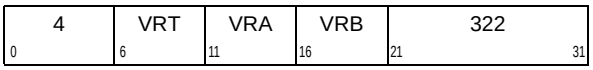

```
do i = 0 to 1
   aop ← VR[VRA].dword[i]
   bop ← VR[VRB].dword[i]
   VR[VRT].dword[i] ← (aop >ui bop) ? aop : bop
end
```

For each integer value i from 0 to 1, do the following.
The unsigned integer value in doubleword element i of VR[VRA] is compared to the unsigned integer value in doubleword element i of VR[VRB]. The larger of the two values is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Maximum Signed Halfword VX-form

vmaxsh VRT,VRA,VRB



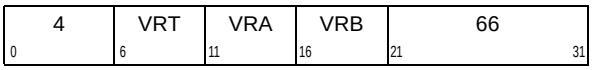

```
do i=0 to 127 by 16
   aop ← EXTS((VRA)i:i+15)
   bop ← EXTS((VRB)i:i+15
   VRTi:i+15 ← ( aop >si bop ) ? (VRA)i:i+15 : (VRB)i:i+15
end
```

For each integer value i from 0 to 7, do the following.
Signed-integer halfword element i in VRA is compared to signed-integer halfword element i in VRB. The larger of the two values is placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Maximum Unsigned Halfword VX-form

vmaxuh VRT,VRA,VRB



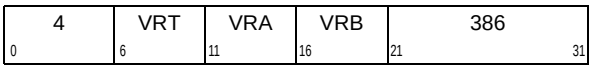

```
do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← (aop >ui bop) ? (VRA)i:i+15 : (VRB)i:i+15
end
```

For each integer value i from 0 to 7, do the following.
Unsigned-integer halfword element i in VRA is compared to unsigned-integer halfword element i in VRB. The larger of the two values is placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Maximum Signed Word VX-form

vmaxsw VRT,VRA,VRB



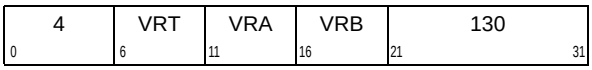

```
do i=0 to 127 by 32
   aop ← EXTS((VRA)i:i+31)
   bop ← EXTS((VRB)i:i+31)
   VRTi:i+31 ← ( aop >si bop ) ? (VRA)i:i+31 : (VRB)i:i+31
end
```

For each integer value i from 0 to 3, do the following.
Signed-integer word element i in VRA is compared to signed-integer word element i in VRB. The larger of the two values is placed into word element i of VRT.

Special Registers Altered:
None

Vector Maximum Unsigned Word VX-form

vmaxuw VRT,VRA,VRB



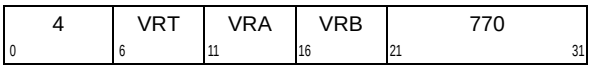

```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← (aop >ui bop) ? (VRA)i:i+31 : (VRB)i:i+31
end
```

For each integer value i from 0 to 3, do the following.
Unsigned-integer word element i in VRA is compared to unsigned-integer word element i in VRB. The larger of the two values is placed into word element i of VRT.

Special Registers Altered:
None

Vector Minimum Signed Byte VX-form

vminsb VRT,VRA,VRB



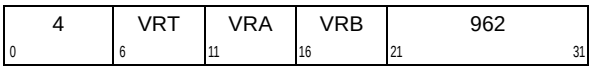

```
do i=0 to 127 by 8
   aop ← EXTS((VRA)i:i+7)
   bop ← EXTS((VRB)i:i+7)
   VRTi:i+7 ← (aop <si bop) ? (VRA)i:i+7 : (VRB)i:i+7
end
```

For each integer value i from 0 to 15, do the following.
Signed-integer byte element i in VRA is compared to signed-integer byte element i in VRB. The smaller of the two values is placed into byte element i of VRT.

Special Registers Altered:
None

Vector Minimum Signed Doubleword VX-form

vminsd VRT,VRA,VRB



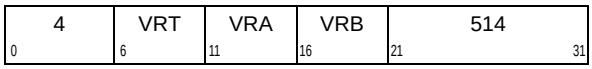

```
do i = 0 to 1
   aop ← VR[VRA].dword[i]
   bop ← VR[VRB].dword[i]
   VR[VRT].dword[i] ← (EXTS(aop) <si EXTS(bop)) ? aop : bop
end
```

For each integer value i from 0 to 1, do the following.
The signed integer value in doubleword element i of VR[VRA] is compared to the signed integer value in doubleword element i of VR[VRB]. The smaller of the two values is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Minimum Unsigned Byte VX-form

vminub VRT,VRA,VRB



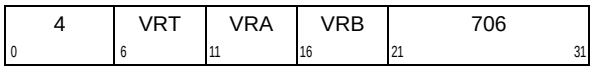

```
do i=0 to 127 by 8
   aop ← EXTZ((VRA)i:i+7)
   bop ← EXTZ((VRB)i:i+7
   VRTi:i+7 ← ( aop <ui bop ) ? (VRA)i:i+7 : (VRB)i:i+7
end
```

For each integer value i from 0 to 15, do the following.
Unsigned-integer byte element i in VRA is compared to unsigned-integer byte element i in VRB. The smaller of the two values is placed into byte element i of VRT.

Special Registers Altered:
None

Vector Minimum Unsigned Doubleword VX-form

vminud VRT,VRA,VRB



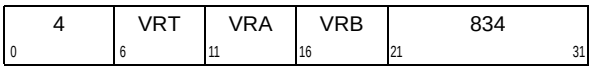

```
do i = 0 to 1
   aop ← VR[VRA].dword[i]
   bop ← VR[VRB].dword[i]
   VR[VRT].dword[i] ← (aop <ui bop) ? aop : bop
end
```

For each integer value i from 0 to 1, do the following.
The unsigned integer value in doubleword element i of VR[VRA] is compared to the unsigned integer value in doubleword element i of VR[VRB]. The smaller of the two values is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Minimum Signed Halfword VX-form

vminsh VRT,VRA,VRB




```
do i=0 to 127 by 16
   aop ← EXTS((VRA)i:i+15)
   bop ← EXTS((VRB)i:i+15)
   VRTi:i+15 ← ( aop <si bop ) ? (VRA)i:i+15 : (VRB)i:i+15
end
```

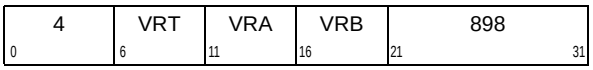
For each integer value i from 0 to 7, do the following.
Signed-integer halfword element i in VRA is compared to signed-integer halfword element i in VRB. The smaller of the two values is placed into halfword element i of VRT.

Special Registers Altered:

None

Vector Minimum Signed Word VX-form

vminsw VRT,VRA,VRB




```
do i=0 to 127 by 32
   aop ← EXTS((VRA)i:i+31)
   bop ← EXTS((VRB)i:i+31)
   VRTi:i+31 ← ( aop <si bop ) ? (VRA)i:i+31 : (VRB)i:i+31
end
```

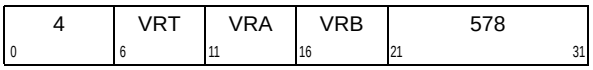
For each integer value i from 0 to 3, do the following.
Signed-integer word element i in VRA is compared to signed-integer word element i in VRB. The smaller of the two values is placed into word element i of VRT.

Special Registers Altered:

None

Vector Minimum Unsigned Halfword VX-form

vminuh VRT,VRA,VRB




```
do i=0 to 127 by 16
   aop ← EXTZ((VRA)i:i+15)
   bop ← EXTZ((VRB)i:i+15)
   VRTi:i+15 ← ( aop <ui bop ) ? (VRA)i:i+15 : (VRB)i:i+15
end
```

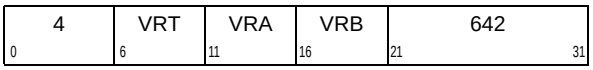
For each integer value i from 0 to 7, do the following.
Unsigned-integer halfword element i in VRA is compared to unsigned-integer halfword element i in VRB. The smaller of the two values is placed into halfword element i of VRT.

Special Registers Altered:

None

Vector Minimum Unsigned Word VX-form

vminuw VRT,VRA,VRB




```
do i=0 to 127 by 32
   aop ← EXTZ((VRA)i:i+31)
   bop ← EXTZ((VRB)i:i+31)
   VRTi:i+31 ← ( aop <ui bop ) ? (VRA)i:i+31 : (VRB)i:i+31
end
```

For each integer value i from 0 to 3, do the following.
Unsigned-integer word element i in VRA is compared to unsigned-integer word element i in VRB. The smaller of the two values is placed into word element i of VRT.

Special Registers Altered:

None

6.9.3 Vector Integer Compare Instructions

The *Vector Integer Compare* instructions compare two Vector Registers element by element, interpreting the elements as unsigned or signed-integers depending on the instruction, and set the corresponding element of the target Vector Register to all 1s if the relation being tested is true and to all 0s if the relation being tested is false.

If Rc=1 CR Field 6 is set to reflect the result of the comparison, as follows.

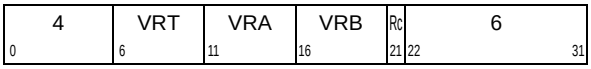
Bit	Description
0	The relation is true for all element pairs (i.e., VRT is set to all 1s)
1	0
2	The relation is false for all element pairs (i.e., VRT is set to all 0s)
3	0

Programming Note

***vcmpequb*[.]**, ***vcmpequh*[.]**, ***vcmpequw*[.]**, and ***vcmpequd*[.]** can be used for unsigned or signed-integers.

Vector Compare Equal Unsigned Byte VC-form

vcmpequb VRT,VRA,VRB (Rc=0)
vcmpequb. VRT,VRA,VRB (Rc=1)



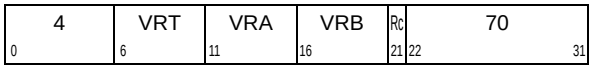

```
do i=0 to 127 by 8
   VRT_{i:i+7} ← ((VRA)_{i:i+7} =int (VRB)_{i:i+7}) ? 8^1 : 8^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 15, do the following.
Unsigned-integer byte element i in VRA is compared to unsigned-integer byte element i in VRB. Byte element i in VRT is set to all 1s if unsigned-integer byte element i in VRA is equal to unsigned-integer byte element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:
CR field 6 . . . . . .(if Rc=1)

Vector Compare Equal Unsigned Halfword VC-form

vcmpequh VRT,VRA,VRB (Rc=0)
vcmpequh. VRT,VRA,VRB (Rc=1)



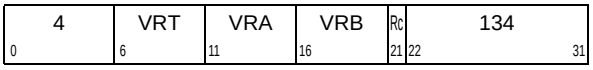

```
do i=0 to 127 by 16
   VRT_{i:i+15} ←((VRA)_{i:i+15} =int (VRB)_{i:i+15}) ? 16^1 : 16^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 7, do the following.
Unsigned-integer halfword element i in VRA is compared to unsigned-integer halfword element element i in VRB. Halfword element i in VRT is set to all 1s if unsigned-integer halfword element i in VRA is equal to unsigned-integer halfword element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:
CR field 6 . . . . . (if Rc=1)

Vector Compare Equal Unsigned Word VC-form

vcmpequw VRT,VRA,VRB (Rc=0)
vcmpequw. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 32
   VRT[i:i+31] ← ((VRA)[i:i+31] =int (VRB)[i:i+31]) ? 32^1 : 32^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 3, do the following.

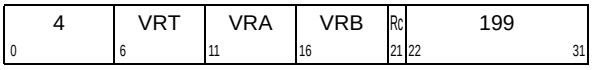
The unsigned integer value in word element i in VR[VRA] is compared to the unsigned integer value in word element i in VR[VRB]. Word element i in VR[VRT] is set to all 1s if unsigned-integer word element i in VR[VRA] is equal to unsigned-integer word element i in VR[VRB], and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 . . . . . (if Rc=1)

Vector Compare Equal Unsigned Doubleword VX-form

vcmpequd VRT,VRA,VRB (Rc=0)
vcmpequd. VRT,VRA,VRB (Rc=1)




```
do i = 0 to 1
   aop ← EXTZ(VR[VRA].dword[i])
   bop ← EXTZ(VR[VRB].dword[i])
   if (aop = bop) then do
      VR[VRT].dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
      flag.bit[i] ← 0b1
   end
   else do
      VR[VRT].dword[i] ← 0x0000_0000_0000_0000
      flag.bit[i] ← 0b0
   end
end
if Rc=1 then do
   CR.bit[24] ← (flag=0b11)
   CR.bit[25] ← 0b0
   CR.bit[26] ← (flag=0b00)
   CR.bit[27] ← 0b0
end
```

For each integer value i from 0 to 1, do the following.

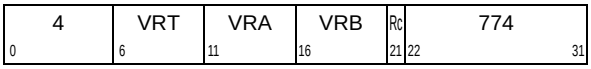
The unsigned integer value in doubleword element i of VR[VRA] is compared to the unsigned integer value in doubleword element i of VR[VRB]. Doubleword element i of VR[VRT] is set to all 1s if the unsigned integer value in doubleword element i of VR[VRA] is equal to the unsigned integer value in doubleword element i of VR[VRB], and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 . . . . . (if Rc=1)

Vector Compare Greater Than Signed Byte VC-form

vcmpgtsb VRT,VRA,VRB (Rc=0)
vcmpgtsb. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 8
   VRT_i:i+7 ← ((VRA)_i:i+7 >si (VRB)_i:i+7) ? 8^1 : 8^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 15, do the following.

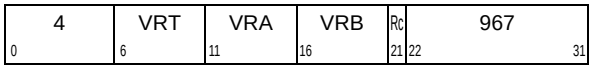
The signed integer value in byte element i in VR[VRA] is compared to the signed integer value in byte element i in VR[VRB]. Byte element i in VR[VRT] is set to all 1s if signed-integer byte element i in VR[VRA] is greater than to signed-integer byte element i in VR[VRB], and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 . . . . . (if Rc=1)

Vector Compare Greater Than Signed Doubleword VX-form

vcmpgtsd VRT,VRA,VRB (Rc=0)
vcmpgtsd. VRT,VRA,VRB (Rc=1)




```
do i = 0 to 1
   aop ← EXTS(VR[VRA].dword[i])
   bop ← EXTS(VR[VRB].dword[i])
   if (aop >si bop) then do
      VR[VRT].dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
      flag.bit[i] ← 0b1
   end
   else do
      VR[VRT].dword[i] ← 0x0000_0000_0000_0000
      flag.bit[i] ← 0b0
   end
end
if "vcmpgtsd." then do
   CR.bit[24] ← (flag=0b11)
   CR.bit[25] ← 0b0
   CR.bit[26] ← (flag=0b00)
   CR.bit[27] ← 0b0
end
```

For each integer value i from 0 to 1, do the following.

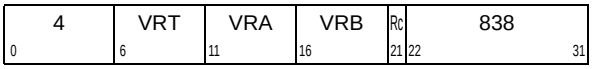
The signed integer value in doubleword element i of VR[VRA] is compared to the signed integer value in doubleword element i of VR[VRB]. Doubleword element i of VR[VRT] is set to all 1s if the signed integer value in doubleword element i of VR[VRA] is greater than the signed integer value in doubleword element i of VR[VRB], and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 . . . . . (if Rc=1)

Vector Compare Greater Than Signed Halfword VC-form

vcmpgtsh VRT,VRA,VRB (Rc=0)
vcmpgtsh. VRT,VRA,VRB (Rc=1)



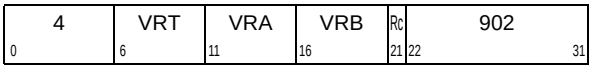

```
do i=0 to 127 by 16
   VRT_i:i+15 ← ((VRA)_i:i+15 >si (VRB)_i:i+15) ? 16^1 : 16^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 7, do the following.
Signed-integer halfword element i in VRA is compared to signed-integer halfword element i in VRB. Halfword element i in VRT is set to all 1s if signed-integer halfword element i in VRA is greater than signed-integer halfword element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:
CR field 6 . . . . . .(if Rc=1)

Vector Compare Greater Than Signed Word VC-form

vcmpgtsw VRT,VRA,VRB (Rc=0)
vcmpgtsw. VRT,VRA,VRB (Rc=1)



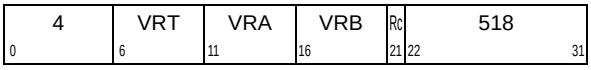

```
do i=0 to 127 by 32
   VRT_i:i+31 ← ((VRA)_i:i+31 >si (VRB)_i:i+31) ? 32^1 : 32^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 3, do the following.
Signed-integer word element i in VRA is compared to signed-integer word element i in VRB. Word element i in VRT is set to all 1s if signed-integer word element i in VRA is greater than signed-integer word element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:
CR field 6 . . . . . (if Rc=1)

Vector Compare Greater Than Unsigned Byte VC-form

vcmpgtub VRT,VRA,VRB (Rc=0)
vcmpgtub. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 8
   VRT_i:i+7 ← ((VRA)_i:i+7 >ui (VRB)_i:i+7) ? 8^1 : 8^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 15, do the following.

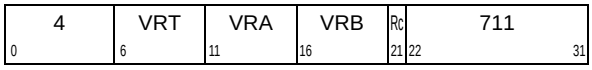
Unsigned-integer byte element i in VRA is compared to unsigned-integer byte element i in VRB. Byte element i in VRT is set to all 1s if unsigned-integer byte element i in VRA is greater than to unsigned-integer byte element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 . . . . . (if Rc=1)

Vector Compare Greater Than Unsigned Doubleword VX-form

vcmpgtud VRT,VRA,VRB (Rc=0)
vcmpgtud. VRT,VRA,VRB (Rc=1)




```
do i = 0 to 1
   aop ← EXTZ(VR[VRA].dword[i])
   bop ← EXTZ(VR[VRB].dword[i])
   if (EXTZ(aop) >ui EXTZ(bop)) then do
      VR[VRT].dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
      flag.bit[i] ← 0b1
   end
   else do
      VR[VRT].dword[i] ← 0x0000_0000_0000_0000
      flag.bit[i] ← 0b1
   end
end
if "vcmpgtud." then do
   CR.bit[24] ← (flag=0b11)
   CR.bit[25] ← 0b0
   CR.bit[26] ← (flag=0b00)
   CR.bit[27] ← 0b0
end
```

For each integer value i from 0 to 1, do the following.

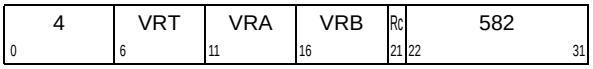
The unsigned integer value in doubleword element i of VR[VRA] is compared to the unsigned integer value in doubleword element i of VR[VRB]. Doubleword element i of VR[VRT] is set to all 1s if the unsigned integer value in doubleword element i of VR[VRA] is greater than the unsigned integer value in doubleword element i of VR[VRB], and is set to all 0s otherwise.

Special Registers Altered:

CR field 6 (if Rc=1)

Vector Compare Greater Than Unsigned Halfword VC-form

vcmpgtuh VRT,VRA,VRB (Rc=0)
vcmpgtuh. VRT,VRA,VRB (Rc=1)



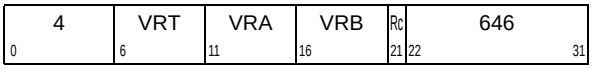

```
do i=0 to 127 by 16
   VRT_{i:i+15} ← ((VRA)_{i:i+15} >ui (VRB)_{i:i+15}) ? 16^1 : 16^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 7, do the following.
Unsigned-integer halfword element i in VRA is compared to unsigned-integer halfword element i in VRB. Halfword element i in VRT is set to all 1s if unsigned-integer halfword element i in VRA is greater than to unsigned-integer halfword element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:
CR field 6 . . . . . .(if Rc=1)

Vector Compare Greater Than Unsigned Word VC-form

vcmpgtuw VRT,VRA,VRB (Rc=0)
vcmpgtuw. VRT,VRA,VRB (Rc=1)



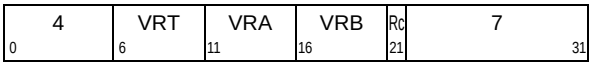

```
do i=0 to 127 by 32
   VRT_{i:i+31} ← ((VRA)_{i:i+31} >ui (VRB)_{i:i+31}) ? 32^1 : 32^0
end
if Rc=1 then do
   t ← (VRT=128^1)
   f ← (VRT=128^0)
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 3, do the following.
Unsigned-integer word element i in VRA is compared to unsigned-integer word element i in VRB. Word element i in VRT is set to all 1s if unsigned-integer word element i in VRA is greater than to unsigned-integer word element i in VRB, and is set to all 0s otherwise.

Special Registers Altered:
CR field 6 . . . . . (if Rc=1)

Vector Compare Not Equal Byte VX-form

vcmpneb VRT,VRA,VRB (if Rc=0)
vcmpneb. VRT,VRA,VRB (if Rc=1)




```
if MSR.VEC=0 then Vector_Unavailable()

for i = 0 to 15
   src1 ← VR[VRA].byte[i]
   src2 ← VR[VRB].byte[i]
   if (src1 != src2) then
      VR[VRT].byte[i] ← 0xFF
   else
      VR[VRT].byte[i] ← 0x00
end

all_true  ← (VR[VRT]=0xFFFF_FFFF_FFFF_FFFF_FFF_FFFF_FFFF_FFFF)
all_false ← (VR[VRT]=0x0000_0000_0000_0000_0000_0000_0000_0000)

if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)
```

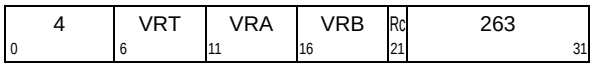
For each integer value i from 0 to 15, do the following.
The integer value in byte element i in VR[VRA] is compared to the integer value in byte element i in VR[VRB]. The contents of byte element i in VR[VRT] are set to 0xFF if integer value in byte element i in VR[VRA] is not equal to the integer value in byte element i in VR[VRB], and are set to 0x00 otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:
CR field 6 (if Rc=1)

Vector Compare Not Equal or Zero Byte VX-form

vcmpnezb VRT,VRA,VRB (if Rc=0)
vcmpnezb. VRT,VRA,VRB (if Rc=1)




```
if MSR.VEC=0 then Vector_Unavailable()

for i = 0 to 15
   src1 ← VR[VRA].byte[i]
   src2 ← VR[VRB].byte[i]
   if (src1 = 0) | (src2 = 0) | (src1 != src2) then
      VR[VRT].byte[i] ← 0xFF
   else
      VR[VRT].byte[i] ← 0x00
end

all_true  ← (VR[VRT]=0xFFFF_FFFF_FFFF_FFFF_FFF_FFFF_FFFF_FFFF)
all_false ← (VR[VRT]=0x0000_0000_0000_0000_0000_0000_0000_0000)

if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)
```

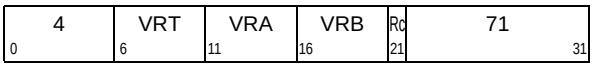
For each integer value i from 0 to 15, do the following.
The integer value in byte element i in VR[VRA] is compared to the integer value in byte element i in VR[VRB]. The contents of byte element i in VR[VRT] are set to 0xFF if integer value in byte element i in VR[VRA] is not equal to the integer value in byte element i in VR[VRB] or either value is equal to 0x00, and are set to 0x00 otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:
CR field 6 (if Rc=1)

Vector Compare Not Equal Halfword VX-form

vcmpneh VRT,VRA,VRB (if Rc=0)
vcmpneh. VRT,VRA,VRB (if Rc=1)




```
if MSR.VEC=0 then Vector_Unavailable()

for i = 0 to 7
   src1 ← VR[VRA].hword[i]
   src2 ← VR[VRB].hword[i]
   if (src1 != src2) then
      VR[VRT].hword[i] ← 0xFFFF
   else
      VR[VRT].hword[i] ← 0x0000
end

all_true  ← (VR[VRT]=0xFFFF_FFFF_FFFF_FFFF_FFF_FFFF_FFFF_FFFF)
all_false ← (VR[VRT]=0x0000_0000_0000_0000_0000_0000_0000_0000)

if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)
```

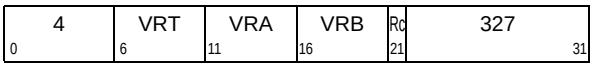
For each integer value i from 0 to 7, do the following.
The integer value in halfword element i in VR[VRA] is compared to the integer value in halfword element i in VR[VRB]. The contents of halfword element i in VR[VRT] are set to 0xFFFF if integer value in halfword element i in VR[VRA] is not equal to the integer value in halfword element i in VR[VRB], and are set to 0x0000 otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:
CR field 6 (if Rc=1)

Vector Compare Not Equal or Zero Halfword VX-form

vcmpnezh VRT,VRA,VRB (if Rc=0)
vcmpnezh. VRT,VRA,VRB (if Rc=1)




```
if MSR.VEC=0 then Vector_Unavailable()

for i = 0 to 7
   src1 ← VR[VRA].hword[i]
   src2 ← VR[VRB].hword[i]
   if (src1=0) | (src2=0) | (src1≠src2) then
      VR[VRT].hword[i] ← 0xFFFF
   else
      VR[VRT].hword[i] ← 0x0000
end

all_true  ← (VR[VRT]=0xFFFF_FFFF_FFFF_FFFF_FFF_FFFF_FFFF_FFFF)
all_false ← (VR[VRT]=0x0000_0000_0000_0000_0000_0000_0000_0000)

if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)
```

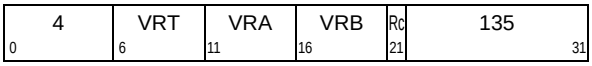
For each integer value i from 0 to 7, do the following.
The integer value in halfword element i in VR[VRA] is compared to the integer value in halfword element i in VR[VRB]. The contents of halfword element i in VR[VRT] are set to 0xFFFF if integer value in halfword element i in VR[VRA] is not equal to the integer value in halfword element i in VR[VRB] or either value is equal to 0x0000, and are set to 0x0000 otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:
CR field 6 (if Rc=1)

Vector Compare Not Equal Word VX-form

vcmpnew VRT,VRA,VRB (if Rc=0)
vcmpnew. VRT,VRA,VRB (if Rc=1)




```
if MSR.VEC=0 then Vector_Unavailable()

for i = 0 to 3
   src1 ← VR[VRA].word[i]
   src2 ← VR[VRB].word[i]
   if (src1 != src2) then
      VR[VRT].word[i] ← 0xFFFF_FFFF
   else
      VR[VRT].word[i] ← 0x0000_0000
end

all_true  ← (VR[VRT]=0xFFFF_FFFF_FFFF_FFFF_FFF_FFFF_FFFF_FFFF)
all_false ← (VR[VRT]=0x0000_0000_0000_0000_0000_0000_0000_0000)

if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)
```

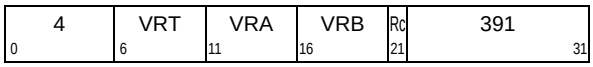
For each integer value i from 0 to 3, do the following.
The integer value in word element i in VR[VRA] is compared to the integer value in word element i in VR[VRB]. The contents of word element i in VR[VRT] are set to 0xFFFF_FFFF if integer value in word element i in VR[VRA] is not equal to the integer value in word element i in VR[VRB], and are set to 0x0000_0000 otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:
CR field 6 (if Rc=1)

Vector Compare Not Equal or Zero Word VX-form

vcmpnezw VRT,VRA,VRB (if Rc=0)
vcmpnezw. VRT,VRA,VRB (if Rc=1)




```
if MSR.VEC=0 then Vector_Unavailable()

for i = 0 to 3
   src1 ← VR[VRA].word[i]
   src2 ← VR[VRB].word[i]
   if (src1=0) | (src2=0) | (src1≠src2) then
      VR[VRT].word[i] ← 0xFFFF_FFFF
   else
      VR[VRT].word[i] ← 0x0000_0000
end

all_true  ← (VR[VRT]=0xFFFF_FFFF_FFFF_FFFF_FFF_FFFF_FFFF_FFFF)
all_false ← (VR[VRT]=0x0000_0000_0000_0000_0000_0000_0000_0000)

if Rc=1 then CR.bit[56:59] ← (all_true<<3) + (all_false<<1)
```

For each integer value i from 0 to 3, do the following.
The integer value in word element i in VR[VRA] is compared to the integer value in word element i in VR[VRB]. The contents of word element i in VR[VRT] are set to 0xFFFF_FFFF if integer value in word element i in VR[VRA] is not equal to the integer value in word element i in VR[VRB] or either value is equal to 0x0000_0000, and are set to 0x0000_0000 otherwise.

If Rc=1, CR field 6 is set to indicate whether all vector elements compared true and whether all vector elements compared false.

Special Registers Altered:
CR field 6 (if Rc=1)

6.9.4 Vector Logical Instructions

Extended mnemonics for vector logical operations

Extended mnemonics are provided that use the Vector OR and Vector NOR instructions to copy the contents of one Vector Register to another, with and without complementing. These are shown as examples with the two instructions.

Vector Move Register

Several vector instructions can be coded in a way such that they simply copy the contents of one Vector Register to another. An extended mnemonic is provided to convey the idea that no computation is being performed but merely data movement (from one register to another).

The following instruction copies the contents of register Vy to register Vx.

vmr Vx,Vy (equivalent to: vor Vx,Vy,Vy)

Vector Complement Register

The *Vector NOR* instruction can be coded in a way such that it complements the contents of one Vector Register and places the result into another Vector Register. An extended mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of register Vy and places the result into register Vx.

vnot Vx,Vy (equivalent to: vnor Vx,Vy,Vy)

Vector Logical AND VX-form

vand VRT,VRA,VRB

4	VRT	VRA	VRB	1028
0	6	11	16	21 31

```
VR[VRT] ← VR[VRA] & VR[VRB]
```

The contents of VR[VRA] are ANDed with the contents of VR[VRB] and the result is placed into VR[VRT].

Special Registers Altered:

None

Vector Logical AND with Complement VX-form

vandc VRT,VRA,VRB

4	VRT	VRA	VRB	1092
0	6	11	16	21 31

```
VR[VRT] ← VR[VRA] & ¬VR[VRB]
```

The contents of VR[VRA] are ANDed with the complement of the contents of VR[VRB] and the result is placed into VR[VRT].

Special Registers Altered:

None

Vector Logical Equivalence VX-form

veqv VRT,VRA,VRB

4	VRT	VRA	VRB	1668
0	6	11	16	21 31

```
VR[VRT] ← VR[VRA] h VR[VRB]
```

The contents of VR[VRA] are XORed with the contents of VR[VRB] and the complemented result is placed into VR[VRT].

Special Registers Altered:

None

Vector Logical NAND VX-form

vnand VRT,VRA,VRB

4	VRT	VRA	VRB	1412
0	6	11	16	21 31

```
if MSR.VEC=0 then VECTOR_UNAVAILABLE()

VR[VRT] ← ¬( VR[VRA] & VR[VRB] )
```

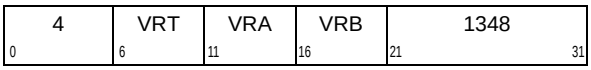
The contents of VR[VRA] are ANDed with the contents of VR[VRB] and the complemented result is placed into VR[VRT].

Special Registers Altered:

None

Vector Logical OR with Complement VX-form

vorc VRT,VRA,VRB




```
VR[VRT] ← VR[VRA] | ¬VR[VRB]
```

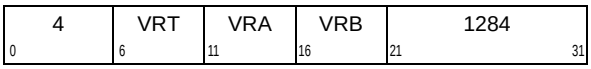
The contents of VR[VRA] are ORed with the complement of the contents of VR[VRB] and the result is placed into VR[VRT].

Special Registers Altered:

None

Vector Logical NOR VX-form

vnor VRT,VRA,VRB




```
VR[VRT] ← ¬( VR[VRA] | VR[VRB] )
```

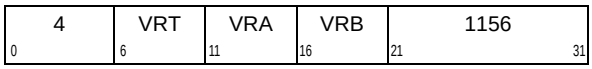
The contents of VR[VRA] are ORed with the contents of VR[VRB] and the complemented result is placed into VR[VRT].

Special Registers Altered:

None

Vector Logical OR VX-form

vor VRT,VRA,VRB




```
VR[VRT] ← VR[VRA] | VR[VRB]
```

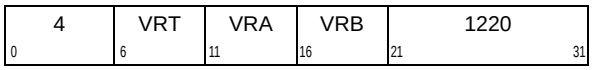
The contents of VR[VRA] are ORed with the contents of VR[VRB] and the result is placed into VR[VRT].

Special Registers Altered:

None

Vector Logical XOR VX-form

vxor VRT,VRA,VRB




```
VR[VRT] ← VR[VRA] ⊕ VR[VRB]
```

The contents of VR[VRA] are XORed with the contents of VR[VRB] and the result is placed into VR[VRT].

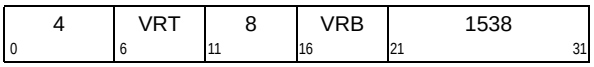
Special Registers Altered:

None

6.9.5 Vector Parity Byte Instructions

Vector Parity Byte Word VX-form

vprtybw VRT,VRB



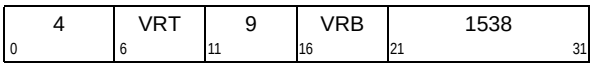

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   s ← 0
   do j = 0 to 3
      s ← s ^ VR[VRB].word[i].byte[j].bit[7]
   end
   VR[VRT].word[i] ← Chop(EXTZ(s), 32)
end
```

For each integer value i from 0 to 3, do the following
If the sum of the least significant bit in each byte sub-element of word element i of VR[VRB] is odd, the value 1 is placed into word element i of VR[VRT]; otherwise the value 0 is placed into word element i of VR[VRT].

Special Registers Altered:
None

Vector Parity Byte Doubleword VX-form

vprtybd VRT,VRB



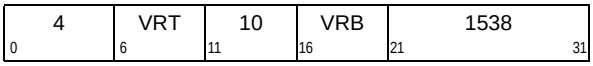

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   s ← 0
   do j = 0 to 7
      s ← s ^ VR[VRB]dword[i].byte[j].bit[7]
   end
   VR[VRT].dword[i] ← Chop(EXTZ(s), 64)
end
```

For each integer value i from 0 to 1, do the following
If the sum of the least significant bit in each byte sub-element of doubleword element i of VR[VRB] is odd, the value 1 is placed into doubleword element i of VR[VRT]; otherwise the value 0 is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Parity Byte Quadword VX-form

vprtybq VRT,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

s ← 0
do j = 0 to 15
   s ← s ^ VR[VRB].byte[j].bit[7]
end
VR[VRT] ← Chop(EXTZ(s), 128)
```

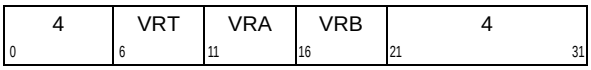
If the sum of the least significant bit in each byte element of VR[VRB] is odd, the value 1 is placed into VR[VRT]; otherwise the value 0 is placed into VR[VRT].

Special Registers Altered:
None

6.9.6 Vector Integer Rotate and Shift Instructions

Vector Rotate Left Byte VX-form

vrlb VRT,VRA,VRB




```
do i=0 to 127 by 8
   sh ← (VRB)i+5:i+7
   VRTi:i+7 ← (VRA)i:i+7 <<< sh
end
```

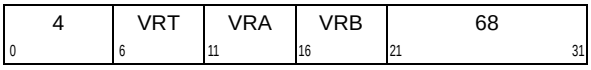
For each integer value i from 0 to 15, do the following.
Byte element i in VRA is rotated left by the number of bits specified in the low-order 3 bits of the corresponding byte element i in VRB.

The result is placed into byte element i in VRT.

Special Registers Altered:
None

Vector Rotate Left Halfword VX-form

vrlh VRT,VRA,VRB




```
do i=0 to 127 by 16
   sh ← (VRB)i+12:i+15
   VRTi:i+15 ← (VRA)i:i+15 <<< sh
end
```

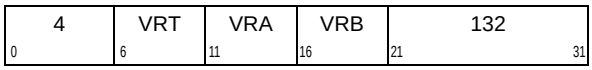
For each integer value i from 0 to 7, do the following.
Halfword element i in VRA is rotated left by the number of bits specified in the low-order 4 bits of the corresponding halfword element i in VRB.

The result is placed into halfword element i in VRT.

Special Registers Altered:
None

Vector Rotate Left Word VX-form

vrlw VRT,VRA,VRB




```
do i=0 to 127 by 32
   sh ← (VRB)i+27:i+31
   VRTi:i+31 ← (VRA)i:i+31 <<< sh
end
```

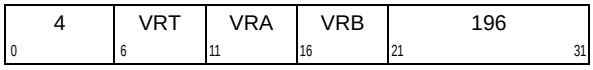
For each integer value i from 0 to 3, do the following.
Word element i in VRA is rotated left by the number of bits specified in the low-order 5 bits of the corresponding word element i in VRB.

The result is placed into word element i in VRT.

Special Registers Altered:
None

Vector Rotate Left Doubleword VX-form

vrld VRT,VRA,VRB




```
do i = 0 to 1
   sh ← VR[VRB].dword[i].bit[58:63]
   VR[VRT].dword[i] ← VR[VRA].dword[i] <<< sh
end
```

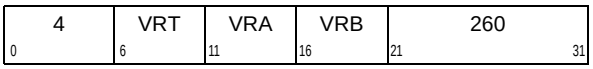
For each integer value i from 0 to 1, do the following.
The contents of doubleword element i of VR[VRA] are rotated left by the number of bits specified in bits 58:63 of doubleword element i of VR[VRB].

The result is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Shift Left Byte VX-form

vslb VRT,VRA,VRB




```
do i=0 to 127 by 8
   sh ← (VRB)i+5:i+7
   VRTi:i+7 ← (VRA)i:i+7 << sh
end
```

For each integer value i from 0 to 15, do the following.
Byte element i in VRA is shifted left by the number of bits specified in the low-order 3 bits of byte element i in VRB.

– Bits shifted out of bit 0 are lost.

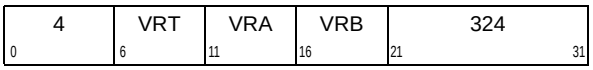
– Zeros are supplied to the vacated bits on the right.

The result is placed into byte element i of VRT.

Special Registers Altered:
None

Vector Shift Left Halfword VX-form

vslh VRT,VRA,VRB




```
do i=0 to 127 by 16
   sh ← (VRB)i+12:i+15
   VRTi:i+15 ← (VRA)i:i+15 << sh
end
```

For each integer value i from 0 to 7, do the following.
Halfword element i in VRA is shifted left by the number of bits specified in the low-order 4 bits of halfword element i in VRB.

– Bits shifted out of bit 0 are lost.

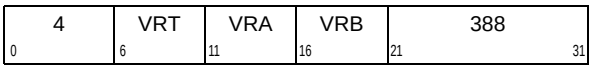
– Zeros are supplied to the vacated bits on the right.

The result is placed into halfword element i of VRT.

Special Registers Altered:
None

Vector Shift Left Word VX-form

vslw VRT,VRA,VRB




```
do i=0 to 127 by 32
   sh ← (VRB)i+27:i+31
   VRTi:i+31 ← (VRA)i:i+31 << sh
end
```

For each integer value i from 0 to 3, do the following.
Word element i in VRA is shifted left by the number of bits specified in the low-order 5 bits of word element i in VRB.

– Bits shifted out of bit 0 are lost.

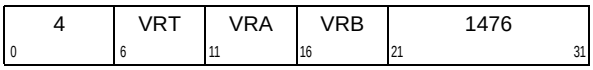
– Zeros are supplied to the vacated bits on the right.

The result is placed into word element i of VRT.

Special Registers Altered:
None

Vector Shift Left Doubleword VX-form

vsld VRT,VRA,VRB




```
do i = 0 to 1
   sh ← VR[VRB].dword[i].bit[58:63]
   VR[VRT].dword[i] ← VR[VRA].dword[i] << sh
end
```

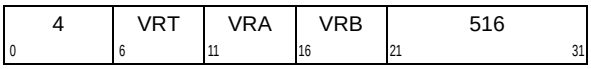
For each integer value i from 0 to 1, do the following.
The contents of doubleword element i of VR[VRA] are shifted left by the number of bits specified in bits 58:63 of doubleword element i of VR[VRB].
– Bits shifted out of bit 0 are lost.
– Zeros are supplied to the vacated bits on the right.

The result is placed into doubleword element i of VR[VRT].

Special Registers Altered:
None

Vector Shift Right Byte VX-form

vsrb VRT,VRA,VRB




```
do i=0 to 127 by 8
   sh ← (VRB)i+5:i+7
   VRTi:i+7 ← (VRA)i:i+7 >>ui sh
end
```

For each integer value i from 0 to 15, do the following.

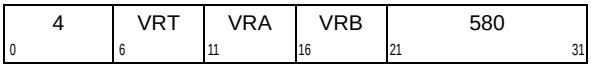
Byte element i in VRA is shifted right by the number of bits specified in the low-order 3 bits of byte element i in VRB. Bits shifted out of the least-significant bit are lost. Zeros are supplied to the vacated bits on the left. The result is placed into byte element i of VRT.

Special Registers Altered:

None

Vector Shift Right Halfword VX-form

vsrh VRT,VRA,VRB




```
do i=0 to 127 by 16
   sh ← (VRB)i+12:i+15
   VRTi:i+15 ← (VRA)i:i+15 >>ui sh
end
```

For each integer value i from 0 to 7, do the following.

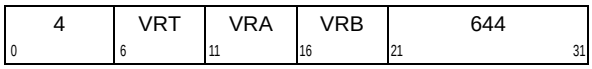
Halfword element i in VRA is shifted right by the number of bits specified in the low-order 4 bits of halfword element i in VRB. Bits shifted out of the least-significant bit are lost. Zeros are supplied to the vacated bits on the left. The result is placed into halfword element i of VRT.

Special Registers Altered:

None

Vector Shift Right Word VX-form

vsrw VRT,VRA,VRB




```
do i=0 to 127 by 32
   sh ← (VRB)i+27:i+31
   VRTi:i+31 ← (VRA)i:i+31 >>ui sh
end
```

For each integer value i from 0 to 3, do the following.

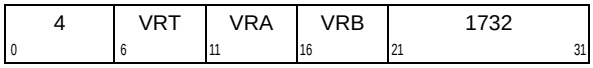
Word element i in VRA is shifted right by the number of bits specified in the low-order 5 bits of word element i in VRB. Bits shifted out of the least-significant bit are lost. Zeros are supplied to the vacated bits on the left. The result is placed into word element i of VRT.

Special Registers Altered:

None

Vector Shift Right Doubleword VX-form

vsrd VRT,VRA,VRB




```
do i = 0 to 1
   sh ← VR[VRB].dword[i].bit[58:63]
   VR[VRT].dword[i] ← VR[VRA].dword[i] >>ui sh
end
```

For each integer value i from 0 to 1, do the following.

The contents of doubleword element i of VR[VRA] are shifted right by the number of bits specified in bits 58:63 of doubleword element i of VR[VRB]. Zeros are supplied to the vacated bits on the left.

The result is placed into doubleword element i of VR[VRT].

Special Registers Altered:

None

Vector Shift Right Algebraic Byte VX-form

vsrab VRT,VRA,VRB

4	VRT	VRA	VRB	772
0	6	11	16	21 31

```
do i=0 to 127 by 8
   sh ← (VRB)i+5:i+7
   VRTi:i+7 ← (VRA)i:i+7 >>si sh
end
```

For each integer value i from 0 to 15, do the following.

Byte element i in VRA is shifted right by the number of bits specified in the low-order 3 bits of the corresponding byte element i in VRB. Bits shifted out of bit 7 of the byte element are lost. Bit 0 of the byte element is replicated to fill the vacated bits on the left. The result is placed into byte element i of VRT.

Special Registers Altered:

None

Vector Shift Right Algebraic Halfword VX-form

vsrah VRT,VRA,VRB

4	VRT	VRA	VRB	836
0	6	11	16	21 31

```
do i=0 to 127 by 16
   sh ← (VRB)i+12:i+15
   VRTi:i+15 ← (VRA)i:i+15 >>si sh
end
```

For each integer value i from 0 to 7, do the following.

Halfword element i in VRA is shifted right by the number of bits specified in the low-order 4 bits of the corresponding halfword element i in VRB. Bits shifted out of bit 15 of the halfword are lost. Bit 0 of the halfword is replicated to fill the vacated bits on the left. The result is placed into halfword element i of VRT.

Special Registers Altered:

None

Vector Shift Right Algebraic Word VX-form

vsraw VRT,VRA,VRB

4	VRT	VRA	VRB	900
0	6	11	16	21 31

```
do i=0 to 127 by 32
   sh ← (VRB)i+27:i+31
   VRTi:i+31 ← (VRA)i:i+31 >>si sh
end
```

For each integer value i from 0 to 3, do the following.

Word element i in VRA is shifted right by the number of bits specified in the low-order 5 bits of the corresponding word element i in VRB. Bits shifted out of bit 31 of the word are lost. Bit 0 of the word is replicated to fill the vacated bits on the left. The result is placed into word element i of VRT.

Special Registers Altered:

None

Vector Shift Right Algebraic Doubleword VX-form

vsrad VRT,VRA,VRB

4	VRT	VRA	VRB	964
0	6	11	16	21 31

```
do i = 0 to 1
   sh ← VR[VRB].dword[i].bit[58:63]
   VR[VRT].dword[i] ← VR[VRA].dword[i] >>si sh
end
```

For each integer value i from 0 to 1, do the following.

The contents of doubleword element i of VR[VRA] are shifted right by the number of bits specified in bits 58:63 of doubleword element i of VR[VRB]. Bit 0 of doubleword element i of VR[VRA] is replicated to fill the vacated bits on the left.

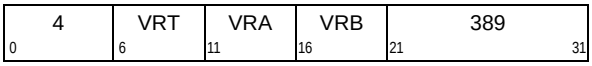
The result is placed into doubleword element i of VR[VRT].

Special Registers Altered:

None

Vector Rotate Left Word then AND with Mask VX-form

vrlwnm VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1.word[0] ← VR[VRA].word[i]
   src1.word[1] ← VR[VRA].word[i]
   src2 ← VR[VRB].word[i]

   b ← src2.bit[11:15]
   e ← src2.bit[19:23]
   n ← src2.bit[27:31]
   r ← src1.bit[n:n+31]
   m ← MASK(b, e)

   VR[VRT].word[i] ← r & m
end
```

For each integer value i from 0 to 3, do the following.

Let src1 be the contents of word element i of VR[VRA].

Let src2 be the contents of word element i of VR[VRB].

Let mb be the contents of bits 11:15 of src2.
Let me be the contents of bits 19:23 of src2.
Let sh be the contents of bits 27:31 of src2.

src1 is rotated left sh bits. A mask is generated having 1-bits from bit mb through bit me and 0-bits elsewhere. The rotated data are ANDed with the generated mask.

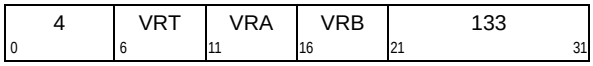
The result is placed into word element i of VR[VRT].

Special Registers Altered:

None

Vector Rotate Left Word then Mask Insert VX-form

vrlwmi VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   src1.word[0] ← VR[VRA].word[i]
   src1.word[1] ← VR[VRA].word[i]
   src2 ← VR[VRB].word[i]
   src3 ← VR[VRT].word[i]

   b ← src2.bit[11:15]
   e ← src2.bit[19:23]
   n ← src2.bit[27:31]
   r ← src1.bit[n:n+31]
   m ← MASK(b, e)

   VR[VRT].word[i] ← (r & m) | (src3 & ¬m)
end
```

For each integer value i from 0 to 3, do the following.

Let src1 be the contents of word element i of VR[VRA].

Let src2 be the contents of word element i of VR[VRB].

Let src3 be the contents of word element i of VR[VRT].

Let mb be the contents of bits 11:15 of src2.
Let me be the contents of bits 19:23 of src2.
Let sh be the contents of bits 27:31 of src2.

src1 is rotated left sh bits. A mask is generated having 1-bits from bit mb through bit me and 0-bits elsewhere. The rotated data are inserted into src3 under control of the generated mask.

The result is placed into word element i of VR[VRT].

Special Registers Altered:

None

Vector Rotate Left Doubleword then AND with Mask VX-form

vrldnm VRT,VRA,VRB

4	VRT	VRA	VRB	453
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1.dword[0] ← VR[VRA].dword[i]
   src1.dword[1] ← VR[VRA].dword[i]
   src2 ← VR[VRB].dword[i]

   b ← src2.bit[42:47]
   e ← src2.bit[50:55]
   n ← src2.bit[58:63]
   r ← src1.bit[n:n+63]
   m ← MASK(b, e)

   VR[VRT].dword[i] ← r & m
end
```

For each integer value `i` from 0 to 1, do the following.

Let `src1` be the contents of doubleword element `i` of `VR[VRA]`.

Let `src2` be the contents of doubleword element `i` of `VR[VRB]`.

Let `mb` be the contents of bits `42:47` of `src2`.
Let `me` be the contents of bits `50:55` of `src2`.
Let `sh` be the contents of bits `58:63` of `src2`.

`src1` is rotated left `sh` bits. A mask is generated having 1-bits from bit `mb` through bit `me` and 0-bits elsewhere. The rotated data are ANDed with the generated mask.

The result is placed into doubleword element `i` of `VR[VRT]`.

Special Registers Altered:

None

Vector Rotate Left Doubleword then Mask Insert VX-form

vrldmi VRT,VRA,VRB

4	VRT	VRA	VRB	197
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   src1.dword[0] ← VR[VRA].dword[i]
   src1.dword[1] ← VR[VRA].dword[i]
   src2 ← VR[VRB].dword[i]
   src3 ← VR[VRT].dword[i]

   b ← src2.bit[42:47]
   e ← src2.bit[50:55]
   n ← src2.bit[58:63]
   r ← src1.bit[n:n+63]
   m ← MASK(b, e)

   VR[VRT].dword[i] ← (r & m) | (src3 & ¬m)
end
```

For each integer value `i` from 0 to 1, do the following.

Let `src1` be the contents of doubleword element `i` of `VR[VRA]`.

Let `src2` be the contents of doubleword element `i` of `VR[VRB]`.

Let `src3` be the contents of doubleword element `i` of `VR[VRT]`.

Let `mb` be the contents of bits `42:47` of `src2`.
Let `me` be the contents of bits `50:55` of `src2`.
Let `sh` be the contents of bits `58:63` of `src2`.

`src1` is rotated left `sh` bits. A mask is generated having 1-bits from bit `mb` through bit `me` and 0-bits elsewhere. The rotated data are inserted into `src3` under control of the generated mask.

The result is placed into doubleword element `i` of `VR[VRT]`.

Special Registers Altered:

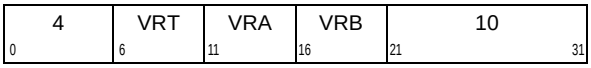
None

6.10 Vector Floating-Point Instruction Set

6.10.1 Vector Floating-Point Arithmetic Instructions

Vector Add Floating-Point VX-form

vaddfp VRT,VRA,VRB



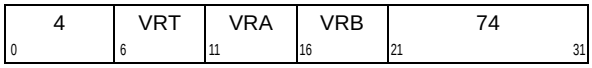

```
do i=0 to 127 by 32
   VRT(i:i+31) ← RoundToNearSP((VRA)(i:i+31) +fp (VRB)(i:i+31))
end
```

For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRA is added to single-precision floating-point element i in VRB. The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element i of VRT.

Special Registers Altered:
None

Vector Subtract Floating-Point VX-form

vsubfp VRT,VRA,VRB



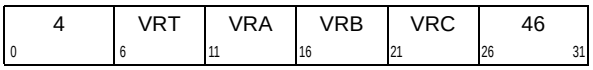

```
do i=0 to 127 by 32
   VRT(i:i+31) ← RoundToNearSP((VRA)(i:i+31) -fp (VRB)(i:i+31))
end
```

For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRB is subtracted from single-precision floating-point element i in VRA. The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element i of VRT.

Special Registers Altered:
None

Vector Multiply-Add Floating-Point VA-form

vmaddfp VRT,VRA,VRC,VRB




```
do i=0 to 127 by 32
   prod ← (VRA)i:i+31 ×fp (VRC)i:i+31
   VRTi:i+31 ← RoundToNearSP( prod +fp (VRB)i:i+31 )
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point element i in VRA is multiplied by single-precision floating-point element i in VRC. Single-precision floating-point element i in VRB is added to the infinitely-precise product. The intermediate result is rounded to the nearest single-precision floating-point number and placed into word element i of VRT.

Special Registers Altered:

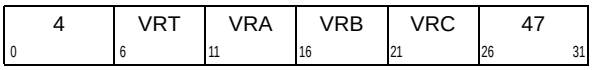
None

Programming Note

To use a multiply-add to perform an IEEE or Java compliant multiply, the addend must be -0.0. This is necessary to insure that the sign of a zero result will be correct when the product is -0.0 (+0.0 + -0.0 ≥ +0.0, and -0.0 + -0.0≥ -0.0). When the sign of a resulting 0.0 is not important, then +0.0 can be used as an addend which may, in some cases, avoid the need for a second register to hold a -0.0 in addition to the integer 0/floating-point +0.0 that may already be available.

Vector Negative Multiply-Subtract Floating-Point VA-form

vnmsubfp VRT,VRA,VRC,VRB




```
do i=0 to 127 by 32
   prod0:inf ← (VRA)i:i+31 ×fp (VRC)i:i+31
   VRTi:i+31 ← -RoundToNearSP(prod0:inf -fp (VRB)i:i+31)
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point element i in VRA is multiplied by single-precision floating-point element i in VRC. Single-precision floating-point element i in VRB is subtracted from the infinitely-precise product. The intermediate result is rounded to the nearest single-precision floating-point number, then negated and placed into word element i of VRT.

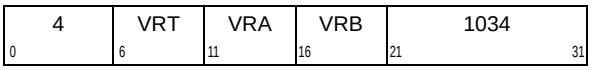
Special Registers Altered:

None

6.10.2 Vector Floating-Point Maximum and Minimum Instructions

Vector Maximum Floating-Point VX-form

vmaxfp VRT,VRA,VRB




```
do i=0 to 127 by 32
   gt_flag ← ( (VRA)i:i+31 >fp (VRB)i:i+31 )
   VRTi:i+31 ← gt_flag ? (VRA)i:i+31 : (VRB)i:i+31
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point element i in VRA is compared to single-precision floating-point element i in VRB. The larger of the two values is placed into word element i of VRT.

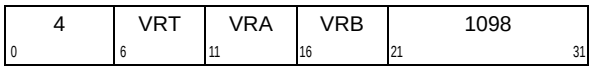
The maximum of +0 and -0 is +0. The maximum of any value and a NaN is a QNaN.

Special Registers Altered:

None

Vector Minimum Floating-Point VX-form

vminfp VRT,VRA,VRB




```
do i=0 to 127 by 32
   lt_flag ← ( (VRA)i:i+31 <fp (VRB)i:i+31 )
   VRTi:i+31 ← lt_flag ? (VRA)i:i+31 : (VRB)i:i+31
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point element i in VRA is compared to single-precision floating-point element i in VRB. The smaller of the two values is placed into word element i of VRT.

The minimum of +0 and -0 is -0. The minimum of any value and a NaN is a QNaN.

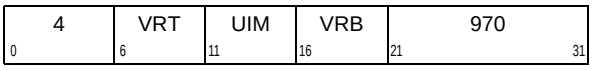
Special Registers Altered:

None

6.10.3 Vector Floating-Point Rounding and Conversion Instructions

Vector Convert with truncate Floating-Point To Signed Word format Saturate VX-form

vctsxs VRT,VRB,UIM




```
do i=0 to 127 by 32
   VRT_i:i+31 ← ConvertSPtoSXWsaturate((VRB)_i:i+31, UIM)
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point word element i in VRB is multiplied by 2^{UIM}. The product is converted to a 32-bit signed fixed-point integer using the rounding mode Round toward Zero.

- If the intermediate result is greater than $2^{31}-1$ the result saturates to $2^{31}-1$.
- If the intermediate result is less than -2^{31} the result saturates to -2^{31}.

The result is placed into word element i of VRT.

Special Registers Altered:

SAT

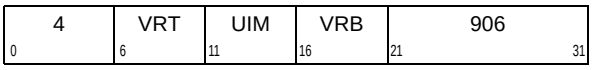
Extended Mnemonics:

Example of an extended mnemonics for *Vector Convert to Signed Fixed-Point Word Saturate*:

Extended:		Equivalent to:	
vcfpsxws	VRT,VRB,UIM	vctsxs	VRT,VRB,UIM

Vector Convert with truncate Floating-Point To Unsigned Word format Saturate VX-form

vctuxs VRT,VRB,UIM




```
do i=0 to 127 by 32
   VRT_i:i+31 ← ConvertSPtoUXWsaturate((VRB)_i:i+31, UIM)
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point word element i in VRB is multiplied by 2^{UIM}. The product is converted to a 32-bit unsigned fixed-point integer using the rounding mode Round toward Zero.

- If the intermediate result is greater than $2^{32}-1$ the result saturates to $2^{32}-1$.

The result is placed into word element *i* of VRT.

Special Registers Altered:

SAT

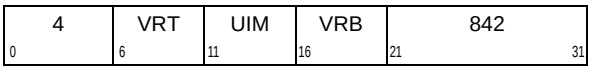
Extended Mnemonics:

Example of an extended mnemonics for *Vector Convert to Unsigned Fixed-Point Word Saturate*:

Extended:		Equivalent to:	
vcfpuxws	VRT,VRB,UIM	vctuxs	VRT,VRB,UIM

Vector Convert with round to nearest Signed Word format VX-form

vcfsx VRT,VRB,UIM




```
do i=0 to 127 by 32
   VRT[i:i+31] ← ConvertSXWtoSP( (VRB)[i:i+31] ) ÷fp 2^UIM
end
```

For each integer value i from 0 to 3, do the following.
Signed fixed-point word element i in VRB is converted to the nearest single-precision floating-point value. Each result is divided by 2^{UIM} and placed into word element *i* of VRT.

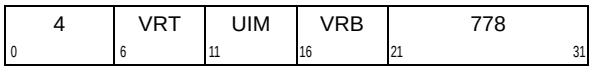
Special Registers Altered:
None

Extended Mnemonics:
Examples of extended mnemonics for *Vector Convert from Signed Fixed-Point Word*

Extended:		Equivalent to:	
vcsxwfp	VRT,VRB,UIM	vcfsx	VRT,VRB,UIM

Vector Convert with round to nearest Unsigned Word format VX-form

vcfux VRT,VRB,UIM




```
do i=0 to 127 by 32
   VRT[i:i+31] ← ConvertUXWtoSP( (VRB)[i:i+31] ) ÷fp 2^UIM
end
```

For each integer value i from 0 to 3, do the following.
Unsigned fixed-point word element i in VRB is converted to the nearest single-precision floating-point value. The result is divided by 2^{UIM} and placed into word element *i* of VRT.

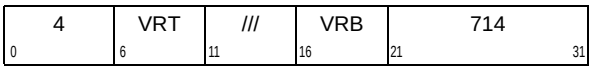
Special Registers Altered:
None

Extended Mnemonics:
Examples of extended mnemonics for *Vector Convert from Unsigned Fixed-Point Word*

Extended:		Equivalent to:	
vcuxwfp	VRT,VRB,UIM	vcfux	VRT,VRB,UIM

Vector Round to Floating-Point Integer toward -Infinity VX-form

vrfim VRT,VRB




```
do i=0 to 127 by 32
   VRT0:31 ← RoundToSPIntFloor( (VRB)0:31 )
end
```

For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRB is rounded to a single-precision floating-point integer using the rounding mode Round toward -Infinity.

The result is placed into the corresponding word element i of VRT.

Special Registers Altered:
None

Programming Note

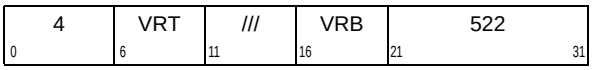
The *Vector Convert To Fixed-Point Word* instructions support only the rounding mode Round toward Zero. A floating-point number can be converted to a fixed-point integer using any of the other three rounding modes by executing the appropriate *Vector Round to Floating-Point Integer* instruction before the *Vector Convert To Fixed-Point Word* instruction.

Programming Note

The fixed-point integers used by the *Vector Convert* instructions can be interpreted as consisting of 32-UIM integer bits followed by UIM fraction bits.

Vector Round to Floating-Point Integer Nearest VX-form

vrfin VRT,VRB




```
do i=0 to 127 by 32
   VRT0:31 ← RoundToSPIntNear( (VRB)0:31 )
end
```

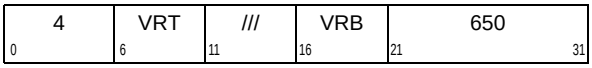
For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRB is rounded to a single-precision floating-point integer using the rounding mode Round to Nearest.

The result is placed into the corresponding word element i of VRT.

Special Registers Altered:
None

Vector Round to Floating-Point Integer toward +Infinity VX-form

vrfip VRT,VRB




```
do i=0 to 127 by 32
   VRT0:31 ← RoundToSPIntCeil( (VRB)0:31 )
end
```

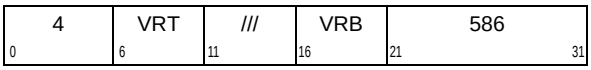
For each integer value i from 0 to 3, do the following.
Single-precision floating-point element i in VRB is rounded to a single-precision floating-point integer using the rounding mode Round toward +Infinity.

The result is placed into the corresponding word element i of VRT.

Special Registers Altered:
None

Vector Round to Floating-Point Integer toward Zero VX-form

vrfiz VRT,VRB




```
do i=0 to 127 by 32
   VRT0:31 ← RoundToSPIntTrunc( (VRB)0:31 )
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point element i in VRB is rounded to a single-precision floating-point integer using the rounding mode Round toward Zero.

The result is placed into the corresponding word element i of VRT.

Special Registers Altered:

None

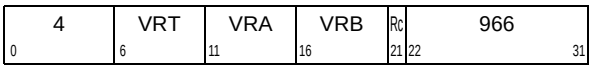
6.10.4 Vector Floating-Point Compare Instructions

The *Vector Floating-Point Compare* instructions compare two Vector Registers word element by word element, interpreting the elements as single-precision floating-point numbers. With the exception of the *Vector Compare Bounds Floating-Point* instruction, they set the target Vector Register, and CR Field 6 if Rc=1, in the same manner as do the *Vector Integer Compare* instructions; see Section 6.9.3.

The *Vector Compare Bounds Floating-Point* instruction sets the target Vector Register, and CR Field 6 if Rc=1, to indicate whether the elements in VRA are within the bounds specified by the corresponding element in VRB, as explained in the instruction description. A single-precision floating-point value x is said to be "within the bounds" specified by a single-precision floating-point value y if $-y \le x \le y$.

Vector Compare Bounds Floating-Point VC-form

vcmpbfp VRT,VRA,VRB (Rc=0)
vcmpbfp. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 32
   le ← ( (VRA)i:i+31 ≤fp (VRB)i:i+31 )
   ge ← ( (VRA)i:i+31 ≥fp -(VRB)i:i+31 )
   VRTi:i+31 ← ¬le || ¬ge || 30 0
end
if Rc=1 then do
   ib ← (VRT=128 0)
   CR6 ← 0b00 || ib || 0b0
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point word element i in VRA is compared to single-precision floating-point word element i in VRB. A 2-bit value is formed that indicates whether the element in VRA is within the bounds specified by the element in VRB, as follows.

- Bit 0 of the 2-bit value is set to 0 if the element in VRA is less than or equal to the element in VRB, and is set to 1 otherwise.
- Bit 1 of the 2-bit value is set to 0 if the element in VRA is greater than or equal to the negation of the element in VRB, and is set to 1 otherwise.

The 2-bit value is placed into the high-order two bits of word element *i* of VRT and the remaining bits of element *i* are set to 0.

If Rc=1, CR field 6 is set as follows.

Bit	Description
0	Set to 0
1	Set to 0
2	Set to indicate whether all four elements in VRA are within the bounds specified by the corresponding element in VRB, otherwise set to 0.
3	Set to 0

Special Registers Altered:

CR field 6 . . . . . (if Rc=1)

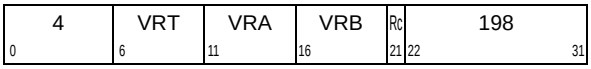
Programming Note

Each single-precision floating-point word element in VRB should be non-negative; if it is negative, the corresponding element in VRA will necessarily be out of bounds.

One exception to this is when the value of an element in VRB is -0.0 and the value of the corresponding element in VRA is either +0.0 or -0.0. +0.0 and -0.0 compare equal to -0.0.

Vector Compare Equal Floating-Point VC-form

vcmpeqfp VRT,VRA,VRB (Rc=0)
vcmpeqfp. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 32
   VRT_i:i+31 ← ((VRA)_i:i+31 =fp (VRB)_i:i+31) ? 32 1 : 32 0
end
if Rc=1 then do
   t ← ( VRT=128 1 )
   f ← ( VRT=128 0 )
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point element i in VRA is compared to single-precision floating-point element i in VRB. Word element i in VRT is set to all 1s if single-precision floating-point element i in VRA is equal to single-precision floating-point element i in VRB, and is set to all 0s otherwise.

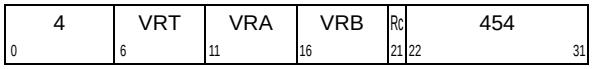
If the source element i in VRA or the source element i in VRB is a NaN, VRT is set to all 0s, indicating “not equal to”. If the source element i in VRA and the source element i in VRB are both infinity with the same sign, VRT is set to all 1s, indicating “equal to”.

Special Registers Altered:

CR field 6 . . . . . .(if Rc=1)

Vector Compare Greater Than or Equal Floating-Point VC-form

vcmpgefp VRT,VRA,VRB (Rc=0)
vcmpgefp. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 32
   VRT_i:i+31 ←((VRA)_i:i+31 ≥fp (VRB)_i:i+31) ? 32 1 : 32 0
end
if Rc=1 then do
   t ← ( VRT=128 1 )
   f ← ( VRT=128 0 )
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point element i in VRA is compared to single-precision floating-point element i in VRB. Word element i in VRT is set to all 1s if single-precision floating-point element i in VRA is greater than or equal to single-precision floating-point element i in VRB, and is set to all 0s otherwise.

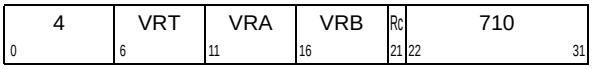
If the source element i in VRA or the source element i in VRB is a NaN, VRT is set to all 0s, indicating “not greater than or equal to”. If the source element i in VRA and the source element i in VRB are both infinity with the same sign, VRT is set to all 1s, indicating “greater than or equal to”.

Special Registers Altered:

CR field 6 . . . . . (if Rc=1)

Vector Compare Greater Than Floating-Point VC-form

vcmpgtfp VRT,VRA,VRB (Rc=0)
vcmpgtfp. VRT,VRA,VRB (Rc=1)




```
do i=0 to 127 by 32
   VRT_{i:i+31} ← ((VRA)_{i:i+31} >_fp (VRB)_{i:i+31}) ? 32^1 : 32^0
end
if Rc=1 then do
   t ← ( VRT=128^1 )
   f ← ( VRT=128^0 )
   CR6 ← t || 0b0 || f || 0b0
end
```

For each integer value i from 0 to 3, do the following.

Single-precision floating-point element i in VRA is compared to single-precision floating-point element i in VRB. Word element i in VRT is set to all 1s if single-precision floating-point element i in VRA is greater than single-precision floating-point element i in VRB, and is set to all 0s otherwise.

If the source element i in VRA or the source element i in VRB is a NaN, VRT is set to all 0s, indicating “not greater than”. If the source element i in VRA and the source element i in VRB are both infinity with the same sign, VRT is set to all 0s, indicating “not greater than”.

Special Registers Altered:

CR field 6 . . . . . .(if Rc=1)

6.10.5 Vector Floating-Point Estimate Instructions

Vector 2 Raised to the Exponent Estimate Floating-Point VX-form

vexptefp VRT,VRB

4	VRT	///	VRB	394
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT[i:i+31] ← Power2EstimateSP( (VRB)[i:i+31] )
end
```

For each integer value i from 0 to 3, do the following.

The single-precision floating-point estimate of 2 raised to the power of single-precision floating-point element *i* in VRB is placed into word element *i* of VRT.

Let x be any single-precision floating-point input value. Unless x< -146 or the single-precision floating-point result of computing 2 raised to the power x would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16. The most significant 12 bits of the estimate's significand are monotonic. An integral input value returns an integral value when the result is representable.

The result for various special cases of the source value is given below.

Value	Result
- Infinity	+0
-0	+1
+0	+1
+Infinity	+Infinity
NaN	QNaN

Special Registers Altered:

None

Vector Log Base 2 Estimate Floating-Point VX-form

vlogefp VRT,VRB

4	VRT	///	VRB	458
0	6	11	16	21 31

```
do i=0 to 127 by 32
   VRT[i:i+31] ← LogBase2EstimateSP((VRB)[i:i+31])
end
```

For each integer value i from 0 to 3, do the following.

The single-precision floating-point estimate of the base 2 logarithm of single-precision floating-point element *i* in VRB is placed into the corresponding word element of VRT.

Let x be any single-precision floating-point input value. Unless | x-1 | is less than or equal to 0.125 or the single-precision floating-point result of computing the base 2 logarithm of x would be an infinity or a QNaN, the estimate has an absolute error in precision (absolute value of the difference between the estimate and the infinitely precise value) no greater than 2^{-5}. Under the same conditions, the estimate has a relative error in precision no greater than one part in 8.

The most significant 12 bits of the estimate's significand are monotonic. The estimate is exact if $x=2^y$, where y is an integer between -149 and +127 inclusive. Otherwise the value placed into the element of register VRT may vary between implementations, and between different executions on the same implementation.

The result for various special cases of the source value is given below.

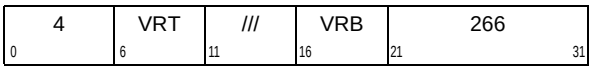
Value	Result
- Infinity	QNaN
< 0	QNaN
- 0	- Infinity
+0	- Infinity
+Infinity	+Infinity
NaN	QNaN

Special Registers Altered:

None

Vector Reciprocal Estimate Floating-Point VX-form

vrefp VRT,VRB




```
do i=0 to 127 by 32
   VRT_i:i+31 ← ReciprocalEstimateSP( (VRB)_i:i+31 )
end
```

For each integer value i from 0 to 3, do the following.
The single-precision floating-point estimate of the reciprocal of single-precision floating-point element i in VRB is placed into word element i of VRT.

Unless the single-precision floating-point result of computing the reciprocal of a value would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 4096.

Note that results may vary between implementations, and between different executions on the same implementation.

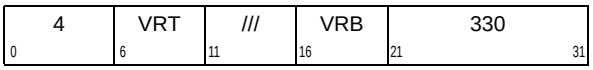
The result for various special cases of the source value is given below.

Value	Result
- Infinity	-0
- 0	- Infinity
+0	+ Infinity
+Infinity	+0
NaN	QNaN

Special Registers Altered:
None

Vector Reciprocal Square Root Estimate Floating-Point VX-form

vrsqrtefp VRT,VRB




```
do i=0 to 127 by 32
   VRT_i:i+31 ← RecipSquareRootEstimateSP((VRB)_i:i+31)
end
```

For each integer value i from 0 to 3, do the following.
The single-precision floating-point estimate of the reciprocal of the square root of single-precision floating-point element i in VRB is placed into word element i of VRT.

Let x be any single-precision floating-point value. Unless the single-precision floating-point result of computing the reciprocal of the square root of x would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 4096.

Note that results may vary between implementations, and between different executions on the same implementation.

The result for various special cases of the source value is given below.

Value	Result
- Infinity	QNaN
< 0	QNaN
- 0	- Infinity
+0	+ Infinity
+Infinity	+0
NaN	QNaN

Special Registers Altered:
None

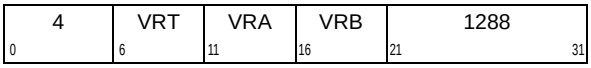
6.11 Vector Exclusive-OR-based Instructions

6.11.1 Vector AES Instructions

This section describes a set of instructions that support the Federal Information Processing Standards Publication 197 Advanced Encryption Standard for encryption and decryption.

Vector AES Cipher VX-form

vcipher VRT,VRA,VRB




```
State    ← VR[VRA]
RoundKey ← VR[VRB]
vtemp1   ← SubBytes(State)
vtemp2   ← ShiftRows(vtemp1)
vtemp3   ← MixColumns(vtemp2)
VR[VRT]  ← vtemp3 ^ RoundKey
```

Let State be the contents of VR[VRA], representing the intermediate state array during AES cipher operation.

Let RoundKey be the contents of VR[VRB], representing the round key.

One round of an AES cipher operation is performed on the intermediate State array, sequentially applying the transforms, SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey(), as defined in FIPS-197.

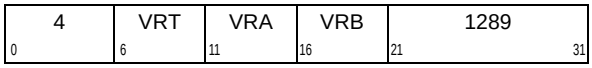
The result is placed into VR[VRT], representing the new intermediate state of the cipher operation.

Special Registers Altered:

None

Vector AES Cipher Last VX-form

vcipherlast VRT,VRA,VRB




```
State    ← VR[VRA]
RoundKey ← VR[VRB]
vtemp1   ← SubBytes(State)
vtemp2   ← ShiftRows(vtemp1)
VR[VRT]  ← vtemp2 ^ RoundKey
```

Let State be the contents of VR[VRA], representing the intermediate state array during AES cipher operation.

Let RoundKey be the contents of VR[VRB], representing the round key.

The final round in an AES cipher operation is performed on the intermediate State array, sequentially applying the transforms, SubBytes(), ShiftRows(), AddRoundKey(), as defined in FIPS-197.

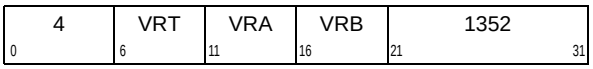
The result is placed into VR[VRT], representing the final state of the cipher operation.

Special Registers Altered:

None

Vector AES Inverse Cipher VX-form

vncipher VRT,VRA,VRB




```
State    ← VR[VRA]
RoundKey ← VR[VRB]
vtemp1   ← InvShiftRows(State)
vtemp2   ← InvSubBytes(vtemp1)
vtemp3   ← vtemp2 ^ RoundKey
VR[VRT]  ← InvMixColumns(vtemp3)
```

Let State be the contents of VR[VRA], representing the intermediate state array during AES inverse cipher operation.

Let RoundKey be the contents of VR[VRB], representing the round key.

One round of an AES inverse cipher operation is performed on the intermediate State array, sequentially applying the transforms, InvShiftRows(), InvSubBytes(), AddRoundKey(), and InvMixColumns(), as defined in FIPS-197.

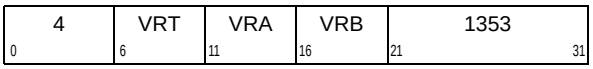
The result is placed into VR[VRT], representing the new intermediate state of the inverse cipher operation.

Special Registers Altered:

None

Vector AES Inverse Cipher Last VX-form

vncipherlast VRT,VRA,VRB




```
State    ← VR[VRA]
RoundKey ← VR[VRB]
vtemp1   ← InvShiftRows(State)
vtemp2   ← InvSubBytes(vtemp1)
VR[VRT]  ← vtemp2 ^ RoundKey
```

Let State be the contents of VR[VRA], representing the intermediate state array during AES inverse cipher operation.

Let RoundKey be the contents of VR[VRB], representing the round key.

The final round in an AES inverse cipher operation is performed on the intermediate State array, sequentially applying the transforms, InvShiftRows(), InvSubBytes(), and AddRoundKey(), as defined in FIPS-197.

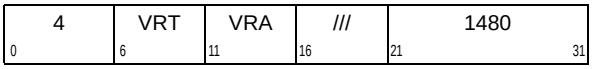
The result is placed into VR[VRT], representing the final state of the inverse cipher operation.

Special Registers Altered:

None

Vector AES SubBytes VX-form

vsbox VRT,VRA




```
State   ← VR[VRA]
VR[VRT] ← SubBytes(State)
```

Let State be the contents of VR[VRA], representing the intermediate state array during AES cipher operation.

The result of applying the transform, SubBytes() on State, as defined in FIPS-197, is placed into VR[VRT].

Special Registers Altered:

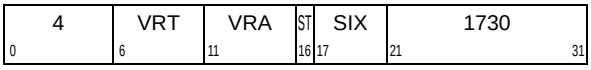
None

6.11.2 Vector SHA-256 and SHA-512 Sigma Instructions

This section describes a set of instructions that support the Federal Information Processing Standards Publication 180-3 Secure Hash Standard.

Vector SHA-512 Sigma Doubleword VX-form

vshasigmad VRT,VRA,ST,SIX




```
do i = 0 to 1
   src ← VR[VRA].doubleword[i]
   if ST=0 & SIX.bit[2×i]=0 then  // SHA-512 σ0 function
      VR[VRT].dword[i] ← (src >>>  1) ^
                         (src >>>  8) ^
                         (src >> 7)
   if ST=0 & SIX.bit[2×i]=1 then  // SHA-512 σ1 function
      VR[VRT].dword[i] ← (src >>> 19) ^
                         (src >>> 61) ^
                         (src >> 6)
   if ST=1 & SIX.bit[2×i]=0 then  // SHA-512 Σ0 function
      VR[VRT].dword[i] ← (src >>> 28) ^
                         (src >>> 34) ^
                         (src >>> 39)
   if ST=1 & SIX.bit[2×i]=1  then  // SHA-512 Σ1 function
      VR[VRT].dword[i] ← (src >>> 14) ^
                         (src >>> 18) ^
                         (src >>> 41)
end
```

For each integer value i from 0 to 1, do the following.

When ST=0 and bit 2×i of SIX is 0, a SHA-512 σ0 function is performed on the contents of doubleword element i of VR[VRA] and the result is placed into doubleword element i of VR[VRT].

When ST=0 and bit 2×i of SIX is 1, a SHA-512 σ1 function is performed on the contents of doubleword element i of VR[VRA] and the result is placed into doubleword element i of VR[VRT].

When ST=1 and bit 2×i of SIX is 0, a SHA-512 Σ0 function is performed on the contents of doubleword element i of VR[VRA] and the result is placed into doubleword element i of VR[VRT].

When ST=1 and bit 2×i of SIX is 1, a SHA-512 Σ1 function is performed on the contents of doubleword element i of VR[VRA] and the result is placed into doubleword element i of VR[VRT].

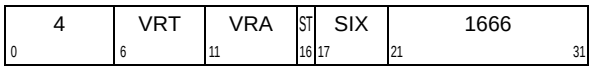
Bits 1 and 3 of SIX are reserved.

Special Registers Altered:

None

Vector SHA-256 Sigma Word VX-form

vshasigmaw VRT,VRA,ST,SIX




```
do i = 0 to 3
   src ← VR[VRA].word[i]
   if ST=0 & SIX.bit[i]=0 then  // SHA-256 σ0 function
      VR[VRT].word[i] ← (src >>>  7) ^
                        (src >>> 18) ^
                        (src >>  3)
   if ST=0 & SIX.bit[i]=1 then  // SHA-256 σ1 function
      VR[VRT].word[i] ← (src >>> 17) ^
                        (src >>> 19) ^
                        (src >> 10)
   if ST=1 & SIX.bit[i]=0 then  // SHA-256 Σ0 function
      VR[VRT].word[i] ← (src >>>  2) ^
                        (src >>> 13) ^
                        (src >>> 22)
   if ST=1 & SIX.bit[i]=1 then  // SHA-256 Σ1 function
      VR[VRT].word[i] ← (src >>>  6) ^
                        (src >>> 11) ^
                        (src >>> 25)
end
```

For each integer value i from 0 to 3, do the following.

When ST=0 and bit i of SIX is 0, a SHA-256 σ0 function is performed on the contents of word element i of VR[VRA] and the result is placed into word element i of VR[VRT].

When ST=0 and bit i of SIX is 1, a SHA-256 σ1 function is performed on the contents of word element i of VR[VRA] and the result is placed into word element i of VR[VRT].

When ST=1 and bit i of SIX is 0, a SHA-256 Σ0 function is performed on the contents of word element i of VR[VRA] and the result is placed into word element i of VR[VRT].

When ST=1 and bit i of SIX is 1, a SHA-256 Σ1 function is performed on the contents of word element i of VR[VRA] and the result is placed into word element i of VR[VRT].

Special Registers Altered:

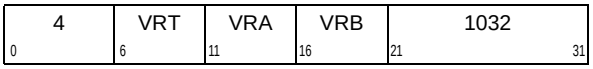
None

6.11.3 Vector Binary Polynomial Multiplication Instructions

This section describes a set of binary polynomial multiply-sum instructions. Corresponding elements are multiplied and the exclusive-OR of each even-odd pair of products sum, useful for a variety of finite field arithmetic operations.

Vector Polynomial Multiply-Sum Byte VX-form

vpmsumb VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()
do i = 0 to 15
   prod[i].bit[0:14] ← 0
   srcA ← VR[VRA].byte[i]
   srcB ← VR[VRB].byte[i]
   do j = 0 to 7
      do k = 0 to j
         gbit ← srcA.bit[k] & srcB.bit[j-k]
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
   do j = 8 to 14
      do k = j-7 to 7
         gbit ← (srcA.bit[k] & srcB.bit[j-k])
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
end
do i = 0 to 7
   VR[VRT].hword[i] ← 0b0 » (prod[2×i] ^ prod[2×i+1])
end
```

For each integer value i from 0 to 15, do the following.
Let prod[i] be the 15-bit result of a binary polynomial multiplication of the contents of byte element i of VR[VRA] and the contents of byte element i of VR[VRB].

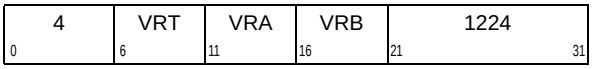
For each integer value i from 0 to 7, do the following.
The exclusive-OR of prod[2×i] and prod[2×i+1] is placed in bits 1:15 of halfword element i of VR[VRT]. Bit 0 of halfword element i of VR[VRT] is set to 0.

Special Registers Altered:

None

Vector Polynomial Multiply-Sum Doubleword VX-form

vpmsumd VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()
do i = 0 to 1
   prod[i].bit[0:126] ← 0
   srcA ← VR[VRA].doubleword[i]
   srcB ← VR[VRB].doubleword[i]
   do j = 0 to 63
      do k = 0 to j
         gbit ← srcA.bit[k] & srcB.bit[j-k]
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
   do j = 64 to 126
      do k = j-63 to 63
         gbit ← (srcA.bit[k] & srcB.bit[j-k])
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
end
VR[VRT] ← 0b0 » (prod[0] ^ prod[1])
```

Let prod[0] be the 127-bit result of a binary polynomial multiplication of the contents of doubleword element 0 of VR[VRA] and the contents of doubleword element 0 of VR[VRB].

Let prod[1] be the 127-bit result of a binary polynomial multiplication of the contents of doubleword element 1 of VR[VRA] and the contents of doubleword element 1 of VR[VRB].

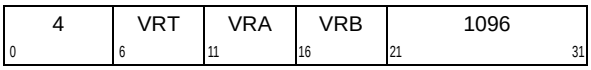
The exclusive-OR of prod[0] and prod[1] is placed in bits 1:127 of VR[VRT]. Bit 0 of VR[VRT] is set to 0.

Special Registers Altered:

None

Vector Polynomial Multiply-Sum Halfword VX-form

vpmsumh VRT,VRA,VRB




```
do i = 0 to 7
   prod.bit[0:30] ← 0
   srcA ← VR[VRA].halfword[i]
   srcB ← VR[VRB].halfword[i]
   do j = 0 to 15
      do k = 0 to j
         gbit ← srcA.bit[k] & srcB.bit[j-k]
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
   do j = 16 to 30
      do k = j-15 to 15
         gbit ← (srcA.bit[k] & srcB.bit[j-k])
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
end
VR[VRT].word[0] ← 0b0 » (prod[0] ^ prod[1])
VR[VRT].word[1] ← 0b0 » (prod[2] ^ prod[3])
VR[VRT].word[2] ← 0b0 » (prod[4] ^ prod[5])
VR[VRT].word[3] ← 0b0 » (prod[6] ^ prod[7])
```

For each integer value i from 0 to 7, do the following.
Let prod[i] be the 31-bit result of a binary polynomial multiplication of the contents of halfword element i of VR[VRA] and the contents of halfword element i of VR[VRB].

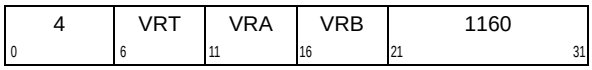
For each integer value i from 0 to 3, do the following.
The exclusive-OR of prod[2×i] and prod[2×i+1] is placed in bits 1:31 of word element i of VR[VRT]. Bit 0 of word element i of VR[VRT] is set to 0.

Special Registers Altered:

None

Vector Polynomial Multiply-Sum Word VX-form

vpmsumw VRT,VRA,VRB




```
do i = 0 to 3
   prod[i].bit[0:62] ← 0
   srcA ← VR[VRA].word[i]
   srcB ← VR[VRB].word[i]
   do j = 0 to 31
      do k = 0 to j
         gbit ← srcA.bit[k] & srcB.bit[j-k]
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
   do j = 32 to 62
      do k = j-31 to 31
         gbit ← (srcA.bit[k] & srcB.bit[j-k])
         prod[i].bit[j] ← prod[i].bit[j] ^ gbit
      end
   end
end
VR[VRT].dword[0] ← 0b0 » (prod[0] ^ prod[1])
VR[VRT].dword[1] ← 0b0 » (prod[2] ^ prod[3])
```

For each integer value i from 0 to 3, do the following.
Let prod[i] be the 63-bit result of a binary polynomial multiplication of the contents of word element i of VR[VRA] and the contents of word element i of VR[VRB].

For each integer value i from 0 to 1, do the following.
The exclusive-OR of prod[2×i] and prod[2×i+1] is placed in bits 1:63 of doubleword element i of VR[VRT]. Bit 0 of doubleword element i of VR[VRT] is set to 0.

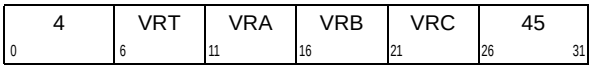
Special Registers Altered:

None

6.11.4 Vector Permute and Exclusive-OR Instruction

Vector Permute and Exclusive-OR VA-form

vpermxor VRT,VRA,VRB,VRC




```
do i = 0 to 15
   indexA ← VR[VRC].byte[i].bit[0:3]
   indexB ← VR[VRC].byte[i].bit[4:7]
   src1   ← VR[VRA].byte[indexA]
   src2   ← VR[VRB].byte[indexB]
   VSR[VRT].byte[i] ← src1 ^ src2
end
```

For each integer value i from 0 to 15, do the following.

Let indexA be the contents of bits 0:3 of byte element i of VR[VRC].

Let indexB be the contents of bits 4:7 of byte element i of VR[VRC].

The exclusive OR of the contents of byte element indexA of VR[VRA] and the contents of byte element indexB of VR[VRB] is placed into byte element i of VR[VRT].

Special Registers Altered:

None

6.12 Vector Gather Instruction

Vector Gather Bits by Bytes by Doubleword VX-form

vgbbd VRT,VRB

4	VRT	///	VRB	1292
0	6	11	16	21 31

```
do i = 0 to 1
    do j = 0 to 7
        do k = 0 to 7
            b ← VSR[VRB].dword[i].byte[k].bit[j]
            VSR[VRT].dword[i].byte[j].bit[k] ← b
        end
    end
end
```

Let src be the contents of VR[VRB], composed of two doubleword elements numbered 0 and 1.

Let each doubleword element be composed of eight bytes numbered 0 through 7.

An 8-bit × 8-bit bit-matrix transpose is performed on the contents of each doubleword element of VR[VRB] (see Figure 104).

For each integer value i from 0 to 1, do the following,

The contents of bit 0 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 0 of doubleword element i of VR[VRT].

The contents of bit 1 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 1 of doubleword element i of VR[VRT].

The contents of bit 2 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 2 of doubleword element i of VR[VRT].

The contents of bit 3 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 3 of doubleword element i of VR[VRT].

The contents of bit 4 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 4 of doubleword element i of VR[VRT].

The contents of bit 5 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 5 of doubleword element i of VR[VRT].

The contents of bit 6 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 6 of doubleword element i of VR[VRT].

The contents of bit 7 of each byte of doubleword element i of VR[VRB] are concatenated and placed into byte 7 of doubleword element i of VR[VRT].

Special Registers Altered:

None

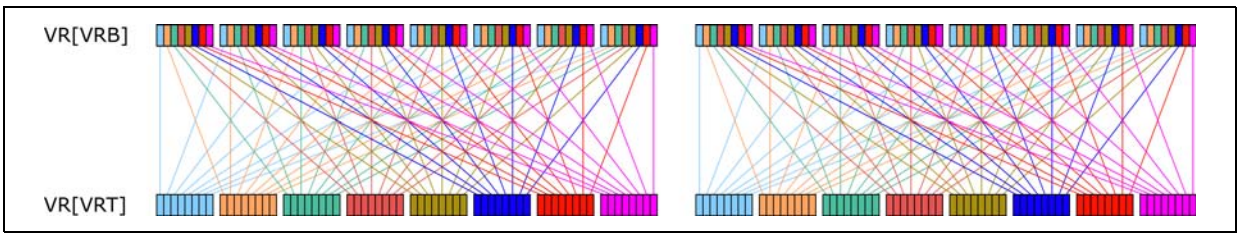
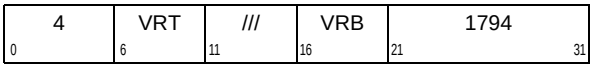


Figure 104.Vector Gather Bits by Bytes by Doubleword

6.13 Vector Count Leading Zeros Instructions

Vector Count Leading Zeros Byte VX-form

vclzb VRT,VRB



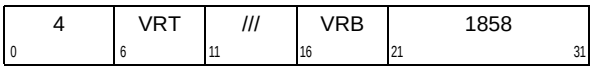

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   n ← 0
   do while n < 8
      if VR[VRB].byte[i].bit[n] = 0b1 then leave
      n ← n + 1
   end
   VSR[VRT].byte[i] ← n
end
```

For each integer value i from 0 to 15, do the following.
A count of the number of consecutive zero bits starting at bit 0 of byte element i of VR[VRB] is placed into byte element i of VR[VRT]. This number ranges from 0 to 8, inclusive.

Special Registers Altered:
None

Vector Count Leading Zeros Halfword VX-form

vclzh VRT,VRB



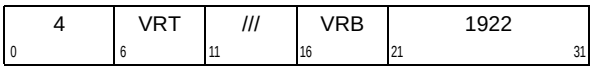

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   n ← 0
   do while n < 16
      if VR[VRB].hword[i].bit[n] = 0b1 then leave
      n ← n + 1
   end
   VSR[VRT].hword[i] ← n
end
```

For each integer value i from 0 to 7, do the following.
A count of the number of consecutive zero bits starting at bit 0 of halfword element i of VR[VRB] is placed into halfword element i of VR[VRT]. This number ranges from 0 to 16, inclusive.

Special Registers Altered:
None

Vector Count Leading Zeros Word VX-form

vclzw VRT,VRB



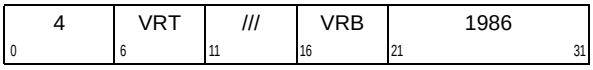

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   n ← 0
   do while n < 32
      if VR[VRB].word[i].bit[n] = 0b1 then leave
      n ← n + 1
   end
   VSR[VRT].word[i] ← n
end
```

For each integer value i from 0 to 3, do the following.
A count of the number of consecutive zero bits starting at bit 0 of word element i of VR[VRB] is placed into word element i of VR[VRT]. This number ranges from 0 to 32, inclusive.

Special Registers Altered:
None

Vector Count Leading Zeros Doubleword VX-form

vclzd VRT,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   n ← 0
   do while (n<64) & (VR[VRB].dword[i].bit[n]=0b0)
      n ← n + 1
   end
   VSR[VRT].dword[i] ← n
end
```

For each integer value i from 0 to 1, do the following.
A count of the number of consecutive zero bits starting at bit 0 of doubleword element i of VR[VRB] is placed into doubleword element i of VR[VRT]. This number ranges from 0 to 64, inclusive.

Special Registers Altered:
None

6.14 Vector Count Trailing Zeros Instructions

Vector Count Trailing Zeros Byte VX-form

vctzb VRT,VRB

4	VRT	28	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   n ← 0
   do while n < 8
      if VR[VRB].byte[i].bit[7-n] = 0b1 then leave
      n ← n + 1
   end
   VR[VRT].byte[i] ← Chop(EXTZ(n), 8)
end
```

For each integer value i from 0 to 15, do the following.
A count of the number of consecutive zero bits starting at bit 7 of byte element i of VR[VRB] is placed into byte element i of VR[VRT]. This number ranges from 0 to 8, inclusive.

Special Registers Altered:
None

Vector Count Trailing Zeros Halfword VX-form

vctzh VRT,VRB

4	VRT	29	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   n ← 0
   do while n < 16
      if VR[VRB].hword[i].bit[15-n] = 0b1 then leave
      n ← n + 1
   end
   VR[VRT].hword[i] ← Chop(EXTZ(n), 16)
end
```

For each integer value i from 0 to 7, do the following.
A count of the number of consecutive zero bits starting at bit 15 of halfword element i of VR[VRB] is placed into halfword element i of VR[VRT]. This number ranges from 0 to 16, inclusive.

Special Registers Altered:
None

Vector Count Trailing Zeros Word VX-form

vctzw VRT,VRB

4	VRT	30	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   n ← 0
   do while n < 32
      if VR[VRB].word[i].bit[31-n] = 0b1 then leave
      n ← n + 1
   end
   VR[VRT].word[i] ← Chop(EXTZ(n), 32)
end
```

For each integer value i from 0 to 3, do the following.
A count of the number of consecutive zero bits starting at bit 31 of word element i of VR[VRB] is placed into word element i of VR[VRT]. This number ranges from 0 to 32, inclusive.

Special Registers Altered:
None

Vector Count Trailing Zeros Doubleword VX-form

vctzd VRT,VRB

4	VRT	31	VRB	1538
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   n ← 0
   do while n < 64
      if VR[VRB].dword[i].bit[63-n] = 0b1 then leave
      n ← n + 1
   end
   VR[VRT].dword[i] ← Chop(EXTZ(n), 64)
end
```

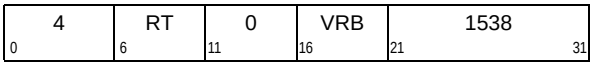
For each integer value i from 0 to 1, do the following.
A count of the number of consecutive zero bits starting at bit 63 of doubleword element i of VR[VRB] is placed into doubleword element i of VR[VRT]. This number ranges from 0 to 64, inclusive.

Special Registers Altered:
None

6.14.1 Vector Count Leading/Trailing Zero LSB Instructions

Vector Count Leading Zero Least-Significant Bits Byte VX-form

vclzlsbb RT,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

count ← 0
do while count < 16
   if (VR[VRB].byte[count].bit[7]=1) break
   count ← count + 1
end

GPR[RT] ← EXTZ64(count)
```

Let count be the number of contiguous leading byte elements in VR[VRB] having a zero least-significant bit.

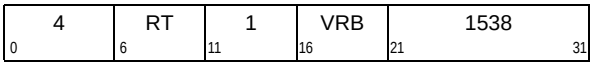
count is placed into GPR[RT].

Special Registers Altered:

None

Vector Count Trailing Zero Least-Significant Bits Byte VX-form

vctzlsbb RT,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

count ← 0
do while count < 16
   if (VR[VRB].byte[15-count].bit[7]=1) break
   count ← count + 1
end

GPR[RT] ← EXTZ64(count)
```

Let count be the number of contiguous trailing byte elements in VR[VRB] having a zero least-significant bit.

count is placed into GPR[RT].

Special Registers Altered:

None

6.14.2 Vector Extract Element Instructions

Vector Extract Unsigned Byte Left-Indexed VX-form

vextublx RT,RA,VRB

4	RT	RA	VRB	1549
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]

GPR[RT] ← EXTZ64(VR[VRB].byte[index])
```

Let index be the contents of bits 60:63 of GPR[RA].

The contents of byte element index of VR[VRB] are placed into bits 56:63 of GPR[RT].

The contents of bits 0:55 of GPR[RT] are set to 0.

Special Registers Altered:

None

Vector Extract Unsigned Halfword Left-Indexed VX-form

vextuhlx RT,RA,VRB

4	RT	RA	VRB	1613
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]

GPR[RT] ← EXTZ64(VR[VRB].byte[index:index+1])
```

Let index be the contents of bits 60:63 of GPR[RA].

The contents of byte elements index:index+1 of VR[VRB] are placed into bits 48:63 of GPR[RT].

The contents of bits 0:47 of GPR[RT] are set to 0.

If the value of index is greater than 14, the results are undefined.

Special Registers Altered:

None

Vector Extract Unsigned Byte Right-Indexed VX-form

vextubrx RT,RA,VRB

4	RT	RA	VRB	1805
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]

GPR[RT] ← EXTZ64(VR[VRB].byte[15-index])
```

Let index be the contents of bits 60:63 of GPR[RA].

The contents of byte element 15-index of VR[VRB] are placed into bits 56:63 of GPR[RT].

The contents of bits 0:55 of GPR[RT] are set to 0.

Special Registers Altered:

None

Vector Extract Unsigned Halfword Right-Indexed VX-form

vextuhrx RT,RA,VRB

4	RT	RA	VRB	1869
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]

GPR[RT] ← EXTZ64(VR[VRB].byte[14-index:15-index])
```

Let index be the contents of bits 60:63 of GPR[RA].

The contents of byte elements 14-index:15-index of VR[VRB] are placed into bits 48:63 of GPR[RT].

The contents of bits 0:47 of GPR[RT] are set to 0.

If the value of index is greater than 14, the results are undefined.

Special Registers Altered:

None

Vector Extract Unsigned Word Left-Indexed VX-form

vextuwlx RT,RA,VRB

4	RT	RA	VRB	1677
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]

GPR[RT] ← EXTZ64(VR[VRB].byte[index:index+3])
```

Let index be the contents of bits 60:63 of GPR[RA].

The contents of byte elements index:index+3 of VR[VRB] are placed into bits 32:63 of GPR[RT].

The contents of bits 0:31 of GPR[RT] are set to 0.

If the value of index is greater than 12, the results are undefined.

Special Registers Altered:

None

Vector Extract Unsigned Word Right-Indexed VX-form

vextuwrx RT,RA,VRB

4	RT	RA	VRB	1933
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

index ← GPR[RA].bit[60:63]

GPR[RT] ← EXTZ64(VR[VRB].byte[12-index:15-index])
```

Let index be the contents of bits 60:63 of GPR[RA].

The contents of byte elements index:index+3 of VR[VRB] are placed into bits 32:63 of GPR[RT].

The contents of bits 0:31 of GPR[RT] are set to 0.

If the value of index is greater than 12, the results are undefined.

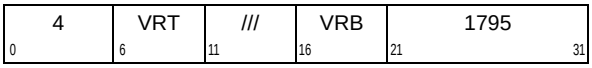
Special Registers Altered:

None

6.15 Vector Population Count Instructions

Vector Population Count Byte VX-form

vpopcntb VRT,VRB



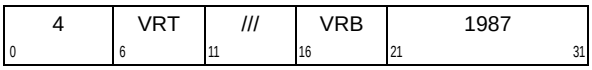

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   n ← 0
   do j = 0 to 7
      n ← n + VR[VRB].byte[i].bit[j]
   end
   VSR[VRT].byte[i] ← n
end
```

For each integer value i from 0 to 15, do the following.
A count of the number of bits set to 1 in byte element i of VR[VRB] is placed into byte element i of VR[VRT]. This number ranges from 0 to 8, inclusive.

Special Registers Altered:
None

Vector Population Count Doubleword VX-form

vpopcntd VRT,VRB



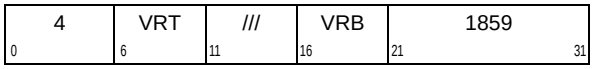

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   n ← 0
   do j = 0 to 63
      n ← n + VR[VRB].dword[i].bit[j]
   end
   VSR[VRT].dword[i] ← n
end
```

For each integer value i from 0 to 1, do the following.
A count of the number of bits set to 1 in doubleword element i of VR[VRB] is placed into doubleword element i of VR[VRT]. This number ranges from 0 to 64, inclusive.

Special Registers Altered:
None

Vector Population Count Halfword VX-form

vpopcnth VRT,VRB



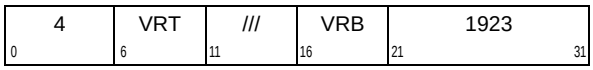

```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 7
   n ← 0
   do j = 0 to 15
      n ← n + VR[VRB].hword[i].bit[j]
   end
   VSR[VRT].hword[i] ← n
end
```

For each integer value i from 0 to 7, do the following.
A count of the number of bits set to 1 in halfword element i of VR[VRB] is placed into halfword element i of VR[VRT]. This number ranges from 0 to 16, inclusive.

Special Registers Altered:
None

Vector Population Count Word VX-form

vpopcntw VRT,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 3
   n ← 0
   do j = 0 to 31
      n ← n + VR[VRB].word[i].bit[j]
   end
   VSR[VRT].word[i] ← n
end
```

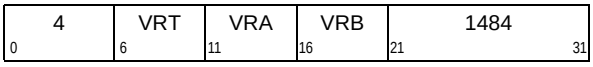
For each integer value i from 0 to 3, do the following.
A count of the number of bits set to 1 in word element i of VR[VRB] is placed into word element i of VR[VRT]. This number ranges from 0 to 32, inclusive.

Special Registers Altered:
None

6.16 Vector Bit Permute Instruction

Vector Bit Permute Doubleword VX-form

vbpermd VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 1
   do j = 0 to 7
      index ← VR[VRB].dword[i].byte[j]
      if index < 64 then
         perm.bit[j] ← VR[VRA].dword[i].bit[index]
      else
         perm.bit[j] ← 0
   end

   VR[VRT].dword[i] ← EXTZ64(perm)
end
```

For each integer value i from 0 to 1, and for each integer value j from 0 to 7, do the following.

Let index be the contents of byte sub-element j of doubleword element i of VR[VRB].

If index is less than 64, then the contents of bit index of doubleword i of VR[VRA] are placed into bit 56+j of doubleword element i of VR[VRT]. Otherwise, bit 56+j of doubleword element i of VR[VRT] is set to 0.

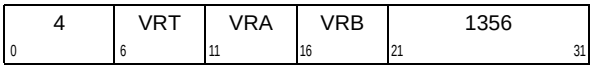
The contents of bits 0:55 of doubleword element i of VR[VRT] are set to 0.

Special Registers Altered:

None

Vector Bit Permute Quadword VX-form

vbpermq VRT,VRA,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   index ← VR[VRB].byte[i]
   if index < 128 then
      perm.bit[i] ← VR[VRA].bit[index]
   else
      perm.bit[i] ← 0
end
VR[VRT].dword[0] ← Chop(EXTZ(perm),64)
VR[VRT].dword[1] ← 0x0000_0000_0000_0000
```

For each integer value i from 0 to 15, do the following.

Let index be the contents of byte element i of VR[VRB].

If index is less than 128, then the contents of bit index of VR[VRA] are placed into bit 48+i of doubleword element i of VR[VRT]. Otherwise, bit 48+i of doubleword element i of VR[VRT] is set to 0.

The contents of bits 0:47 of VR[VRT] are set to 0.
The contents of bits 64:127 of VR[VRT] are set to 0.

Special Registers Altered:

None

Programming Note

The fact that the permuted bit is 0 if the corresponding index value exceeds 127 permits the permuted bits to be selected from a 256-bit quantity, using a single index register. For example, assume that the 256-bit quantity Q, from which the permuted bits are to be selected, is in registers v2 (high-order 128 bits of Q) and v3 (low-order 128 bits of Q), that the index values are in register v1, with each byte of v1 containing a value in the range 0:255, and that each byte of register v4 contains the value 128. The following code sequence selects eight permuted bits from Q and places them into the low-order byte of v6.

```
vbpermq   v6,v1,v2    # select from high-order half
                          of Q
vxor      v0,v1,v4    # adjust index values
vbpermq   v5,v0,v3    # select from low-order half
                          of Q
vor       v6,v6,v5    # merge the two selections
```

6.17 Decimal Integer Instructions

A *valid encoding* of a packed decimal integer value requires the following properties.

- Each of the 31 4-bit digits of the operand's magnitude (bits 0:123) must be in the range 0-9.
- The sign code (bits 124:127) must be in the range 10-15.

Source operands with sign codes of `0b1010`, `0b1100`, `0b1110`, and `0b1111` are interpreted as positive values.

Source operands with sign codes of `0b1011` and `0b1101` are interpreted as negative values.

Positive and zero results are encoded with a either sign code of `0b1100` or `0b1111`, depending on the preferred sign (indicated as an immediate operand).

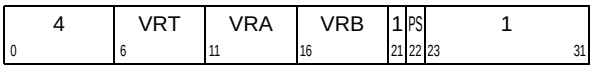
Negative results are encoded with a sign code of `0b1101`.

6.17.1 Decimal Integer Arithmetic Instructions

The *Decimal Integer Arithmetic* instructions operate on decimal integer values only in signed packed decimal format. Signed packed decimal format consists of 31 4-bit base-10 digits of magnitude and a trailing 4-bit sign code. Operations are performed as sign-magnitude, and produce a decimal result placed in a Vector Register (i.e., ***bcdadd***, ***bcdsub***).

Decimal Add Modulo VX-form

bcdadd. VRT,VRA,VRB,PS




```
if MSR.VEC=0 then Vector_Unavailable()

VR[VRT] ← Signed_BCD_Add(VR[VRA],VR[VRB],PS)

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← ox_flag | inv_flag
```

Let src1 be the decimal integer value in VR[VRA].
Let src2 be the decimal integer value in VR[VRB].

src1 is added to src2.

If the unbounded result is equal to zero, do the following.

If PS=0, the sign code of the result is set to 0b1100.
If PS=1, the sign code of the result is set to 0b1111.

CR field 6 is set to 0b0010.

If the unbounded result is greater than zero, do the following.

If PS=0, the sign code of the result is set to 0b1100.
If PS=1, the sign code of the result is set to 0b1111.

If the operation overflows, CR field 6 is set to 0b0101. Otherwise, CR field 6 is set to 0b0100.

If the unbounded result is less than zero, do the following.

The sign code of the result is set to 0b1101.

If the operation overflows, CR field 6 is set to 0b1001. Otherwise, CR field 6 is set to 0b1000.

The low-order 31 digits of the magnitude of the result are placed in bits 0:123 of VR[VRT].

The sign code is placed in bits 124:127 of VR[VRT].

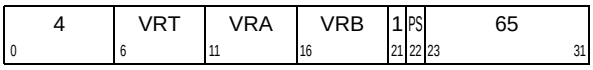
If either src1 or src2 is an *invalid encoding* of a 31-digit signed decimal value, the result is undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

CR field 6

Decimal Subtract Modulo VX-form

bcdsub. VRT,VRA,VRB,PS




```
if MSR.VEC=0 then Vector_Unavailable()

VR[VRT] ← Signed_BCD_Subtract(VR[VRA],VR[VRB],PS)

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← ox_flag | inv_flag
```

Let src1 be the decimal integer value in VR[VRA].
Let src2 be the decimal integer value in VR[VRB].

src1 is subtracted by src2.

If the unbounded result is equal to zero, do the following.

If PS=0, the sign code of the result is set to 0b1100.
If PS=1, the sign code of the result is set to 0b1111.

CR field 6 is set to 0b0010.

If the unbounded result is greater than zero, do the following.

If PS=0, the sign code of the result is set to 0b1100.
If PS=1, the sign code of the result is set to 0b1111.

If the operation overflows, CR field 6 is set to 0b0101. Otherwise, CR field 6 is set to 0b0100.

If the unbounded result is less than zero, do the following.

The sign code of the result is set to 0b1101.

If the operation overflows, CR field 6 is set to 0b1001. Otherwise, CR field 6 is set to 0b1000.

The low-order 31 digits of the magnitude of the result are placed in bits 0:123 of VR[VRT].

The sign code is placed in bits 124:127 of VR[VRT].

If either src1 or src2 is an *invalid encoding* of a 31-digit signed decimal value, the result is undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

CR field 6

Programming Note

Software should take care when interoperability with the Decimal Floating-Point facilities is required. The register format defined for 31-digit signed decimal values employed by ***bcdadd.*** and ***bcdsub.*** is a single 128-bit VR. The register format defined for 31-digit signed decimal values employed by the *Decimal Floating-Point* instructions ***ddedpdq***[**.**] and ***denbcdq***[**.**] is a pair of 64-bit FPRs. ***xxpermdi*** can be used to convert between the two register formats as well as move data beween the FPR and VR halves of the Vector-Scalar Registers.

gew and ***fmrgow*** are provided to support direct move operations in 32-bit mode.

Programming Note

`bcdsub. vTmp,vA,vB,0` can be used to compare decimal operands `vA` and `vB`. Bits `0:2` of `CR` field `6` will be set to indicate `vA` is less than `vB` (`LT`), `vA` is greater than `vB` (`GT`), and `vA` is equal to `vB` (`EQ`).

`bcdsub. vTmp,vA,vA,0` can be used to test if an operand `vA` is an *invalid encoding* of a decimal value.

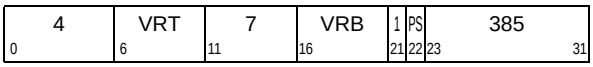
Programming Note

When bit `3` of `CR` field `6` is set to `1` by ***bcdadd.*** or ***bcdsub.***, either an overflow occurred or one or both operands are not valid encodings of decimal values. Discerning whether an overflow occurred can be accomplished by performing the other decimal instruction on the operands. For example, if ***bcdadd.*** caused bit `3` of `CR` field `6` to be set to `1`, performing ***bcdsub.*** on the same set of operands will cause bit `3` of `CR` field `6` to be set to `1` if and only if one or both of the operands is an invalid encoding. If bit `3` of `CR` field `6` is not set by ***bcdsub.*** then the ***bcdadd.*** can be asserted to have overflowed. Likewise, ***bcdadd.*** can be used in a similar manner to determine the cause of bit `3` of `CR` field `6` getting set by a ***bcdsub.***.

6.17.2 Decimal Integer Format Conversion Instructions

Decimal Convert From National VX-form

bcdcfn. VRT,VRB,PS




```
if MSR.VEC=0 then Vector_Unavailable()

src_sign ← (VR[VRB].hword[7] = 0x002D)
eq_flag  ← 1

/* check for valid sign */
inv_flag ← (VR[VRB].hword[7] != 0x002B) &
           (VR[VRB].hword[7] != 0x002D)

do i = 0 to 6
   eq_flag  ← eq_flag  & (VR[VRB].hword[i] = 0x0030)
   /* check for valid digit */
   inv_flag ← inv_flag | (VR[VRB].hword[i] < 0x0030)
                       | (VR[VRB].hword[i] > 0x0039)
end

lt_flag  ← (eq_flag=0) & (src_sign=1)
gt_flag  ← (eq_flag=0) & (src_sign=0)

do i = 0 to 23
   result.nibble[i] ← 0x0
end
do i = 0 to 6
   result.nibble[i+24] ← VR[VRB].hword[i].nibble[3]
end
result.nibble[31] ← (src_sign=0) ? ((PS=0) ? 0xC : 0xF) : 0xD

VR[VRT]    ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag
```

Let src be the national decimal value in VR[VRB].

src is placed in VR[VRT] in packed decimal format.

A valid encoding of a national decimal value requires the following.

– The contents of halfword 7 (sign code) must be either 0x002B or 0x002D.
– The contents of halfwords 0 to 6 must be in the range 0x0030 to 0x0039.

National decimal values having a sign code of 0x002B are interpreted as positive values.

National decimal values having a sign code of 0x002D are interpreted as negative values.

For each integer value i from 0 to 23, do the following.
The contents of nibble element i of VR[VRT] are set to 0x0.

For each integer value i from 0 to 6, do the following.
The contents of nibble 3 of halfword element i of src are placed into nibble element i+24 of VR[VRT].

For PS=0, the contents of nibble element 31 (i.e., sign code) of VR[VRT] are set to 0xC for positive values and to 0xD for negative values.

For PS=1, the contents of nibble element 31 (i.e., sign code) of VR[VRT] are set to 0xF for positive values and to 0xD for negative values.

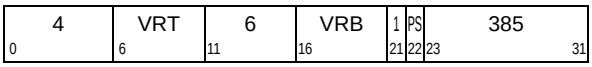
CR field 6 is set to reflect src compared to zero.

If src is an *invalid encoding* of a national decimal value, the contents of VR[VRT] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:
CR field 6

Decimal Convert From Zoned VX-form

bcdcfz. VRT,VRB,PS




```
if MSR.VEC=0 then Vector_Unavailable()

/* check for valid sign */
inv_flag ← ((VR[VRB].byte[15].nibble[0] < 0xA) & (PS=1)) |
           (VR[VRB].byte[15].nibble[1] > 0x9)

/* check for valid digits */
MIN ← (PS=0) ? 0x30 : 0xF0
MAX ← (PS=0) ? 0x39 : 0xF9
do i = 0 to 14
   inv_flag ← inv_flag | (VR[VRB].byte[i] < MIN)
                       | (VR[VRB].byte[i] > MAX)
end

if PS=0 then
   src_sign ←  VR[VRB].nibble[30].bit[1]
else
   src_sign ← (VR[VRB].nibble[30] = 0b1011) |
              (VR[VRB].nibble[30] = 0b1101)

eq_flag  ← 1

do i = 0 to 14
   result.nibble[i] ← 0x0
end
do i = 0 to 15
   result.nibble[i+15] ← VR[VRB].byte[i].nibble[1]
   eq_flag  ← eq_flag & (VR[VRB].byte[i].nibble[1]=0x0)
end

lt_flag  ← (eq_flag=0) & (src_sign=1)
gt_flag  ← (eq_flag=0) & (src_sign=0)

result.nibble[31] ← (src_sign=0) ? 0xC : 0xD

VR[VRT]    ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag
```

Let src be the zoned decimal value in VR[VRB].

src is placed in VR[VRT] in packed decimal format.

When PS=0, do the following.

A valid encoding of a zoned decimal value requires the following.

- The contents of bits 0:3 of byte 15 (sign code) can be any value in the range 0x0 to 0xF.
- The contents of bits 0:3 of bytes 0 to 14 must be the value 0x3.
- The contents of bits 4:7 of bytes 0 to 15 must be a value in the range 0x0 to 0x9.

Zoned decimal values having a sign code of 0x0, 0x1, 0x2, 0x3, 0x8, 0x9, 0xA, or 0xB are interpreted as positive values.

Zoned decimal values having a sign code of 0x4, 0x5, 0x6, 0x7, 0xC, 0xD, 0xE, or 0xF are interpreted as negative values.

When PS=1, do the following.

A valid encoding of a zoned decimal source operand requires the following.

- The contents of bits 0:3 of byte 15 (sign code) must be a value in the range 0xA to 0xF.
- The contents of bits 0:3 of bytes 0 to 14 must be the value 0xF.
- The contents of bits 4:7 of bytes 0 to 15 must be a value in the range 0x0 to 0x9.

Zoned decimal source operands having a sign code of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Zoned decimal source operands having a sign code of 0xB or 0xD are interpreted as negative values.

Positive packed decimal results are returned with a sign code of 0xC.

Negative packed decimal results are returned with a sign code of 0xD.

For each integer value i from 0 to 14,

The contents of nibble element i of VR[VRT] are set to 0x0.

For each integer value i from 0 to 15,

The contents of nibble 1 of byte element i of src are placed into nibble element i+15 of VR[VRT].

CR field 6 is set to reflect src compared to zero.

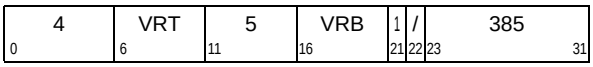
If src is an *invalid encoding* of a zoned decimal value, the contents of VR[VRT] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

CR field 6

Decimal Convert To National VX-form

bcdctn. VRT,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

ox_flag ← 0
do i = 0 to 23
   ox_flag ← ox_flag | (VR[VRB].nibble[i] != 0x0)
end

inv_flag ← (VR[VRB].nibble[31] < 0xA)
do i = 0 to 30
   inv_flag ← inv_flag | (VR[VRB].nibble[i] > 0x9)
end

src_sign ← (VR[VRB].nibble[31] = 0xB) |
           (VR[VRB].nibble[31] = 0xD)

eq_flag  ← (VR[VRB].nibble[0:30] = 0)
lt_flag  ← (eq_flag=0) & (src_sign=1)
gt_flag  ← (eq_flag=0) & (src_sign=0)

do i = 0 to 6
   result.hword[i].nibble[0:2] ← 0x003
   result.hword[i].nibble[3]   ← VR[VRB].nibble[i+24]
end

result.hword[7] ← (src_sign=1) ? 0x002D : 0x002B

VR[VRT]    ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag | ox_flag
```

Let src be the packed decimal value in VR[VRB].

src is placed into VR[VRT] in national decimal format.

A valid encoding of a signed packed decimal value requires the following.

- The contents of nibble 31 (sign code) must be a value in the range 0xA to 0xF.
- The contents of each nibble 0-30 must be a value in the range 0x0 to 0x9.

Packed decimal values with sign codes of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Packed decimal values with sign codes of 0xB or 0xD are interpreted as negative values.

Values greater in magnitude than $10^7 - 1$ are too large to be represented in national decimal format.

For each integer value i from 0 to 6, do the following.

The value 0x003 is placed into nibbles 0:2 of halfword element i of VR[VRT].

The contents of nibble element i+24 of VR[VRB] are placed into nibble 3 of halfword element i of VR[VRT].

The contents of halfword element 7 (i.e., sign code) of VR[VRT] are set to 0x002B for positive values and to 0x002D for negative values.

CR field 6 is set to reflect src compared to zero, including whether or not src is too large to be represented in national decimal format.

If src is an *invalid encoding* of a packed decimal value, the contents of VR[VRT] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

CR field 6

Decimal Convert To Zoned VX-form

bcdctz. VRT,VRB,PS

4	VRT	4	VRB	1	PS	385
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

inv_flag ← (VR[VRB].nibble[31] < 0xA)
do i = 0 to 30
    inv_flag ← inv_flag | (VR[VRB].nibble[i] > 0x9)
end

ox_flag ← 0
do i = 0 to 15
   ox_flag ← ox_flag | (VR[VRB].nibble[i] != 0x0)
end

src_sign ← (VR[VRB].nibble[31] = 0xB) |
           (VR[VRB].nibble[31] = 0xD)

eq_flag  ← (VR[VRB].nibble[0:30] = 0)
lt_flag  ← (eq_flag=0) & (src_sign=1)
gt_flag  ← (eq_flag=0) & (src_sign=0)

do i = 0 to 14
   result.byte[i].nibble[0] ← (PS=0) ? 0x3 : 0xF
   result.byte[i].nibble[1] ← VR[VRB].nibble[i+15]
end

if src.sign=0 then
   result.byte[15].nibble[0] ← (PS=0) ? 0x3 : 0xC
else
   result.byte[15].nibble[0] ← (PS=0) ? 0x7 : 0xD

result.byte[15].nibble[1] ← VR[VRB].nibble[30]

VR[VRT]    ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag | ox_flag
```

Let src be the packed decimal value in VR[VRB].

src is placed into VR[VRT] in zoned decimal format.

A valid encoding of a signed packed decimal value requires the following.

- The contents of nibble 31 (sign code) must be a value in the range 0xA to 0xF.
- The contents of each nibble 0-30 must be a value in the range 0x0 to 0x9.

Packed decimal values with sign codes of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Packed decimal values with sign codes of 0xB or 0xD are interpreted as negative values.

Values greater in magnitude than 10^{16} - 1 are too large to be represented in zoned decimal format.

For PS=0, do the following.

The leftmost nibble of each digit 0-14 of the zoned decimal result is set to 0x3.

Positive zoned decimal results are returned with a sign code of 0x3.

Negative zoned decimal results are returned with a sign code of 0x7.

For PS=1, do the following.

The leftmost nibble of each digit 0-14 of the zoned decimal result is set to 0xF.

Positive zoned decimal results are returned with a sign code of 0xC.

Negative zoned decimal results are returned with a sign code of 0xD.

For each integer value i from 0 to 15, do the following.

The rightmost nibble of each digit i of the zoned decimal result is set to the contents of nibble i+15 of src.

The result is placed into VR[VRT].

CR field 6 is set to reflect src compared to zero, including whether or not src is too large to be represented in zoned decimal format.

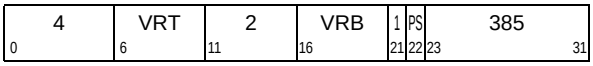
If src is an *invalid encoding* of a packed decimal value, the contents of VR[VRT] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

CR field 6

Decimal Convert From Signed Quadword VX-form

bcdcfsq. VRT,VRB,PS




```
if MSR.VEC=0 then Vector_Unavailable()

ox_flag ← (EXTS(VR[VRB]) >  10^31-1) |
          (EXTS(VR[VRB]) < -10^31-1)
lt_flag ← (EXTS(VR[VRB]) <  0)
gt_flag ← (EXTS(VR[VRB]) >  0)
eq_flag ← (EXTS(VR[VRB]) =  0)

if ox_flag=0 then
   result ← ConvertSItoBCD(EXTS(VR[VRB]),PS)
else
   result ← 0xUUUU_UUUU_UUUU_UUUU_UUUU_UUUU_UUUU_UUUU

VR[VRT]    ← ox_flag ? undefined : result

CR.bit[56] ← lt_flag
CR.bit[57] ← gt_flag
CR.bit[58] ← eq_flag
CR.bit[59] ← ox_flag
```

Let src be the signed integer value in VR[VRB].

src is placed into VR[VRT] in signed packed decimal format.

For PS=0, the contents of nibble element 31 (i.e., sign code) of VR[VRT] are set to 0xC for values greater than or equal to 0 and to 0xD for values less than 0.

For PS=1, the contents of nibble element 31 (i.e., sign code) of VR[VRT] are set to 0xF for values greater than or equal to 0 and to 0xD for values less than 0.

If the signed integer value in VR[VRB] is greater than 10^{31}-1 or less than -10^{31}-1, the value is too large to be represented in packed decimal format, and the contents of VR[VRT] are undefined.

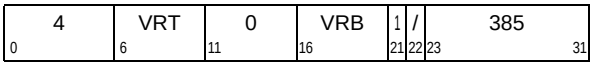
CR field 6 is set to reflect src compared to zero and whether or not src is too large in magnitude to be represented in packed decimal format.

Special Registers Altered:

CR field 6

Decimal Convert To Signed Quadword VX-form

bcdctsq. VRT,VRB




```
if MSR.VEC=0 then Vector_Unavailable()

inv_flag ← (VR[VRB].nibble[31] < 0xA)
do i = 0 to 30
    inv_flag ← inv_flag | (VR[VRB].nibble[i] > 0x9)
end

src_sign ← (VR[VRB].nibble[31] = 0xB) |
           (VR[VRB].nibble[31] = 0xD)

eq_flag  ← (VR[VRB].nibble[0:30] = 0)
lt_flag  ← (eq_flag=0) & (src_sign=1)
gt_flag  ← (eq_flag=0) & (src_sign=0)

result ← Chop(ConvertBCDtoSI(VR[VRB]), 128)

VR[VRT]    ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag
```

Let src be the packed decimal value in VR[VRB].

src is placed into VR[VRT] in signed integer format.

A valid encoding of a signed packed decimal value requires the following.

- The contents of nibble 31 (sign code) must be a value in the range 0xA to 0xF.
- The contents of each nibble 0-30 must be a value in the range 0x0 to 0x9.

Packed decimal values with sign codes of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Packed decimal values with sign codes of 0xB or 0xD are interpreted as negative values.

CR field 6 is set to reflect src compared to zero.

If src is an *invalid encoding* of a packed decimal value, the contents of VR[VRT] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

CR field 6

Vector Multiply-by-10 Unsigned Quadword VX-form

vmul10uq VRT,VRA

4	VRT	VRA	///	513
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src ← EXTZ(VR[VRA])
prod ← (src << 3) + (src << 1)
VR[VRT] ← Chop(prod, 128)
```

Let src be the unsigned integer value in VR[VRA].

The rightmost 128 bits of the product of src multiplied by the value 10 are placed into VR[VRT].

Special Registers Altered:

None

Vector Multiply-by-10 & write Carry Unsigned Quadword VX-form

vmul10cuq VRT,VRA

4	VRT	VRA	///	1
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src ← EXTZ(VR[VRA])
prod ← (src << 3) + (src << 1)
VR[VRT] ← Chop(prod>>128, 128)
```

Let src be the unsigned integer value in VR[VRA].

The product of src multiplied by the value 10 is shifted right by 128 bits. The rightmost 128 bits of the shifted result is placed into VR[VRT].

Special Registers Altered:

None

Vector Multiply-by-10 Extended Unsigned Quadword VX-form

vmul10euq VRT,VRA,VRB

4	VRT	VRA	VRB	577
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src ← EXTZ(VR[VRA])
cin ← EXTZ(VR[VRB].bit[124:127])
prod ← (src << 3) + (src << 1) + cin
VR[VRT] ← Chop(prod, 128)
```

Let src be the unsigned integer value in VR[VRA].

Let cin be the unsigned packed decimal value in bits 124:127 of VR[VRB]. Values of cin greater than 9 are undefined.

The rightmost 128 bits of the sum of cin and the product of src multiplied by the value 10 are placed into VR[VRT].

Special Registers Altered:

None

Vector Multiply-by-10 Extended & write Carry Unsigned Quadword VX-form

vmul10ecuq VRT,VRA,VRB

4	VRT	VRA	VRB	65
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

src ← EXTZ(VR[VRA])
cin ← EXTZ(VR[VRB].bit[124:127])
prod ← (src << 3) + (src << 1) + cin
VR[VRT] ← Chop(prod>>128, 128)
```

Let src be the unsigned integer value in VR[VRA].

Let cin be the unsigned packed decimal value in bits 124:127 of VR[VRA]. Values of cin greater than 9 are undefined.

The sum of cin and the product of src multiplied by the value 10 is shifted right by 128 bits. The rightmost 128 bits of the shifted result is placed into VR[VRT].

Special Registers Altered:

None

6.17.3 Decimal Integer Sign Manipulation Instructions

Decimal Copy Sign VX-form

bcdcpsgn. VRT,VRA,VRB

4	VRT	VRA	VRB	833
0	6	11	16	21 31

```
if MSR.VEC=0 then Vector_Unavailable()

inv_flag ← (VR[VRA].nibble[31] < 0xA) |
           (VR[VRB].nibble[31] < 0xA)
do i = 0 to 30
    inv_flag ← inv_flag | (VR[VRA].nibble[i] > 0x9)
                        | (VR[VRB].nibble[i] > 0x9)
end

src_sign ← (VR[VRB].nibble[31] = 0xB) |
           (VR[VRB].nibble[31] = 0xD)

eq_flag  ← (VR[VRA].nibble[0:30] = 0)
lt_flag  ← (eq_flag=0) & (src_sign=1)
gt_flag  ← (eq_flag=0) & (src_sign=0)

result.nibble[0:30] ← VR[VRA].nibble[0:30]
result.nibble[31]   ← VR[VRB].nibble[31]

VR[VRT]    ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag
```

The decimal value in VR[VRA] is placed into VR[VRT] with the sign code of the decimal value in VR[VRB].

CR field 6 is set to reflect the result compared to zero.

If either the decimal value in VR[VRA] or the decimal value in VR[VRB] is an *invalid encoding*, the contents of VR[VRT] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

CR field 6

Decimal Set Sign VX-form

bcdsetsgn. VRT,VRB,PS

4	VRT	31	VRB	1	PS	385
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

inv_flag ← (VR[VRB].nibble[31] < 0xA)
do i = 0 to 30
    inv_flag ← inv_flag | (VR[VRB].nibble[i] > 0x9)
end

src_sign ← (VR[VRB].nibble[31] = 0xB) |
           (VR[VRB].nibble[31] = 0xD)

eq_flag  ← (VR[VRB].nibble[0:30] = 0)
lt_flag  ← (eq_flag=0) & (src_sign=1)
gt_flag  ← (eq_flag=0) & (src_sign=0)

result.nibble[0:30] ← VR[VRB].nibble[0:30]
result.nibble[31]   ← (src_sign=0) ? ((PS=0) ? 0xC:0xF) : 0xD

VR[VRT]    ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag
```

Let src be the packed decimal value in VR[VRB].

Packed decimal values with sign codes of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Packed decimal values with sign codes of 0xB or 0xD are interpreted as negative values.

If src is negative, src is placed into VR[VRT] with the sign code set to 0xD.

If src is positive and PS=0, src is placed into VR[VRT] with the sign code set to 0xC.

If src is positive and PS=1, src is placed into VR[VRT] with the sign code set to 0xF.

CR field 6 is set to reflect src compared to zero.

If src is an *invalid encoding* of a packed decimal value, the contents of VR[VRT] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

CR field 6

6.17.4 Decimal Integer Shift and Round Instructions

Decimal Shift VX-form

bcds. VRT,VRA,VRB,PS

4	VRT	VRA	VRB	1	PS	193
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

n ← EXTS(VR[VRA].byte[7])

inv_flag ← (VR[VRB].nibble[31] < 0xA)
do i = 0 to 30
    inv_flag ← inv_flag | (VR[VRB].nibble[i] > 0x9)
end

src_sign ← (VR[VRB].nibble[31] = 0xB) |
           (VR[VRB].nibble[31] = 0xD)

eq_flag  ← (VR[VRB].nibble[0:30] = 0)
lt_flag  ← (eq_flag=0) & (src_sign=1)
gt_flag  ← (eq_flag=0) & (src_sign=0)

if (n >si 0) then do  // shift left
   shcnt               ← (n<32) ? n : 31
   src.nibble[0:30]    ← VR[VRB].nibble[0:30]
   src.nibble[31:61]   ← DUP(0b0000,31)
   result.nibble[0:30] ← src.data.nibble[shcnt:shcnt+30]
   ox_flag ← (shcnt > 0) & (src.nibble[0:shcnt-1] != 0)
end
else do              // shift right
   shcnt               ← ((¬n+1)<32) ? (¬n+1) : 31
   src.nibble[0:30]    ← DUP(0b0000,31)
   src.nibble[31:61]   ← VR[VRB].nibble[0:30]
   result.nibble[0:30] ← src.nibble[31-shcnt:61-shcnt]
   ox_flag ← 0b0
end

result.nibble[31] ← (src_sign=0) ? ((PS=0) ? 0xC : 0xF) : 0xD

VR[VRT]    ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag | ox_flag
```

Let n be the signed integer value in byte element 7 of VR[VRA].

Let src be the signed packed decimal value in VR[VRB].

A valid encoding of a signed packed decimal value requires the following.

- The contents of nibble 31 (sign code) must be a value in the range 0xA to 0xF.
- The contents of each nibble 0-30 must be a value in the range 0x0 to 0x9.

Packed decimal source operands with sign codes of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Packed decimal source operands with sign codes of 0xB or 0xD are interpreted as negative values.

If n is greater than zero, src is shifted left n digits. Zeros are supplied to vacated digits on the right. If any non-zero digits are shifted out, an overflow occurs.

If n is less than zero, src is shifted right -n digits. Zeros are supplied to vacated digits on the left.

If the packed decimal value in VR[VRB] is negative, the sign code of the result is set to 0b1101.

If the packed decimal value in VR[VRB] is positive, the sign code of the result is set to 0b1100 if PS=0 and is set to 0b1111 if PS=1.

The shifted result is placed into VR[VRT].

CR field 6 is set to reflect src compared to zero, including whether or not significant digits were shifted out when the shift count is positive (i.e., left shift operation).

If src is an *invalid encoding* of a packed decimal value, the contents of VR[VRT] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

CR field 6

Decimal Unsigned Shift VX-form

bcdus. VRT,VRA,VRB

4	VRT	VRA	VRB	1	/	129
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

n ← EXTS(VR[VRA].byte[7])

inv_flag ← 0
do i = 0 to 31
    inv_flag ← inv_flag | (VR[VRB].nibble[i] > 0x9)
end

eq_flag  ← (VR[VRB].nibble[0:31] = 0)
gt_flag  ← (eq_flag=0)

if (n >si 0) then do  // shift left
   shcnt ← (n<33) ? n : 32
   src.nibble[0:31] ← VR[VRB]
   src.nibble[32:63] ← DUP(0b0000,32)
   result ← src.nibble[shcnt:shcnt+31]
   ox_flag ← (shcnt > 0) & (src.nibble[0:shcnt-1] != 0)
end
else do              // shift right
   shcnt ← ((¬n+1)<33) ? (¬n+1) : 32
   src.nibble[0:31]  ← DUP(0b0000,32)
   src.nibble[32:63] ← VR[VRB]
   result ← src.nibble[32-shcnt:63-shcnt]
   ox_flag ← 0
end

VR[VRT]    ← inv_flag ? undefined : result

CR.bit[56] ← 0b0
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag | ox_flag
```

Let `n` be the signed integer value in byte element `7` of `VR[VRA]`.

Let `src` be the unsigned packed decimal value in `VR[VRB]`.

A valid encoding of an unsigned packed decimal value requires the contents of each nibble 0-31 must be a value in the range `0x0` to `0x9`.

If `n` is greater than zero, `src` is shifted left `n` digits. Zeros are supplied to vacated digits on the right. If any non-zero digits are shifted out, an overflow occurs.

If `n` is less than zero, `src` is shifted right `-n` digits. Zeros are supplied to vacated digits on the left.

The shifted result is placed into `VR[VRT]`.

CR field 6 is set to reflect `src` compared to zero, including whether or not significant digits were shifted out when the shift count is positive (i.e., left shift operation).

If `src` is an *invalid encoding* of a packed decimal value, the contents of `VR[VRT]` are undefined and `CR` field `6` is set to `0b0001`.

Special Registers Altered:

CR field 6

Decimal Shift and Round VX-form

bcdsr. VRT,VRA,VRB,PS

4	VRT	VRA	VRB	1	PS	449
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

n ← EXTS(VR[VRA].byte[7])

inv_flag ← (VR[VRB].nibble[31] < 0xA)
do i = 0 to 30
    inv_flag ← inv_flag | (VR[VRB].nibble[i] > 0x9)
end

src_sign ← (VR[VRB].nibble[31] = 0xB) |
           (VR[VRB].nibble[31] = 0xD)

eq_flag  ← (VR[VRB].nibble[0:30] = 0)
lt_flag  ← (eq_flag=0) & (src_sign=1)
gt_flag  ← (eq_flag=0) & (src_sign=0)

if (n >si 0) then do  // shift left
   shcnt ← Clamp(n, 0, 31)
   src.nibble[0:30]    ← VR[VRB].nibble[0:30]
   src.nibble[31:61]   ← DUP(0b0000,31)
   result.nibble[0:30] ← src.nibble[shcnt:shcnt+30]
   ox_flag ← (shcnt > 0) & (src.nibble[0:shcnt-1] != 0)
   g_flag  ← 0
end
else do             // shift right
   shcnt ← Clamp(¬n + 1, 0, 31)
   src.nibble[0:30]    ← DUP(0b0000,31)
   src.nibble[31:61]   ← VR[VRB].nibble[0:30]
   result.nibble[0:30] ← src.nibble[31-shcnt:61-shcnt]
   ox_flag ← 0
   g_flag  ← (shcnt > 0) & (src.nibble[62-shcnt] >=ui 5)
end
result.nibble[31] ← (src_sign=0) ? ((PS=0) ? 0xC : 0xF) : 0xD

result ← (g_flag=0) ? result : result +bcd 1

VR[VRT]    ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag | ox_flag
```

Let n be the signed integer value in byte element 7 of VR[VRA].

Let src be the signed packed decimal value in VR[VRB].

A valid encoding of a signed packed decimal source operand requires the following.

- The contents of nibble 31 (sign code) must be a value in the range 0xA to 0xF.
- The contents of each nibble 0-30 must be a value in the range 0x0 to 0x9.

Packed decimal source operands with sign codes of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Packed decimal source operands with sign codes of 0xB or 0xD are interpreted as negative values.

If n is greater than zero, src is shifted left n digits. Zeros are supplied to vacated digits on the right. If any non-zero digits are shifted out, an overflow occurs.

If n is less than zero, src is shifted right -n digits. Zeros are supplied to vacated digits on the left. If the value of the last digit shifted out on the right was greater than or equal to 5, the magnitude of the result is incremented by 1.

If src is negative, the sign code of the result is set to 0b1101.

If src is positive, the sign code of the result is set to 0b1100 if PS=0 and is set to 0b1111 if PS=1.

The shifted and rounded result is placed into VR[VRT].

CR field 6 is set to reflect src compared to zero, including whether or not significant digits were shifted out when the shift count is positive (i.e., left shift operation).

If src is an *invalid encoding* of a packed decimal value, the contents of VR[VRT] are undefined and CR field 6 is set to 0b0001.

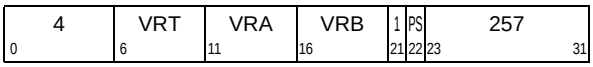
Special Registers Altered:

CR field 6

6.17.5 Decimal Integer Truncate Instructions

Decimal Truncate VX-form

bcdtrunc. VRT,VRA,VRB,PS




```
if MSR.VEC=0 then Vector_Unavailable()

inv_flag ← (VR[VRB].nibble[31] < 0xA)
do i = 0 to 30
    inv_flag ← inv_flag | (VR[VRB].nibble[i] > 0x9)
end

length  ← VR[VRA].bit[48:63]
ox_flag ← 0

src_sign ← (VR[VRB].nibble[31] = 0xB) |
           (VR[VRB].nibble[31] = 0xD)

eq_flag  ← (VR[VRB].nibble[0:30] = 0)
lt_flag  ←  src_sign & ¬eq_flag
gt_flag  ← ¬src_sign & ¬eq_flag

if length < 31 then do
   do i = 0 to 30-length
      if VR[VRB].nibble[i]!=0b0000 then ox_flag ← 1
      result.nibble[i] ← 0b0000
   end
   if length > 0 then do
      do i = 31-length to 30
         result.nibble[i] ← VR[VRB].nibble[i]
      end
   end
end
else result.nibble[0:30] ← VR[VRB].nibble[0:30]

result.nibble[31] ← (src_sign=0) ? ((PS=0) ? 0xC : 0xF) : 0xD

VR[VRT]    ← inv_flag ? undefined : result

CR.bit[56] ← inv_flag ? 0b0 : lt_flag
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag | ox_flag
```

Let length be the integer value in bits 48:63 of VR[VRA].

Let src be the signed decimal value in VR[VRB].

A valid encoding of a packed decimal source operand requires the following.

– The contents of nibble 31 (sign code) must be a value in the range 0xA to 0xF.
– The contents of each nibble 0-30 must be a value in the range 0x0 to 0x9.

Packed decimal values with sign codes of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.

Packed decimal values with sign codes of 0xB or 0xD are interpreted as negative values.

If src is negative, the sign code of the result is set to 0b1101.

If src is positive, the sign code of the result is set to 0b1100 if PS=0 and is set to 0b1111 if PS=1.

src is copied into VR[VRT] with the leftmost 31-length digits each set to 0b0000. If any of the leftmost 31-length digits of the signed decimal value in VR[VRB] are non-zero, an overflow occurs.

CR field 6 is set to reflect src compared to zero, including whether or not significant digits were truncated.

If src is an *invalid encoding* of a packed decimal value, the contents of VR[VRT] are undefined and CR field 6 is set to 0b0001.

Special Registers Altered:

CR field 6

Decimal Unsigned Truncate VX-form

bcdutrunc. VRT,VRA,VRB

4	VRT	VRA	VRB	1	/	321
0	6	11	16	21	22	23 31

```
if MSR.VEC=0 then Vector_Unavailable()

inv_flag ← 0
do i = 0 to 31
    inv_flag ← inv_flag | (VR[VRB].nibble[i] > 0x9)
end

length ← VR[VRA].bit[48:63]
ox_flag ← 0

eq_flag  ← (VR[VRB].nibble[0:31]  = 0)
gt_flag  ← (VR[VRB].nibble[0:31] != 0)

if length < 32 then do
   do i = 0 to 31-length
      if VR[VRB].nibble[i]!=0b0000 then ox_flag ← 1
      result.nibble[i] ← 0b0000
   end
   if length > 0 then do
      do i = 32-length to 31
         result.nibble[i] ← VR[VRB].nibble[i]
      end
   end
end
else result ← VR[VRB]

VR[VRT]    ← inv_flag ? undefined : result

CR.bit[56] ← 0b0
CR.bit[57] ← inv_flag ? 0b0 : gt_flag
CR.bit[58] ← inv_flag ? 0b0 : eq_flag
CR.bit[59] ← inv_flag | ox_flag
```

Let `length` be the integer value in bits 48:63 of `VR[VRA]`.

Let `src` be the unsigned decimal value in `VR[VRB]`.

A valid encoding of a packed decimal source operand requires the contents of each nibble 0-31 must be a value in the range `0x0` to `0x9`.

`src` is copied into `VR[VRT]` with the leftmost `32-length` digits each set to `0b0000`. If any of the leftmost `32-length` digits of the signed decimal value in `VR[VRB]` are non-zero, an overflow occurs.

`CR` field 6 is set to reflect `src` compared to zero, including whether or not significant digits were truncated.

If `src` is an *invalid encoding* of a packed decimal value, the contents of `VR[VRT]` are undefined and `CR` field `6` is set to `0b0001`.

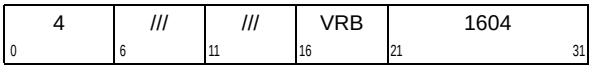
Special Registers Altered:

CR field 6

6.18 Vector Status and Control Register Instructions

Move To Vector Status and Control Register VX-form

mtvscr VRB



VSCR ← (VRB)$_{96:127}$

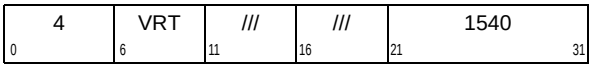
The contents of word element 3 of VRB are placed into the VSCR.

Special Registers Altered:

None

Move From Vector Status and Control Register VX-form

mfvscr VRT



VRT ← $^{96}0$ || (VSCR)

The contents of the VSCR are placed into word element 3 of VRT.

The remaining word elements in VRT are set to 0.

Special Registers Altered:

None

Chapter 7. Vector-Scalar Floating-Point Operations

7.1 Introduction

7.1.1 Overview of the Vector-Scalar Extension

Vector-Scalar Extension (VSX) provides facilities supporting vector and scalar binary floating-point operations. The following VSX features are provided to increase opportunities for vectorization.

– A unified register file, a set of Vector-Scalar Registers (VSR), supporting both scalar and vector operations is provided, eliminating the overhead of vector-scalar data transfer through storage.

– Support for word-aligned storage accesses for both scalar and vector operations is provided.

– Robust support for IEEE-754 for both vector and scalar floating-point operations is provided.

Combining the Floating-Point Registers (FPR) defined in Chapter 4. Floating-Point Facility and the Vector Registers (VR) defined in Chapter 6. Vector Facility provides additional registers to support more aggressive compiler optimizations for both vector and scalar operations.

7.1.1.1 Compatibility with Floating-Point and Decimal Floating-Point Operations

The instruction sets defined in Chapter 4. Floating-Point Facility and Chapter 5. Decimal Floating-Point retain their definition with one primary difference. The FPRs are mapped to doubleword element 0 of VSRs 0-31. The contents of doubleword 1 of the VSR corresponding to a source FPR specified by an instruction are ignored. The contents of doubleword 1 of a VSR corresponding to the target FPR specified by an instruction are undefined.

Programming Note

Application binary interfaces extended to support VSX require special care of vector data written to VSRs 0-31 (i.e., VSRs corresponding to FPRs). Legacy scalar function calls employ doubleword-based loads and stores to preserve the contents of any nonvolatile registers, This has the adverse effect of not preserving the contents of doubleword 1 of these VSRs.

7.1.1.2 Compatibility with Vector Operations

The instruction set defined in Chapter 6. Vector Facility, retains its definition with one primary difference. The VRs are mapped to VSRs 32-63.

7.2 VSX Registers

7.2.1 Vector-Scalar Registers

Sixty-four 128-bit VSRs are provided. See Figure 105 All VSX floating-point computations and other data manipulation are performed on data residing in Vector-Scalar Registers, and results are placed into a VSR.

Depending on the instruction, the contents of a VSR are interpreted as a sequence of equal-length elements (words or doublewords) or as a quadword. Each of the elements is aligned within the VSR, as shown in Figure 105. Many instructions perform a given operation in parallel on all elements in a VSR. Depending on the instruction, a word element can be interpreted as a signed integer word (SW), an unsigned integer word (UW), a logical mask value (MW), or a single-precision floating-point value (SP); a doubleword element can be interpreted as a doubleword signed integer (SD), a doubleword unsigned integer (UD), a doubleword mask (DM), or a double-precision floating-point value (DP). In the instructions descriptions, phrases like *signed integer word element* are used as shorthand for *word element, interpreted as a signed integer*.

Load and *Store* instructions are provided that transfer a byte, halfword, word, doubleword, or quadword between storage and a VSR.

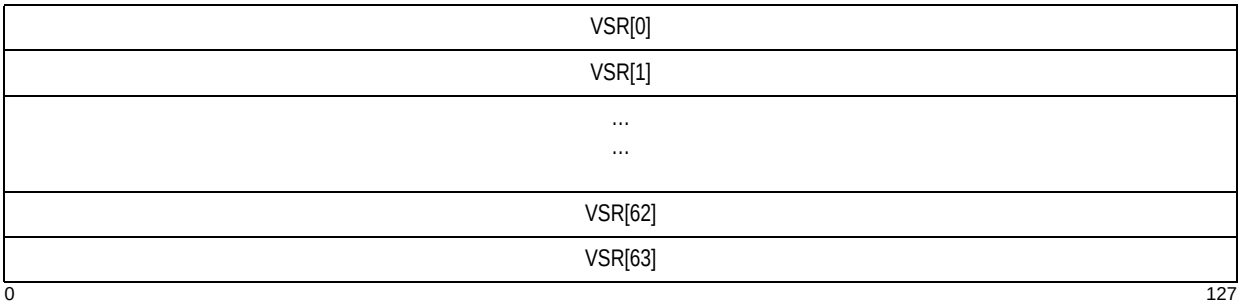


Figure 105.Vector-Scalar Registers

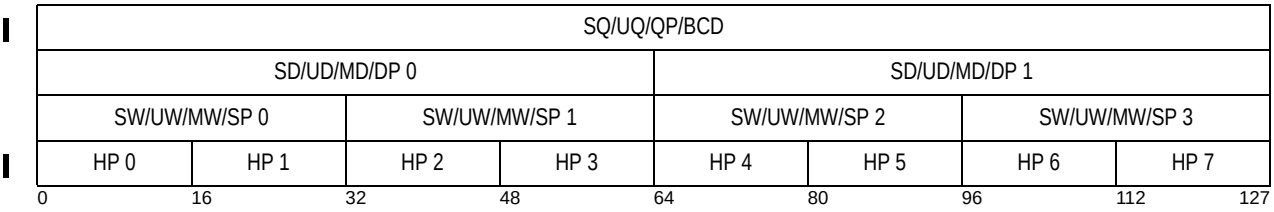


Figure 106.Vector-Scalar Register Elements

7.2.1.1 Floating-Point Registers

Chapter 4. Floating-Point Facility provides 32 64-bit FPRs. Chapter 5. Decimal Floating-Point also employs FPRs in decimal floating-point (DFP) operations. When VSX is implemented, the 32 FPRs are mapped to doubleword 0 of VSRs 0-31. For example, FPR[0] is located in doubleword element 0 of VSR[0], FPR[1] is located in doubleword element 0 of VSR[1], and so forth.

All instructions that operate on an FPR are redefined to operate on doubleword element 0 of the corresponding VSR. The contents of doubleword element 1 of the VSR corresponding to a source FPR or FPR pair for these instructions are ignored and the contents of doubleword element 1 of the VSR corresponding to the target FPR or FPR pair for these instructions are undefined.

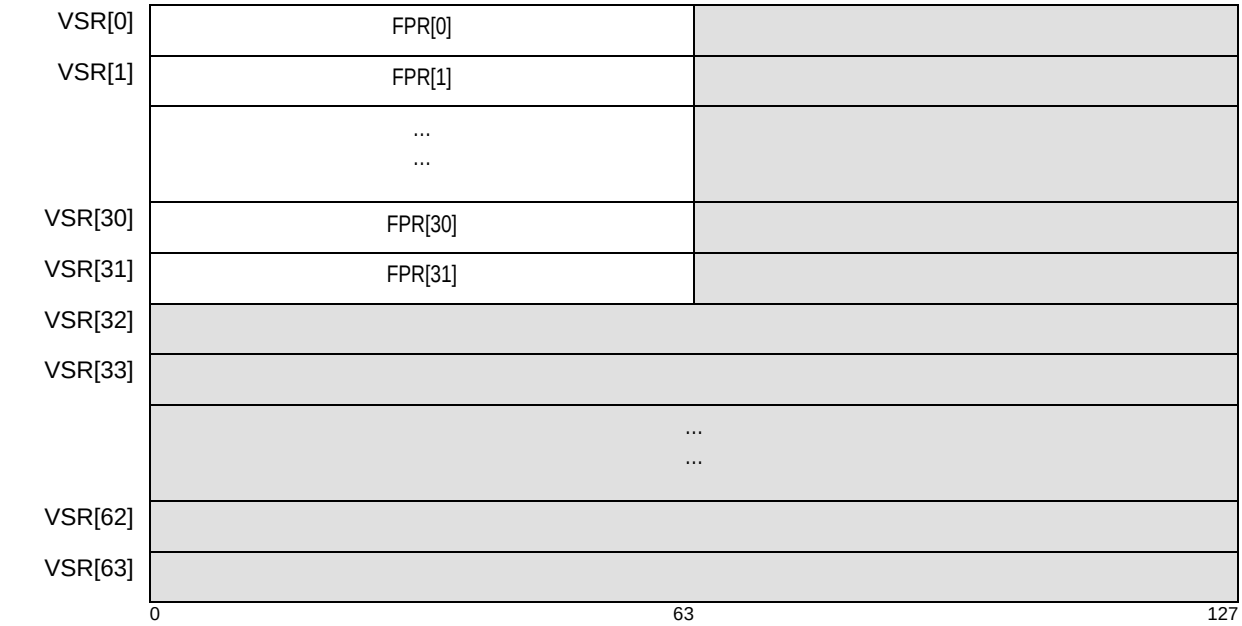


Figure 107.Floating-Point Registers as part of VSRs

7.2.1.2 Vector Registers

Chapter 6. Vector Facility provides 32 128-bit VRs. When VSX is implemented, the 32 VRs are mapped to VSRs 32-63. For example, VR[0] is located in VSR[32], VR[1] is located in VSR[33], and so forth.

All instructions that operate on a VR are redefined to operate on the corresponding VSR.

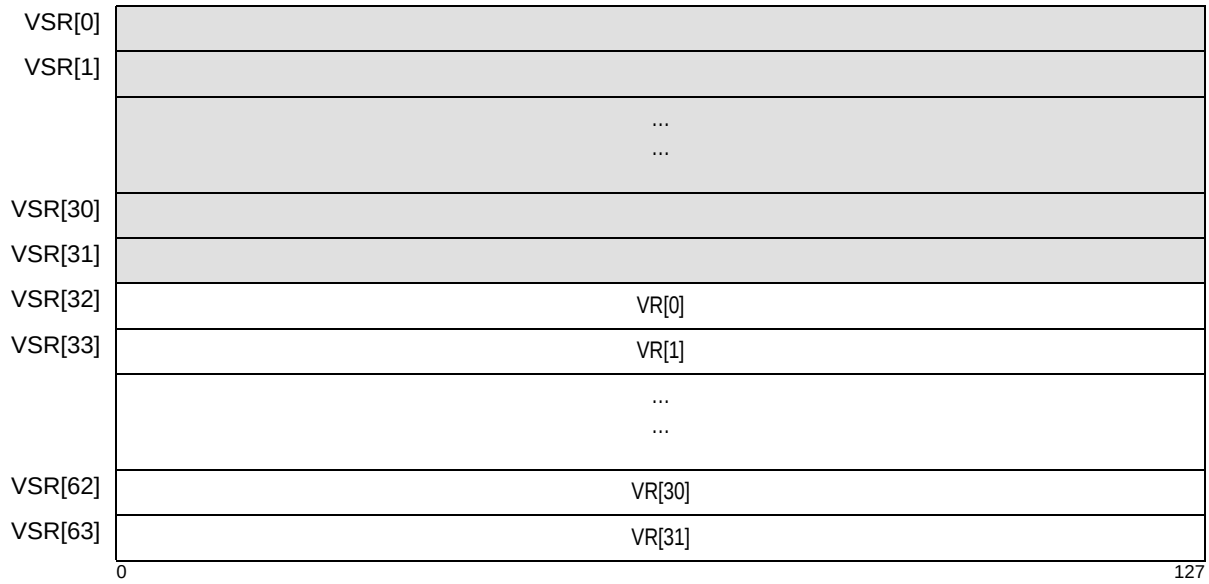


Figure 108.Vector Registers as part of VSRs

7.2.2 Floating-Point Status and Control Register

The Floating-Point Status and Control Register (FPSCR) controls the handling of floating-point exceptions and records status resulting from the floating-point operations. Bits 0:19 and 32:55 are status bits. Bits 56:63 are control bits.

The exception status bits in the FPSCR (bits 35:44, 53:55) are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an ***mcrfs***, ***mtfsfi***, ***mtfsf***, or ***mtfsb0*** instruction. The exception summary bits in the FPSCR (FX, FEX, and VX, which are bits 32:34) are not considered to be "exception status bits", and only FX is sticky.

Programming Note

Access to *Move To FPSCR* and *Move From FPSCR* instructions requires FP=1.

FEX and VX are simply the ORs of other FPSCR bits. Therefore these two bits are not listed among the FPSCR bits affected by the various instructions.

The bit definitions for the FPSCR are as follows.

Bits **Definition**

0:28 **Decimal Floating-Point Rounding Control** (DRN)
This field is not used by VSX instructions.

32 **Floating-Point Exception Summary** (FX)
Every floating-point instruction, except ***mtfsfi*** and ***mtfsf***, implicitly sets FX to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** can alter FX explicitly.

Programming Note

FX is defined not to be altered implicitly by ***mtfsfi*** and ***mtfsf*** because permitting these instructions to alter FX implicitly can cause a paradox. An example is an ***mtfsfi*** or ***mtfsf*** instruction that supplies 0 for FX and 1 for OX, and is executed when OX=0. See also the Programming Notes with the definition of these two instructions.

33 **Floating-Point Enabled Exception Summary** (FEX)
This bit is the OR of all the floating-point exception bits masked by their respective enable bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter FEX explicitly.

Bits **Definition**

34 **Floating-Point Invalid Operation Exception Summary** (VX)
This bit is the OR of all the Invalid Operation exception bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter VX explicitly.

35 **Floating-Point Overflow Exception** (OX)
This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic*, *VSX Vector Floating-Point Arithmetic*, *VSX Scalar DP-SP Conversion* or *VSX Vector DP-SP Conversion* class instruction causes an Overflow exception. See Section 7.4.3 , "Floating-Point Overflow Exception" on page 404.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

36 **Floating-Point Underflow Exception** (UX)
This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic*, *VSX Vector Floating-Point Arithmetic*, *VSX Scalar DP-SP Conversion* or *VSX Vector DP-SP Conversion* class instruction causes an Underflow exception. See Section 7.4.4 , "Floating-Point Underflow Exception" on page 409.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

37 **Floating-Point Zero Divide Exception** (ZX)
This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic* or *VSX Vector Floating-Point Arithmetic* class instruction causes an Zero Divide exception. See Section 7.4.2 , "Floating-Point Zero Divide Exception" on page 401.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

38 **Floating-Point Inexact Exception** (XX)
This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic*, *VSX Vector Floating-Point Arithmetic*, *VSX Scalar Integer Conversion*, *VSX Vector Integer Conversion*, *VSX Scalar Round to Floating-Point Integer*, or *VSX Vector Round to Floating-Point Integer* class instruction causes an Inexact exception. See Section 7.4.5 , "Floating-Point Inexact Exception" on page 414.

This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.

Bits	Definition
39	**Floating-Point Invalid Operation Exception (SNAN)** (VXSNAN) This bit is set to 1 when a *VSX Scalar Floating-Point* and *VSX Vector Floating-Point* class instruction causes an SNaN type Invalid Operation exception. See Section 7.4.1 , “Floating-Point Invalid Operation Exception” on page 390. This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.
40	**Floating-Point Invalid Operation Exception (Inf-Inf)** (VXISI) This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic* and *VSX Vector Floating-Point Arithmetic* class instruction causes an Infinity – Infinity type Invalid Operation exception. See Section 7.4.1 , “Floating-Point Invalid Operation Exception” on page 390. This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.
41	**Floating-Point Invalid Operation Exception (Inf÷Inf)** (VXIDI) This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic* and *VSX Vector Floating-Point Arithmetic* class instruction causes an Infinity ÷ Infinity type Invalid Operation exception. See Section 7.4.1 , “Floating-Point Invalid Operation Exception” on page 390. This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.
42	**Floating-Point Invalid Operation Exception (Zero÷Zero)** (VXZDZ) This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic* and *VSX Vector Floating-Point Arithmetic* class instruction causes a Zero ÷ Zero type Invalid Operation exception. See Section 7.4.1 , “Floating-Point Invalid Operation Exception” on page 390. This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.
43	**Floating-Point Invalid Operation Exception (Inf×Zero)** (VXIMZ) This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic* and *VSX Vector Floating-Point Arithmetic* class instruction causes a Infinity × Zero type Invalid Operation exception. See Section 7.4.1 , “Floating-Point Invalid Operation Exception” on page 390. This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.
44	**Floating-Point Invalid Operation Exception (Invalid Compare)** (VXVC) This bit is set to 1 when a *VSX Scalar Compare Double-Precision*, *VSX Vector Compare Double-Precision*, or *VSX Vector Compare Single-Precision* class instruction causes an Invalid Compare type Invalid Operation exception. See Section 7.4.1 , “Floating-Point Invalid Operation Exception” on page 390. This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.
45	**Floating-Point Fraction Rounded** (FR) This bit is set to 0 or 1 by *VSX Scalar Floating-Point Arithmetic*, *VSX Scalar Integer Conversion*, and *VSX Scalar Round to Floating-Point Integer* class instructions to indicate whether or not the fraction was incremented during rounding. See Section 7.3.2.6 , “Rounding” on page 381. This bit is not sticky.
46	**Floating-Point Fraction Inexact** (FI) This bit is set to 0 or 1 by *VSX Scalar Floating-Point Arithmetic*, *VSX Scalar Integer Conversion*, and *VSX Scalar Round to Floating-Point Integer* class instructions to indicate whether or not the rounded result is inexact or the instruction caused a disabled Overflow exception. See Section 7.3.2.6 on page 381. This bit is not sticky. See the definition of XX, above, regarding the relationship between FI and XX.

Bits	Definition
47:51	**Floating-Point Result Flags** (FPRF) *VSX Scalar Floating-Point Arithmetic*, *VSX Scalar DP-SP Conversion*, *VSX Scalar Convert Integer to Double-Precision*, and *VSX Scalar Round to Double-Precision Integer* class instructions set this field based on the result placed into the target register and on the target precision, except that if any portion of the result is undefined then the value placed into FPRF is undefined. For *VSX Scalar Convert Double-Precision to Integer* class instructions, the value placed into FPRF is undefined. Additional details are as follows.
47	**Floating-Point Result Class Descriptor** (C) *VSX Scalar Floating-Point Arithmetic*, *VSX Scalar DP-SP Conversion*, *VSX Scalar Convert Integer to Double-Precision*, and *VSX Scalar Round to Double-Precision Integer* class instructions set this bit with the FPCC bits, to indicate the class of the result as shown in Table 2, "Floating-Point Result Flags," on page 371.
48:51	**Floating-Point Condition Code** (FPCC) *VSX Scalar Compare Double-Precision* instruction sets one of the FPCC bits to 1 and the other three FPCC bits to 0 based on the relative values of the operands being compared. *VSX Scalar Floating-Point Arithmetic*, *VSX Scalar DP-SP Conversion*, *VSX Scalar Convert Integer to Double-Precision*, and *VSX Scalar Round to Double-Precision Integer* class instructions set the FPCC bits with the C bit, to indicate the class of the result as shown in Table 2, "Floating-Point Result Flags," on page 371. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.
48	**Floating-Point Less Than or Negative** (FL)
49	**Floating-Point Greater Than or Positive** (FG)
50	**Floating-Point Equal or Zero** (FE)
51	**Floating-Point Unordered or NaN** (FU)

Bits	Definition
52	Reserved
53	**Floating-Point Invalid Operation Exception (Software-Defined Condition)** (VXSOFT) This bit can be altered only by ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, or ***mtfsb1***. See Section 7.4.1 , "Floating-Point Invalid Operation Exception" on page 390. **Programming Note** VXSOFT can be used by software to indicate the occurrence of an arbitrary, software-defined, condition that is to be treated as an Invalid Operation exception. For example, the bit could be set by a program that computes a base 10 logarithm if the supplied input is negative.
54	**Floating-Point Invalid Operation Exception (Invalid Square Root)** (VXSQRT) This bit is set to 1 when a *VSX Scalar Floating-Point Arithmetic* or *VSX Vector Floating-Point Arithmetic* class instruction causes a Invalid Square Root type Invalid Operation exception. See Section 7.4.1 , "Floating-Point Invalid Operation Exception" on page 390. This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.
55	**Floating-Point Invalid Operation Exception (Invalid Integer Convert)** (VXCVI) This bit is set to 1 when a *VSX Scalar Convert Double-Precision to Integer*, *VSX Vector Convert Double-Precision to Integer*, or *VSX Vector Convert Single-Precision to Integer* class instruction causes a Invalid Integer Convert type Invalid Operation exception. See Section 7.4.1 , "Floating-Point Invalid Operation Exception" on page 390. This bit can be set to 0 or 1 by a *Move To FPSCR* class instruction.
56	**Floating-Point Invalid Operation Exception Enable** (VE) This bit is used by *VSX Scalar Floating-Point* and *VSX Vector Floating-Point* class instructions to enable trapping on Invalid Operation exceptions. See Section 7.4.1 , "Floating-Point Invalid Operation Exception" on page 390.

Bits	Definition
57	**Floating-Point Overflow Exception Enable** (OE) This bit is used by *VSX Scalar Floating-Point* and *VSX Vector Floating-Point* class instructions to enable trapping on Overflow exceptions. See Section 7.4.3 , "Floating-Point Overflow Exception" on page 404.
58	**Floating-Point Underflow Exception Enable** (UE) This bit is used by *VSX Scalar Floating-Point* and *VSX Vector Floating-Point* class instructions to enable trapping on Underflow exceptions. See Section 7.4.4 , "Floating-Point Underflow Exception" on page 409.
59	**Floating-Point Zero Divide Exception Enable** (ZE) This bit is used by *VSX Scalar Floating-Point* and *VSX Vector Floating-Point* class instructions to enable trapping on Zero Divide exceptions. See Section 7.4.2 , "Floating-Point Zero Divide Exception" on page 401.
60	**Floating-Point Inexact Exception Enable** (XE) This bit is used by *VSX Scalar Floating-Point* and *VSX Vector Floating-Point* class instructions to enable trapping on Inexact exceptions. See Section 7.4.5 , "Floating-Point Inexact Exception" on page 414.
61	**Floating-Point Non-IEEE Mode** (NI) Floating-point non-IEEE mode is optional. If floating-point non-IEEE mode is not implemented, this bit is treated as reserved, and the remainder of the definition of this bit does not apply. If floating-point non-IEEE mode is implemented, this bit has the following meaning. 0 The processor is not in floating-point non-IEEE mode (i.e., all floating-point operations conform to the IEEE standard). 1 The processor is in floating-point non-IEEE mode.

Bits	Definition
61	**Floating-Point Non-IEEE Mode** (NI) (*continued*) When the processor is in floating-point non-IEEE mode, the remaining FPSCR bits is permitted to have meanings different from those given in this document, and floating-point operations need not conform to the IEEE standard. The effects of executing a given floating-point instruction with NI=1, and any additional requirements for using non-IEEE mode, are implementation-dependent. The results of executing a given instruction in non-IEEE mode is permitted to vary between implementations, and between different executions on the same implementation. **Programming Note** When the processor is in floating-point non-IEEE mode, the results of floating-point operations is permitted to be approximate, and performance for these operations might be better, more predictable, or less data-dependent than when the processor is not in non-IEEE mode. For example, in non-IEEE mode an implementation is permitted to return 0 instead of a denormalized number and return a large number instead of an infinity.
62:63	**Floating-Point Rounding Control** (RN) This field is used by *VSX Scalar Floating-Point* and *VSX Vector Floating-Point* class instructions that round their result and the rounding mode is not implied by the opcode. This bit can be explicitly set or reset by a new Move To FPSCR class instruction. See Section 7.3.2.6 , "Rounding" on page 381. 00 Round to Nearest Even 01 Round toward Zero 10 Round toward +Infinity 11 Round toward -Infinity

Result Flags					Result Value Class
C	FL	FG	FE	FU	
1	0	0	0	1	Quiet NaN
0	1	0	0	1	- Infinity
0	1	0	0	0	- Normalized Number
1	1	0	0	0	- Denormalized Number
1	0	0	1	0	- Zero
0	0	0	1	0	+ Zero
1	0	1	0	0	+ Denormalized Number
0	0	1	0	0	+ Normalized Number
0	0	1	0	1	+ Infinity

Table 2. Floating-Point Result Flags

7.3 VSX Operations

7.3.1 VSX Floating-Point Arithmetic Overview

This section describes the floating-point arithmetic and exception model supported by Vector-Scalar Extension. Except for extensions to support 32-bit single-precision floating-point vector operations, the models are identical to that described in Chapter 4. Floating-Point Facility.

The processor (augmented by appropriate software support, where required) implements a floating-point system compliant with the ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic* (hereafter referred to as *the IEEE standard*). That standard defines certain required "operations" (addition, subtraction, and so on). Herein, the term, floating-point operation, is used to refer to one of these required operations and to additional operations defined (e.g., those performed by *Multiply-Add* or *Reciprocal Estimate* instructions). A Non-IEEE mode is also provided. This mode, which is permitted to produce results not in strict compliance with the IEEE standard, allows shorter latency.

Instructions are provided to perform arithmetic, rounding, conversion, comparison, and other operations in VSRs; to move floating-point data between storage and these registers.

These instructions are divided into two categories.

– computational instructions

  The computational instructions are those that perform addition, subtraction, multiplication, division, extracting the square root, rounding, conversion, comparison, and combinations of these operations. These instructions provide the floating-point operations. There are two forms of computational instructions, scalar, which perform a single floating-point operation, and vector, which perform either two double-precision floating-point operations or four single-precision operations. Computational instructions place status information into the Floating-Point Status and Control Register. They are the instructions described in Sections 7.6.1.3 through 7.6.1.8.2.

– noncomputational instructions

  The noncomputational instructions are those that perform loads and stores, move the contents of a VSR to another floating-point register possibly altering the sign, and select the value from one of two VSRs based on the value in a third VSR. The operations performed by these instructions are not considered floating-point operations. These instructions do not alter the Floating-Point Status and Control Register. They are the instructions listed in Sections 7.6.1.1, 7.6.1.2.1, and 7.6.1.12 through 7.6.1.13.

A floating-point number consists of a signed exponent and a signed significand. The quantity expressed by this number is the product of the significand and the number $2^{exponent}$. Encodings are provided in the data format to represent finite numeric values, ±Infinity, and values that are “Not a Number” (NaN). Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation. Their encoding permits a variable diagnostic information field. NaNs might be used to indicate such things as uninitialized variables and can be produced by certain invalid operations.

There is one class of exceptional events that occur during instruction execution that is unique to Vector-Scalar Extension and Floating-Point: the Floating-Point Exception. Floating-point exceptions are signaled with bits set in the FPSCR. They can cause the system floating-point enabled exception error handler to be invoked, precisely or imprecisely, if the proper control bits are set.

Floating-Point Exceptions

The following floating-point exceptions are detected by the processor:

– Invalid Operation exception (VX)
  - SNaN (VXSNAN)
  - Infinity - Infinity (VXISI)
  - Infinity÷Infinity (VXIDI)
  - Zero÷Zero (VXZDZ)
  - Infinity×Zero (VXIMZ)
  - Invalid Compare (VXVC)
  - Software-Defined Condition (VXSOFT)
  - Invalid Square Root (VXSQRT)
  - Invalid Integer Convert (VXCVI)

– Zero Divide exception (ZX)
– Overflow exception (OX)
– Underflow exception (UX)
– Inexact exception (XX)

Each floating-point exception, and each category of Invalid Operation exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. See Section 7.2.2, “Floating-Point Status and Control Register” on page 367 for a description of these exception and enable bits, and Section 7.3.3 , “VSX Floating-Point Execution Models” on page 384 for a detailed discussion of floating-point exceptions, including the effects of the enable bits.

7.3.2 VSX Floating-Point Data

7.3.2.1 Data Format

This architecture defines the representation of a floating-point value in three different binary fixed-length formats, 16-bit half-precision, 32-bit single-precision format, 64-bit double-precision format, and 128-bit quad-precision format. The half-precision format is used for half-precision data in storage and registers. The single-precision format is used for single-precision data in storage and registers. The double-precision format is used for double-precision data in storage and registers. The quad-precision format is used for quad-precision floating-point data in storage and registers.

The lengths of the exponent and the fraction fields differ between these three formats. The structure of the half-precision, single-precision, double-precision, and quad-precision formats is shown below.

Values in floating-point format are composed of three fields:

`S` sign bit
`EXP` exponent+bias
`FRACTION` fraction

Representation of numeric values in the floating-point formats consists of a sign bit (`S`), a biased exponent (`EXP`), and the fraction portion (`FRACTION`) of the significand. The significand consists of a leading implied bit concatenated on the right with the `FRACTION`. This leading implied bit is 1 for normalized numbers and 0 for denormalized (subnormal) numbers or zero and is located in the unit bit position (that is, the first bit to the left of the binary point). Values representable within the three floating-point formats can be specified by the parameters listed in Table 3.

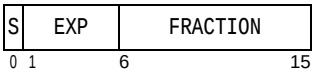


Figure 109. Floating-point half-precision format

Figure 110. Floating-point single-precision format

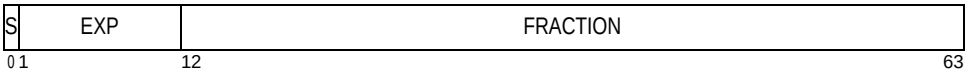


Figure 111.Floating-point double-precision format

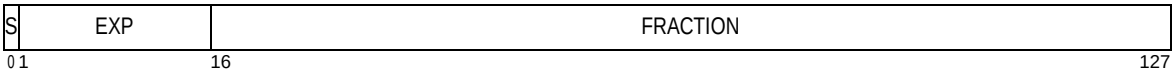


Figure 112.Floating-point quad-precision format (binary128)

	binary16	binary32	binary64	binary128
Exponent Bias	+15	+127	+1023	+16383
Maximum Exponent (Emax)	+15	+127	+1023	+16383
Minimum Exponent (Emin)	-14	-126	-1022	-16382
Widths (bits):				128
Format	16	32	64	1
Sign	1	1	1	15
Exponent	5	8	11	112
Fraction	10	23	52	113
Significand	11	24	53	
Nmax	$(2-2^{-10}) \times 2^{15} \approx 6.6 \times 10^{4}$	$(1-2^{-24}) \times 2^{128} \approx 3.4 \times 10^{38}$	$(1-2^{-53}) \times 2^{1024} \approx 1.8 \times 10^{308}$	$(1-2^{-113}) \times 2^{16384} \approx 1.2 \times 10^{4932}$
Nmin	$1.0 \times 2^{-14} \approx 6.1 \times 10^{-5}$	$1.0 \times 2^{-126} \approx 1.2 \times 10^{-38}$	$1.0 \times 2^{-1022} \approx 2.2 \times 10^{-308}$	$1.0 \times 2^{-16382} \approx 3.4 \times 10^{-4932}$
Dmin	$1.0 \times 2^{-24} \approx 6.0 \times 10^{-8}$	$1.0 \times 2^{-149} \approx 1.4 \times 10^{-45}$	$1.0 \times 2^{-1074} \approx 4.9 \times 10^{-324}$	$1.0 \times 2^{-16494} \approx 6.5 \times 10^{-4966}$

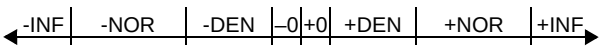
≈ Value is approximate
Dmin Smallest (in magnitude) representable denormalized number.
Nmax Largest (in magnitude) representable number.
Nmin Smallest (in magnitude) representable normalized number.

Table 3. IEEE floating-point fields

7.3.2.2 Value Representation

This architecture defines numeric and nonnumeric values representable within each of the three supported formats. The numeric values are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The nonnumeric values representable are the infinities and the Not a Numbers (NaNs). The infinities are adjoined to the real numbers, but are not numbers themselves, and the standard rules of arithmetic do not hold when they are used in an operation. They are related to the real numbers by order alone. It is possible however to define restricted operations among numbers and infinities as defined below. The relative location on the real number line for each of the defined entities is shown in Figure 113.

Figure 113.Approximation to real numbers



The NaNs are not related to the numeric values or infinities by order or value but are encodings used to convey diagnostic information such as the representation of uninitialized variables.

The following is a description of the different floating-point values defined in the architecture:

Binary floating-point numbers

Machine representable values used as approximations to real numbers. Three categories of numbers are supported: normalized numbers, denormalized numbers, and zero values.

Normalized numbers (±NOR)

These are values that have a biased exponent value in the range:

1 to 30 in half-precision format
1 to 254 in single-precision format
1 to 2046 in double-precision format
1 to 32766 in quad-precision format

They are values in which the implied unit bit is 1. Normalized numbers are interpreted as follows:

$$NOR = (-1)^{s} \times 2^{E} \times (1.fraction)$$

where s is the sign, E is the unbiased exponent, and 1.fraction is the significand, which is composed of a leading unit bit (implied bit) and a fraction part.

Zero values (±0)

These are values that have a biased exponent value of zero and a fraction value of zero. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (that is, comparison regards +0 as equal to -0).

Denormalized numbers (±DEN)

These are values that have a biased exponent value of zero and a nonzero fraction value. They are nonzero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is 0. Denormalized numbers are interpreted as follows:

$$DEN = (-1)^{s} \times 2^{Emin} \times (0.fraction)$$

where Emin is the minimum representable exponent value.

-14 for half-precision
-126 for single-precision
-1022 for double-precision
-16382 for quad-precision.

Infinities (±INF)

These are values that have the maximum biased exponent value:

31 in half-precision format
255 in single-precision format
2047 in double-precision format
32767 in quad-precision format

and a zero fraction value. They are used to approximate values greater in magnitude than the maximum normalized value.

Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and the real numbers can be related by ordering in the affine sense:

-Infinity < every finite number < +Infinity

Arithmetic on infinities is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in Section 7.4.1 , “Floating-Point Invalid Operation Exception” on page 390.

For comparison operations, +Infinity compares equal to +Infinity and -Infinity compares equal to -Infinity.

Not a Numbers (NaNs)

These are values that have the maximum biased exponent value and a nonzero fraction value. The sign bit is ignored (that is, NaNs are neither positive nor negative). If the high-order bit of the fraction field is 0, the NaN is a *Signaling NaN*; otherwise it is a *Quiet NaN*.

Signaling NaNs are used to signal exceptions when they appear as operands of computational instructions.

Quiet NaNs are used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when Invalid Operation exception is disabled (VE=0). Quiet NaNs propagate through all floating-point operations except ordered comparison and conversion to integer. Quiet NaNs do not signal exceptions, except for ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of floating-point operations, and used to convey diagnostic information to help identify results from invalid operations.

Assume the following generic arithmetic templates.

```
f(src1,src3,src2)
  ex: result = (src1 x src3) - src2

f(src1,src2)
  ex: result = src1 x src2
  ex: result = src1 + src2

f(src1)
  ex: result = f(src1)
```

When a QNaN is the result of a floating-point operation because one of the operands is a NaN or because a QNaN was generated due to a trap-disabled Invalid Operation exception, the following rule is applied to determine the NaN with the high-order fraction bit set to 1 that is to be stored as the result.

```
if src1 is a NaN
  then result = Quiet(src1)
  else if src2 is a NaN (if there is a src2)
    then result = Quiet(src2)
    else if src3 is a NaN (if there is a src3)
       then result = Quiet(src3)
       else if disabled invalid operation exception
          then result = generated QNaN
```

where `Quiet(x)` means `x` if `x` is a QNaN and `x` converted to a QNaN if `x` is an SNaN. Any instruction that generates a QNaN as the result of a disabled Invalid Operation exception generates the value,

`0x7E00` for half-precision results,

`0x7FC0_0000` for single-precision results,

`0x7FF8_0000_0000_0000` for double-precision results,

`0x7FFF_8000_0000_0000_0000_0000_0000_0000` for quad-precision results.

Note that the M-form multiply-add-type instructions use the `B` source operand to specify `src3` and the `T` target operand to specify `src2`, whereas A-form multiply-add-type instructions use the B source operand to specify `src2` and the `T` target operand to specify `src3`.

A double-precision NaN is considered to be representable in single-precision format if and only if the low-order 29 bits of the double-precision NaN's fraction are zero.

7.3.2.3 Sign of Result

The following rules govern the sign of the result of an arithmetic, rounding, or conversion operation, when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

– The sign of the result of an add operation is the sign of the operand having the larger absolute value. If both operands have the same signs, the sign of the result of an add operation is the same as the sign of the operands. The sign of the result of the subtract operation `x-y` is the same as the sign of the result of the add operation `x+(-y)`.

  When the sum of two operands with opposite sign, or the difference of two operands with the same signs, is exactly zero, the sign of the result is positive in all rounding modes except Round toward -Infinity, in which mode the sign is negative.

– The sign of the result of a multiply or divide operation is the Exclusive OR of the signs of the operands.

– The sign of the result of a *Square Root* or *Reciprocal Square Root Estimate* operation is always positive, except that the square root of `-0` is `-0` and the reciprocal square root of `-0` is -Infinity.

– The sign of the result of a *Convert From Integer* or *Round to Floating-Point Integer* operation is the sign of the operand being converted.

For the *Multiply-Add* instructions, the rules given above are applied first to the multiply operation and then to the add or subtract operation (one of the inputs to the add or subtract operation is the result of the multiply operation).

7.3.2.4 Normalization and Denormalization

The intermediate result of an arithmetic instruction can require normalization and/or denormalization as described below. Normalization and denormalization do not affect the sign of the result.

When an arithmetic or rounding instruction produces an intermediate result which carries out of the significand, or in which the significand is nonzero but has a leading zero bit, it is not a normalized number and must be normalized before it is stored. For the carry-out case, the significand is shifted right one bit, with a one shifted into the leading significand bit, and the exponent is incremented by one. For the leading-zero case, the significand is shifted left while decrementing its exponent by one for each bit shifted, until the leading significand bit becomes one. The Guard bit and the Round bit (see Section 7.3.3.1, “VSX Execution Model for IEEE Operations” on page 384) participate in the shift with zeros shifted into the Round bit. The exponent is regarded as if its range were unlimited.

After normalization, or if normalization was not required, the intermediate result can have a nonzero significand and an exponent value that is less than the minimum value that can be represented in the format specified for the result. In this case, the intermediate result is said to be “Tiny” and the stored result is determined by the rules described in Section 7.4.4 , “Floating-Point Underflow Exception” on page 409. These rules can require denormalization.

A number is denormalized by shifting its significand right while incrementing its exponent by 1 for each bit shifted, until the exponent is equal to the format’s minimum value. If any significant bits are lost in this shifting process, “Loss of Accuracy” has occurred (See Section 7.4.4 , “Floating-Point Underflow Exception” on page 409) and Underflow exception is signaled.

Engineering Note

When denormalized numbers are operands of multiply, divide, and square root operations, some implementations might prenormalize the operands internally before performing the operations.

7.3.2.5 Data Handling and Precision

Scalar double-precision floating-point data is represented in double-precision format in VSRs and storage.

Vector double-precision floating-point data is represented in double-precision format in VSRs and storage.

Scalar single-precision floating-point data is represented in double-precision format in VSRs and in single-precision format in storage.

Vector single-precision floating-point data is represented in single-precision format in VSRs and storage.

Double-precision operands may be used as input for double-precision scalar arithmetic operations.

Double-precision operands may be used as input for single-precision scalar arithmetic operations when trapping on overflow and underflow exceptions is disabled.

Single-precision operands may be used as input for double-precision and single-precision scalar arithmetic operations.

Double-precision operands may be used as input for double-precision vector arithmetic operations.

Single-precision operands may be used as input for single-precison vector arithmetic operations.

Instructions are also provided for manipulations which do not require double-precision or single-precision. In addition, instructions are provided to access an integer representation in GPRs.

Half-Precision Operands

Instructions are provided to convert between half-precision and single-precision formats for vector data in VSRs and between half-precision and double-precision formats for scalar data. Note that scalar double-precision format is identical to scalar single-precision format.

An instruction is provided to explicitly convert half-precision format operands in a VSR to single-precision format. Scalar single-precision floating-point is enabled with six types of instruction.

1. *VSX Scalar Convert Half-Precision to Double-Precision format XX2-form*

   The half-precision floating-point value in the rightmost halfword in doubleword element 0 of the source VSR is placed into the doubleword element 0 of the target VSR in double-precision format.

2. *VSX Scalar Convert with round Double-Precision to Half-Precision format XX2-form*

   The double-precision value in doubleword element `0` of the source `VSR` is rounded to to half-precision, checking the exponent for half-precision range

and handling any exceptions according to respective enable bits, and places the result into the rightmost halfword of doubleword element 0 of the target VSR in half-precision format.

Source operand values greater in magnitude than 2^{39} when Overflow is enabled (OE=1) produce undefined results because the value cannot be scaled into the half-precision normalized range.

Source operand values smaller in magnitude than 2^{-38} when Underflow is enabled (UE=1) produce undefined results because the value cannot be scaled into the half-precision normalized range.

3. *VSX Vector Convert Half-Precision to Single-Precision format XX2-form*

   The half-precision floating-point value in the rightmost halfword of each word element of the source VSR is placed into the corresponding word element of the target VSR in single-precision format.

4. *VSX Vector Convert with round Single-Precision to Half-Precision format XX2-form*

   The single-precision floating-point value in each word element i of the source VSR is rounded to half-precision and placed into the rightmost halfword of the corresponding word element of the target VSR in half-precision format.

Single-Precision Operands

For single-precision scalar data, a conversion from single-precision format to double-precision format is performed when loading from storage into a VSR and a conversion from double-precision format to single-precision format is performed when storing from a VSR to storage. No floating-point exceptions are caused by these instructions.

Instructions are provided to convert between single-precision and double-precision formats for scalar and vector data in VSRs.

An instruction is provided to explicitly convert a double format operand in a VSR to single-precision. Scalar single-precision floating-point is enabled with six types of instruction.

1. Load Scalar Single-Precision

   This form of instruction accesses a floating-point operand in single-precision format in storage, converts it to double-precision format, and loads it into a VSR. No floating-point exceptions are caused by these instructions.

2. *Scalar Round to Single-Precision*

   xsrsp rounds a double-precision operand to single-precision, checking the exponent for single-precision range and handling any exceptions according to respective enable bits, and places that operand into a VSR in double-precision format. For results produced by single-precision arithmetic instructions, single-precision loads, and other instances of ***xsrsp***, ***xsrsp*** does not alter the value. Values greater in magnitude than 2^{319} when Overflow is enabled (OE=1) produce undefined results because the value cannot be scaled back into the normalized range. Values smaller in magnitude than 2^{-318} when Underflow is enabled (UE=1) produce undefined results because the value cannot be scaled back into the normalized range.

3. Scalar Convert Single-Precision to Double-Precision

   xscvspdp accesses a floating-point operand in single-precision format from word element 0 of the source VSR, converts it to double-precision format, and places it into doubleword element 0 of the target VSR.

4. Scalar Convert Double-Precision to Single-Precision

   xscvdpsp rounds the double-precision floating-point value in doubleword element 0 of the source VSR to single-precision, and places the result into word element 0 of the target VSR in single-precision format. This function would be used to port scalar floating-point data to a format compatible for single-precision vector operations. Values greater in magnitude than 2^{319} when Overflow is enabled (OE=1) produce undefined results because the value cannot be scaled back into the normalized range. Values smaller in magnitude than 2^{-318} when Underflow is enabled (UE=1) produce undefined results because the value cannot be scaled back into the normalized range.

5. VSX Scalar Single-Precision Arithmetic

   This form of instruction takes operands from the VSRs in double format, performs the operation as if it produced an intermediate result having infinite precision and unbounded exponent range, and then coerces this intermediate result to fit in single-precision format. Status bits, in the FPSCR and optionally in the Condition Register, are set to reflect the single-precision result. The result is then placed into the target VSR in double-precision format. The result lies in the range supported by the single format.

If any input value is not representable in single-precision format and either OE=1 or UE=1, the result placed into the target VSR and the setting of status bits in the FPSCR are undefined.

For ***xsresp*** or ***xsrsqrtesp***, if the input value is finite and has an unbiased exponent greater than +127, the input value is interpreted as an Infinity.

6. Store VSX Scalar Single-Precision

   stxsspx converts a single-precision value that is in double-precision format to single-precision format and stores that operand into storage. No floating-point exceptions are caused by ***stxsspx***. (The value being stored is effectively assumed to be the result of an instruction of one of the preceding five types.)

When the result of a *Load VSX Scalar Single-Precision* (***lxsspx***), a *VSX Scalar Round to Single-Precision* (***xsrsp***), or a *VSX Scalar Single-Precision Arithmetic*[1] instruction is stored in a VSR, the low-order 29 bits of FRACTION are zero.

Programming Note

VSX Scalar Round to Single-Precision (***xsrsp***) is provided to allow value conversion from double-precision to single-precision with appropriate exception checking and rounding. ***xsrsp*** should be used to convert double-precision floating-point values to single-precision values prior to storing them into single format storage elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision load and arithmetic instructions are already single-precision values and can be stored directly into single format storage elements, or used directly as operands for single-precision arithmetic instructions, without preceding the store, or the arithmetic instruction, by an ***xsrsp***.

Programming Note

A single-precision value can be used in double-precision scalar arithmetic operations.

Except for ***xsresp*** or ***xsrsqrtesp***, any double-precision value can be used in single-precision scalar arithmetic operations when OE=0 and UE=0. When OE=1 or UE=1, or if the instruction is ***xsresp*** or ***xsrsqrtesp***, source operands must be respresentable in single-precision format.

Some implementations may execute single-precision arithmetic instructions faster than double-precision arithmetic instructions. Therefore, if double-precision accuracy is not required, single-precision data and instructions should be used.

Programming Note

Both single-precision and double-precision forms are provided for most scalar floating-point instructions. Some scalar floating-point instructions are only provided in double-precision form since their operation is identical to the equivalent scalar single-precision operation.

Of the operations for which only a double-precision form of the instruction is provided,

– instructions that return the absolute value, the negative absolute value, or the negated value (***xsnabsdp***, ***xsabsdp***, ***xsnegdp***) can be used to perform these operations on scalar single-precision operands,

– instructions that perform a comparison (***xscmpodp***, ***xscmpudp***) can be used to perform these operations on scalar single-precision operands,

– instructions that determine the maximum (***xsmaxdp***) or minimum (***xsmindp***) can be used to perform these operations on scalar single-precision operands, and

– instructions that perform an extraction or insertion of the exponent or significand (***xscmpexpdp***, ***xsiexpdp***, ***xststdcdp***, ***xststdcsp***, ***xsxexpdp***, ***xsxsigdp***) can be used to perform these operations on scalar single-precision operands.

1. *VSX Scalar Single-Precision Arithmetic* instructions:
   xsaddsp, ***xsdivsp***, ***xsmulsp***, ***xsresp***, ***xssubsp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***

Integer-Valued Operands

Instructions are provided to round floating-point operands to integer values in floating-point format. To facilitate exchange of data between the floating-point and integer processing, instructions are provided to convert between floating-point double and single-precision format and integer word and doubleword format in a VSR. Computation on integer-valued operands can be performed using arithmetic instructions of the required precision. (The results might not be integer values.) The three groups of instructions provided specifically to support integer-valued operands are described below.

1. Rounding to a floating-point integer

   VSX Scalar Round to Double-Precision Integer[1] instructions round a double-precision operand to an integer value in double-precision format. These instructions can also be used for single-precision operands represented in double-precision format.

   VSX Vector Round to Double-Precision Integer[2] instructions round each double-precision vector operand element to an integer value in double-precision format.

   VSX Vector Round to Single-Precision Integer[3] instructions round each single-precision vector operand element to an integer value in single-precision format.

   Except for ***xsrdpic***, ***xvrdpic***, and ***xvrspic***, rounding is performed using the rounding mode specified by the opcode. For ***xsrdpic***, ***xvrdpic***, and ***xvrspic***, rounding is performed using the rounding mode specified by RN.

   VSX Round to Floating-Point Integer[4] instructions can cause Invalid Operation (VXSNAN) exceptions.

   xsrdpic, ***xvrdpic***, and ***xvrspic*** can also cause Inexact exception.

   See Sections 7.3.2.6 and 7.3.3.1 for more information about rounding.

2. Converting floating-point format to integer format

   VSX Scalar Double-Precision to Integer Format Conversion[5] instructions convert a double-precision operand to 32-bit or 64-bit signed or unsigned integer format. These instructions can also be used for single-precision operands represented in double-precision format.

   VSX Vector Double-Precision to Integer Format Conversion[6] instructions convert either double-precision or single-precision vector operand elements to 32-bit or 64-bit signed or unsigned integer format.

   VSX Vector Single-Precision to Integer Doubleword Format Conversion[7] instructions converts the single-precision value in each odd-numbered word element of the source vector operand to a 64-bit signed or unsigned integer format.

   VSX Vector Single-Precision to Integer Word Format Conversion[8] instructions converts the single-precision value in each word element of the source vector operand to either a 32-bit signed or unsigned integer format.

   Rounding is performed using Round Towards Zero rounding mode. These instructions can cause Invalid Operation (VXSNAN, VXCVI) and Inexact exceptions.

3. Converting integer format to floating-point format

   VSX Scalar Integer Doubleword to Double-Precision Format Conversion[9] instructions convert a 64-bit signed or unsigned integer to a double-precision floating-point value and returns the result in double-precision format.

   VSX Scalar Integer Doubleword to Single-Precision Format Conversion[10]

1. *VSX Scalar Round to Double-Precision Integer* instructions:
   xsrdpi, ***xsrdpip***, ***xsrdpim***, ***xsrdpiz***, ***xsrdpic***
2. *VSX Vector Round to Double-Precision Integer* instructions:
   xvrdpi, ***xvrdpip***, ***xvrdpim***, ***xvrdpiz***, ***xvrdpic***
3. *VSX Vector Round to Single-Precision Integer* instructions:
   xvrspi, ***xvrspip***, ***xvrspim***, ***xvrspiz***, ***xvrspic***
4. *VSX Round to Floating-Point Integer* instructions:
   xsrdpi, ***xsrdpip***, ***xsrdpim***, ***xsrdpiz***, ***xsrdpic***, ***xvrdpi***, ***xvrdpip***, ***xvrdpim***, ***xvrdpiz***, ***xvrdpic***, ***xvrspi***, ***xvrspip***, ***xvrspim***, ***xvrspiz***, and ***xvrspic***
5. *VSX Scalar Double-Precision to Integer Format Conversion* instructions:
   xscvdpsxds, ***xscvdpsxws***, ***xscvdpuxds***, ***xscvdpuxws***
6. *VSX Vector Double-Precision to Integer Format Conversion* instructions:
   xvcvdpsxds, ***xvcvdpsxws***, ***xvcvdpuxds***, ***xvcvdpuxws***
7. *VSX Vector Single-Precision to Integer Doubleword Format Conversion* instructions:
   xvcvspsxds, ***xvcvspuxds***
8. *VSX Vector Single-Precision to Integer Word Format Conversion* instructions:
   xvcvspsxws, ***xvcvspuxws***
9. *VSX Scalar Integer Doubleword to Double-Precision Format Conversion* instructions:
   xscvsxddp, ***xscvuxddp***

instructions converts a 64-bit signed or unsigned integer to a single-precision floating-point value and returns the result in double-precision format.

VSX Vector Integer Doubleword to Double-Precision Format Conversion[1] instructions converts the 64-bit signed or unsigned integer in each doubleword element in the source vector operand to double-precision floating-point format.

VSX Vector Integer Word to Double-Precision Format Conversion[2] instructions converts the 32-bit signed or unsigned integer in each odd-numbered word element in the source vector operand to double-precision floating-point format.

VSX Vector Integer Doubleword to Single-Precision Format Conversion[3] instructions convert the 64-bit signed or unsigned integer in each doubleword element in the source vector operand to single-precision floating-point format.

VSX Vector Integer Word to Single-Precision Format Conversion[4] instructions convert the 32-bit signed or unsigned integer in each word element in the source vector operand to single-precision floating-point format.

Rounding is performed using the rounding mode specificed in RN. Because of the limitations of the source format, only an Inexact exception can be generated.

7.3.2.6 Rounding

The material in this section applies to operations that have numeric operands (that is, operands that are not infinities or NaNs). Rounding the intermediate result of such an operation can cause an Overflow exception, an Underflow exception, or an Inexact exception. The remainder of this section assumes that the operation causes no exceptions and that the result is numeric. See Section 7.3.2.2, “Value Representation” and Section 7.4, “VSX Floating-Point Exceptions” for the cases not covered here.

The floating-point arithmetic, and rounding and conversion instructions round their intermediate results. With the exception of the estimate instructions, these instructions produce an intermediate result that can be regarded as having unbounded precision and exponent range. All but two groups of these instructions normalize or denormalize the intermediate result prior to rounding and then place the final result into the target element of the target VSR in either double-precision, single-precision, or quad-precision format.

The scalar round to double-precision integer, vector round to double-precision integer, and convert double-precision to integer instructions with biased exponents ranging from 1022 through 1074 are prepared for rounding by repetitively shifting the significand right one position and incrementing the biased exponent until it reaches a value of 1075. (Intermediate results with biased exponents 1075 or larger are already integers, and with biased exponents 1021 or less round to zero.) After rounding, the final result for round to double-precision integer instructions is normalized and put in double-precision format, and, for the convert double-precision to integer instructions, is converted to a signed or unsigned integer.

The vector round to single-precision integer and vector convert single-precision to integer instructions with biased exponents ranging from 126 through 178 are prepared for rounding by repetitively shifting the significand right one position and incrementing the biased exponent until it reaches a value of 179. (Intermediate results with biased exponents 179 or larger are already integers, and with biased exponents 125 or less round to zero.) After rounding, the final result for vector round to single-precision integer is normalized and put in double-precision format, and for vector convert single-precision to integer is converted to a signed or unsigned integer.

FR and FI generally indicate the results of rounding. Each of the scalar instructions which rounds its intermediate result sets these bits. There are no vector instructions that modify FR and FI. If the fraction is incremented during rounding, FR is set to 1, otherwise FR is set to 0. If the result is inexact, FI is set to 1, otherwise FI is set to zero. The scalar round to double-precision integer instructions are exceptions to this rule, setting FR and FI to 0. The scalar double-precision estimate instructions set FR and FI to undefined values. The remaining scalar floating-point instructions do not alter FR and FI.

10. *VSX Scalar Integer Doubleword to Single-Precision Format Conversion* instructions:
xscvsxdsp, ***xscvuxdsp***
1. *VSX Vector Integer Doubleword to Double-Precision Format Conversion* instructions:
xscvsxddp, ***xscvuxddp***
2. *VSX Vector Integer Word to Double-Precision Format Conversion* instructions:
xscvsxwdp, ***xscvuxwdp***
3. *VSX Vector Integer Doubleword to Single-Precision Format Conversion* instructions:
xscvsxdsp, ***xscvuxdsp***
4. *VSX Vector Integer Word to Single-Precision Format Conversion* instructions:
xscvsxwsp, ***xscvuxwsp***

Four user-selectable rounding modes are provided through the Floating-Point Rounding Control field in the FPSCR. See Section 7.2.2, “Floating-Point Status and Control Register” on page 367. These are encoded as follows.

RN	Rounding Mode
00	Round to Nearest Even
01	Round towards Zero
10	Round towards +Infinity
11	Round towards -Infinity

A fifth rounding mode is provided in the round to floating-point integer instructions (Section 7.6.1.8.2 on page 430), Round to Nearest Away.

A sixth rounding mode is provided in the quad-precision floating-point instructions, Round to Odd.

Programming Note

Round to Odd rounding mode is useful when the results of a Quad-Precision Arithmetic instruction are required to be rounded to a shorter precision while avoiding a double rounding error. In this case, the rounding mode of the Quad-Precision Arithmetic instruction is overridden as Round To Odd by setting the RO bit in the instruction encoding to 1, then the result of that Quad-Precision Arithmetic instruction can be rounded to the desired shorter precision using the rounding mode specified in RN by following with a VSX Scalar Round Quad-Precision to Double-Extended-Precision for 15-bit exponent range and 64-bit significand precision, VSX Scalar Round Quad-Precision to Double-Precision for 11-bit exponent range and 53-bit significand precision, or VSX Scalar Round Quad-Precision to Single-Precision for 8-bit exponent range and 24-bit significand precision. For example,

```
xsaddqpo   Tx,A,B       ; use Round to Odd override (RO=1)
xsrqpxp    Tdxp,Tx      ; final QP result rounded to DXP
```

To return a quad-precision result rounded to double-precision requires a 3-instruction sequence,

```
xsaddqpo   Tx,A,B       ; use Round to Odd override (RO=1)
xscvqpdp   Temp,Tx      ; QP result rounded & converted to DP
xscvdpqp   Tdp,Temp     ; final QP result rounded to DP
```

To return a quad-precision result rounded to single-precision requires a 4-instruction sequence,

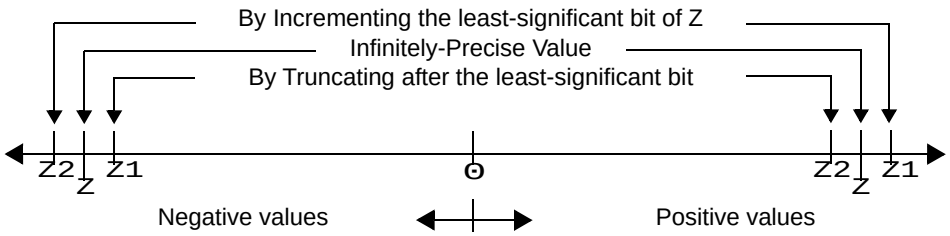
```
xsaddqpo   Tx,A,B       ; use Round to Odd override (RO=1)
xscvqpdpo  Temp,Tx      ; QP result rounded to DP using Round to Odd & converted to DP format
xsrsp      Temp,Temp    ; DP result is rounded to SP
xscvdpqp   Tsp,Temp     ; final QP result rounded to SP
```

Let Z be the intermediate arithmetic result or the operand of a convert operation. If Z can be represented exactly in the target format, the result in all rounding modes is Z as represented in the target format. If Z cannot be represented exactly in the target format, let Z1 and Z2 bound Z as the next larger and next smaller numbers representable in the target format. Then Z1 or Z2 can be used to approximate the result in the target format.

Figure 114 shows the relation of Z, Z1, and Z2 in this case. The following rules specify the rounding in the four modes.

See Section 7.3.3.1, “VSX Execution Model for IEEE Operations” on page 384 for a detailed explanation of rounding.

Figure 114 also summarizes the rounding actions for floating-point intermediate result for all supported rounding modes.



Round to Nearest Away

Choose Z if Z is representable in the target precision.

Otherwise, choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the one that is furthest away from 0.

Round to Nearest Even

Choose Z if Z is representable in the target precision.

Otherwise, choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the one that is even (least significant bit is 0).

Round to Odd

Choose Z if Z is representable in the target precision.

Otherwise, choose the value (Z1 or Z2) that is odd (least significant bit is 1).

Round toward Zero

Choose Z if Z is representable in the target precision.

Otherwise, choose the smaller in magnitude (Z1 or Z2).

Round toward +Infinity

Choose Z if Z is representable in the target precision.

Otherwise, choose Z1.

Round toward -Infinity

Choose Z if Z is representable in the target precision.

Otherwise, choose Z2.

Figure 114.Selection of Z1 and Z2

7.3.3 VSX Floating-Point Execution Models

All implementations of this architecture must provide the equivalent of the following execution models to ensure that identical results are obtained.

Special rules are provided in the definition of the computational instructions for the infinities, denormalized numbers and NaNs. The material in the remainder of this section applies to instructions that have numeric operands and a numeric result (that is, operands and result that are not infinities or NaNs), and that cause no exceptions. See Section 7.3.2.2 and Section 7.3.3 for the cases not covered here.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic makes use of two additional bits to avoid potential transient overflow and underflow conditions. One extra bit is required when denormalized double-precision numbers are prenormalized. The second bit is required to permit the computation of the adjusted exponent value in the following cases when the corresponding exception enable bit is 1:

– Underflow during multiplication using a denormalized operand.

– Overflow during division using a denormalized divisor.

– Undeflow during division using denormalized dividend and a large divisor.

The IEEE standard includes 32-bit and 64-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands.

VSX defines both scalar and vector double-precision floating-point operations to operate only on double-precision operands. VSX also defines vector single-precision floating-point operations to operate only on single-precision operands.

7.3.3.1 VSX Execution Model for IEEE Operations

IEEE-conforming significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits `0:p-1` comprise the significand of the intermediate result (where `p` is the length of the significand).

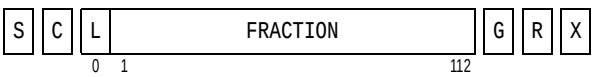


Figure 115.IEEE quad-precision (binary128) floating-point execution model (`p=113`)

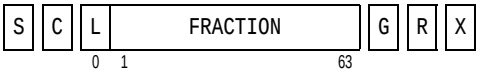


Figure 116.IEEE double-extended-precision floating-point execution model (`p=64`)

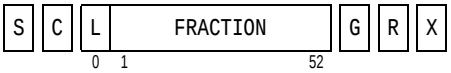


Figure 117.IEEE double-precision (binary64) floating-point execution model (`p=53`)

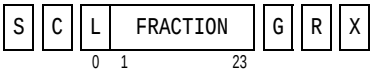


Figure 118.IEEE single-precision (binary32) floating-point execution model (`p=24`)

The S bit is the sign bit.

The C bit is the carry bit, which captures the carry out of the significand.

The L bit is the leading unit bit of the significand, which receives the implicit bit from the operand.

For the quad-precision execution model, `FRACTION` is a 112-bit field that accepts the fraction of the operand.

For the double-extended-precision execution model, `FRACTION` is a 63-bit field that accepts the fraction of the operand. This model is used only by the *VSX Scalar Round to Double-Extended-Precision* instruction.

For the double-precision execution model, `FRACTION` is a 52-bit field that accepts the fraction of the operand.

For the single-precision execution model, `FRACTION` is a 23-bit field that accepts the fraction of the operand.

The Guard (`G`), Round (`R`), and Sticky (`X`) bits are extensions to the low-order bits of the accumulator to provide the effect of an unbounded significand. The G and R bits are required for postnormalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits that appear to the low-order side of the R bit, resulting from either shifting the accumulator right or to other generation of low-order result bits. The G and R bits participate in the left shifts with zeros being shifted into the R bit. Table 4 shows the significance of the G, R, and X bits with respect to the intermediate result (IR), the representable number next lower in magnitude (NL),

and the representable number next higher in magnitude (NH).

G	R	X	Interpretation
0	0	0	IR is exact
0	0	1	IR closer to NL
0	1	0	
0	1	1	
1	0	0	IR midway between NL and NH
1	0	1	IR closer to NH
1	1	0	
1	1	1	

Table 4. Interpretation of G, R, and X bits

Table 5 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers relative to the accumulator illustrated in Figures 109, 110, 111, and 112.

Format	Guard	Round	Sticky
Double	G bit	R bit	X bit
Single	24	25	OR of bits 26:52, G, R, X

Table 5. Location of the Guard, Round, and Sticky bits in the IEEE execution model

The significand of the intermediate result is prepared for rounding by shifting its contents right, if required, until the least significant bit to be retained is in the low-order bit position of the fraction.

Six rounding modes are provided as described in Section 7.3.2.6, “Rounding” on page 381. The rules for rounding in each mode are as follows.

– ***Round to Nearest Even***
If IR is exact, choose IR.
Otherwise, if IR is closer to NL, choose NL.
Otherwise, if IR is closer to NH, choose NH.
Otherwise, if IR is midway between NL and NH, choose whichever of NL and NH is even.

– ***Round towards Zero***
If IR is exact, choose IR.
Otherwise, choose NL.

– ***Round towards +Infinity***
If IR is exact, choose IR.
Otherwise, if positive, choose NH.
Otherwise, if negative, choose NL.

– ***Round towards -Infinity***
If IR is exact, choose IR.
Otherwise, if positive, choose NL.
Otherwise, if negative, choose NH.

– ***Round to Nearest Away***
If IR is exact, choose IR.
Otherwise, if G=0, choose NL.
Otherwise, if G=1, choose NH.

– ***Round to Odd***
If IR is exact, choose IR.
Otherwise, choose NL, and if G=1, R=1, or X=1, the least-significant bit of the result is set to 1.

Four of the rounding modes are user-selectable through RN.

RN	Rounding Mode
0b00	Round to Nearest Even
0b01	Round toward Zero
0b10	Round toward +Infinity
0b11	Round toward -Infinity

Round to Nearest Away is provided in the *VSX Round to Floating-Point Integer* instructions (Section 7.6.1.8.2 on page 430).

Round to Odd is provided in the *VSX Quad-Precision Floating-Point Arithmetic* instructions as an override to the rounding mode selected by RN with the rules for rounding as follows.

If G=1, R=1, or X=1, the result is inexact.

If rounding results in a carry into C, the significand is shifted right one position and the exponent is incremented by one. This yields an inexact result, and possibly also exponent overflow. Fraction bits are stored to the target VSR.

7.3.3.2 VSX Execution Model for Multiply-Add Type Instructions

This architecture provides a special form of instruction that performs up to three operations in one instruction (a multiplication, an addition, and a negation). With this added capability comes the special ability to produce a more exact intermediate result as input to the rounder. 32-bit arithmetic is similar, except that the FRACTION field is smaller.

Multiply-add significand arithmetic is considered to be performed with a floating-point accumulator having the

following format, where bits 0:106 comprise the significand of the intermediate result.

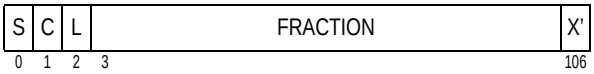


Figure 119.Multiply-add 64-bit execution model

The first part of the operation is a multiplication. The multiplication has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the `C` bit), the significand is shifted right one position, shifting the `L` bit (leading unit bit) into the most significant bit of the `FRACTION` and shifting the `C` bit (carry out) into the `L` bit. All 106 bits (`L` bit, the `FRACTION`) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount that is added to that exponent to make it equal to the other input's exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the `X′` bit. The add operation also produces a result conforming to the above model with the `X′` bit taking part in the add operation.

The result of the addition is then normalized, with all bits of the addition result, except the `X′` bit, participating in the shift. The normalized result serves as the intermediate result that is input to the rounder.

For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. Figure 6 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers in the multiply-add execution model.

Format	Guard	Round	Sticky
Double	53	54	OR of 55:105, X'
Single	24	25	OR of 26:105, X'

Table 6. Location of the Guard, Round, and Sticky bits in the multiply-add execution model

The rules for rounding the intermediate result are the same as those given in Section 7.3.3.1.

If the instruction is a negative multiply-add or negative multiply-subtract type instruction, the final result is negated.

7.4 VSX Floating-Point Exceptions

This architecture defines the following floating-point exceptions under the IEEE-754 exception model:

– Invalid Operation exception

  SNaN
  Infinity - Infinity
  Infinity÷Infinity
  Zero÷Zero
  Infinity×Zero
  Invalid Compare
  Software-Defined Condition
  Invalid Square Root
  Invalid Integer Convert

– Zero Divide exception
– Overflow exception
– Underflow exception
– Inexact exception

These exceptions, other than Invalid Operation exception resulting from a Software-Defined Condition, can occur during execution of computational instructions. An Invalid Operation exception resulting from a Software-Defined Condition occurs when a *Move To FPSCR* instruction sets VXSOFT to 1.

Each floating-point exception, and each category of Invalid Operation exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. The exception bit indicates the occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the FE0 and FE1 bits (see page 388), whether and how the system floating-point enabled exception error handler is invoked. In general, the enabling specified by the enable bit is of invoking the system error handler, not of permitting the exception to occur. The occurrence of an exception depends only on the instruction and its inputs, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an Underflow exception depends on the setting of the enable bit.

A single instruction, other than ***mtfsfi*** or ***mtfsf***, can set more than one exception bit only in the following cases:

– An Inexact exception can be set with an Overflow exception.

– An Inexact exception can be set with an Underflow exception.

– An Invalid Operation exception (SNaN) is set with an Invalid Operation exception (Infinity×0) for multiply-add class instructions for which the values being multiplied are infinity and zero and the value being added is an SNaN.

– An Invalid Operation exception (SNaN) can be set with an Invalid Operation exception (Invalid Compare) for ordered comparison instructions.

– An Invalid Operation exception (SNaN) can be set with an Invalid Operation exception (Invalid Integer Convert) for convert to integer instructions.

When an exception occurs, the writing of a result to the target register can be suppressed, or a result can be delivered, depending on the exception.

The writing of a result to the target register is suppressed for the certain kinds of exceptions, based on whether the instruction is a vector or a scalar instruction, so that there is no possibility that one of the operands is lost. For other kinds of exceptions and also depending on whether the instruction is a vector or a scalar instruction, a result is generated and written to the destination specified by the instruction causing the exception. The result can be a different value for the enabled and disabled conditions for some of these exceptions. Table 7 lists the types of exceptions and indicates whether a result is written to the target VSR or suppressed.

On exception type...	Scalar Instruction Results	Vector Instruction Results
Enabled Invalid Operation	suppressed	suppressed
Enabled Zero Divide	suppressed	suppressed
Enabled Overflow	written	suppressed
Enabled Underflow	written	suppressed
Enabled Inexact	written	suppressed
Disabled Invalid Operation	written	written

Table 7. Exception Types Result Suppression

On exception type...	Scalar Instruction Results	Vector Instruction Results
Disabled Zero Divide	written	written
Disabled Overflow	written	written
Disabled Underflow	written	written
Disabled Inexact	written	written

Table 7. Exception Types Result Suppression

The subsequent sections define each of the floating-point exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of *traps* and *trap handlers*. In this architecture, an FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the trap enabled case; the expectation is that the exception is detected by software, which revises the result. An FPSCR exception enable bit of 0 causes generation of the default result value specified for the trap disabled (or no trap occurs or trap is not implemented) case. The expectation is that the exception is not detected by software, which uses the default result. The result to be delivered in each case for each exception is described in the following sections.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is required for all exceptions, all FPSCR exception enable bits must be set to 0, and Ignore Exceptions Mode (see below) should be used. In this case, the system floating-point enabled exception error handler is not invoked, even if floating-point exceptions occur: software can inspect the FPSCR exception bits, if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to 1, and a mode other than Ignore Exceptions Mode must be used. In this case, the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs. The system floating-point enabled exception error handler is also invoked if a *Move To FPSCR* instruction causes an exception bit and the corresponding enable bit both to be 1. The *Move To FPSCR* instruction is considered to cause the enabled exception.

The FE0 and FE1 bits control whether and how the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs. The location of these bits and the requirements for altering them are described in Book III. The system floating-point enabled exception error handler is never invoked because of a disabled floating-point exception. The effects of the four possible settings of these bits are as follows.

FE0 FE1 Description

0 0 **Ignore Exceptions Mode**
Floating-point exceptions do not cause the system floating-point enabled exception error handler to be invoked.

0 1 **Imprecise Nonrecoverable Mode**
The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. It may not be possible to identify the excepting instruction or the data that caused the exception. Results produced by the excepting instruction might have been used by or might have affected subsequent instructions that are executed before the error handler is invoked.

1 0 **Imprecise Recoverable Mode**
The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. Sufficient information is provided to the error handler for it to identify the excepting instruction, the operands, and correct the result. No results produced by the excepting instruction have been used by or affected subsequent instructions that are executed before the error handler is invoked.

1 1 **Precise Mode**
The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.

In all cases, the question of whether a floating-point result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by the value of the FE0 and FE1 bits.

In all cases in which the system floating-point enabled exception error handler is invoked, all instructions before the instruction at which the system floating-point enabled exception error handler is invoked have been completed, and no instruction after the instruction at which the system floating-point enabled exception error handler is invoked has begun execution. The instruction at which the system floating-point enabled exception error handler is invoked has completed if it is the excepting instruction,

and there is only one such instruction. Otherwise, it has not begun execution, or has been partially executed in some cases, as described in Book III.

Programming Note

In any of the three non-Precise modes, a *Floating-Point Status and Control Register* instruction can be used to force any exceptions, because of instructions initiated before the *Floating-Point Status and Control Register* instruction, to be recorded in the FPSCR. (This forcing is superfluous for Precise Mode.)

In both Imprecise modes, a *Floating-Point Status and Control Register* instruction can be used to force any invocations of the system floating-point enabled exception error handler that result from instructions initiated before the *Floating-Point Status and Control Register* instruction to occur. This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.

The last sentence of the paragraph preceding this Programming Note can apply only in the Imprecise modes, or if the mode has just been changed from Ignore Exceptions Mode to some other mode. It always applies in the latter case.

To obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines.

– If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used with all FPSCR exception enable bits set to 0.

– If the IEEE default results are not acceptable to the application, Imprecise Nonrecoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with FPSCR exception enable bits set to 1 for those exceptions for which the system floating-point enabled exception error handler is to be invoked.

– Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to 1.

– Precise Mode can degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

7.4.1 Floating-Point Invalid Operation Exception

7.4.1.1 Definition

An Invalid Operation exception occurs when an operand is invalid for the specified operation. The invalid operations are:

SNaN
Any floating-point operation on a Signaling NaN.

Infinity–Infinity
Magnitude subtraction of infinities.

Infinity÷Infinity
Floating-point division of infinity by infinity.

Zero÷Zero
Floating-point division of zero by zero.

Infinity × Zero
Floating-point multiplication of infinity by zero.

Invalid Compare
Floating-point ordered comparison involving a NaN.

Invalid Square Root
Floating-point square root or reciprocal square root of a nonzero negative number.

Invalid Integer Convert
Floating-point-to-integer convert involving a number too large in magnitude to be represented in the target format, or involving an infinity or a NaN.

An Invalid Operation exception also occurs when an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction is executed that sets VXSOFT to 1 (Software-Defined Condition).

The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the FPSCR.

7.4.1.2 Action for VE=1

When Invalid Operation exception is enabled (`VE=1`) and an Invalid Operation exception occurs, the following actions are taken:

For *VSX Scalar Floating-Point Arithmetic*, *VSX Scalar DP-SP Conversion*, *VSX Scalar Convert Floating-Point to Integer*, and *VSX Scalar Round to Floating-Point Integer* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

`VXSNAN`	(if SNaN)
`VXISI`	(if Infinity–Infinity)
`VXIDI`	(if Infinity÷Infinity)
`VXZDZ`	(if Zero÷Zero)
`VXIMZ`	(if Infinity×Zero)
`VXSQRT`	(if Invalid Square Root)
`VXCVI`	(if Invalid Integer Convert)

2. Update of `VSR[XT]` is suppressed.

3. `FR` and `FI` are set to zero.

4. `FPRF` is unchanged.

For *VSX Scalar Floating-Point Compare* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

`VXSNAN`	(if SNaN)
`VXVC`	(if Invalid Compare)

2. `FR`, `FI`, and `C` are unchanged.

3. FPCC is set to reflect unordered.

For any of the following instructions,

VSX Scalar Quad-Precision Arithmetic instructions:
xsaddqp[***o***], ***xsdivqp***[***o***], ***xsmulqp***[***o***], ***xssqrtqp***[***o***], ***xssubqp***[***o***]
xsmaddqp[***o***], ***xsmsubqp***[***o***], ***xsnmaddqp***[***o***], ***xsnmsubqp***[***o***]

VSX Scalar Quad-Precision Convert to Integer instructions:
xscvqpsdz, ***xscvqpswz***, ***xscvqpudz***, ***xscvqpuwz***

VSX Scalar Round Quad-Precision to Double-Extended-Precision (***xsrqpxp***)
VSX Scalar Round to Quad-Precision Integer (***xsrqpi***)
VSX Scalar Convert with round Quad-Precision to Double-Precision format [*using round to Odd*] (***xscvqpdp***[***o***])

do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXISI (if Infinity - Infinity)
   VXIDI (if Infinity ÷ Infinity)
   VXZDZ (if Zero ÷ Zero)
   VXIMZ (if Infinity × Zero)
   VXSQRT (if Invalid Square Root)
   VXCVI (if Invalid Integer Convert)

2. VSR[VRT+32] is not modified.
3. FR and FI are set to zero. FPRF is not modified.

For any of the following instructions,

VSX Scalar Compare Ordered Quad-Precision (***xscmpoqp***)
VSX Scalar Compare Unordered Quad-Precision (***xscmpuqp***)

do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXVC (if Invalid Compare)

2. FR, FI, and C are not modified. FPCC is set to reflect unordered.

For any of the following instructions,

VSX Scalar Convert Half-Precision to Double-Precision format (***xscvhpdp***)
VSX Scalar Convert with round Double-Precision to Half-Precision format (***xscvdphp***)

do the following.

1. VXSNAN is set to 1.
2. VSR[XT] is not modified.
3. FR and FI are set to 0. FPRF is not modified.

For any of the following instructions,

VSX Vector Convert Half-Precision to Single-Precision format (***xvcvhpsp***)
VSX Vector Convert with round Single-Precision to Half-Precision format (***xvcvsphp***)

do the following.

1. VXSNAN is set to 1.
2. VSR[XT] is not modified.
3. FR, FI, and FPRF are not modified.

For any of the following instructions,

VSX Vector Floating-Point Arithmetic instructions:
VSX Vector Floating-Point Compare instructions:
VSX Vector DP-SP Conversion instructions:
VSX Vector Convert Floating-Point to Integer instructions:
VSX Vector Round to Floating-Point Integer instructions:

do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

VXSNAN	(if SNaN)
VXISI	(if Infinity – Infinity)
VXIDI	(if Infinity ÷ Infinity)
VXZDZ	(if Zero ÷ Zero)
VXIMZ	(if Infinity × Zero)
VXVC	(if Invalid Compare)
VXSQRT	(if Invalid Square Root)
VXCVI	(if Invalid Integer Convert)

2. Update of VSR[XT] is suppressed for all vector elements.

3. FR and FI are unchanged.

4. FPRF is unchanged.

7.4.1.3 Action for VE=0

When Invalid Operation exception is disabled (VE=0) and an Invalid Operation exception occurs, the following actions are taken:

For the *VSX Scalar Convert with round Double-Precision to Single-Precision format* (***xscvdpsp***) instruction:

1. VXSNAN is set to 1.

2. The single-precision representation of a Quiet NaN is placed into word element 0 of VSR[XT]. The contents of word elements 1-3 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is set to indicate the class of the result (Quiet NaN).

For the *VSX Vector Single-Precision Arithmetic* instructions, *VSX Vector Single-Precision Maximum/Minimum* instructions, the *VSX Vector Convert with round Double-Precision to Single-Precision format* (***xvcvdpsp***) instruction, and the *VSX Vector Round to Single-Precision Integer* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXISI (if Infinity – Infinity)
   VXIDI (if Infinity ÷ Infinity)
   VXZDZ (if Zero ÷ Zero)
   VXIMZ (if Infinity × Zero)
   VXSQRT (if Invalid Square Root)

2. The single-precision representation of a Quiet NaN is placed into its respective word element of VSR[XT].

3. FR, FI, and FPRF are not modified.

For the *VSX Scalar Double-Precision Arithmetic* instructions, *VSX Scalar Double-Precision Maximum/Minimum* instructions, the *VSX Scalar Convert Single-Precision to Double-Precision format* (***xscvspdp***) instruction, and the *VSX Scalar Round to Double-Precision Integer* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXISI (if Infinity – Infinity)
   VXIDI (if Infinity ÷ Infinity)
   VXZDZ (if Zero ÷ Zero)
   VXIMZ (if Infinity × Zero)
   VXSQRT (if Invalid Square Root)

2. The double-precision representation of a Quiet NaN is placed into doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is set to indicate the class of the result (Quiet NaN).

For any of the following instructions,

VSX Scalar Quad-Precision Arithmetic instructions:
xsaddqp[***o***], ***xsdivqp***[***o***], ***xsmulqp***[***o***], ***xssqrtqp***[***o***], ***xssubqp***[***o***]
xsmaddqp[***o***], ***xsmsubqp***[***o***], ***xsnmaddqp***[***o***], ***xsnmsubqp***[***o***]

VSX Scalar Quad-Precision Round to Integer (***xsrqpi***)

do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXISI (if Infinity - Infinity)
   VXIDI (if Infinity ÷ Infinity)
   VXZDZ (if Zero ÷ Zero)
   VXIMZ (if Infinity × Zero)
   VXSQRT (if Invalid Square Root)

2. The quad-precision representation of a Quiet NaN is placed into VSR[VRT+32].

3. FR and FI are set to 0. FPRF is set to indicate the class of the result (Quiet NaN).

For *VSX Scalar Round Quad-Precision to Double-Extended-Precision* (***xsrqpxp***), do the following.

1. VXSNAN is set to 1.

2. The Quiet NaN is placed into VSR[VRT+32] in quad-precision format.

3. FR and FI are set to 0. FPRF is set to indicate the class of the result (Quiet NaN).

For any of the following instructions,

VSX Scalar Compare Ordered Quad-Precision (***xscmpoqp***)
VSX Scalar Compare Unordered Quad-Precision (***xscmpoqp***)

do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXVC (if Invalid Compare)

2. FR, FI and C are unchanged. FPCC is set to reflect unordered.

For *VSX Scalar Convert with round Quad-Precision to Double-Precision format* [using round *to Odd*] (***xscvqpdp***[***o***]), do the following.

1. VXSNAN is set to 1.

2. The double-precision Quiet NaN result is placed into doubleword element 0 of VSR[VRT+32] in double-precision format.

   0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].

3. FR and FI are set to 0. FPRF is set to indicate the class of the result (Quiet NaN).

For *VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format* (***xscvqpsdz***), do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x7FFF_FFFF_FFFF_FFFF is placed into doubleword element 0 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a positive number or +Infinity.

   0x8000_0000_0000_0000 is placed into doubleword element 0 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a negative number, -Infinity, or NaN.

   0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].

3. FR and FI are set to 0. FPRF is undefined.

For *VSX Scalar Convert with round to zero Quad-Precision to Signed Word format* (***xscvqpswz***), do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)

VXCVI (if Invalid Integer Convert)

2. 0x7FFF_FFFF is placed into word element 1 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a positive number or +Infinity.

   0x8000_0000 is placed into word element 1 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a negative number, -Infinity, or NaN.

   0x0000_0000 is placed into word elements 0, 2, and 3 of VSR[VRT+32].

3. FR and FI are set to 0. FPRF is undefined.

For *VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format* (***xscvqpudz***), do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0xFFFF_FFFF_FFFF_FFFF is placed into doubleword element 0 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a positive number or +Infinity.

   0x0000_0000_0000_0000 is placed into doubleword element 0 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a negative number, -Infinity, or NaN.

   0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].

3. FR and FI are set to 0. FPRF is undefined.

For *VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format* (***xscvqpuwz***), do the following.

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0xFFFF_FFFF is placed into word element 1 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a positive number or +Infinity.

   0x0000_0000 is placed into word element 1 of VSR[VRT+32] if the quad-precision operand in VSR[VRB+32] is a negative number, -Infinity, or NaN.

   0x0000_0000 is placed into word elements 0, 2, and 3 of VSR[VRT+32].

3. FR and FI are set to 0. FPRF is undefined.

For *VSX Scalar Convert with round Double-Precision to Half-Precision format* (***xscvdphp***), do the following.

1. VXSNAN is set to 1.

2. The half-precision representation of a Quiet NaN is placed into the rightmost halfword of doubleword element 0 of VSR[XT]. The contents of the leftmost 3 halfwords of doubleword element 0 of VSR[XT] are set to 0. The contents of doubleword element 1 of VSR[XT] are undefined.

3. FR and FI are set to 0. FPRF is set to indicate the class of the result (Quiet NaN).

For *VSX Scalar Convert Half-Precision to Double-Precision format* (***xscvhpdp***), do the following.

1. VXSNAN is set to 1.

2. The double-precision representation of a Quiet NaN is placed into doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.

3. FR and FI are set to 0. FPRF is set to indicate the class of the result (Quiet NaN).

For the *VSX Vector Double-Precision Arithmetic* instructions, *VSX Vector Double-Precision Maximum/Minimum* instructions, the *VSX Vector Convert Single-Precision to Double-Precision format* (***xvcvspdp***) instruction, and the *VSX Vector Round to Double-Precision Integer* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXISI (if Infinity – Infinity)
   VXIDI (if Infinity ÷ Infinity)
   VXZDZ (if Zero ÷ Zero)
   VXIMZ (if Infinity × Zero)
   VXSQRT (if Invalid Square Root)

2. The double-precision representation of a Quiet NaN is placed into its respective doubleword element of VSR[XT].

3. FR, FI, and FPRF are not modified.

For the *VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format* (***xscvdpsxd***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x7FFF_FFFF_FFFF_FFFF is placed into doubleword element 0 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

   0x8000_0000_0000_0000 is placed into doubleword element 0 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of doubleword element 1 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is undefined.

For the *VSX Scalar Convert with round to zero Double-Precision to Unsigned Doubleword format* (***xscvdpuxd***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0xFFFF_FFFF_FFFF_FFFF is placed into doubleword element 0 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

   0x0000_0000_0000_0000 is placed into doubleword element 0 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, -Infinity, or NaN.

The contents of doubleword element 1 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is undefined.

For the *VSX Scalar Convert with round to zero Double-Precision to Signed Word format* (***xscvdpsxw***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x7FFF_FFFF is placed into word element 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

   0x8000_0000 is placed into word element 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is undefined.

For the *VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format* (***xscvdpuxw***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0xFFFF_FFFF is placed into word element 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a positive number or +Infinity.

   0x0000_0000 is placed into word element 1 of VSR[XT] if the double-precision operand in doubleword element 0 of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is undefined.

For the *VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format* (***xvcvdpsxd***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x7FFF_FFFF_FFFF_FFFF is placed into doubleword element i of VSR[XT] if the double-precision operand in the corresponding doubleword element of VSR[XB] is a positive number or +Infinity.

0x8000_0000_0000_0000 is placed into its respective doubleword element i of VSR[XT] if the double-precision operand in the corresponding doubleword element of VSR[XB] is a negative number, -Infinity, or NaN.

3. FR, FI, and FPRF are not modified.

For the *VSX Vector Convert with round to zero Double-Precision to Unsigned Doubleword format* (***xvcvdpuxd***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0xFFFF_FFFF_FFFF_FFFF is placed into doubleword element i of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a positive number or +Infinity.

   0x0000_0000_0000_0000 is placed into doubleword element i of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a negative number, -Infinity, or NaN.

3. FR, FI, and FPRF are not modified.

For the *VSX Vector Convert with round to zero Double-Precision to Signed Word format* (***xvcvdpsxw***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x7FFF_FFFF is placed intoword element i×2 of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a positive number or +Infinity.

   0x8000_0000 is placed into word element i×2 of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of word element i×2+1 of VSR[XT] are undefined.

3. FR, FI, and FPRF are not modified.

For the *VSX Vector Convert with round to zero Double-Precision to Unsigned Word format* (***xvcvdpuxw***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0xFFFF_FFFF is placed into word element i×2 of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a positive number or +Infinity.

   0x0000_0000 is placed into word element i×2 of VSR[XT] if the double-precision operand in doubleword element i of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of word element i×2+1 of VSR[XT] are undefined.

3. FR, FI, and FPRF are not modified.

For the *VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format* (***xvcvspsxd***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x7FFF_FFFF_FFFF_FFFF is placed into doubleword element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a positive number or +Infinity.

   0x8000_0000_0000_0000 is placed into doubleword element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a negative number, -Infinity, or NaN.

3. FR, FI, and FPRF are not modified.

For the *VSX Vector Convert with round to zero Single-Precision to Unsigned Doubleword format* (***xvcvspuxd***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0xFFFF_FFFF_FFFF_FFFF is placed into doubleword element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a positive number or +Infinity.

   0x0000_0000_0000_0000 is placed into doubleword element i of VSR[XT] if the single-precision operand in word element i×2 of VSR[XB] is a negative number, -Infinity, or NaN.

3. FR, FI, and FPRF are not modified.

For the *VSX Vector Convert with round to zero Single-Precision to Signed Word format* (***xvcvspsxw***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x7FFF_FFFF is placed into word element i of VSR[XT] if the single-precision operand in word element i of VSR[XB] is a positive number or +Infinity.

   0x8000_0000 is placed into word element i of VSR[XT] if the single-precision operand in word element i of VSR[XB] is a negative number, -Infinity, or NaN.

   The contents of word element 2×i+1 of VSR[XT] are undefined.

3. FR, FI, and FPRF are not modified.

For the *VSX Vector Convert with round to zero Single-Precision to Unsigned Word format* (***xvcvspuxw***) instruction:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0xFFFF_FFFF is placed into word element i of VSR[XT] if the single-precision operand in the corresponding word element 2×i of VSR[XB] is a positive number or +Infinity.

0x0000_0000 is placed into word element i of VSR[XT] if the single-precision operand in word element 2×i of VSR[XB] is a negative number, -Infinity, or NaN.

The contents of word element 2×i+1 of VSR[XT] are undefined.

3. FR, FI, and FPRF are not modified.

For the *VSX Scalar Floating-Point Compare* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. FR, FI and C are unchanged.

3. FPCC is set to reflect unordered.

For the VSX *Vector Compare Single-Precision* instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x0000_0000 is placed into its respective word element of VSR[XT].

3. FR, FI, and FPRF are not modified.

For the vector double-precision compare instructions:

1. One or two of the following Invalid Operation exceptions are set to 1.

   VXSNAN (if SNaN)
   VXCVI (if Invalid Integer Convert)

2. 0x0000_0000_0000_0000 is placed into its respective doubleword element of VSR[XT].

3. FR, FI, and FPRF are not modified.

For *VSX Vector Convert with round Single-Precision to Half-Precision format* (***xscvsphp***), do the following.

1. VXSNAN is set to 1.

2. The half-precision representation of a Quiet NaN is placed into the rightmost halfword of its respective word element of VSR[XT]. The contents of the leftmost halfword of its respective word element of VSR[XT] are set to 0.

3. FR, FI, and FPRF are not modified.

For *VSX Vector Convert Half-Precision to Single-Precision format* (***xscvhpsp***), do the following.

1. VXSNAN is set to 1.

2. The half-precision representation of a Quiet NaN is placed into the rightmost halfword of its respective word element of VSR[XT]. The contents of the leftmost halfword of its respective word element of VSR[XT] are set to 0.

3. FR, FI, and FPRF are not modified.

7.4.2 Floating-Point Zero Divide Exception

7.4.2.1 Definition

A Zero Divide exception occurs when a *VSX Floating-Point Divide*[1] instruction is executed with a zero divisor value and a finite nonzero dividend value.

A Zero Divide exception also occurs when a *VSX Floating-Point Reciprocal Estimate*[2] instruction or a *VSX Floating-Point Reciprocal Square Root Estimate*[3] instruction is executed with an operand value of zero.

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

7.4.2.2 Action for ZE=1

When Zero Divide exception is enabled (ZE=1) and a Zero Divide exception occurs, the following actions are taken:

For any of the following instructions,

VSX Scalar Floating-Point Divide instructions:
xsdivdp, ***xsdivsp***

VSX Scalar Floating-Point Reciprocal Estimate instructions
xsredp, ***xsresp***

VSX Scalar Floating-Point Reciprocal Square Root Estimate instructions
xsrsqrtedp, ***xsrsqrtesp***

do the following.

1. ZX is set to 1.
2. Update of VSR[XT] is suppressed.
3. FR and FI are set to 0.
4. FPRF is unchanged.

For *VSX Scalar Divide Quad-Precision* (***xsdivqp***), do the following.

1. ZX is set to 1.
2. Update of VSR[VRT+32] is suppressed.
3. FR and FI are set to 0. FPRF is not modified.

For any of the following instructions,

VSX Vector Floating-Point Divide instructions
xsdivdp, ***xsdivsp***, ***xvdivdp***, ***xvdivsp***

VSX Vector Floating-Point Reciprocal Estimate instructions
xsredp, ***xsresp***, ***xvredp***, ***xvresp***

VSX Vector Floating-Point Reciprocal Square Root Estimate instructions
xsrsqrtedp, ***xsrsqrtesp***, ***xvrsqrtedp***, ***xvrsqrtesp***

1. *VSX Vector Floating-Point Divide* instructions:
   xsdivdp, ***xsdivsp***, ***xvdivdp***, ***xvdivsp***
2. *VSX Floating-Point Reciprocal Estimate* instructions:
   xsredp, ***xsresp***, ***xvredp***, ***xvresp***
3. *VSX Floating-Point Reciprocal Square Root Estimate* instructions:
   xsrsqrtedp, ***xsrsqrtesp***, ***xvrsqrtedp***, ***xvrsqrtesp***

do the following.

1. ZX is set to 1.
2. Update of VSR[XT] is suppressed for all vector elements.
3. FR and FI are unchanged.
4. FPRF is unchanged.

7.4.2.3 Action for ZE=0

When Zero Divide exception is disabled (ZE=0) and a Zero Divide exception occurs, the following actions are taken:

For *VSX Scalar Floating-Point Divide*[1] instructions, do the following.

1. ZX is set to 1.
2. An Infinity, having a sign determined by the XOR of the signs of the source operands, is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
3. FR and FI are set to 0.
4. FPRF is set to indicate the class and sign of the result (± Infinity).

For *VSX Scalar Divide Quad-Precision* (***xsdivqp***), do the following.

1. ZX is set to 1.
2. An Infinity, having a sign determined by the XOR of the signs of the source operands, is placed into VSR[VRT+32] in quad-precision format.
3. FR and FI are set to 0. FPRF is set to indicate the class and sign of the result (± Infinity).

For *VSX Vector Divide Double-Precision* (***xvdivdp***), do the following.

1. ZX is set to 1.
2. For each vector element causing a Zero Divide exception, an Infinity, having a sign determined by the XOR of the signs of the source operands, is placed into its respective doubleword element of VSR[XT] in double-precision format.
3. FR, FI, and FPRF are not modified.

For *VSX Vector Divide Single-Precision* (***xvdivsp***), do the following.

1. ZX is set to 1.
2. For each vector element causing a Zero Divide exception, an Infinity, having a sign determined by the XOR of the signs of the source operands, is placed into its respective word element of VSR[XT] in single-precision format.
3. FR, FI, and FPRF are not modified.

1. *VSX Scalar Floating-Point Divide* instructions:
   xsdivdp, ***xsdivsp***

For *VSX Scalar Floating-Point Reciprocal Estimate*[1] instructions and *VSX Scalar Floating-Point Reciprocal Square Root Estimate*[2] instructions, do the following.

1. ZX is set to 1.

2. An Infinity, having the sign of the source operand, is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.

3. FR and FI are set to 0.

4. FPRF is set to indicate the class and sign of the result (± Infinity).

For the *VSX Vector Reciprocal Estimate Double-Precision* (***xvredp***) and *VSX Vector Reciprocal Square Root Estimate Double-Precision* (***xvrsqrtedp***) instructions:

1. ZX is set to 1.

2. For each vector element causing a Zero Divide exception, an Infinity, having the sign of the source operand, is placed into its respective doubleword element of VSR[XT] in double-precision format.

3. FR, FI, and FPRF are not modified.

For the *VSX Vector Reciprocal Estimate Single-Precision* (***xvresp***) and *VSX Vector Reciprocal Square Root Estimate Single-Precision* (***xvrsqrtesp***) instructions:

1. ZX is set to 1.

2. For each vector element causing a Zero Divide exception, an Infinity, having the sign of the source operand, is placed into its respective word element of VSR[XT] in single-precision format.

3. FR, FI, and FPRF are not modified.

1. *VSX Scalar Floating-Point Reciprocal Estimate* instructions:
   xsredp, ***xsresp***
2. *VSX Scalar Floating-Point Reciprocal Square Root Estimate* instructions:
   xsrsqrtedp, ***xsrsqrtesp***

7.4.3 Floating-Point Overflow Exception

7.4.3.1 Definition

An Overflow exception occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.

7.4.3.2 Action for OE=1

When Overflow exception is enabled (OE=1) and an Overflow exception occurs, the following actions are taken:

For the *VSX Vector round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***) instruction:

1. OX is set to 1.
2. If the unbiased exponent of the normalized intermediate result is less than or equal to 318 (Emax+192), the exponent is adjusted by subtracting 192. Otherwise the result is undefined.
3. The adjusted rounded result is placed into word element 0 of VSR[XT] in single-precision format. The contents of word elements 1-3 of VSR[XT] are undefined.
4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).

For *VSX Scalar Double-Precision Arithmetic*[1] instructions, do the following.

1. OX is set to 1.
2. The exponent of the normalized intermediate result is adjusted by subtracting 1536.
3. The adjusted rounded result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
4. FPRF is set to indicate the class and sign of the result (±Normal Number).

For *VSX Scalar Single-Precision Arithmetic*[2] instructions, do the following.

1. OX is set to 1.
2. The exponent is adjusted by subtracting 192.
3. The adjusted and rounded result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
4. FPRF is set to indicate the class and sign of the result (±Normal Number).

1. *VSX Scalar Double-Precision Arithmetic* instructions:
   xsadddp, ***xsdivdp***, ***xsmuldp***, ***xsredp***, ***xssubdp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***
2. *VSX Scalar Single-Precision Arithmetic* instructions:
   xsaddsp, ***xsdivsp***, ***xsmulsp***, ***xsresp***, ***xssubsp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***

For any of the following instruction classes,

VSX Scalar Quad-Precision Arithmetic instructions:
xsaddqp[***o***], ***xsdivqp***[***o***], ***xsmulqp***[***o***], ***xssqrtqp***[***o***], ***xssubqp***[***o***]
xsmaddqp[***o***], ***xsmsubqp***[***o***], ***xsnmaddqp***[***o***], ***xsnmsubqp***[***o***]

VSX Scalar Round Quad-Precision to Double-Extended-Precision (***xsrqpxp***)

do the following.

1. OX is set to 1.
2. The exponent is adjusted by subtracting 24576.
3. The adjusted, rounded result is placed into VSR[VRT+32] in quad-precision format.
4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).

For *VSX Scalar Convert with round Quad-Precision to Double-Precision format* [using round *to Odd*] (***xscvqpdp***), do the following.

1. OX is set to 1.
2. The exponent is adjusted by subtracting 1536. If the adjusted exponent is greater than +1023 (Emax), the result is undefined.
3. The adjusted, rounded result is placed into doubleword element 0 of VSR[VRT+32] in double-precision format.

   0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].
4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).

For *VSX Scalar Convert with round Double-Precision to Half-Precision format* (***xscvdphp***), do the following.

1. OX is set to 1.
2. The exponent is adjusted by subtracting 24. If the adjusted exponent is greater than +15 (Emax), the result is undefined.
3. The adjusted, rounded result is placed into rightmost halfword of doubleword element 0 of VSR[XT] in half-precision format.

   The contents of the leftmost 3 halfwords of doubleword element 0 of VSR[XT] are set to 0.

   The contents of doubleword element 1 of VSR[XT] are undefined.
4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).

For *VSX Vector Double-Precision Arithmetic*[1] instructions, *VSX Vector Single-Precision Arithmetic*[2] instructions, and *VSX Vector round and Convert Double-Precision to Single-Precision format* instruction (***xvcvdpsp***), do the following.

1. OX is set to 1.
2. Update of VSR[XT] is suppressed for all vector elements.
3. FR, FI, and FPRF are not modified.

For *VSX Vector Convert with round Single-Precision to Half-Precision format* (***xvcvsphp***), do the following.

1. OX is set to 1.
2. VSR[XT] is not modified.
3. FR, FI, and FPRF are not modified.

1. *VSX Vector Double-Precision Arithmetic* instructions:
   xvadddp, ***xvdivdp***, ***xvmuldp***, ***xvredp***, ***xvsubdp***, ***xvmaddadp***, ***xsmaddmdp***, ***xvmsubadp***, ***xvmsubmdp***, ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***
2. *VSX Vector Single-Precision Arithmetic* instructions:
   xvaddsp, ***xvdivsp***, ***xvmulsp***, ***xvresp***, ***xvsubsp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvsmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***

7.4.3.3 Action for OE=0

When Overflow exception is disabled (OE=0) and an Overflow exception occurs, the following actions are taken:

1. OX and XX are set to 1.

2. The result is determined by the rounding mode (RN) and the sign of the intermediate result as follows:

   Round to Nearest Even
   For negative overflow, the result is -Infinity.
   For positive overflow, the result is +Infinity.

   Round toward Zero
   For negative overflow, the result is the format's most negative finite number.
   For positive overflow, the result is the format's most positive finite number.

   Round toward +Infinity
   For negative overflow, the result is the format's most negative finite number.
   For positive overflow, the result is +Infinity.

   Round toward -Infinity
   For negative overflow, the result is -Infinity.
   For positive overflow, the result is the format's most positive finite number.

For *VSX Scalar round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***):

3. The result is placed into word element 0 of VSR[XT] as a single-precision value. The contents of word elements 1-3 of VSR[XT] are undefined.

4. FR is undefined.

5. FI is set to 1.

6. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Double-Precision Arithmetic*[1] instructions and *VSX Scalar Single-Precision Arithmetic*[2] instructions, do the following.

3. The result is placed into doubleword element 0 of VSR[XT] as a double-precision value. The contents of doubleword element 1 of VSR[XT] are undefined.

4. FR is undefined.

5. FI is set to 1.

6. FPRF is set to indicate the class and sign of the result.

For any of the following instructions,

VSX Scalar Quad-Precision Arithmetic instructions:
xsaddqp[***o***], ***xsdivqp***[***o***], ***xsmulqp***[***o***], ***xssubqp***[***o***]
xsmaddqp[***o***], ***xsmsubqp***[***o***], ***xsnmaddqp***[***o***], ***xsnmsubqp***[***o***]

VSX Scalar Quad-Precision Round to Double-Extended-Precision (***xsrqpxp***)

1. *VSX Scalar Double-Precision Arithmetic* instructions:
   xsadddp, ***xsdivdp***, ***xsmuldp***, ***xsredp***, ***xssubdp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***
2. *VSX Scalar Single-Precision Arithmetic* instructions:
   xsaddsp, ***xsdivsp***, ***xsmulsp***, ***xsresp***, ***xssubsp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***

do the following.

3. The result is placed into VSR[VRT+32] in quad-precision format.

4. FR is undefined. FI is set to 1. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Convert with round Quad-Precision to Double-Precision format* (***xscvqpdp***), do the following.

3. The result is placed into doubleword element 0 of VSR[VRT+32] as a double-precision value.

   0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].

4. FR is undefined. FI is set to 1. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Convert with round Double-Precision to Half-Precision format* (***xscvdphp***), do the following.

1. OX and XX are set to 1.

2. The result is placed into the rightmost halfword of doubleword element 0 of VSR[XT] as a half-precision value.

   The contents of the leftmost 3 halfwords of doubleword element 0 of VSR[XT] are set to 0.

   The contents of doubleword element 1 of VSR[XT] are undefined.

3. FR is undefined. FI is set to 1. FPRF is set to indicate the class and sign of the result.

For *VSX Vector Double-Precision Arithmetic*[1] instructions, do the following.

3. For each vector element causing an Overflow exception, the result is placed into its respective doubleword element of VSR[XT] in double-precision format.

4. FR, FI, and FPRF are not modified.

For *VSX Vector Single-Precision Arithmetic*[2] instructions and *VSX Vector round and Convert Double-Precision to Single-Precision format* (***xvcvdpsp***), do the following.

3. For each vector element causing an Overflow exception, the result is placed into its respective word element of VSR[XT] in single-precision format.

4. FR, FI, and FPRF are not modified.

For *VSX Vector Convert with round Single-Precision to Half-Precision format* (***xvcvsphp***), do the following.

1. OX and XX are set to 1.

2. For each vector element causing an Overflow exception, the result is placed into the rightmost halfword of its respective word element of VSR[XT] in half-precision format.

   The contents of the leftmost halfword of its respective word element of VSR[XT] are set to 0.

3. FR, FI, and FPRF are not modified.

1. *VSX Vector Double-Precision Arithmetic* instructions:
   xvadddp, ***xvdivdp***, ***xvmuldp***, ***xvredp***, ***xvsubdp***, ***xvmaddadp***, ***xvmaddmdp***, ***xvmsubadp***, ***xvmsubmdp***, ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***
2. *VSX Vector Single-Precision Arithmetic* instructions:
   xvaddsp, ***xvdivsp***, ***xvmulsp***, ***xvresp***, ***xvsubsp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***

7.4.4 Floating-Point Underflow Exception

7.4.4.1 Definition

Underflow exception is defined separately for the enabled and disabled states:

Enabled:
Underflow occurs when the intermediate result is “Tiny”.

Disabled:
Underflow occurs when the intermediate result is “Tiny” and there is “Loss of Accuracy”.

A *tiny* result is detected before rounding, when a nonzero intermediate result computed as though both the precision and the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is tiny and Underflow exception is disabled (`UE=0`), the intermediate result is denormalized (see Section 7.3.2.4 , “Normalization and Denormalization” on page 377) and rounded (see Section 7.3.2.6 , “Rounding” on page 381) before being placed into the target VSR.

Loss of accuracy is detected when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

7.4.4.2 Action for UE=1

When Underflow exception is enabled (`UE=1`) and an Underflow exception occurs, the following actions are taken:

For *VSX Scalar round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***), do the following.

1. `UX` is set to 1.
2. If the unbiased exponent of the normalized intermediate result is greater than or equal to -319 (`Emin-192`), the exponent is adjusted by adding 192. Otherwise the result is undefined.
3. The adjusted rounded result is placed into word element 0 of `VSR[XT]` in single-precision format. The contents of word elements 1-3 of `VSR[XT]` are undefined.
4. Unless the result is undefined, `FPRF` is set to indicate the class and sign of the result (±Normal Number).

For *VSX Scalar Double-Precision Arithmetic*[1] instructions *and VSX Scalar Double-Precision Reciprocal Estimate* (***xsredp***), do the following.

1. UX is set to 1.
2. The exponent of the normalized intermediate result is adjusted by adding 1536.
3. The adjusted rounded result is placed into doubleword element 0 of `VSR[XT]` in double-precision format. The contents of doubleword element 1 of `VSR[XT]` are undefined.
4. `FPRF` is set to indicate the class and sign of the result (±Normal Number).

1. *VSX Scalar Double-Precision Arithmetic* instructions:
xsadddp, ***xsdivdp***, ***xsmuldp***, ***xssubdp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***

For any of the following instructions,

VSX Scalar Quad-Precision Arithmetic instructions:
xsaddqp[***o***], ***xsdivqp***[***o***], ***xsmulqp***[***o***], ***xssubqp***[***o***]
xsmaddqp[***o***], ***xsmsubqp***[***o***], ***xsnmaddqp***[***o***], ***xsnmsubqp***[***o***]

VSX Scalar Round Quad-Precision to Double-Extended-Precision (***xsrqpxp***)

do the following.

1. UX is set to 1.
2. The exponent of the normalized intermediate result is adjusted by adding 24576.
3. The adjusted, rounded result is placed into VSR[VRT+32] in quad-precision format.
4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).

For *VSX Scalar Convert with round Quad-Precision to Double-Precision format* [*using round to Odd*] (***xscvqpdp***[***o***]), do the following.

1. UX is set to 1.
2. The exponent of the normalized intermediate result is adjusted by adding 1536. If the adjusted exponent is less than -1022, the result is undefined.
3. The adjusted, rounded result is placed into doubleword element 0 of VSR[VRT+32] in double-precision format.

   0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].
4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).

For *VSX Scalar Single-Precision Arithmetic*[1] instructions and *VSX Scalar Single-Precision Reciprocal Estimate* (***xsresp***), do the following.

1. UX is set to 1.
2. The exponent is adjusted by adding 192.
3. The adjusted rounded result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.
4. FPRF is set to indicate the class and sign of the result (±Normal Number).

Programming Note

The FR and FI bits are provided to allow the system floating-point enabled exception error handler, when invoked because of an Underflow exception, to simulate a “trap disabled” environment. That is, the FR and FI bits allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized and correctly rounded.

For *VSX Scalar Convert with round Double-Precision to Half-Precision with round* (***xscvdphp***), do the following.

1. UX is set to 1.

1. *VSX Scalar Single-Precision Arithmetic* instructions:
   xsaddsp, ***xsdivsp***, ***xsmulsp***, ***xssubsp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***

2. The exponent of the normalized intermediate result is adjusted by adding 24. If the adjusted exponent is less than -14, the result is undefined.

3. The adjusted, rounded result is placed into rightmost halfword of doubleword element 0 of VSR[XT] in half-precision format.

   The contents of the leftmost 3 halfwords of doubleword element 0 of VSR[XT] are set to 0.

   The contents of doubleword element 1 of VSR[XT] are undefined.

4. Unless the result is undefined, FPRF is set to indicate the class and sign of the result (±Normal Number).

For *VSX Vector Floating-Point Arithmetic*[1] instructions, *VSX Vector Floating-Point Reciprocal Estimate*[2] instructions, and *VSX Vector round and Convert Double-Precision to Single-Precision format* (***xvcvdpsp***), do the following.

1. UX is set to 1.

2. Update of VSR[XT] is suppressed for all vector elements.

3. FR, FI, and FPRF are not modified.

For *VSX Vector Convert with round Single-Precision to Half-Precision format* (***xvcvsphp***), do the following.

1. UX is set to 1.
2. VSR[XT] is not modified.
3. FR, FI, and FPRF are not modified.

7.4.4.3 Action for UE=0

When Underflow exception is disabled (UE=0) and an Underflow exception occurs, the following actions are taken:

For *VSX Scalar round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***), do the following.

1. UX is set to 1.

2. The result is placed into word element 0 of VSR[XT] in single-precision format. The contents of word elements 1-3 of VSR[XT] are undefined.

3. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Floating-Point Arithmetic*[3] instructions and *VSX Scalar Reciprocal Estimate*[4] instructions, do the following.

1. UX is set to 1.

2. The result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.

1. *VSX Vector Arithmetic* instructions:
   xvadddp, ***xvdivdp***, ***xvmuldp***, ***xvsubdp***, ***xvaddsp***, ***xvdivsp***, ***xvmulsp***, ***xvsubsp***, ***xvmaddadp***, ***xvmaddmdp***, ***xvmsubadp***, ***xvmsubmdp***, ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***
2. *VSX Vector Floating-Point Reciprocal Estimate* instructions:
   xvredp, ***xvresp***
3. *VSX Scalar Floating-Point Arithmetic* instructions:
   xsadddp, ***xsdivdp***, ***xsmuldp***, ***xssubdp***, ***xsaddsp***, ***xsdivsp***, ***xsmulsp***, ***xssubsp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***
4. *VSX Scalar Reciprocal Estimate* instructions:
   xsredp, ***xsresp***

3. FPRF is set to indicate the class and sign of the result.

For any of the following instructions,

VSX Scalar Quad-Precision Arithmetic instructions:
xsaddqp[***o***], ***xsdivqp***[***o***], ***xsmulqp***[***o***], ***xssubqp***[***o***]
xsmaddqp[***o***], ***xsmsubqp***[***o***], ***xsnmaddqp***[***o***], ***xsnmsubqp***[***o***]

VSX Scalar Round Quad-Precision to Double-Extended-Precision (***xsrqpxp***)

do the following.

1. UX is set to 1.
2. The result is placed into VSR[VRT+32] in quad-precision format.
3. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Convert with round Quad-Precision to Double-Precision format* (***xscvqpdp***), do the following.

1. UX is set to 1.
2. The result is placed into doubleword element 0 of VSR[VRT+32] in double-precision format.

   0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].
3. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Convert with round Double-Precision to Half-Precision format* (***xscvdphp***), do the following.

1. UX is set to 1.
2. The result is placed into the rightmost halfword of doubleword element 0 of VSR[XT] as a half-precision value.

   The contents of the leftmost 3 halfwords of doubleword element 0 of VSR[XT] are set to 0.

   The contents of doubleword element 1 of VSR[XT] are undefined.
3. FPRF is set to indicate the class and sign of the result.

For *VSX Vector Double-Precision Arithmetic*[1] instructions and *VSX Vector Reciprocal Estimate Double-Precision* (***xvredp***), do the following.

1. UX is set to 1.
2. For each vector element causing an Underflow exception, the result is placed into its respective doubleword element of VSR[XT] in double-precision format.
3. FR, FI, and FPRF are not modified.

For *VSX Vector Single-Precision Arithmetic*[2] instructions, *VSX Vector Reciprocal Estimate Single-Precision* (***xvresp***), and *VSX Vector round and Convert Double-Precision to Single-Precision format* (***xvcvdpsp***), do the following.

1. UX is set to 1.

1. *VSX Vector Double-Precision Arithmetic* instructions:
   xvadddp, ***xvdivdp***, ***xvmuldp***, ***xvsubdp***, ***xvmaddadp***, ***xvmaddmdp***, ***xvmsubadp***, ***xvmsubmdp***, ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***
2. *VSX Vector Single-Precision Arithmetic* instructions:
   xvaddsp, ***xvdivsp***, ***xvmulsp***, ***xvsubsp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***

2. For each vector element causing an Underflow exception, the result is placed into its respective word element of VSR[XT] in single-precision format.

3. FR, FI, and FPRF are not modified.

For *VSX Vector Convert with round Single-Precision to Half-Precision format* (***xvcvsphp***), do the following.

1. UX is set to 1.

2. For each vector element causing an Underflow exception, the result is placed into the rightmost halfword of its respective word element of VSR[XT] in half-precision format.

   The contents of the leftmost halfword of its respective word element of VSR[XT] are set to 0.

3. FR, FI, and FPRF are not modified.

7.4.5 Floating-Point Inexact Exception

7.4.5.1 Definition

An Inexact exception occurs when one of two conditions occur during rounding:

1. The rounded result differs from the intermediate result assuming both the precision and the exponent range of the intermediate result to be unbounded. In this case the result is said to be inexact. (If the rounding causes an enabled Overflow exception or an enabled Underflow exception, an Inexact exception also occurs only if the significands of the rounded result and the intermediate result differ.)

2. The rounded result overflows and Overflow exception is disabled.

The action to be taken depends on the setting of the Inexact Exception Enable bit of the FPSCR.

7.4.5.2 Action for XE=1

Programming Note

In some implementations, enabling Inexact exceptions can degrade performance more than does enabling other types of floating-point exception.

When Inexact exception is enabled (UE=1) and an Inexact exception occurs, the following actions are taken:

For the *VSX Vector round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***) instruction:

1. XX is set to 1.

2. The result is placed into word element 0 of VSR[XT] in single-precision format. The contents of word elements 1-3 of VSR[XT] are undefined.

3. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Floating-Point Arithmetic*[1] instructions, *VSX Scalar Round to Double-Precision Integer Exact using Current rounding mode* (***xsrdpic***), and *VSX Scalar Integer to Floating-Point Format Conversion*[2] instructions, do the following.

1. XX is set to 1.

2. The result is placed into doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined.

3. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Floating-Point to Integer Word Format Conversion*[3] instructions, do the following.

1. XX is set to 1.

2. The result is placed into word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

3. FPRF is set to indicate the class and sign of the result.

1. *VSX Scalar Floating-Point Arithmetic* instructions:
   xsadddp, ***xsdivdp***, ***xsmuldp***, ***xssubdp***, ***xsaddsp***, ***xsdivsp***, ***xsmulsp***, ***xssubsp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***
2. *VSX Scalar Integer to Floating-Point Format Conversion* instructions:
   xscvsxddp, ***xscvuxddp***, ***xscvsxdsp***, ***xscvuxdsp***
3. *VSX Scalar Floating-Point to Integer Word Format Conversion* instructions:
   xscvdpsxws, ***xscvdpuxws***

For any of the following instructions,

VSX Scalar Quad-Precision Arithmetic instructions:
xsaddqp[***o***], ***xsdivqp***[***o***], ***xsmulqp***[***o***], ***xssqrtqp***[***o***], ***xssubqp***[***o***]
xsmaddqp[***o***], ***xsmsubqp***[***o***], ***xsnmaddqp***[***o***], ***xsnmsubqp***[***o***]

VSX Scalar Quad-Precision Round instructions:
xsrqpi, ***xsrqpxp***

do the following.

1. XX is set to 1.
2. The result is placed into VSR[VRT+32] in quad-precision format.
3. FR is set to indicate if the rounded result was incremented. FI is set to 1. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Convert with round Quad-Precision to Double-Precision format* (***xscvqpdp***), do the following.

1. XX is set to 1.
2. The result is placed into doubleword element 0 of VSR[VRT+32] in double-precision format.

   0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].
3. FR is set to indicate if the rounded result was incremented. FI is set to 1. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar truncate & Convert Quad-Precision to Signed Doubleword* (***xscvqpsdz***), do the following.

1. XX is set to 1.
2. The result is placed into doubleword element 0 of VSR[XT] in signed integer format.

   0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].
3. FR is set to 0. FI is set to 1. FPRF is undefined.

For *VSX Scalar truncate & Convert Quad-Precision to Signed Word* (***xscvqpswz***), do the following.

1. XX is set to 1.
2. The result is placed into word element 1 of VSR[XT] in signed integer format.

   0x0000_0000 is placed into word elements 0, 2, and 3 of VSR[VRT+32].
3. FR is set to 0. FI is set to 1. FPRF is undefined.

For *VSX Scalar truncate & Convert Quad-Precision to Unsigned Doubleword* (***xscvqpudz***), do the following.

1. XX is set to 1.
2. The result is placed into doubleword element 0 of VSR[XT] in unsigned integer format.

   0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].
3. FR is set to 0. FI is set to 1. FPRF is undefined.

For *VSX Scalar truncate & Convert Quad-Precision to Unsigned Word* (***xscvqpuwz***), do the following.

1. XX is set to 1.
2. The result is placed into word element 1 of VSR[XT] in unsigned integer format.

   0x0000_0000 is placed into word elements 0, 2, and 3 of VSR[VRT+32].
3. FR is set to 0. FI is set to 1. FPRF is undefined.

For *VSX Scalar Convert with round Double-Precision to Half-Precision truncate* (***xscvdphp***), do the following.

1. XX is set to 1.
2. The result is placed into the rightmost halfword of doubleword element 0 of VSR[XT] as a half-precision value.

   The contents of the leftmost 3 halfwords of doubleword element 0 of VSR[XT] are set to 0.

   The contents of doubleword element 1 of VSR[XT] are undefined.
3. FR is set to indicate if the rounded result was incremented. FI is set to 1. FPRF is set to indicate the class and sign of the result.

For *VSX Vector Floating-Point Arithmetic*[1] instructions, *VSX Vector Floating-Point Reciprocal Estimate*[2] instructions, *VSX Vector round and Convert Double-Precision to Single-Precision format* (***xvcvdpsp***), *VSX Vector Double-Precision to Integer Format Conversion*[3] instructions, and *VSX Vector Integer to Floating-Point Format Conversion*[4] instructions, do the following.

1. XX is set to 1.
2. Update of VSR[XT] is suppressed for all vector elements.
3. FR, FI, and FPRF are not modified.

For *VSX Vector Convert with round Single-Precision to Half-Precision format* (***xvcvsphp***), do the following.

1. XX is set to 1.
2. VSR[XT] is not modified.
3. FR, FI, and FPRF are not modified.

1. *VSX Vector Floating-Point Arithmetic* instructions:
   xvadddp, ***xvdivd***p, ***xvmuld***p, ***xvsubd***p, ***xsaddsp***, ***xvdivsp***, ***xvmulsp***, ***xvsubsp***, ***xvmaddadp***, ***xvmaddmdp***, ***xvmsubadp***, ***xvmsubmdp***, ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***
2. *VSX Vector Floating-Point Reciprocal Estimate* instructions:
   xvredp, ***xvresp***
3. *VSX Vector Double-Precision to Integer Format Conversion* instructions:
   xvcvdpsxds, ***xvcvdpsxws***, ***xvcvdpuxds***, ***xvcvdpuxws***
4. *VSX Vector Integer to Floating-Point Format Conversion* instructions:
   xvcvsxddp, ***xvcvuxddp***, ***xvcvsxdsp***, ***xvcvuxdsp***, ***xvcvsxwsp***, ***xvcvuxwsp***

7.4.5.3 Action for XE=0

When Inexact exception is disabled (XE=0) and an Inexact exception occurs, the following actions are taken:

For *VSX Scalar round and Convert Double-Precision to Single-Precision format* (***xscvdpsp***), do the following.

1. XX is set to 1.
2. The result is placed into word element 0 of VSR[XT] as a single-precision value. The contents of word elements 1-3 of VSR[XT] are undefined.
3. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Double-Precision Arithmetic*[1] instructions, *VSX Scalar Single-Precision Arithmetic*[2] instructions, *VSX Scalar Round to Single-Precision* (***xsrsp***), the *VSX Scalar Round to Double-Precision Integer Exact using Current rounding mode* (***xsrdpic***), and *VSX Scalar Integer to Double-Precision Format Conversion*[3] instructions, do the following.

1. XX is set to 1.
2. The result is placed into doubleword element 0 of VSR[XT] as a double-precision value. The contents of doubleword element 1 of VSR[XT] are undefined.
3. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Convert with round to zero Double-Precision To Signed Word format* (***xscvdpsxws***) and *VSX Scalar Convert with round to zero Double-Precision To Unsigned Word format* (***xscvdpuxws***), do the following.

1. XX is set to 1.
2. The result is placed into word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.
3. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar Convert with round Quad-Precision to Double-Precision format* (***xscvqpdp***), do the following.

1. XX is set to 1.
2. The result is placed into the rightmost halfword of doubleword element 0 of VSR[XT] as a half-precision value.

   The contents of the leftmost 3 halfwords of doubleword element 0 of VSR[XT] are set to 0.

   The contents of doubleword element 1 of VSR[XT] are undefined.
3. FR is set to indicate if the rounded result was incremented. FI is set to 1. FPRF is set to indicate the class and sign of the result.

1. *VSX Scalar Double-Precision Arithmetic* instructions:
   xsadddp, ***xssubdp***, ***xsmuldp***, ***xsdivdp***, ***xssqrtdp***, ***xsmaddadp***, ***xsmaddmdp***, ***xsmsubadp***, ***xsmsubmdp***, ***xsnmaddadp***, ***xsnmaddmdp***, ***xsnmsubadp***, ***xsnmsubmdp***
2. *VSX Scalar Single-Precision Arithmetic* instructions:
   xsaddsp, ***xssubsp***, ***xsmulsp***, ***xsdivsp***, ***xssqrtsp***, ***xsmaddasp***, ***xsmaddmsp***, ***xsmsubasp***, ***xsmsubmsp***, ***xsnmaddasp***, ***xsnmaddmsp***, ***xsnmsubasp***, ***xsnmsubmsp***
3. *VSX Scalar Integer to Double-Precision Format Conversion* instructions:
   xscvsxddp, ***xscvuxddp***

For *VSX Vector Double-Precision Arithmetic* instructions,
xvadddp, ***xvsubdp***, ***xvmuldp***, ***xvdivdp***, ***xvsqrtdp***, ***xvmaddadp***, ***xvmaddmdp***, ***xvmsubadp***, ***xvmsubmdp***, ***xvnmaddadp***, ***xvnmaddmdp***, ***xvnmsubadp***, ***xvnmsubmdp***

do the following.

1. XX is set to 1.

2. For each vector element causing an Inexact exception, the result is placed into its respective doubleword element of VSR[XT] in double-precision format.

3. FR, FI, and FPRF are not modified.

For any of the following instructions,

VSX Scalar Quad-Precision Arithmetic instructions:
xsaddqp[o], ***xsdivqp[o]***, ***xsmulqp[o]***, ***xssqrtqp[o]***, ***xssubqp[o]***
xsmaddqp[***o***], ***xsmsubqp***[***o***], ***xsnmaddqp***[***o***], ***xsnmsubqp***[***o***]

VSX Scalar Round Quad-Precision to Double-Extended-Precision (***xsrqpxp***)
VSX Scalar Round to Quad-Precision Integer (**xsrqpi**)

do the following.

1. XX is set to 1.

2. The result is placed into VSR[VRT+32] in quad-precision format.

3. FR is set to indicate if the rounded result was incremented. FI is set to 1. FPRF is set to indicate the class and sign of the result.

For *VSX Scalar round & Convert Quad-Precision to Double-Precision* (***xscvqpdp***), do the following.

1. XX is set to 1.

2. The result is placed into doubleword element 0 of VSR[VRT+32] in double-precision format.

   0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].

3. FR is set to indicate if the rounded result was incremented. FI is set to 1. FPRF is set to indicate the class and sign of the result.

For any of the following instructions,

VSX Scalar truncate & Convert Quad-Precision to Signed Doubleword (***xscvqpsdz***)
VSX Scalar truncate & Convert Quad-Precision to Signed Word (***xscvqpswz***)

do the following.

1. XX is set to 1.

2. The result is placed into doubleword element 0 of VSR[VRT+32] in signed integer format.

   0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].

3. FR is set to 0. FI is set to 1. FPRF is undefined.

For any of the following instructions,

VSX Scalar truncate & Convert Quad-Precision to Unsigned Doubleword (***xscvqpudz***)
VSX Scalar truncate & Convert Quad-Precision to Unsigned Word (***xscvqpuwz***)

do the following.

1. XX is set to 1.
2. The result is placed into doubleword element 0 of VSR[VRT+32] in unsigned integer format.

   0x0000_0000_0000_0000 is placed into doubleword element 1 of VSR[VRT+32].
3. FR is set to 0. FI is set to 1. FPRF is undefined.

For *VSX Vector Convert with round Single-Precision to Half-Precision format* (***xvcvsphp***), do the following.

1. XX is set to 1.
2. For each vector element causing an Underflow exception, the result is placed into the rightmost halfword of its respective word element of VSR[XT] in half-precision format.

   The contents of the leftmost halfword of its respective word element of VSR[XT] are set to 0.
3. FR, FI, and FPRF are not modified.

For *VSX Vector Single-Precision Arithmetic*[1] instructions, do the following.

1. XX is set to 1.
2. For each vector element causing an Inexact exception, the result is placed into its respective word element of VSR[XT] in single-precision format.
3. FR, FI, and FPRF are not modified.

1. *VSX Vector Single-Precision Arithmetic* instructions:
   xvaddsp, ***xvsubsp***, ***xvmulsp***, ***xvdivsp***, ***xvsqrtsp***, ***xvmaddasp***, ***xvmaddmsp***, ***xvmsubasp***, ***xvmsubmsp***, ***xvnmaddasp***, ***xvnmaddmsp***, ***xvnmsubasp***, ***xvnmsubmsp***

7.5 VSX Storage Access Operations

The *VSX Storage Access* instructions compute the effective address (EA) of the storage to be accessed as described in Power ISA Book I.

7.5.1 Accessing Aligned Storage Operands

The following quadword-aligned array, AH, consists of 8 halfwords.

```
short   AW[4] = { 0x0001_0203,
                  0x0405_0607,
                  0x0809_0A0B,
                  0x0C0D_0E0F };
```

Figure 120 illustrates the Big-Endian storage image of array AW.

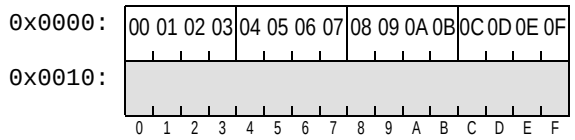


Figure 120.Big-Endian storage image of array AW

Figure 121 illustrates the Little-Endian storage image of array AW.

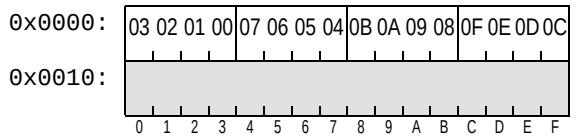


Figure 121.Little-Endian storage image of array AW

Figure 122 shows the result of loading that quadword into a VSR or, equivalently, shows the contents that must be in a VSR if storing that VSR is to produce the storage contents shown in Figure 120 for Big-Endian. Note that Figure shows the effect of loading the quadword from both Big-Endian storage and Little-Endian storage.

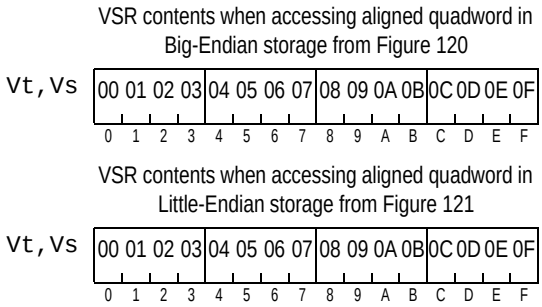


Figure 122.Vector-Scalar Register contents for aligned quadword Load or Store VSX Vector

7.5.2 Accessing Unaligned Storage Operands

The following array, B, consists of 5 word elements.

```
int   B[5];
B[0] = 0x01234567;
B[1] = 0x00112233;
B[2] = 0x44556677;
B[3] = 0x8899AABB;
B[4] = 0xCCDDEEFF;
```

Figure 123 illustrates both Big-Endian and Little-Endian storage images of array B.

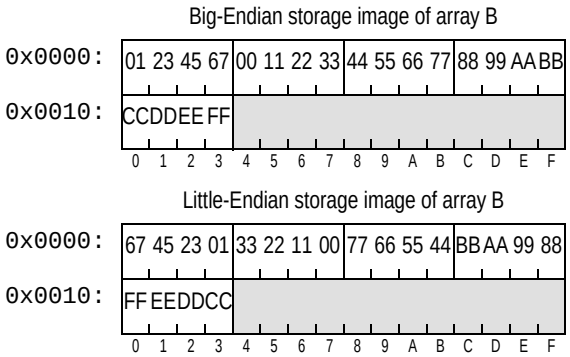


Figure 123.Storage images of array B

Though this example shows the array starting at a quadword-aligned address, if the subject data of interest are elements 1 through 4, accessing elements 1 through 4 of array B involves an unaligned quadword storage access that spans two aligned quadwords.

Loading an Unaligned Quadword from Big-Endian Storage

Loading elements from elements 1 through 4 of B (see Figure 123) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using Big-Endian byte ordering.

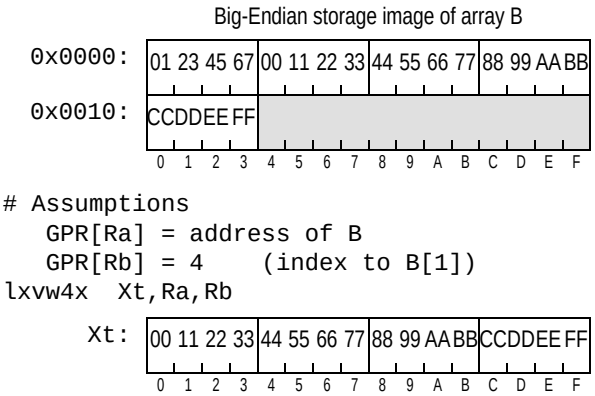


Figure 124.Process to load unaligned quadword from Big-Endian storage using Load VSX Vector Word*4 Indexed

Loading an Unaligned Quadword from Little-Endian Storage

Loading elements from elements 1 through 4 of B (see Figure 123) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using Little-Endian byte ordering.

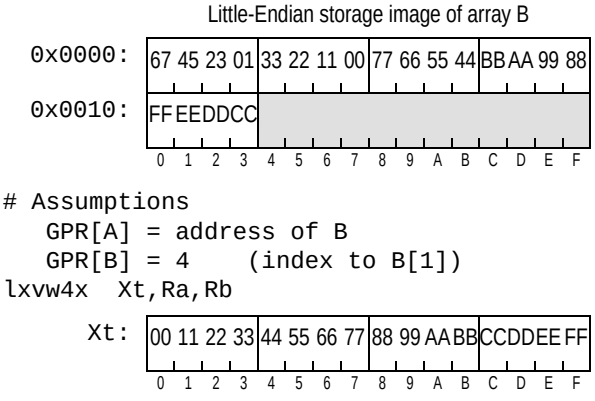
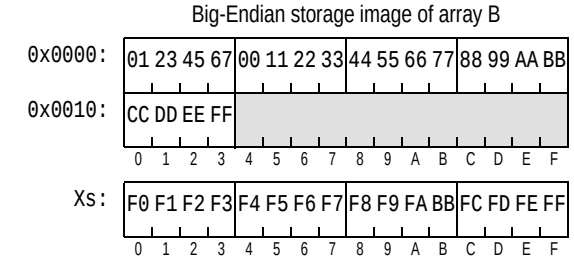


Figure 125.Process to load unaligned quadword from Little-Endian storage Load VSX Vector Word*4 Indexed

Storing an Unaligned Quadword to Big-Endian Storage

Storing a VSR to elements 1 through 4 of B (see Figure 123) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using Big-Endian byte ordering.




```
# Assumptions
   GPR[Ra] = address of B
   GPR[Rb] = 4    (index to B[1])

stxvw4x  Xs,Ra,Rb
```

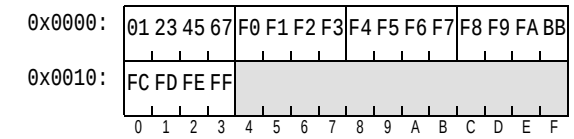
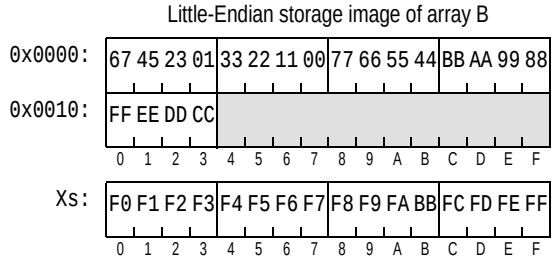


Figure 126.Process to store unaligned quadword to Big-Endian storage using Store VSX Vector Word*4 Indexed

Storing an Unaligned Quadword to Little-Endian Storage

Storing a VSR to elements 1 through 4 of B (see Figure 123) into VR[VT] involves an unaligned quadword storage access.

VSX supports word-aligned vector and scalar storage accesses using Little-Endian byte ordering.




```
# Assumptions
   GPR[A] = address of B
   GPR[B] = 4    (index to B[1])

stxvw4x  Xs,Ra,Rb
```

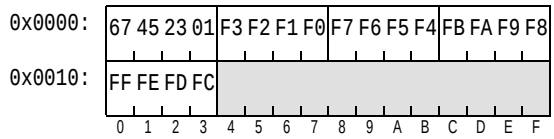


Figure 127.Process to store unaligned quadword to Little-Endian storage Store VSX Vector Word*4 Indexed

7.5.3 Storage Access Exceptions

Storage accesses cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (*Store* only), or if the program attempts to access storage that is unavailable.

7.6 VSX Instruction Set

7.6.1 VSX Instruction Set Summary

7.6.1.1 VSX Storage Access Instructions

There are two basic forms of scalar load and scalar store instructions, word and doubleword. *VSX Scalar Load* instructions place a copy of the contents of the addressed word or doubleword in storage into the left-most word or doubleword element of the target VSR. The contents of the right-most element(s) of the target VSR are undefined. *VSX Scalar Store* instructions place a copy of the contents of the left-most word or doubleword element in the source VSR into the addressed word or doubleword in storage.

There are two basic forms of vector load and vector store instructions, a vector of 4 word elements and a vector of two doublewords. Both forms access a quadword in storage.

There is one basic form of vector load and splat instruction, doubleword. *VSX Vector Load and Splat* instruction places a copy of the contents of the addressed doubleword in storage into both doubleword elements of the target VSR.

7.6.1.1.1 VSX Scalar Storage Access Instructions

Mnemonic	Instruction Name	Page
lxsd	Load VSX Scalar Dword	480
lxsdx	Load VSX Scalar Dword Indexed	480
lxsibzx	Load VSX Scalar as Integer Byte & Zero Indexed	482
lxsihax	Load VSX Scalar as Integer Hword & Zero Indexed	482
lxsiwax	Load VSX Scalar as Integer Word Algebraic Indexed	483
lxsiwzx	Load VSX Scalar as Integer Word & Zero Indexed	484
lxssp	Load VSX Scalar Single-Precision	485
lxsspx	Load VSX Scalar Single-Precision Indexed	485

Table 8. VSX Scalar Load Instructions

Mnemonic	Instruction Name	Page
stxsd	Store VSX Scalar Dword	498
stxsdx	Store VSX Scalar Dword Indexed	498
stxsibx	Store VSX Scalar as Integer Byte Indexed	499
stxsihx	Store VSX Scalar as Integer Hword Indexed	499
stxsiwx	Store VSX Scalar as Integer Word Indexed	500
stxssp	Store VSX Scalar Single-Precision	501
stxsspx	Store VSX Scalar Single-Precision Indexed	502

Table 9. VSX Scalar Store Instructions

7.6.1.1.2 VSX Vector Storage Access Instructions

Mnemonic	Instruction Name	Page
lxv	Load VSX Vector	492
lxvb16x	Load VSX Vector Byte*16 Indexed	487
lxvd2x	Load VSX Vector Dword*2 Indexed	488
lxvh8x	Load VSX Vector Hword*8 Indexed	495
lxvw4x	Load VSX Vector Word*4 Indexed	496
lxvx	Load VSX Vector Indexed	492

Table 10.VSX Vector Load Instructions

Mnemonic	Instruction Name	Page
lxvdsx	Load VSX Vector Dword and Splat Indexed	494
lxvwsx	Load VSX Vector Word & Splat Indexed	497

Table 11.VSX Vector Load & Splat Instructions

Mnemonic	Instruction Name	Page
lxvl	Load VSX Vector with Length	489
lxvll	Load VSX Vector with Length Left-justified	491

Table 12.VSX Vector Load with Length Instructions

Mnemonic	Instruction Name	Page
stxv	Store VSX Vector	507
stxvb16x	Store VSX Vector Byte*16 Indexed	503
stxvd2x	Store VSX Vector Dword*2 Indexed	504
stxvh8x	Store VSX Vector Hword*8 Indexed	505
stxvw4x	Store VSX Vector Word*4 Indexed	506
stxvx	Store VSX Vector Indexed	510

Table 13.VSX Vector Store Instructions

Mnemonic	Instruction Name	Page
stxvl	Store VSX Vector with Length	507
stxvll	Store VSX Vector with Length Left-justified	509

Table 14.VSX Vector Store w/ Length Instructions

7.6.1.2 VSX Binary Floating-Point Sign Manipulation Instructions

7.6.1.2.1 VSX Scalar Binary Floating-Point Sign Manipulation Instructions

Mnemonic	Instruction Name	Page
xsabsdp	VSX Scalar Absolute Double-Precision	512
xsabsqp	VSX Scalar Absolute Quad-Precision	512
xscpsgndp	VSX Scalar Copy Sign Double-Precision	533
xscpsgnqp	VSX Scalar Copy Sign Quad-Precision	533
xsnabsdp	VSX Scalar Negative Absolute Double-Precision	606
xsnabsqp	VSX Scalar Negative Absolute Quad-Precision	606
xsnegdp	VSX Scalar Negate Double-Precision	607
xsnegqp	VSX Scalar Negate Quad-Precision	607

Table 15.VSX Scalar BFP Sign Manipulation Instructions

7.6.1.2.2 VSX Vector Binary Floating-Point Sign Manipulation Instructions

Mnemonic	Instruction Name	Page
xvabsdp	VSX Vector Absolute Value Double-Precision	658
xvabssp	VSX Vector Absolute Value Single-Precision	658
xvcpsgndp	VSX Vector Copy Sign Double-Precision	671
xvcpsgnsp	VSX Vector Copy Sign Single-Precision	671
xvnabsdp	VSX Vector Negative Absolute Value Double-Precision	725
xvnabssp	VSX Vector Negative Absolute Value Single-Precision	725
xvnegdp	VSX Vector Negate Double-Precision	726
xvnegsp	VSX Vector Negate Single-Precision	726

Table 16.VSX Vector BFP Sign Manipulation Instructions

7.6.1.3 VSX Binary Floating-Point Arithmetic Instructions

7.6.1.3.1 VSX Scalar Binary Floating-Point Arithmetic Instructions

Mnemonic	Instruction Name	Page
xsadddp	VSX Scalar Add Double-Precision	513
xsaddqp[o]	VSX Scalar Add Quad-Precision [using round to Odd]	520
xsaddsp	VSX Scalar Add Single-Precision	518
xsdivdp	VSX Scalar Divide Double-Precision	562
xsdivqp[o]	VSX Scalar Divide Quad-Precision [using round to Odd]	564
xsdivsp	VSX Scalar Divide Single-Precision	566
xsmuldp	VSX Scalar Multiply Double-Precision	600
xsmulqp[o]	VSX Scalar Multiply Quad-Precision [using round to Odd]	602
xsmulsp	VSX Scalar Multiply Single-Precision	604
xssqrtdp	VSX Scalar Square Root Double-Precision	641
xssqrtqp[o]	VSX Scalar Square Root Quad-Precision [using round to Odd]	642
xssqrtsp	VSX Scalar Square Root Single-Precision	644
xssubdp	VSX Scalar Subtract Double-Precision	645
xssubqp[o]	VSX Scalar Subtract Quad-Precision [using round to Odd]	647
xssubsp	VSX Scalar Subtract Single-Precision	649

Table 17.VSX Scalar BFP Elementary Arithmetic Instructions

Mnemonic	Instruction Name	Page
xsmaddadp	VSX Scalar Multiply-Add Type-A Double-Precision	570
xsmaddasp	VSX Scalar Multiply-Add Type-A Single-Precision	573

Table 18.VSX Scalar BFP Multiply-Add-class Instructions

Mnemonic	Instruction Name	Page
xsmaddmdp	VSX Scalar Multiply-Add Type-M Double-Precision	570
xsmaddmsp	VSX Scalar Multiply-Add Type-M Single-Precision	573
xsmaddqp[o]	VSX Scalar Multiply-Add Quad-Precision [using round to Odd]	576
xsmsubadp	VSX Scalar Multiply-Subtract Type-A Double-Precision	591
xsmsubasp	VSX Scalar Multiply-Subtract Type-A Single-Precision	594
xsmsubmdp	VSX Scalar Multiply-Subtract Type-M Double-Precision	591
xsmsubmsp	VSX Scalar Multiply-Subtract Type-M Single-Precision	594
xsmsubqp[o]	VSX Scalar Multiply-Subtract Quad-Precision [using round to Odd]	597
xsnmaddadp	VSX Scalar Negative Multiply-Add Type-A Double-Precision	608
xsnmaddasp	VSX Scalar Negative Multiply-Add Type-A Single-Precision	613
xsnmaddmdp	VSX Scalar Negative Multiply-Add Type-M Double-Precision	608
xsnmaddmsp	VSX Scalar Negative Multiply-Add Type-M Single-Precision	613
xsnmaddqp[o]	VSX Scalar Negative Multiply-Add Quad-Precision [using round to Odd]	616
xsnmsubadp	VSX Scalar Negative Multiply-Subtract Type-A Double-Precision	619
xsnmsubasp	VSX Scalar Negative Multiply-Subtract Type-A Single-Precision	622
xsnmsubmdp	VSX Scalar Negative Multiply-Subtract Type-M Double-Precision	619
xsnmsubmsp	VSX Scalar Negative Multiply-Subtract Type-M Single-Precision	622
xsnmsubqp[o]	VSX Scalar Negative Multiply-Subtract Quad-Precision [using round to Odd]	625

Table 18.VSX Scalar BFP Multiply-Add-class Instructions

Mnemonic	Instruction Name	Page
xsredp	VSX Scalar Reciprocal Estimate Double-Precision	632
xsresp	VSX Scalar Reciprocal Estimate Single-Precision	633
xsrsqrtedp	VSX Scalar Reciprocal Square Root Estimate Double-Precision	639
xsrsqrtesp	VSX Scalar Reciprocal Square Root Estimate Single-Precision	640
xstdivdp	VSX Scalar Test for software Divide Double-Precision	651
xstsqrtdp	VSX Scalar Test for software Square Root Double-Precision	652

Table 19.VSX Scalar Software BFP Divide/Square Root Instructions

7.6.1.3.2 VSX Vector BFP Arithmetic Instructions

Mnemonic	Instruction Name	Page
xvadddp	VSX Vector Add Double-Precision	659
xvaddsp	VSX Vector Add Single-Precision	663
xvdivdp	VSX Vector Divide Double-Precision	696
xvdivsp	VSX Vector Divide Single-Precision	698
xvmuldp	VSX Vector Multiply Double-Precision	721
xvmulsp	VSX Vector Multiply Single-Precision	723
xvsqrtdp	VSX Vector Square Root Double-Precision	751
xvsqrtsp	VSX Vector Square Root Single-Precision	752
xvsubdp	VSX Vector Subtract Double-Precision	753
xvsubsp	VSX Vector Subtract Single-Precision	755

Table 20.VSX Vector BFP Elementary Arithmetic Instructions

Mnemonic	Instruction Name	Page
xvmaddadp	VSX Vector Multiply-Add Type-A Double-Precision	701
xvmaddasp	VSX Vector Multiply-Add Type-A Single-Precision	704
xvmaddmdp	VSX Vector Multiply-Add Type-M Double-Precision	701
xvmaddmsp	VSX Vector Multiply-Add Type-M Single-Precision	704
xvmsubadp	VSX Vector Multiply-Subtract Type-A Double-Precision	715
xvmsubasp	VSX Vector Multiply-Subtract Type-A Single-Precision	718
xvmsubmdp	VSX Vector Multiply-Subtract Type-M Double-Precision	715
xvmsubmsp	VSX Vector Multiply-Subtract Type-M Single-Precision	718
xvnmaddadp	VSX Vector Negative Multiply-Add Type-A Double-Precision	727
xvnmaddasp	VSX Vector Negative Multiply-Add Type-A Single-Precision	732
xvnmaddmdp	VSX Vector Negative Multiply-Add Type-M Double-Precision	727
xvnmaddmsp	VSX Vector Negative Multiply-Add Type-M Single-Precision	732
xvnmsubadp	VSX Vector Negative Multiply-Subtract Type-A Double-Precision	735
xvnmsubasp	VSX Vector Negative Multiply-Subtract Type-A Single-Precision	738
xvnmsubmdp	VSX Vector Negative Multiply-Subtract Type-M Double-Precision	735
xvnmsubmsp	VSX Vector Negative Multiply-Subtract Type-M Single-Precision	738

Table 21.VSX Vector BFP Multiply-Add-class Instructions

Mnemonic	Instruction Name	Page
xvredp	VSX Vector Reciprocal Estimate Double-Precision	744
xvresp	VSX Vector Reciprocal Estimate Single-Precision	745
xvrsqrtedp	VSX Vector Reciprocal Square Root Estimate Double-Precision	748
xvrsqrtesp	VSX Vector Reciprocal Square Root Estimate Single-Precision	750
xvtdivdp	VSX Vector Test for software Divide Double-Precision	757
xvtdivsp	VSX Vector Test for software Divide Single-Precision	758
xvtsqrtdp	VSX Vector Test for software Square Root Double-Precision	759
xvtsqrtsp	VSX Vector Test for software Square Root Single-Precision	759

Table 22.VSX Vector BFP Software Divide/Square Root Instructions

7.6.1.4 VSX Binary Floating-Point Compare Instructions

7.6.1.4.1 VSX Scalar BFP Compare Instructions

Mnemonic	Instruction Name	Page
xscmpodp	VSX Scalar Compare Ordered Double-Precision	527
xscmpoqp	VSX Scalar Compare Ordered Quad-Precision	529
xscmpudp	VSX Scalar Compare Unordered Double-Precision	530
xscmpuqp	VSX Scalar Compare Unordered Quad-Precision	532

Table 23.VSX Scalar BFP Compare Instructions

Mnemonic	Instruction Name	Page
xscmpeqdp	VSX Scalar Compare Equal Double-Precision	524
xscmpgedp	VSX Scalar Compare Greater Than or Equal Double-Precision	525
xscmpgtdp	VSX Scalar Compare Greater Than Double-Precision	526

Table 24.VSX Scalar BFP Predicate Compare Instructions

Mnemonic	Instruction Name	Page
xsmaxcdp	VSX Scalar Maximum Type-C Double-Precision	581
xsmaxdp	VSX Scalar Maximum Double-Precision	579
xsmaxjdp	VSX Scalar Maximum Type-J Double-Precision	583
xsmincdp	VSX Scalar Minimum Type-C Double-Precision	587
xsmindp	VSX Scalar Minimum Double-Precision	585
xsminjdp	VSX Scalar Minimum Type-J Double-Precision	589

Table 25.VSX Scalar BFP Maximum/Minimum Instructions

7.6.1.4.2 VSX Vector BFP Compare Instructions

Mnemonic	Instruction Name	Page
xvcmpeqdp[.]	VSX Vector Compare Equal To Double-Precision	665
xvcmpeqsp[.]	VSX Vector Compare Equal To Single-Precision	666
xvcmpgedp[.]	VSX Vector Compare Greater Than or Equal To Double-Precision	667
xvcmpgesp[.]	VSX Vector Compare Greater Than or Equal To Single-Precision	668
xvcmpgtdp[.]	VSX Vector Compare Greater Than Double-Precision	669
xvcmpgtsp[.]	VSX Vector Compare Greater Than Single-Precision	670

Table 26.VSX Vector BFP Predicate Compare Instructions

Mnemonic	Instruction Name	Page
xvmaxdp	VSX Vector Maximum Double-Precision	707
xvmaxsp	VSX Vector Maximum Single-Precision	709
xvmindp	VSX Vector Minimum Double-Precision	711
xvminsp	VSX Vector Minimum Single-Precision	713

Table 27.VSX Vector BFP Maximum/Minimum Instructions

7.6.1.5 VSX Binary Floating-Point Round to Shorter Precision Instructions

Mnemonic	Instruction Name	Page
xsrqpxp	VSX Scalar Round Quad-Precision to Double-Extended-Precision	636
xsrsp	VSX Scalar Round Double-Precision to Single-Precision	638

Table 28.VSX Scalar BFP Round to Shorter Precision Instructions

7.6.1.6 VSX Binary Floating-Point Convert to Shorter Precision Instructions

Mnemonic	Instruction Name	Page
xscvdphp	VSX Scalar Convert w/ round Double-Precision to Half-Precision format	534
xscvdpsp	VSX Scalar Convert w/ round Double-Precision to Single-Precision format	536
xscvdpspn	VSX Scalar Convert Double-Precision to Single-Precision format Non-signalling	537
xscvqpdp[o]	VSX Scalar Convert w/ round Quad-Precision to Double-Precision format [using round to Odd]	638

Table 29.VSX Scalar BFP Convert to Shorter Precision Instructions

Mnemonic	Instruction Name	Page
xvcvdpsp	VSX Vector Convert w/ round Double-Precision to Single-Precision format	672
xvcvsphp	VSX Vector Convert w/ round Single-Precision to Half-Precision format	683

Table 30.VSX Vector BFP Convert to Shorter Precision Instructions

7.6.1.7 VSX Binary Floating-Point Convert to Longer Precision Instructions

Mnemonic	Instruction Name	Page
xscvdpqp	VSX Scalar Convert Double-Precision to Quad-Precision format	535
xscvhpdp	VSX Scalar Convert Half-Precision to Double-Precision format	546
xscvspdp	VSX Scalar Convert Single-Precision to Double-Precision format	557
xscvspdpn	VSX Scalar Convert Single-Precision to Double-Precision format Non-signalling	558

Table 31.VSX Scalar BFP Convert to Longer Precision Instructions

Mnemonic	Instruction Name	Page
xvcvhpsp	VSX Vector Convert Half-Precision to Single-Precision format	681
xvcvspdp	VSX Vector Convert Single-Precision to Double-Precision format	682

Table 32.VSX Vector BFP Convert to Longer Precision Instructions

7.6.1.8 VSX Binary Floating-Point Round to Integral Instructions

7.6.1.8.1 VSX Scalar BFP Round to Integral Instructions

Mnemonic	Instruction Name	Page
xsrdpi	VSX Scalar Round to Double-Precision Integer using round to Nearest Away	628
xsrdpic	VSX Scalar Round to Double-Precision Integer Exact using Current rounding mode	629
xsrdpim	VSX Scalar Round to Double-Precision Integer using round towards -Infinity	630
xsrdpip	VSX Scalar Round to Double-Precision Integer using round towards +Infinity	630
xsrdpiz	VSX Scalar Round to Double-Precision Integer using round towards Zero	631
xsrqpi	VSX Scalar Round to Quad-Precision Integer	634
xsrqpix	VSX Scalar Round Quad-Precision to Integral Exact	634
xvrdpi	VSX Vector Round to Double-Precision Integer using round to Nearest Away	741
xvrdpic	VSX Vector Round to Double-Precision Integer Exact using Current rounding mode	741
xvrdpim	VSX Vector Round to Double-Precision Integer using round towards -Infinity	742
xvrdpip	VSX Vector Round to Double-Precision Integer using round towards +Infinity	742
xvrdpiz	VSX Vector Round to Double-Precision Integer using round towards Zero	743

Table 33.VSX Scalar BFP Round to Integral Instructions

7.6.1.8.2 VSX Vector BFP Round to Integral Instructions

Mnemonic	Instruction Name	Page
xvrdpi	VSX Vector Round to Double-Precision Integer using round to Nearest Away	741
xvrdpic	VSX Vector Round to Double-Precision Integer Exact using Current rounding mode	741
xvrdpim	VSX Vector Round to Double-Precision Integer using round towards -Infinity	742
xvrdpip	VSX Vector Round to Double-Precision Integer using round towards +Infinity	742
xvrdpiz	VSX Vector Round to Double-Precision Integer using round towards Zero	743
xvrspi	VSX Vector Round to Single-Precision Integer using round to Nearest Away	746
xvrspic	VSX Vector Round to Single-Precision Integer Exact using Current rounding mode	746
xvrspim	VSX Vector Round to Single-Precision Integer using round towards -Infinity	747
xvrspip	VSX Vector Round to Single-Precision Integer using round towards +Infinity	747
xvrspiz	VSX Vector Round to Single-Precision Integer using round towards Zero	748

Table 34.VSX Vector BFP Round to Integral Instructions

7.6.1.9 VSX Binary Floating-Point Convert To Integer Instructions

7.6.1.9.1 VSX Scalar BFP Convert To Integer Instructions

Mnemonic	Instruction Name	Page
xscvdpsxds	VSX Scalar Convert w/ truncate Double-Precision to Signed Dword format	537
xscvdpsxws	VSX Scalar Convert w/ truncate Double-Precision to Signed Word format	540
xscvdpuxds	VSX Scalar Convert w/ truncate Double-Precision to Unsigned Dword format	542
xscvdpuxws	VSX Scalar Convert w/ truncate Double-Precision to Unsigned Word format	544
xscvqpsdz	VSX Scalar Convert w/ truncate Quad-Precision to Signed Dword format	548
xscvqpswz	VSX Scalar Convert w/ truncate Quad-Precision to Signed Word format	550
xscvqpudz	VSX Scalar Convert w/ truncate Quad-Precision to Unsigned Dword format	552
xscvqpuwz	VSX Scalar Convert w/ truncate Quad-Precision to Unsigned Word format	554

Table 35.VSX Scalar BFP Convert to Integer Instructions

7.6.1.9.2 VSX Vector BFP Convert To Integer Instructions

Mnemonic	Instruction Name	Page
xvcvdpsxds	VSX Vector Convert w/ truncate Double-Precision to Signed Dword format	673
xvcvdpsxws	VSX Vector Convert w/ truncate Double-Precision to Signed Word format	675
xvcvdpuxds	VSX Vector Convert w/ truncate Double-Precision to Unsigned Dword format	677
xvcvdpuxws	VSX Vector Convert w/ truncate Double-Precision to Unsigned Word format	679
xvcvspsxds	VSX Vector Convert w/ truncate Single-Precision to Signed Dword format	684
xvcvspsxws	VSX Vector Convert w/ truncate Single-Precision to Signed Word format	686
xvcvspuxds	VSX Vector Convert w/ truncate Single-Precision to Unsigned Dword format	688
xvcvspuxws	VSX Vector Convert w/ truncate Single-Precision to Unsigned Word format	690

Table 36.VSX Vector BFP Convert To Integer Instructions

7.6.1.10 VSX Binary Floating-Point Convert From Integer Instructions

7.6.1.10.1 VSX Scalar BFP Convert From Integer Instructions

Mnemonic	Instruction Name	Page
xscvsdqp	VSX Scalar Convert Signed Dword to Quad-Precision format	556
xscvsxddp	VSX Scalar Convert w/ round Signed Dword to Double-Precision format	559
xscvsxdsp	VSX Scalar Convert w/ round Signed Dword to Single-Precision format	559
xscvudqp	VSX Scalar Convert Unsigned Dword to Quad-Precision format	560
xscvuxddp	VSX Scalar Convert w/ round Unsigned Dword to Double-Precision format	561
xscvuxdsp	VSX Scalar Convert w/ round Unsigned Dword to Single-Precision format	561

Table 37.VSX Scalar BFP Convert from Integer Instructions

7.6.1.10.2 VSX Vector BFP Convert From Integer Instructions

Mnemonic	Instruction Name	Page
xvcvsxddp	VSX Vector Convert w/ round Signed Dword to Double-Precision format	692
xvcvsxwdp	VSX Vector Convert Signed Word to Double-Precision format	693
xvcvuxddp	VSX Vector Convert w/ round Unsigned Dword to Double-Precision format	694
xvcvuxwdp	VSX Vector Convert Unsigned Word to Double-Precision format	695
xvcvsxdsp	VSX Vector Convert w/ round Signed Dword to Single-Precision format	692
xvcvsxwsp	VSX Vector Convert w/ round Signed Word to Single-Precision format	693
xvcvuxdsp	VSX Vector Convert w/ round Unsigned Dword to Single-Precision format	694
xvcvuxwsp	VSX Vector Convert w/ round Unsigned Word to Single-Precision format	695

Table 38.VSX Vector BFP Convert From Integer Instructions

7.6.1.11 VSX Binary Floating-Point Math Support Instructions

7.6.1.11.1 VSX Scalar BFP Math Support Instructions

Mnemonic	Instruction Name	Page
xscmpexpdp	VSX Scalar Compare Exponents Double-Precision	522
xscmpexpqp	VSX Scalar Compare Exponents Quad-Precision	523
xsiexpdp	VSX Scalar Insert Exponent Double-Precision	568
xsiexpqp	VSX Scalar Insert Exponent Quad-Precision	569
xststdcdp	VSX Scalar Test Data Class Double-Precision	653
xststdcqp	VSX Scalar Test Data Class Quad-Precision	654
xststdcsp	VSX Scalar Test Data Class Single-Precision	655
xsxexpdp	VSX Scalar Extract Exponent Double-Precision	656
xsxexpqp	VSX Scalar Extract Exponent Quad-Precision	656

Table 39. VSX Scalar BFP Math Support Instructions

Mnemonic	Instruction Name	Page
xsxsigdp	VSX Scalar Extract Significand Double-Precision	657
xsxsigqp	VSX Scalar Extract Significand Quad-Precision	657

Table 39. VSX Scalar BFP Math Support Instructions

7.6.1.11.2 VSX Vector BFP Math Support Instructions

Mnemonic	Instruction Name	Page
xviexpdp	VSX Vector Insert Exponent Double-Precision	700
xviexpsp	VSX Vector Insert Exponent Single-Precision	700
xvtstdcdp	VSX Vector Test Data Class Double-Precision	760
xvtstdcsp	VSX Vector Test Data Class Single-Precision	761
xvxexpdp	VSX Vector Extract Exponent Double-Precision	762
xvxexpsp	VSX Vector Extract Exponent Single-Precision	762
xvxsigdp	VSX Vector Extract Significand Double-Precision	763
xvxsigsp	VSX Vector Extract Significand Single-Precision	763

Table 40. VSX Vector BFP Math Support Instructions

7.6.1.12 VSX Vector Logical Instructions

7.6.1.12.1 VSX Vector Logical Instructions

Mnemonic	Instruction Name	Page
xxland	VSX Vector Logical AND	767
xxlandc	VSX Vector Logical AND with Complement	767
xxleqv	VSX Vector Logical Equivalence	768
xxlnand	VSX Vector Logical NAND	768
xxlnor	VSX Vector Logical NOR	769
xxlor	VSX Vector Logical OR	770
xxlorc	VSX Vector Logical OR with Complement	769
xxlxor	VSX Vector Logical XOR	770

Table 41.VSX Logical Instructions

7.6.1.12.2 VSX Vector Select Instruction

Mnemonic	Instruction Name	Page
xxsel	VSX Vector Select	773

Table 42.VSX Vector Select Instruction

7.6.1.13 VSX Vector Permute-class Instructions

7.6.1.13.1 VSX Vector Byte-Reverse Instructions

Mnemonic	Instruction Name	Page
xxbrd	VSX Vector Byte-Reverse Dword	764
xxbrh	VSX Vector Byte-Reverse Hword	764
xxbrq	VSX Vector Byte-Reverse Qword	765
xxbrw	VSX Vector Byte-Reverse Word	765

Table 43.VSX Vector Byte-Reverse Instructions

7.6.1.13.2 VSX Vector Insert/Extract Instructions

Mnemonic	Instruction Name	Page
xxextractuw	VSX Vector Extract Unsigned Word	766
xxinsertw	VSX Vector Insert Word	766

Table 44.VSX Vector Insert/Extract Instructions

7.6.1.13.3 VSX Vector Merge Instructions

Mnemonic	Instruction Name	Page
xxmrghw	VSX Vector Merge High Word	771
xxmrglw	VSX Vector Merge Low Word	771

Table 45.VSX Vector Merge Instructions

7.6.1.13.4 VSX Vector Splat Instructions

Mnemonic	Instruction Name	Page
xxspltib	VSX Vector Splat Immediate Byte	774
xxspltw	VSX Vector Splat Word	774

Table 46.VSX Vector Splat Instructions

7.6.1.13.5 VSX Vector Permute Instructions

Mnemonic	Instruction Name	Page
xxpermdi	VSX Vector Permute Dword Immediate	773
xxperm	VSX Vector Permute	772
xxpermr	VSX Vector Permute Right-indexed	772

Table 47.VSX Vector Permute Instruction

7.6.1.13.6 VSX Vector Shift Left Double Instructions

Mnemonic	Instruction Name	Page
xxsldwi	VSX Vector Shift Left Double by Word Immediate	774

Table 48.VSX Vector Shift Left Double Instruction

7.6.2 VSX Instruction Description Conventions

7.6.2.1 VSX Instruction RTL Operators

x.bit[y]
Return the contents of bit y of x.

x.bit[y:z]
Return the contents of bits y:z of x.

x.word[y]
Return the contents of word element y of x.

x.word[y:z]
Return the contents of word elements y:z of x.

x.dword[y]
Return the contents of doubleword element y of x.

x.dword[y:z]
Return the contents of doubleword elements y:z of x.

x = y
The value of y is placed into x.

x |= y
The value of y is ORed with the value x and placed into x.

~x
Return the one's complement of x.

!x
Return 1 if the contents of x are equal to 0, otherwise return 0.

x || y
Return the value of x concatenated with the value of y. For example, `0b010 || 0b111` is the same as `0b010111`.

x ^ y
Return the value of x exclusive ORed with the value of y.

x ? y : z
If the value of x is true, return the value of y, otherwise return the value z.

x+y
x and y are integer values.

Return the sum of x and y.

x–y
x and y are integer values.

Return the difference of x and y.

x!=y
x and y are integer values.

Return 1 if x is not equal to y, otherwise return 0.

x<=y
x and y are integer values.

Return 1 if x is less than or equal to y, otherwise return 0.

x>=y
x and y are integer values.

Return 1 if x is greater than or equal to y, otherwise return 0.

7.6.2.2 VSX Instruction RTL Function Calls

AddDP(x,y)

x and y are double-precision floating-point values.

If x or y is an SNaN, vxsnan_flag is set to 1.

If x is an Infinity and y is an Infinity of the opposite sign, vxisi_flag is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x and y are infinities of opposite sign, return the standard QNaN.
Otherwise, return the normalized sum of x and y, having unbounded range and precision.

AddSP(x,y)

x and y are single-precision floating-point values.

If x or y is an SNaN, vxsnan_flag is set to 1.

If x is an Infinity and y is an Infinity of the opposite sign, vxisi_flag is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x and y are infinities of opposite sign, return the standard QNaN.
Otherwise, return the normalized sum of x added to y, having unbounded range and precision.

bfp_ABSOLUTE(x)

x is a binary floating-point value represented in the working floating-point format.

Return x with sign set to 0.

bfp_ADD(x, y)

x is a binary floating-point value represented in the working floating-point format.
y is a binary floating-point value represented in the working floating-point format.

If x or y is an SNaN, vxsnan_flag is set to 1.

If x is an infinity and y is an infinity of the opposite sign, vxisi_flag is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x and y are infinities of opposite sign, return the standard QNaN.
Otherwise, return the normalized sum of x and y, having unbounded range and precision.

bfp_COMPARE_EQ(x, y)

x is a binary floating-point value represented in the working floating-point format.
y is a binary floating-point value represented in the working floating-point format.

Return 0b0 if x is NaN or y is a NaN.
Otherwise, return 0b1 if x is a Zero and y is a Zero.
Otherwise, return 0b1 if x is equal to y.
Otherwise, return 0b0.

bfp_COMPARE_GT(x, y)

x is a binary floating-point value represented in the working floating-point format.
y is a binary floating-point value represented in the working floating-point format.

Return 0b0 if x is NaN or y is a NaN.
Otherwise, return 0b0 if x is a Zero and y is a Zero.
Otherwise, return 0b1 if x is greater than y.
Otherwise, return 0b0.

bfp_COMPARE_LT(x, y)

x is a binary floating-point value represented in the working floating-point format.
y is a binary floating-point value represented in the working floating-point format.

Return 0b0 if x is NaN or y is a NaN.
Otherwise, return 0b0 if x is a Zero and y is a Zero.
Otherwise, return 0b1 if x is less than y.
Otherwise, return 0b0.

bfp_CONVERT_FROM_BFP16(x)

x is a floating-point value represented in half-precision format.

Let exponent be the contents of bits 1:5 of x.
Let fraction be the contents of bits 6:15 of x.

Let result.sign be set to 0.
Let result.exponent be set to 0.
Let result.significand be set to 0.
Let result.class.SNaN be set to 0.
Let result.class.QNaN be set to 0.
Let result.class.Infinity be set to 0.
Let result.class.Zero be set to 0.
Let result.class.Denormal be set to 0.
Let result.class.Normal be set to 0.

If x is a SNaN, do the following.
result.class.SNaN is set to 1.
result.sign is set to the contents of bit 0 of x.

The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:10 of result.significand are set to the value of fraction.

Otherwise, if x is a QNaN, do the following.
result.class.QNaN is set to 1.
result.sign is set to the contents of bit 0 of x.

The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:10 of result.significand are set to the value of fraction.

Otherwise, if x is an Infinity value, do the following.
result.class.Infinity is set to 1.
result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Zero value, do the following.
result.class.Zero is set to 1.
result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Denormal value, do the following.

result.class.Denormal is set to 1.
result.sign is set to the contents of bit 0 of x.
result.exp is set to the value -14.

The contents of bit 0 of result.significand are set to 0.
The contents of bits 1:10 of result.significand are set to the value of fraction.

result.significand is shifted left until the contents bit 0 of result.significand are equal to 1.
result.exponent is decremented by the the number of bits result.significand was shifted.

Otherwise, do the following.

result.class.Normal is set to 1.
result.sign is set to the contents of bit 0 of x.
result.exp is set to the value of exponent subtracted by 15.

The contents of bit 0 of result.significand are set to 1.
The contents of bits 1:10 of result.significand are set to the value of fraction.

Return result.

bfp_CONVERT_FROM_BFP32(x)

x is a floating-point value represented in single-precision format.

Let exponent be the contents of bits 1:8 of x.
Let fraction be the contents of bits 9:31 of x.

Let result.sign be initialized to 0.
Let result.exponent be initialized to 0.
Let result.significand be initialized to 0.
Let result.class.SNaN be initialized to 0.
Let result.class.QNaN be initialized to 0.
Let result.class.Infinity be initialized to 0.
Let result.class.Zero be initialized to 0.
Let result.class.Denormal be initialized to 0.
Let result.class.Normal be initialized to 0.

If x is a SNaN, do the following.
    result.class.SNaN is set to 1.
    result.sign is set to the contents of bit 0 of x.

    The contents of bit 0 of result.significand are set to 0.
    The contents of bits 1:23 of result.significand are set to the value of fraction.

Otherwise, if x is a QNaN, do the following.
    result.class.QNaN is set to 1.
    result.sign is set to the contents of bit 0 of x.

    The contents of bit 0 of result.significand are set to 0.
    The contents of bits 1:23 of result.significand are set to the value of fraction.

Otherwise, if x is an Infinity value, do the following.
    result.class.Infinity is set to 1.
    result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Zero value, do the following.
    result.class.Zero is set to 1.
    result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Denormal value, do the following.
    result.class.Denormal is set to 1.
    result.sign is set to the contents of bit 0 of x.
    result.exponent is set to the value -126.

    The contents of bit 0 of result.significand are set to 0.
    The contents of bits 1:23 of result.significand are set to the value of fraction.

    result.significand is shifted left until the contents bit 0 of result.significand are equal to 1.
    result.exponent is decremented by the the number of bits result.significand was shifted.

Otherwise, do the following.
    result.class.Normal is set to 1.
    result.sign is set to the contents of bit 0 of x.
    result.exponent is set to the value of exponent subtracted by 127.

    The contents of bit 0 of result.significand are set to 1.
    The contents of bits 1:23 of result.significand are set to the value of fraction.

Return result.

bfp_CONVERT_FROM_BFP64(x)

x is a binary floating-point value represented in double-precision format.

Let exponent be the contents of bits 1:11 of x.
Let fraction be the contents of bits 12:63 of x.

result.sign is initialized to 0.
result.exponent is initialized to 0.
result.significand is initialized to 0.
result.class.SNaN is initialized to 0.
result.class.QNaN is initialized to 0.
result.class.Infinity is initialized to 0.
result.class.Zero is initialized to 0.
result.class.Denormal is initialized to 0.
result.class.Normal is initialized to 0.

If x is a SNaN, do the following.
- result.class.SNaN is set to 1.
- result.sign is set to the contents of bit 0 of x.
- The contents of bit 0 of result.significand are set to 0.
- The contents of bits 1:52 of result.significand are set to the value of fraction.
- The contents of the rest of result.significand are set to 0.

Otherwise, if x is a QNaN, do the following.
- result.class.QNaN is set to 1.
- result.sign is set to the contents of bit 0 of x.
- The contents of bit 0 of result.significand are set to 0.
- The contents of bits 1:52 of result.significand are set to the value of fraction.
- The contents of the rest of result.significand are set to 0.

Otherwise, if x is an Infinity, do the following.
- result.class.Infinity is set to 1.
- result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Zero, do the following.
- result.class.Zero is set to 1.
- result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Denormal, do the following.
- result.class.Denormal is set to 1.
- result.sign is set to the contents of bit 0 of x.
- result.exp is set to the value -1022.
- The contents of bit 0 of result.significand are set to 0.
- The contents of bits 1:52 of result.significand are set to the value of fraction.
- The contents of the rest of result.significand are set to 0.
- result.significand is shifted left until the contents bit 0 of result.significand are equal to 1.
- result.exponent is decremented by the the number of bits result.significand was shifted.

Otherwise, do the following.
- result.class.Normal is set to 1.
- result.sign is set to the contents of bit 0 of x.
- result.exp is set to the value of exponent subtracted by 1023.
- The contents of bit 0 of result.significand are set to 1.
- The contents of bits 1:52 of result.significand are set to the value of fraction.
- The contents of the rest of result.significand are set to 0.

Return result (i.e., the value x in the working floating-point format).

bfp_CONVERT_FROM_BFP128(x)

x is a binary floating-point value represented in quad-precision format.

Let exponent be the contents of bits 1:15 of x.
Let fraction be the contents of bits 16:127 of x.

result.sign is initialized to 0.
result.exponent is initialized to 0.
result.significand is initialized to 0.
result.class.SNaN is initialized to 0.
result.class.QNaN is initialized to 0.
result.class.Infinity is initialized to 0.
result.class.Zero is initialized to 0.
result.class.Denormal is initialized to 0.
result.class.Normal is initialized to 0.

If x is a SNaN, do the following.
- result.class.SNaN is set to 1.
- result.sign is set to the contents of bit 0 of x.
- The contents of bit 0 of result.significand are set to 0.
- The contents of bits 1:112 of result.significand are set to the value of fraction.
- The contents of the rest of result.significand are set to 0.

Otherwise, if x is a QNaN, do the following.
- result.class.QNaN is set to 1.
- result.sign is set to the contents of bit 0 of x.
- The contents of bit 0 of result.significand are set to 0.
- The contents of bits 1:112 of result.significand are set to the value of fraction.
- The contents of the rest of result.significand are set to 0.

Otherwise, if x is an Infinity, do the following.
- result.class.Infinity is set to 1.
- result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Zero, do the following.
- result.class.Zero is set to 1.
- result.sign is set to the contents of bit 0 of x.

Otherwise, if x is a Denormal, do the following.
- result.class.Denormal is set to 1.
- result.sign is set to the contents of bit 0 of x.
- result.exp is set to the value -16382.
- The contents of bit 0 of result.significand are set to 0.
- The contents of bits 1:112 of result.significand are set to the value of fraction.
- The contents of the rest of result.significand are set to 0.
- result.significand is shifted left until the contents bit 0 of result.significand are equal to 1.
- result.exponent is decremented by the the number of bits result.significand was shifted.

Otherwise, do the following.
- result.class.Normal is set to 1.
- result.sign is set to the contents of bit 0 of x.
- result.exp is set to the value of exponent subtracted by 16383.
- The contents of bit 0 of result.significand are set to 1.
- The contents of bits 1:112 of result.significand are set to the value of fraction.
- The contents of the rest of result.significand are set to 0.

Return result (i.e., the value x in the working floating-point format).

bfp_CONVERT_FROM_SI64(x)

x is an integer value represented in signed doubleword integer format.

result.sign is initialized to 0.
result.exponent is initialized to 0.
result.significand is initialized to 0.
result.class.SNaN is initialized to 0.
result.class.QNaN is initialized to 0.
result.class.Infinity is initialized to 0.
result.class.Zero is initialized to 0.
result.class.Denormal is initialized to 0.
result.class.Normal is initialized to 0.

If x is equal to 0x0000_0000_0000_0000,
    result.class.Zero is set to 1.

Otherwise, do the following.
    result.class.Normal is set to 1.
    result.sign is set to the contents of bit 0 of x.
    result.exponent is set to the value 64.
    Bits 0:64 of result.significand are set to the value of x sign-extended to 65 bits.

    If bit 0 of result.significand is equal to 1,
        result.sign is set to 1, and
        result.significand is set to the value of the two's complement of result.significand.

    If bit 0 of result.significand is equal to 0,
        result.significand is shifted left until bit 0 of result.significand is equal to 1, and
        result.exponent is decremented by the number of bits result.significand is shifted.

Return result (i.e., the value x in the working floating-point format).

bfp_CONVERT_FROM_UI64(x)

x is an integer value represented in unsigned doubleword integer format.

Return x in the working floating-point format.

result.sign is initialized to 0.
result.exponent is initialized to 0.
result.significand is initialized to 0.
result.class.SNaN is initialized to 0.
result.class.QNaN is initialized to 0.
result.class.Infinity is initialized to 0.
result.class.Zero is initialized to 0.
result.class.Denormal is initialized to 0.
result.class.Normal is initialized to 0.

If x is equal to 0x0000_0000_0000_0000, do the following.
result.class.Zero is set to 1.

Otherwise, do the following.
result.class.Normal is set to 1.
result.sign is set to 0.
result.exponent is set to the value 64.
Bits 0:64 of result.significand is set to the value of x zero-extended to 65 bits.

If bit 0 of result.significand is equal to 0, result.significand is shifted left until bit 0 of result.significand is equal to 1 and result.exponent is decremented by the number of bits result.significand is shifted.

Return result (i.e., the value x in the working floating-point format).

bfp_CONVERT_TO_BFP16(x)

x is a floating-point value represented in the working format.

If x.class.QNaN=1, do the following.
Bit 0 of result is set to the value of x.sign.
Bits 1:5 of result are set to the value 0b11111.
Bits 6:15 of result are set to the value of bits 1:10 of x.significand.

Otherwise, if x.class.Infinity=1, do the following.
Bit 0 of result is set to the value of x.sign.
Bits 1:5 of result are set to the value 0b11111.
Bits 6:15 of result are set to 0.

Otherwise, if x.class.Zero=1, do the following.
Bit 0 of result is set to the value of x.sign.
Bits 1:15 of result are set to 0.

Otherwise, if x.exponent is less than -14 and UE=0, do the following.
Bit 0 of result is set to the value of x.sign.
sh_cnt is set to the difference, -14 - x.exponent.
Bits 1:5 of result are set to 0b00000.
Bits 6:15 of result are set to bits 1:10 of x.significand shifted right by sh_cnt bits.

Otherwise, if x.exponent is less than -14 and UE=1, result is undefined.
Otherwise, if x.exponent is greater than 15 and OE=1, result is undefined.

Otherwise, do the following.
Bit 0 of result is set to the value of x.sign.
Bits 1:5 of result are set to the sum, x.exponent + 15.
Bits 6:15 of result are set to bits 1:10 of x.significand.

Return result.

bfp_CONVERT_TO_BFP32(x)
x is a floating-point value represented in the working format.

If x.class.QNaN=1, do the following.
Bit 0 of result is set to the value of x.sign.
Bits 1:8 of result are set to the value 0b1111_1111.
Bits 9:31 of result are set to the value of bits 1:23 of x.significand.

Otherwise, if x.class.Infinity=1, do the following.
Bit 0 of result is set to the value of x.sign.
Bits 1:9 of result are set to the value 0b1111_1111.
Bits 9:31 of result are set to 0.

Otherwise, if x.class.Zero=1, do the following.
Bit 0 of result is set to the value of x.sign.
Bits 1:31 of result are set to 0.

Otherwise, if x.exponent is less than -126 and UE=0, do the following.
Bit 0 of result is set to the value of x.sign.
sh_cnt is set to the difference, -126 - x.exponent.
Bits 1:8 of result are set to 0b0000_0000.
Bits 9:31 of result are set to bits 1:23 of x.significand shifted right by sh_cnt bits.

Otherwise, if x.exponent is less than -126 and UE=1, result is undefined.
Otherwise, if x.exponent is greater than 127 and OE=1, result is undefined.

Otherwise, do the following.
Bit 0 of result is set to the value of x.sign.
Bits 1:8 of result are set to the sum, x.exponent + 127.
Bits 9:31 of result are set to bits 1:23 of x.significand.

Return result.

bfp_CONVERT_TO_BFP64(x)

x is a floating-point value represented in the working format.

If x.class.QNaN=1, do the following.
- Bit 0 of result is set to the value of x.sign.
- Bits 1:11 of result are set to the value 0b111_1111_1111.
- Bits 12:63 of result are set to the value of bits 1:52 of x.significand.

Otherwise, if x.class.Infinity=1, do the following.
- Bit 0 of result is set to the value of x.sign.
- Bits 1:11 of result are set to the value 0b111_1111_1111.
- Bits 12:63 of result are set to 0.

Otherwise, if x.class.Zero=1, do the following.
- Bit 0 of result is set to the value of x.sign.
- Bits 1:63 of result are set to 0.

Otherwise, if x.exponent is less than -1022 and UE=0, do the following.
- Bit 0 of result is set to the value of x.sign.
- sh_cnt is set to the difference, -1022 - x.exponent.
- Bits 1:11 of result are set to 0b000_0000_0000.
- Bits 12:63 of result are set to bits 1:52 of x.significand shifted right by sh_cnt bits.

Otherwise, if x.exponent is less than -1022 and UE=1, result is undefined.
Otherwise, if x.exponent is greater than 1023 and OE=1, result is undefined.

Otherwise, do the following.
- Bit 0 of result is set to the value of x.sign.
- Bits 1:11 of result are set to the sum, x.exponent + 1023.
- Bits 12:63 of result are set to bits 1:52 of x.significand.

Return result.

bfp_CONVERT_TO_BFP128(x)

x is a quad-precision floating-point value that is represented in the working floating-point format.

If x is a QNaN,
the contents of bit 0 of result are set to the value of x.sign,
the contents of bits 1:15 of result are set to the value 0b111_1111_1111_1111, and
the contents of bits 16:127 of result are set to the value of bits 1:112 of x.significand.

Otherwise, if x is a Zero,
the contents of bit 0 of result are set to the value of x.sign, and
the contents of bits 1:15 of result are set to the value 0b000_0000_0000_0000, and
the contents of bits 16:127 of result are set to the value 0x0000_0000_0000_0000_0000_0000_0000.

Otherwise, if x is an Infinity,
the contents of bit 0 of result are set to the value of x.sign,
the contents of bits 1:15 of result are set to the value 0b111_1111_1111_1111, and
the contents of bits 16:127 of result are set to the value 0x0000_0000_0000_0000_0000_0000_0000.

Otherwise, do the following.
If the exponent of x is less than -16382,
the contents of bit 0 of result are set to the value of x.sign,
the contents of bits 1:15 of result are set to the value 0b000_0000_0000_0000, and
the contents of bits 16:127 of result are set to the value of bits 1:112 of the significand of x shifted right by N bits, where N is the value -16382 subtracted by the value of the exponent of x.
Otherwise,
the contents of bit 0 of result are set to the value of x.sign,
the contents of bits 1:15 of result are set to the sum of the exponent of x and 16383, and
the contents of bits 16:127 of result are set to the value of bits 1:112 of the significand of x.

Return result (i.e., x in quad-precision format).

bfp_CONVERT_TO_SI64(x)

x is an integer value represented in the working floating-point format.

Return the value x in signed doubleword integer format.

bfp_CONVERT_TO_UI64(x)

x is an integer value represented in the working floating-point format.

Return the value x in 64-bit unsigned integer format.

bfp_DENORM(x, y)

x is an integer value specifying the target format's Emin value.
y is a binary floating-point value that is represented in the working floating-point format.

If y.exponent is less than Emin, let sh_cnt be the value Emin - y.exponent.
Otherwise, let sh_cnt be the value 0.

y.significand, having unbounded precision, is shifted right by sh_cnt bits.
y.exponent is incremented by sh_cnt.

Return y in the working floating-point format.

bfp_DIVIDE(x, y)

x is a binary floating-point value that is represented in the working floating-point format.
y is a binary floating-point value that is represented in the working floating-point format.

If x or y is an SNaN, vxsnan_flag is set to 1.
Otherwise, if x and y are infinities, vxidi_flag is set to 1.
Otherwise, if x and y are zeros, vxzdz_flag is set to 1.
Otherwise, if x is a finite value and y is a zero, zx_flag is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x and y are infinities, return the standard QNaN.
Otherwise, if x and y are zeros, return the standard QNaN.
Otherwise, if y is a zero, return infinity, having the sign of the exclusive-OR of the signs of x and y.
Otherwise, return the normalized quotient of x ÷ y, having unbounded range and precision.

bfp_INFINITY()

Return a positive floating-point infinity value, represented in the working format.

```
bfp_INITIALIZE(result)
result.class.Infinity ← 1
return(result)
```

bfp_INITIALIZE(x)

Let x.sign be set to 0.
Let x.exponent be set to 0.
Let x.significand be set to 0.
Let x.class.SNaN be set to 0.
Let x.class.QNaN be set to 0.
Let x.class.Infinity be set to 0.
Let x.class.Zero be set to 0.
Let x.class.Denormal be set to 0.
Let x.class.Normal be set to 0.

Return x.

bfp_MULTIPLY(x, y)

x is a binary floating-point value represented in the working floating-point format.
y is a binary floating-point value represented in the working floating-point format.

If x or y is an SNaN, vxsnan_flag is set to 1.
Otherwise, if x is an infinity and y is a zero, vximz_flag is set to 1.
Otherwise, if x is a zero and y is an infinity, vximz_flag is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x is an infinity and y is a zero, return the standard QNaN.
Otherwise, if x is a zero and y is an infinity, return the standard QNaN.
Otherwise, return the normalized product of x × y, having unbounded range and precision.

bfp_MULTIPLY_ADD(x, y, z)

x is a binary floating-point value represented in the working floating-point format.
y is a binary floating-point value represented in the working floating-point format.
z is a binary floating-point value represented in the working floating-point format.

If x, y, or z is an SNaN, vxsnan_flag is set to 1.
Otherwise, if x is an infinity and y is a zero, vximz_flag is set to 1.
Otherwise, if x is a zero and y is an infinity, vximz_flag is set to 1.
Otherwise, if z and the product of x × y are Infinity values having opposite signs, vxisi_flag is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if z is a QNaN, return z.
Otherwise, if z is an SNaN, return z represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x is an infinity and y is a zero, return the standard QNaN.
Otherwise, if x is a zero and y is an infinity, return the standard QNaN.
Otherwise, if z and the product of x × y are Infinity values having opposite signs, return the standard QNaN.
Otherwise, return the sum of z and the normalized product of x × y, having unbounded range and precision.

bfp_NEGATE(x)

x is a binary floating-point value that is represented in the working floating-point format.

Return x with its sign complemented.

bfp_NMAX_BFP16()

Return the largest, positive, normalized half-precision floating-point value, $(2-2^{-10})\times2^{+15}$, represented in the working format.

```
bfp_INITIALIZE(result)
result.exponent ← +15
result.significand.bit[0:10] ← 0b111_1111_1111
result.class.Normal ← 1
return(result)
```

bfp_NMAX_BFP64

Return the largest finite double-precision value (i.e., $2^{1024}-2^{1024-53}$) in the working floating-point format.

```
return( bfp_CONVERT_FROM_BFP64(0x7FEF_FFFF_FFFF_FFFF) )
```

bfp_NMAX_BFP80

Return the largest finite double-extended-precision value (i.e., $2^{16384}-2^{16384-65}$) in the working floating-point format.

```
return( bfp_CONVERT_FROM_BFP80(0x7FFE_FFFF_FFFF_FFFF_FFFF) )
```

bfp_NMAX_BFP128

Return the largest finite quad-precision value (i.e., $2^{16384}-2^{16384-113}$) in the working floating-point format.

```
return( bfp_CONVERT_FROM_BFP128(0x7FFE_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF) )
```

bfp_NMIN_BFP16()

Return the smallest, positive, normalized half-precision floating-point value, 2^{-14}, represented in the working format.

```
bfp_INITIALIZE(result)
result.exponent ← -14
result.significand.bit[0:10] ← 0b100_0000_0000
result.class.Normal ← 1
```

```
return(result)
```

bfp_NMIN_BFP64
Return the smallest, positive, normalized double-precision value, 2^{-1022}, represented in the *binary floating-point working format*.

```
return( bfp_CONVERT_FROM_BFP64(0x0010_0000_0000_0000) )
```

bfp_NMIN_BFP80
Return the smallest, positive, normalized double-extended-precision value, 2^{-16382}, represented in the *binary floating-point working format*.

```
return( bfp_CONVERT_FROM_BFP80(0x0001_0000_0000_0000_0000) )
```

bfp_NMIN_BFP128
Return the smallest, positive, normalized quad-precision value, 2^{-16382}, represented in the *binary floating-point working format*.

```
return( bfp_CONVERT_FROM_BFP128(0x0001_0000_0000_0000_0000_0000_0000_0000) )
```

bfp_QUIET(x)
`x` is a Signalling NaN.

Return `x` converted to a Quiet NaN with `x.class.QNaN` set to 1 and `x.class.SNaN` set to 0.

bfp_ROUND_CEIL(p, x)
`x` is a binary floating-point value that is represented in the working floating-point format and has unbounded exponent range and significand precision. `x` must be rounded as presented, without prenormalization.

`p` is an integer value specifying the precision (i.e., number of bits) the significand is rounded to.

Return the smallest floating-point number having unbounded exponent range and a significand with a width of `p` bits that is greater or equal in value to `x`.

`inc_flag` is set to 1 if the magnitude of the value returned is greater than `x`.
`xx_flag` is set to 1 if the value returned is not equal to `x`.

bfp_ROUND_FLOOR(p, x)
`x` is a binary floating-point value that is represented in the working floating-point format and has unbounded exponent range and significand precision. The value must be rounded as presented, without prenormalization.

`p` is an integer value specifying the precision (i.e., number of bits) the significand is rounded to.

Return the largest floating-point number having unbounded exponent range and a significand with a width of `p` bits that is lesser or equal in value to `x`.

`inc_flag` is set to 1 if the magnitude of the value returned is greater than `x`.
`xx_flag` is set to 1 if the value returned is not equal to `x`.

bfp_ROUND_TO_BFP16(x,y)

y is a normalized floating-point value represented in the working format, having unbounded exponent range and significand precision.

x is a 2-bit integer value specifying one of four rounding modes.

0b00	Round to Nearest Even
0b01	Round towards Zero
0b10	Round towards +Infinity
0b11	Round towards - Infinity

If y is an QNaN, Infinity, or Zero, return y. Otherwise, if y is an SNaN, set vxsnan_flag to 1 and return the corresponding QNaN representation of y. Otherwise, return the value y rounded to half-precision format's exponent range and significand precision using the rounding mode specified by x.

```
if y.class.Zero | y.class.Infinity then return(y)

if y.class.QNaN | y.class.SNaN then do
   result ← y
   result.significand.bit[1] ← 1
   result.significand.bit[11:inf] ← 0
   result.class.SNaN ← 0
   result.class.QNaN ← 1
   vxsnan_flag ← y.class.SNaN
   return(result)
end

if bfp_COMPARE_LT(y,bfp_NMIN_BFP16()) then do
   if FPSCR.UE=0 then do
      do while y.exponent < -14    // denormalize y
         y.significand ← y.significand >> 1
         y.exponent ← y.exponent + 1
      end
      if x=0b00 then result ← bfp_ROUND_TO_BFP16_NEAR_EVEN(y)
      if x=0b01 then result ← bfp_ROUND_TO_BFP16_TRUNC(y)
      if x=0b10 then result ← bfp_ROUND_TO_BFP16_CEIL(y)
      if x=0b11 then result ← bfp_ROUND_TO_BFP16_FLOOR(y)
      do while result.significand.bit[0] = 0    // normalize result
         result.significand ← result.significand << 1
         result.exponent ← result.exponent - 1
      end
      ux_flag ← xx_flag
      return(result)
   end
   else do
      y.exponent ← y.exponent + 24
      ux_flag ← 1
   end
end

if x=0b00 then result ← bfp_ROUND_TO_BFP16_NEAR_EVEN(y)
if x=0b01 then result ← bfp_ROUND_TO_BFP16_TRUNC(y)
if x=0b10 then result ← bfp_ROUND_TO_BFP16_CEIL(y)
if x=0b11 then result ← bfp_ROUND_TO_BFP16_FLOOR(y)
```

```
        if bfp_COMPARE_GT(result, bfp_NMAX_BFP16()) then do
           if OE=0 then do
              if x=0b00 then result ← sign ? bfp_NEGATE(bfp_INFINITY())   : bfp_INFINITY()
              if x=0b01 then result ← sign ? bfp_NEGATE(bfp_NMAX_BFP16()) : bfp_NMAX_BFP16()
              if x=0b10 then result ← sign ? bfp_NEGATE(bfp_NMAX_BFP16()) : bfp_INFINITY()
              if x=0b11 then result ← sign ? bfp_NEGATE(bfp_INFINITY())   : bfp_NMAX_BFP16()
              ox_flag  ← 0b1
              xx_flag  ← 0b1
              inc_flag ← 0bU
              return(result)
           end
           else do
              result.exponent ← result.exponent - 24
              ox_flag ← 1
           end
        end

        return(result)
```

bfp_ROUND_TO_BFP16_CEIL(x)

x is a normalized floating-point value represented in the working format, having unbounded exponent range and significand precision.

Return the smallest floating-point number having unbounded exponent range but half-precision significand precision that is greater or equal in value to x.

If the magnitude of the value returned is greater than x, inc_flag is set to 1.

If the value returned is not equal to x, xx_flag is set to 1.

bfp_ROUND_TO_BFP16_FLOOR(x)

x is a normalized floating-point value represented in the working format, having unbounded exponent range and significand precision.

Return the largest floating-point number having unbounded exponent range but half-precision significand precision that is lesser or equal in value to x.

If the magnitude of the value returned is greater than x, inc_flag is set to 1.

If the value returned is not equal to x, xx_flag is set to 1.

bfp_ROUND_TO_BFP16_NEAR_EVEN(x)

x is a normalized floating-point value represented in the working format, having unbounded exponent range and significand precision.

Return the floating-point number having unbounded exponent range but half-precision significand precision that is nearest in value to x (in case of a tie, the floating-point number having unbounded exponent range but half-precision significand precision with the least-significant bit equal to 0 is used).

If the magnitude of the value returned is greater than x, inc_flag is set to 1.

If the value returned is not equal to x, xx_flag is set to 1.

bfp_ROUND_TO_BFP16_TRUNC(x)

x is a normalized floating-point value represented in the working format, having unbounded exponent range and significand precision.

Return the largest floating-point number having unbounded exponent range but half-precision significand precision that is lesser or equal in value to x if x>0, or the smallest floating-point number having unbounded exponent range but half0-precision significand precision that is greater or equal in value to x if x<0.

If the magnitude of the value returned is greater than x, inc_flag is set to 1.

If the value returned is not equal to x, xx_flag is set to 1.

bfp_ROUND_TO_INTEGER(rmode, x)

x is a binary floating-point value that is represented in the working floating-point format and has unbounded exponent range and significand precision.

If x is an SNaN, vxsnan_flag is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

If rmode=0b000 (Round to Nearest Even),
return the double-precision floating-point integer value that is nearest in value to x (in case of a tie, the double-precision floating-point integer value with the least-significant bit equal to 0 is used).

If rmode=0b001 (Round towards Zero),
return the largest double-precision floating-point integer value that is lesser or equal in value to x if x>0, or the smallest double-precision floating-point integer value that is greater or equal in value to x if x<0.

If rmode=0b010 (Round towards +Infinity),
return the smallest double-precision floating-point integer value that is greater or equal in value to x.

If rmode=0b011 (Round towards -Infinity),
return the largest double-precision floating-point integer value that is lesser or equal in value to x.

If rmode=0b100 (Round to Nearest Away),
return the double-precision floating-point integer value that is nearest in value to x (in case of a tie, the double-precision floating-point integer value that is furthest away from 0 is used).

inc_flag is set to 1 if the magnitude of the value returned is greater than x.
xx_flag is set to 1 if the value returned is not equal to x.

bfp_ROUND_ODD(p, x)

x is a binary floating-point value that is represented in the working floating-point format and has unbounded exponent range and significand precision. x must be rounded as presented, without prenormalization.

p is an integer value specifying the precision (i.e., number of bits) the significand is rounded to.

Return x with bit p-1 of the significand set to 1 if any of the bits to the right of bit p-1 of the significand of x are equal to 1, and all bits to the right of bit p-1 of the significand of the value returned are set to 0. Otherwise return x with all bits to the right of bit p-1 of the significand set to 0.

inc_flag is set to 1 if the magnitude of the value returned is greater than x.
xx_flag is set to 1 if the value returned is not equal to x.

bfp_ROUND_NEAR_EVEN(p, x)

x is a binary floating-point value that is represented in the working floating-point format and has unbounded exponent range and significand precision. x must be rounded as presented, without prenormalization.

p is an integer value specifying the precision (i.e., number of bits) the significand is rounded to.

Return the floating-point number having unbounded exponent range and a significand with a width of p bits that is nearest in value to x (in case of a tie, the floating-point number having unbounded exponent range and a p-bit significand with the least-significant bit equal to 0 is used).

inc_flag is set to 1 if the magnitude of the value returned is greater than x.
xx_flag is set to 1 if the value returned is not equal to x.

bfp_ROUND_TRUNC(p, x)

x is a binary floating-point value that is represented in the working floating-point format and has unbounded exponent range and significand precision. x must be rounded as presented, without prenormalization.

p is an integer value specifying the precision (i.e., number of bits) the significand is rounded to.

Return the largest floating-point number having unbounded exponent range and a significand with a width of p bits that is lesser or equal in value to x if x>0, or the smallest floating-point number having unbounded exponent range but double-precision significand precision that is greater or equal in value to x if x<0.

inc_flag is set to 1 if the magnitude of the value returned is greater than x.
xx_flag is set to 1 if the value returned is not equal to x.

bfp_ROUND_TO_BFP128(ro, rmode, x)

x is a normalized binary floating-point value that is represented in the working floating-point format and has unbounded exponent range and significand precision.

ro is a 1-bit unsigned integer and rmode is a 2-bit unsigned integer, together specifying one of five rounding modes to be used in rounding z.

```
ro=0 rmode=0b00   Round to Nearest Even
ro=0 rmode=0b01   Round towards Zero
ro=0 rmode=0b10   Round towards +Infinity
ro=0 rmode=0b11   Round towards -Infinity
ro=1              Round to Odd
```

Return the value x rounded to quad-precision under control of the specified rounding mode.

```
if x.class.QNaN     then return x
if x.class.Infinity then return x
if x.class.Zero     then return x
if bfp_ABSOLUTE(x)<bfp_NMIN_BFP128 then do
   if FPSCR.UE=0 then do
      x ← bfp_DENORM(-16382,x)
      if ro=0 & rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(113,x)
      if ro=0 & rmode=0b01 then r ← bfp_ROUND_TRUNC(113,x)
      if ro=0 & rmode=0b10 then r ← bfp_ROUND_CEIL(113,x)
      if ro=0 & rmode=0b11 then r ← bfp_ROUND_FLOOR(113,x)
      if ro=1              then r ← bfp_ROUND_ODD(113,x)
      ux_flag ← xx_flag
      return(r)
   end
   else do
      x.exponent ← x.exponent + 24576
      ux_flag ← 1
   end
end
if ro=0 & rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(113,x)
if ro=0 & rmode=0b01 then r ← bfp_ROUND_TRUNC(113,x)
if ro=0 & rmode=0b10 then r ← bfp_ROUND_CEIL(113,x)
if ro=0 & rmode=0b11 then r ← bfp_ROUND_FLOOR(113,x)
if ro=1              then r ← bfp_ROUND_ODD(113,x)
if bfp_ABSOLUTE(r)>bfp_NMAX_BFP128 then do
   if FPSCR.OE=0 then do
      if ro=0 & rmode=0b00 then r ← x.sign ? bfp_INFINITY    : bfp_INFINITY
      if ro=0 & rmode=0b01 then r ← x.sign ? bfp_NMAX_BFP128 : bfp_NMAX_BFP128
      if ro=0 & rmode=0b10 then r ← x.sign ? bfp_NMAX_BFP128 : bfp_INFINITY
      if ro=0 & rmode=0b11 then r ← x.sign ? bfp_INFINITY    : bfp_NMAX_BFP128
      if ro=1              then r ← x.sign ? bfp_NMAX_BFP128 : bfp_NMAX_BFP128
      r.sign   ← x.sign
      ox_flag  ← 0b1
      xx_flag  ← 0b1
      inc_flag ← 0bU
      return(r)
   end
   else do
      r.exponent ← r.exponent - 24576
      ox_flag ← 1
   end
end
return(r)
```

bfp_ROUND_TO_BFP80(rmode,x)

x is a normalized binary floating-point value that is represented in the working floating-point format and has unbounded exponent range and significand precision.

rmode is a 2-bit unsigned integer, together specifying one of four rounding modes to be used in rounding x.

rmode=0b00	Round to Nearest Even
rmode=0b01	Round towards Zero
rmode=0b10	Round towards +Infinity
rmode=0b11	Round towards -Infinity

Return the value x rounded to double-extended-precision under control of the specified rounding mode.

```
if x.class.QNaN     then return x
if x.class.Infinity then return x
if x.class.Zero     then return x
if bfp_ABSOLUTE(x)<bfp_NMIN_BFP80  then do
   if FPSCR.UE=0 then do
      x ← bfp_DENORM(-16382,x)
      if rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(64,x)
      if rmode=0b01 then r ← bfp_ROUND_TRUNC(64,x)
      if rmode=0b10 then r ← bfp_ROUND_CEIL(64,x)
      if rmode=0b11 then r ← bfp_ROUND_FLOOR(64,x)
      ux_flag ← xx_flag
      return(r)
   end
   else do
      x.exponent ← x.exponent + 24576
      ux_flag ← 1
   end
end
if rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(64,x)
if rmode=0b01 then r ← bfp_ROUND_TRUNC(64,x)
if rmode=0b10 then r ← bfp_ROUND_CEIL(64,x)
if rmode=0b11 then r ← bfp_ROUND_FLOOR(64,x)
if bfp_ABSOLUTE(r)>bfp_NMAX_BFP80 then do
   if FPSCR.OE=0 then do
      if rmode=0b00 then r ← x.sign ? bfp_INFINITY   : bfp_INFINITY
      if rmode=0b01 then r ← x.sign ? bfp_NMAX_BFP80 : bfp_NMAX_BFP80
      if rmode=0b10 then r ← x.sign ? bfp_NMAX_BFP80 : bfp_INFINITY
      if rmode=0b11 then r ← x.sign ? bfp_INFINITY   : bfp_NMAX_BFP80
      r.sign   ← x.sign
      ox_flag  ← 0b1
      xx_flag  ← 0b1
      inc_flag ← 0bU
      return(r)
   end
   else do
      r.exponent ← r.exponent - 24576
      ox_flag ← 1
   end
end
return(r)
```

bfp_ROUND_TO_BFP64(ro, rmode, x)

x is a normalized binary floating-point value that is represented in the working floating-point format and has unbounded exponent range and significand precision.

ro is a 1-bit unsigned integer and rmode is a 2-bit unsigned integer, together specifying one of five rounding modes to be used in rounding z.

```
ro=0 rmode=0b00   Round to Nearest Even
ro=0 rmode=0b01   Round towards Zero
ro=0 rmode=0b10   Round towards +Infinity
ro=0 rmode=0b11   Round towards -Infinity
ro=1              Round to Odd
```

Return the value x rounded to double-precision under control of the specified rounding mode.

```
if x.class.QNaN     then return x
if x.class.Infinity then return x
if x.class.Zero     then return x
if bfp_ABSOLUTE(x)<bfp_NMIN_BFP64  then do
   if FPSCR.UE=0 then do
      x ← bfp_DENORM(-1022,x)
      if ro=0 & rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(53,x)
      if ro=0 & rmode=0b01 then r ← bfp_ROUND_TRUNC(53,x)
      if ro=0 & rmode=0b10 then r ← bfp_ROUND_CEIL(53,x)
      if ro=0 & rmode=0b11 then r ← bfp_ROUND_FLOOR(53,x)
      if ro=1              then r ← bfp_ROUND_ODD(53,x)
      ux_flag ← xx_flag
      return(r)
   end
   else do
      x.exponent ← x.exponent + 1536
      ux_flag ← 1
   end
end
if ro=0 & rmode=0b00 then r ← bfp_ROUND_NEAR_EVEN(53,x)
if ro=0 & rmode=0b01 then r ← bfp_ROUND_TRUNC(53,x)
if ro=0 & rmode=0b10 then r ← bfp_ROUND_CEIL(53,x)
if ro=0 & rmode=0b11 then r ← bfp_ROUND_FLOOR(53,x)
if ro=1              then r ← bfp_ROUND_ODD(53,x)
if bfp_ABSOLUTE(r)>bfp_NMAX_BFP64 then do
   if FPSCR.OE=0 then do
      if ro=0 & rmode=0b00 then r ← x.sign ? bfp_INFINITY   : bfp_INFINITY
      if ro=0 & rmode=0b01 then r ← x.sign ? bfp_NMAX_BFP64 : bfp_NMAX_BFP64
      if ro=0 & rmode=0b10 then r ← x.sign ? bfp_NMAX_BFP64 : bfp_INFINITY
      if ro=0 & rmode=0b11 then r ← x.sign ? bfp_INFINITY   : bfp_NMAX_BFP64
      if ro=1              then r ← x.sign ? bfp_NMAX_BFP64 : bfp_NMAX_BFP64
      r.sign   ← x.sign
      ox_flag  ← 0b1
      xx_flag  ← 0b1
      inc_flag ← 0bU
      return(r)
   end
   else do
      r.exponent ← r.exponent - 1536
      ox_flag ← 1
   end
end
return(r)
```

bfp_SQUARE_ROOT(x)

x is a binary floating-point value that is represented in the working floating-point format and has unbounded exponent range and significand precision.

If x is an SNaN, vxsnan_flag is set to 1.
Otherwise, if x is negative and non-zero, vxsqrt_flag is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is -Zero, return -Zero.
Otherwise, if x is negative, return the standard QNaN.
Otherwise, return the normalized square root of x, having unbounded range and precision.

ClassDP(x,y)

Return a 5-bit characterization of the double-precision floating-point number x.

0b10001 = Quiet NaN
0b01001 = -Infinity
0b01000 = -Normalized Number
0b11000 = -Denormalized Number
0b10010 = -Zero
0b00010 = +Zero
0b10100 = +Denormalized Number
0b00100 = +Normalized Number
0b00101 = +Infinity

ClassSP(x,y)

Return a 5-bit characterization of the single-precision floating-point number x.

0b10001 = Quiet NaN
0b01001 = -Infinity
0b01000 = -Normalized Number
0b11000 = -Denormalized Number
0b10010 = -Zero
0b00010 = +Zero
0b10100 = +Denormalized Number
0b00100 = +Normalized Number
0b00101 = +Infinity

CompareEQDP(x,y)

x and y are double-precision floating-point values.

If x or y is a NaN, return 0.
Otherwise, if x is equal to y, return 1.
Otherwise, return 0.

CompareEQSP(x,y)

x and y are single-precision floating-point values.

If x or y is a NaN, return 0,
Otherwise, if x is equal to y, return 1.
Otherwise, return 0.

CompareGTDP(x,y)

x and y are double-precision floating-point values.

If x or y is a NaN, return 0,
Otherwise, if x is greater than y, return 1.
Otherwise, return 0.

CompareGTSP(x,y)

x and y are single-precision floating-point values.

If x or y is a NaN, return 0.
Otherwise, if x is greater than y, return 1.
Otherwise, return 0.

CompareLTDP(x,y)

x and y are double-precision floating-point values.

If x or y is a NaN, return 0.
Otherwise, if x is less than y, return 1.
Otherwise, return 0.

CompareLTSP(x,y)

x and y are single-precision floating-point values.

If x or y is a NaN, return 0.
Otherwise, if x is less than y, return 1.
Otherwise, return 0.

ConvertDPtoSD(x)

x is a floating-point value in double-precision format.

If x is a NaN,
    vxcvi_flag is set to 1,
    vxsnan_flag is set to 1 if x is an SNaN, and
    return 0x8000_0000_0000_0000,

Otherwise, do the following.
    Let rnd be the value x truncated to an integral value.

    If rnd is greater than $2^{63}-1$,
        vxcvi_flag is set to 1,
        return 0x7FFF_FFFF_FFFF_FFFF.

    Otherwise, if rnd is less than -2^{63},
        vxcvi_flag is set to 1,
        return 0x8000_0000_0000_0000.

    Otherwise,
        xx_flag is set to 1 if rnd is inexact.
        return rnd in 64-bit signed integer format.

ConvertDPtoSP(x)

x is a floating-point value in double-precision format.

If x is an SNaN, vxsnan_flag is set to 1.

If x is a SNaN, returns x, converted to a QNaN, in single-precision floating-point format.

Otherwise, if x is a QNaN, an Infinity, or a Zero, returns x in single-precision floating-point format.

Otherwise, returns x, rounded to single-precision using the rounding mode specified in RN, in single-precision floating-point format.

ox_flag is set to 1 if rounding x resulted in an Overflow exception.
ux_flag is set to 1 if rounding x resulted in an Underflow exception.
xx_flag is set to 1 if rounding x returns an inexact result.
inc_flag is set to 1 if the significand of the result was incremented during rounding.

ConvertDPtoSP_NS(x)

x is a single-precision floating-point value represented in double-precision format.

Returns x in single-precision format.

```
sign     ← x.bit[0]
exponent ← x.bit[1:11]
fraction ← 0b1 » x.bit[12:63]                              // implicit bit set to 1 (for now)

if (exponent == 0) & (fraction.bit[1:52] != 0) then do     // DP Denormal operand
   exponent ← 0b000_0000_0001                              //   exponent override to DP Emin = 1
   fraction.bit[0] ← 0b0                                   //   implicit bit override to 0
end

if (exponent < 897) && (fraction != 0) then do             // SP tiny operand
   fraction ← fraction >>ui (897 - exponent)               //  denormalize until exponent = SP Emin
   exponent ← 0b011_1000_0000                              //  exponent override to SP Emin-1 = 896
end

return(sign » exponent.bit[0] » exponent.bit[4:10] » fraction.bit[1:23])
```

Programming Note

If x is not representable in single-precision, some exponent and/or significand bits will be discarded, likely producing undesirable results. The low-order 29 bits of the significand of x are discarded, more if the unbiased exponent of x is less than -126 (i.e., denormal). Finite values of x having an unbiased exponent less than -150 will return a result of Zero. Finite values of x having an unbiased exponent greater than +127 will result in discarding significant bits of the exponent. SNaN inputs having no significant bits in the upper 23 bits of the signifcand will return Infinity as the result. No status is set for any of these cases.

ConvertDPtoSW(x)

x is a floating-point value in double-precision format.

If x is a NaN,
- vxcvi_flag is set to 1,
- vxsnan_flag is set to 1 if x is an SNaN, and
- return 0x8000_0000,

Otherwise, do the following.

Let rnd be the value x truncated to an integral value.

If rnd is greater than $2^{31}-1$,
- vxcvi_flag is set to 1,
- return 0x7FFF_FFFF.

Otherwise, if rnd is less than -2^{31},
- vxcvi_flag is set to 1,
- return 0x8000_0000.

Otherwise,
- xx_flag is set to 1 if rnd is inexact.
- return rnd in 32-bit signed integer format.

ConvertDPtoUD(x)

x is a floating-point value in double-precision format.

If x is a NaN,
vxcvi_flag is set to 1,
vxsnan_flag is set to 1 if x is an SNaN, and
return 0x8000_0000_0000_0000,

Otherwise, do the following.
Let rnd be the value x truncated to an integral value.

If rnd is greater than $2^{64}-1$,
vxcvi_flag is set to 1,
return 0xFFFF_FFFF_FFFF_FFFF.

Otherwise, if rnd is less than 0,
vxcvi_flag is set to 1,
return 0x0000_0000_0000_0000.

Otherwise,
xx_flag is set to 1 if rnd is inexact.
return rnd in 64-bit unsigned integer format.

ConvertDPtoUW(x)

x is a floating-point value in double-precision format.

If x is a NaN,
vxcvi_flag is set to 1,
vxsnan_flag is set to 1 if x is an SNaN, and
return 0x0000_0000,

Otherwise, do the following.
Let rnd be the value x truncated to an integral value.

If rnd is greater than $2^{32}-1$,
vxcvi_flag is set to 1,
return 0xFFFF_FFFF.

Otherwise, if rnd is less than 0,
vxcvi_flag is set to 1,
return 0x0000_0000.

Otherwise,
xx_flag is set to 1 if rnd is inexact.
return rnd in 32-bit unsigned integer format.

ConvertFPtoDP(x)

Return the floating-point value x in DP format.

ConvertFPtoSP(x)

Return the floating-point value x in single-precision format.

ConvertSDtoFP(x)

x is a 64-bit signed integer value.
Return the value x converted to floating-point format having unbounded significand precision.

ConvertSPtoDP_NS(x)

x is a single-precision floating-point value.

Returns x in double-precision format.

```
sign     ← x.bit[0]
exponent ← (x.bit[1] » ¬x.bit[1] » ¬x.bit[1] » ¬x.bit[1] » x.bit[2:8])
fraction ← 0b0 » x.bit[9:31] » 0b0_0000_0000_0000_0000_0000_0000_0000

if (x.bit[1:8] == 255) then do                          // Infinity or NaN operand
   exponent ← 2047                                      //   override exponent to DP Emax+1
end

else if (x.bit[1:8] == 0) && (fraction == 0) then do    // SP Zero operand
   exponent ← 0                                         //   override exponent to DP Emin-1
end

else if (x.bit[1:8] == 0) && (fraction != 0) then do    // SP Denormal operand
   exponent ← 897                                       //   override exponent to SP Emin
   do while (fraction.bit[0] == 0)                      //   normalize operand
      fraction ← fraction << 1
      exponent ← exponent - 1
   end
end

return(sign » exponent » fraction.bit[1:52])
```

ConvertSP64toSP(x)

x is a single-precision floating-point value in double-precision format.

Returns the value x in single-precision format. x must be representable in single-precision, or else result returned is undefined. x may require denormalization. No rounding is performed. If x is a SNaN, it is converted to a single-precision SNaN having the same payload as x.

```
sign ← x.bit[0]
exp  ← x.bit[1:11] - 1023
frac ← x.bit[12:63]

if      (exp = -1023) & (frac  = 0) & (sign=0) then return(0x0000_0000) // +Zero
else if (exp = -1023) & (frac  = 0) & (sign=1) then return(0x8000_0000) // -Zero
else if (exp = -1023) & (frac != 0)            then return(0xUUUU_UUUU) // DP denorm
else if (exp < -126) then do // denormalization required
   msb = 1
   do while (exp < -126)     // denormalize operand until exp=Emin
      frac.bit[1:51] ← frac.bit[0:50]
      frac.bit[0]    ← msb
      msb            ← 0
      exp            ← exp + 1
   end
   if (frac = 0) then return(0xUUUU_UUUU)  // value not representable in SP format
   else do // return denormal SP
      result.bit[0]    ← sign
      result.bit[1:8]  ← 0
      result.bit[9:31] ← frac.bit[0:22]
      return(result)
   end
end
else if (exp = +1024) & (frac  = 0) & (sign=0) then return(0x7F80_0000) // +Infinity
else if (exp = +1024) & (frac  = 0) & (sign=1) then return(0xFF80_0000) // -Infinity
else if (exp = +1024) & (frac != 0) then do   // QNaN or SNaN
   result.bit[0]    ← sign
   result.bit[1:8]  ← 255
   result.bit[9:31] ← frac.bit[0:22]
   return(result)
end
else if (exp < +1024) & (exp > +126) then return(0xUUUU_UUUU)  // overflow
else do  // normal value
   result.bit[0]    ← sign
   result.bit[1:8]  ← exp.bit[4:11] + 127
   result.bit[9:31] ← frac.bit[0:22]
   return(result)
end
```

ConvertSPtoDP(x)

x is a single-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is an SNaN, return x represented as a QNaN in double-precision floating-point format.
Otherwise, if x is an QNaN, return x in double-precision floating-point format.
Otherwise, return the value x in double-precision floating-point format.

ConvertSPtoSD(x)

x is a floating-point value in single-precision format.

If x is a NaN,
vxcvi_flag is set to 1, and
vxsnan_flag is set to 1 if x is an SNaN
return 0x8000_0000_0000_0000 and

Otherwise, do the following.
Let rnd be the value x truncated to an integral value.

If rnd is greater than $2^{63}-1$,
vxcvi_flag is set to 1, and
return 0x7FFF_FFFF_FFFF_FFFF.

Otherwise, if rnd is less than -2^{63},
vxcvi_flag is set to 1, and
return 0x8000_0000_0000_0000.

Otherwise,
xx_flag is set to 1 if rnd is inexact, and
return rnd in 64-bit signed integer format.

ConvertSPtoSP64(x)

x is a floating-point value in single-precision format.

Returns the value x in double-precision format. If x is a SNaN, it is converted to a double-precision SNaN having the same payload as x.

```
sign ← x.bit[0]
exp  ← x.bit[1:8] - 127
frac ← x.bit[9:31]

if (exp = –127) & (frac != 0) then do // Normalize the Denormal value
   msb  ← frac.bit[0]
   frac ← frac << 1
   do while (msb = 0)
      msb  ← frac.bit[0]
      frac ← frac << 1
      exp  ← exp – 1
   end
end
else if (exp = -127) & (frac = 0) then exp ← –1023  // Zero value
else if (exp = +128)              then exp ← +1024  // Infinity, NaN

result.bit[0]     ← sign
result.bit[1:11]  ← exp + 1023
result.bit[12:34] ← frac
result.bit[35:63] ← 0
return(result)
```

ConvertSPtoSW(x)

x is a floating-point value in single-precision format.

If x is a NaN,
vxcvi_flag is set to 1,
vxsnan_flag is set to 1 if x is an SNaN, and
return 0x8000_0000.

Otherwise, do the following.
Let rnd be the value x truncated to an integral value.

If rnd is greater than $2^{31}-1$,
vxcvi_flag is set to 1, and
return 0x7FFF_FFFF.

Otherwise, if rnd is less than -2^{31},
vxcvi_flag is set to 1, and
return 0x8000_0000.

Otherwise,
xx_flag is set to 1 if rnd is inexact, and
return rnd in 32-bit signed integer format.

ConvertSPtoUD(x)

x is a floating-point value in single-precision format.

If x is a NaN,
vxcvi_flag is set to 1, and
vxsnan_flag is set to 1 if x is an SNaN
return 0x0000_0000_0000_0000,

Otherwise, do the following.
Let rnd be the value x truncated to an integral value.

If rnd is greater than $2^{64}-1$,
vxcvi_flag is set to 1, and
return 0xFFFF_FFFF_FFFF_FFFF.

Otherwise, if rnd is less than 0,
vxcvi_flag is set to 1, and
return 0x0000_0000_0000_0000.

Otherwise,
xx_flag is set to 1 if rnd is inexact, and
return rnd in 64-bit unsigned integer format.

ConvertSPtoUW(x)

x is a floating-point value in single-precision format.

If x is a NaN,
vxcvi_flag is set to 1,
vxsnan_flag is set to 1 if x is an SNaN, and
return 0x0000_0000.

Otherwise, do the following.
Let rnd be the value x truncated to an integral value.

If rnd is greater than $2^{32}-1$,
vxcvi_flag is set to 1, and
return 0xFFFF_FFFF.

Otherwise, if rnd is less than 0,
vxcvi_flag is set to 1, and
return 0x0000_0000.

Otherwise,
xx_flag is set to 1 if rnd is inexact, and
return rnd in 32-bit unsigned integer format.

ConvertSWtoFP(x)

x is a 32-bit signed integer value.

Return the value x converted to floating-point format having unbounded significand precision.

ConvertUDtoFP(x)

x is a 64-bit unsigned integer value.

Return the value x converted to floating-point format having unbounded significand precision.

ConvertUWtoFP(x)

x is a 32-bit unsigned integer value.

Return the value x converted to floating-point format having unbounded significand precision.

DivideDP(x,y)

x and y are double-precision floating-point values.

If x or y is an SNaN, vxsnan_flag is set to 1.

If x is a Zero and y is a Zero, vxzdz_flag is set to 1.

If x is a finite, nonzero value and y is a Zero, zx_flag is set to 1.

If x is an Infinity and y is an Infinity, vxidi_flag is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x is a Zero and y is a Zero, return the standard QNaN.
Otherwise, if x is a finite, nonzero value and y is a Zero with the same sign as x, return +Infinity.
Otherwise, if x is a finite, nonzero value and y is a Zero with the opposite sign as x, return -Infinity.
Otherwise, if x is an Infinity and y is an Infinity, return the standard QNaN.
Otherwise, return the normalized quotient of x divided by y, having unbounded range and precision.

DivideSP(x,y)

x and y are single-precision floating-point values.

If x or y is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero and y is a Zero, `vxzdz_flag` is set to 1.

If x is a finite, nonzero value and y is a Zero, `zx_flag` is set to 1.

If x is an Infinity and y is an Infinity, `vxidi_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x is a Zero and y is a Zero, return the standard QNaN.
Otherwise, if x is a finite, nonzero value and y is a Zero with the same sign as x, return +Infinity.
Otherwise, if x is a finite, nonzero value and y is a Zero with the opposite sign as x, return -Infinity.
Otherwise, if x is an Infinity and y is an Infinity, return the standard QNaN.
Otherwise, return the normalized quotient of x divided by y, having unbounded range and precision.

DenormDP(x)

x is a floating-point value having unbounded range and precision.

Return the value x with its significand shifted right by a number of bits equal to the difference of the -1022 and the unbiased exponent of x, and its unbiased exponent set to -1022.

DenormSP(x)

x is a floating-point value having unbounded range and precision.

Return the value x with its significand shifted right by a number of bits equal to the difference of the -126 and the unbiased exponent of x, and its unbiased exponent set to -126.

EXTZ32(x)

Result of extending the `b`-bit value `x` on the left with `32-b` zeros, forming a 32-bit value.

```
b ← LENGTH(x)
result.bit[0:31-b]  ← 0
result.bit[32-b:31] ← x
```

EXTZ64(x)

Result of extending the `b`-bit value `x` on the left with `64-b` zeros, forming a 64-bit value.

```
b ← LENGTH(x)
result.bit[0:63-b]  ← 0
result.bit[64-b:63] ← x
```

EXTZ128(x)

Result of extending the `b`-bit value `x` on the left with `128-b` zeros, forming a 128-bit value.

```
b ← LENGTH(x)
result.bit[0:127-b]   ← 0
result.bit[128-b:127] ← x
```

fprf_CLASS_BFP16(x)

x is a floating-point value represented in half-precision format.

Return the 5-bit code that specifies the sign and class of x.

Return 0b10001 if x is a Quiet NaN.
Return 0b01001 if x is a negative infinity.
Return 0b00101 if x is a positive infinity.
Return 0b10010 if x is a negative zero.
Return 0b00010 if x is a positive zero.
Return 0b11000 if x is a negative denormal value when represented in half-precision format.
Return 0b10100 if x is a positive denormal value when represented in half-precision format.
Return 0b01000 if x is a negative normal value when represented in half-precision format.
Return 0b00100 if x is a positive normal value when represented in half-precision format.

fprf_CLASS_BFP64(x)

x is a floating-point value represented in double-precision format.

Return the 5-bit code that specifies the sign and class of x.

Return 0b10001 if x is a Quiet NaN.
Return 0b01001 if x is a negative infinity.
Return 0b00101 if x is a positive infinity.
Return 0b10010 if x is a negative zero.
Return 0b00010 if x is a positive zero.
Return 0b11000 if x is a negative denormal value when represented in double-precision format.
Return 0b10100 if x is a positive denormal value when represented in double-precision format.
Return 0b01000 if x is a negative normal value when represented in double-precision format.
Return 0b00100 if x is a positive normal value when represented in double-precision format.

fprf_CLASS_BFP128(x)

x is binary floating-point value that is represented in quad-precision format.

Return the 5-bit characterization of the sign and class of x.

Return 0b10001 if x is a Quiet NaN.
Return 0b01001 if x is negative and an infinity.
Return 0b01000 if x is negative and a normal number.
Return 0b11000 if x is negative and a denormal number.
Return 0b10010 if x is negative and a zero.
Return 0b00010 if x is positive and a zero.
Return 0b10100 if x is positive and a denormal number.
Return 0b00100 if x is positive and a normal number.
Return 0b00101 if x is positive and an infinity.

IsInf(x)

Return 1 if x is an Infinity, otherwise return 0.

IsNaN(x)

Return 1 if x is either an SNaN or a QNaN, otherwise return 0.

IsNeg(x)

Return 1 if x is a negative, nonzero value, otherwise return 0.

IsSNaN(x)

Return 1 if x is an SNaN, otherwise return 0.

IsZero(x)

Return 1 if x is a Zero, otherwise return 0.

MaximumDP(x,y)

x and y are double-precision floating-point values.

If x or y is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN and y is not a NaN, return y.
Otherwise, if x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return x.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, return the greater of x and y, where +0 is considered greater than –0.

MaximumSP(x,y)

x and y are single-precision floating-point values.

If x or y is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN and y is not a NaN, return y.
Otherwise, if x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return x.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, return the greater of x and y, where +0 is considered greater than –0.

MinimumDP(x,y)

x and y are double-precision floating-point values.

If x or y is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN and y is not a NaN, return y.
Otherwise, if x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return x.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, return the lesser of x and y, where –0 is considered less than +0.

MinimumSP(x,y)

x and y are single-precision floating-point values.

If x or y is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN and y is not a NaN, return y.
Otherwise, if x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return x.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, return the lesser of x and y, where –0 is considered less than +0.

MultiplyAddDP(x,y,z)

x, y and z are double-precision floating-point values.

If x, y or z is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero and y, is an Infinity or x is an Infinity and y is an Zero, `vximz_flag` is set to 1.

If the product of x and y is an Infinity and z is an Infinity of the opposite sign, `vxisi_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if z is a QNaN, return z.
Otherwise, if z is an SNaN, return z represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x is a Zero and y is an Infinity or x is an Infinity and y is an Zero, return the standard QNaN.
Otherwise, if the product of x and y is an Infinity, and z is an Infinity of the opposite sign, return the standard QNaN.
Otherwise, return the normalized sum of z and the product of x and y, having unbounded range and precision.

MultiplyAddSP(x,y,z)

x, y and z are single-precision floating-point values.

If x, y or z is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero and y is an Infinity, or x is an Infinity and y is an Zero, `vximz_flag` is set to 1.

If the product of x and y is an Infinity and z is an Infinity of the opposite sign, `vxisi_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if z is a QNaN, return z.
Otherwise, if z is an SNaN, return z represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x is a Zero and y is an Infinity or x is an Infinity and y is an Zero, return the standard QNaN.
Otherwise, if the product of x and y is an Infinity, and z is an Infinity of the opposite sign, return the standard QNaN.
Otherwise, return the normalized sum of z and the product of x and y, having unbounded range and precision.

MultiplyDP(x,y)

x and y are double-precision floating-point values.

If x or y is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero and y is an Infinity, or x is an Infinity and y is an Zero, `vximz_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x is a Zero and y is as Infinity or x is a Infinity and y is an Zero, return the standard QNaN.
Otherwise, return the normalized product of x and y, having unbounded range and precision.

MultiplySP(x,y)

x and y are single-precision floating-point values.

If x or y is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero and y is an Infinity, or x is an Infinity and y is an Zero, `vximz_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if y is a QNaN, return y.
Otherwise, if y is an SNaN, return y represented as a QNaN.
Otherwise, if x is a Zero and y is as Infinity or x is a Infinity and y is an Zero, return the standard QNaN.
Otherwise, return the normalized product of x and y, having unbounded range and precision.

NegateDP(x)

If the double-precision floating-point value x is a NaN, return x.
Otherwise, return the double-precision floating-point value x with its sign bit complemented.

NegateSP(x)

If the single-precision floating-point value x is a NaN, return x.
Otherwise, return the single-precision floating-point value x with its sign bit complemented.

ReciprocalEstimateDP(x)

x is a double-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero, `zx_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is a Zero, return an Infinity with the sign of x.
Otherwise, if x is an Infinity, return a Zero with the sign of x.
Otherwise, return an estimate of the reciprocal of x having unbounded exponent range.

ReciprocalEstimateSP(x)

x is a single-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero, `zx_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is a Zero, return an Infinity with the sign of x.
Otherwise, if x is an Infinity, return a Zero with the sign of x.
Otherwise, return an estimate of the reciprocal of x having unbounded exponent range.

ReciprocalSquareRootEstimateDP(x)

x is a double-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero, `zx_flag` is set to 1.

If x is a negative, nonzero number, `vxsqrt_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is a negative, nonzero value, return the default QNaN.
Otherwise, return an estimate of the reciprocal of the square root of x having unbounded exponent range.

ReciprocalSquareRootEstimateSP(x)

x is a single-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a Zero, `zx_flag` is set to 1.

If x is a negative, nonzero number, `vxsqrt_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is a negative, nonzero value, return the default QNaN.
Otherwise, return an estimate of the reciprocal of the square root of x having unbounded exponent range.

reset_xflags()

`vxsnan_flag` is set to 0.
`vximz_flag` is set to 0.
`vxidi_flag` is set to 0.
`vxisi_flag` is set to 0.
`vxzdz_flag` is set to 0.
`vxsqrt_flag` is set to 0.
`vxcvi_flag` is set to 0.
`vxvc_flag` is set to 0.
`ox_flag` is set to 0.
`ux_flag` is set to 0.
`xx_flag` is set to 0.
`zx_flag` is set to 0.

RoundToDP(x,y)

x is a 2-bit unsigned integer specifying one of four rounding modes.

0b00	Round to Nearest Even
0b01	Round towards Zero
0b10	Round towards +Infinity
0b11	Round towards - Infinity

y is a normalized floating-point value having unbounded range and precision.

Return the value y rounded to double-precision under control of the rounding mode specified by x.

```
if IsQNaN(y) then return ConvertFPtoDP(y)
if IsInf(y)  then return ConvertFPtoDP(y)
if IsZero(y) then return ConvertFPtoDP(y)
if y<Nmin then do
   if UE=0 then do
      if x=0b00 then r ← RoundToDPNearEven( DenormDP(y) )
      if x=0b01 then r ← RoundToDPTrunc( DenormDP(y) )
      if x=0b10 then r ← RoundToDPCeil( DenormDP(y) )
      if x=0b11 then r ← RoundToDPFloor( DenormDP(y) )
      ux_flag ← xx_flag
      return(ConvertFPtoDP(r))
   end
   else do
      y ← Scalb(y,+1536)
      ux_flag ← 1
   end
end
if x=0b00 then r ← RoundToDPNearEven(y)
if x=0b01 then r ← RoundToDPTrunc(y)
if x=0b10 then r ← RoundToDPCeil(y)
if x=0b11 then r ← RoundToDPFloor(y))
if r>Nmax then do
   if OE=0 then do
      if x=0b00 then r ← sign ? -Inf : +Inf
      if x=0b01 then r ← sign ? -Nmax : +Nmax
      if x=0b10 then r ← sign ? -Nmax : +Inf
      if x=0b11 then r ← sign ? -Inf : +Nmax
      ox_flag  ← 0b1
      xx_flag  ← 0b1
      inc_flag ← 0bU
      return(ConvertFPtoDP(r))
   end
   else do
      r ← Scalb(r,-1536)
      ox_flag ← 1
   end
end
return(ConvertFPtoDP(r))
```

RoundToDPCeil(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the smallest floating-point number having unbounded exponent range but double-precision significand precision that is greater or equal in value to x.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToDPFloor(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the largest floating-point number having unbounded exponent range but double-precision significand precision that is lesser or equal in value to x.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToDPIntegerCeil(x)

x is a double-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the smallest double-precision floating-point integer value that is greater or equal in value to x.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToDPIntegerFloor(x)

x is a double-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the largest double-precision floating-point integer value that is lesser or equal in value to x

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToDPIntegerNearAway(x)

x is a double-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the largest double-precision floating-point integer value that is lesser or equal in value to x+0.5 if x>0, or the smallest double-precision floating-point integer that is greater or equal in value to x-0.5 if x<0.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToDPIntegerNearEven(x)

x is a double-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the double-precision floating-point integer value that is nearest in value to x (in case of a tie, the double-precision floating-point integer value with the least-significant bit equal to 0 is used).

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToDPIntegerTrunc(x)

x is a double-precision floating-point value.

If x is an SNaN, vxsnan_flag is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the largest double-precision floating-point integer value that is lesser or equal in value to x if x>0, or the smallest double-precision floating-point integer value that is greater or equal in value to x if x<0.

If the magnitude of the value returned is greater than x, inc_flag is set to 1.

If the value returned is not equal to x, xx_flag is set to 1.

RoundToDPNearEven(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the floating-point number having unbounded exponent range but double-precision significand precision that is nearest in value to x (in case of a tie, the floating-point number having unbounded exponent range but double-precision significand precision with the least-significant bit equal to 0 is used).

If the magnitude of the value returned is greater than x, inc_flag is set to 1.

If the value returned is not equal to x, xx_flag is set to 1.

RoundToDPTrunc(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the largest floating-point number having unbounded exponent range but double-precision significand precision that is lesser or equal in value to x if x>0, or the smallest floating-point number having unbounded exponent range but double-precision significand precision that is greater or equal in value to x if x<0.

If the magnitude of the value returned is greater than x, inc_flag is set to 1.

If the value returned is not equal to x, xx_flag is set to 1.

RoundToSP(x,y)

x is a 2-bit unsigned integer specifying one of four rounding modes.

0b00	Round to Nearest Even
0b01	Round towards Zero
0b10	Round towards +Infinity
0b11	Round towards - Infinity

y is a normalized floating-point value having unbounded range and precision.

Return the value y rounded to single-precision under control of the rounding mode specified by x.

```
if IsQNaN(y) then return ConvertFPtoSP(y)
if IsInf(y)  then return ConvertFPtoSP(y)
if IsZero(y) then return ConvertFPtoSP(y)
if y<Nmin then do
   if UE=0 then do
      if x=0b00 then r ← RoundToSPNearEven( DenormSP(y) )
      if x=0b01 then r ← RoundToSPTrunc( DenormSP(y) )
      if x=0b10 then r ← RoundToSPCeil( DenormSP(y) )
      if x=0b11 then r ← RoundToSPFloor( DenormSP(y) )
      ux_flag ← xx_flag
      return(ConvertFPtoSP(r))
   end
   else do
      y ← Scalb(y,+192)
      ux_flag ← 1
   end
end
if x=0b00 then r ← RoundToSPNearEven(y)
if x=0b01 then r ← RoundToSPTrunc(y)
if x=0b10 then r ← RoundToSPCeil(y)
if x=0b11 then r ← RoundToSPFloor(y))
if r>Nmax then do
   if OE=0 then do
      if x=0b00 then r ← sign ? -Inf : +Inf
      if x=0b01 then r ← sign ? -Nmax : +Nmax
      if x=0b10 then r ← sign ? -Nmax : +Inf
      if x=0b11 then r ← sign ? -Inf : +Nmax
      ox_flag  ← 0b1
      xx_flag  ← 0b1
      inc_flag ← 0bU
      return(ConvertFPtoSP(r))
   end
   else do
      r ← Scalb(r,-192)
      ox_flag ← 1
   end
end
return(ConvertFPtoSP(r))
```

RoundToSPCeil(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the smallest floating-point number having unbounded exponent range but single-precision significand precision that is greater or equal in value to x.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSPFloor(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the largest floating-point number having unbounded exponent range but single-precision significand precision that is lesser or equal in value to x.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSPIntegerCeil(x)

x is a single-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the smallest single-precision floating-point integer value that is greater or equal in value to x.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSPIntegerFloor(x)

x is a single-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the largest single-precision floating-point integer value that is lesser or equal in value to x.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSPIntegerNearAway(x)

x is a single-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return x if x is a floating-point integer; otherwise return the largest single-precision floating-point integer value that is lesser or equal in value to x+0.5 if x>0, or the smallest single-precision floating-point integer value that is greater or equal in value to x-0.5 if x<0.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSPIntegerNearEven(x)

x is a single-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return x if x is a floating-point integer; otherwise return the single-precision floating-point integer value that is nearest in value to x (in case of a tie, the single-precision floating-point integer value with the least-significant bit equal to 0 is used).

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSPIntegerTrunc(x)

x is a single-precision floating-point value.

If x is a QNaN, return x.

Otherwise, if x is an SNaN, return x represented as a QNaN, and `vxsnan_flag` is set to 1.

Otherwise, if x is an infinity, return x.

Otherwise, do the following.

Return the largest single-precision floating-point integer value that is lesser or equal in value to x if x>0, or the smallest single-precision floating-point integer value that is greater or equal in value to x if x<0.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSPNearEven(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the floating-point number having unbounded exponent range but single-precision significand precision that is nearest in value to x (in case of a tie, the floating-point number having unbounded exponent range but single-precision significand precision with the least-significant bit equal to 0 is used).

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

RoundToSPTrunc(x)

x is a floating-point value having unbounded range and precision.

If x is a QNaN, return x.

Otherwise, if x is an Infinity, return x.

Otherwise, do the following.

Return the largest floating-point number having unbounded exponent range but single-precision significand precision that is lesser or equal in value to x if x>0, or the smallest single-precision floating-point number that is greater or equal in value to x if x<0.

If the magnitude of the value returned is greater than x, `inc_flag` is set to 1.

If the value returned is not equal to x, `xx_flag` is set to 1.

Scalb(x,y)

x is a floating-point value having unbounded range and precision.

y is a signed integer.

Result of multiplying the floating-point value x by 2^y.

SetFX(x)

x is one of the exception flags in the FPSCR.

If the contents of x is 0, FX and x are set to 1.

SquareRootDP(x)

x is a double-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

If x is a negative, nonzero value, `vxsqrt_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is a negative, nonzero value, return the default QNaN.
Otherwise, return the normalized square root of x, having unbounded range and precision.

SquareRootSP(x)

x is a single-precision floating-point value.

If x is an SNaN, `vxsnan_flag` is set to 1.

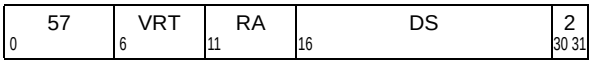
If x is a negative, nonzero value, `vxsqrt_flag` is set to 1.

If x is a QNaN, return x.
Otherwise, if x is an SNaN, return x represented as a QNaN.
Otherwise, if x is a negative, nonzero value, return the default QNaN.
Otherwise, return the normalized square root of x, having unbounded range and precision.

7.6.3 VSX Instruction Descriptions

Load VSX Scalar Doubleword DS-form

lxsd VRT,DS(RA)




```
if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + EXTS(DS)<<2

VSR[VRT+32].dword[0] ← MEM(EA,8)
VSR[VRT+32].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value VRT + 32.

Let EA be the sum of the contents of GPR[RA], or 0 if RA=0, and the signed integer value DS<<2.

When Big-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte 0 of load_data,

– the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data, and so forth until

– the contents of the byte in storage at address EA+7 are placed into byte 7 of load_data.

When Little-Endian byte ordering is employed, let load_data be the contents of the doubleword in storage at address EA such that;

– the contents of the byte in storage at address EA are placed into byte 7 of load_data,

– the contents of the byte in storage at address EA+1 are placed into byte 6 of load_data, and so forth until

– the contents of the byte in storage at address EA+7 are placed into byte 0 of load_data.

load_data is placed into doubleword element 0 of VSR[XT].

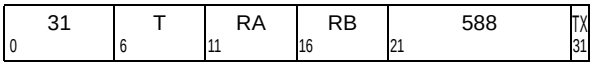
The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered:

None

Load VSX Scalar Doubleword Indexed X-form

lxsdx XT,RA,RB




```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[32×TX+T].dword[0] ← MEM(EA,8)
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×TX + T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte 0 of load_data,

– the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data, and so forth until

– the contents of the byte in storage at address EA+7 are placed into byte 7 of load_data.

When Little-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte 7 of load_data,

– the contents of the byte in storage at address EA+1 are placed into byte 6 of load_data, and so forth until

– the contents of the byte in storage at address EA+7 are placed into byte 0 of load_data.

load_data is placed into doubleword element 0 of VSR[XT].

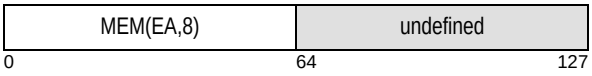
The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

None

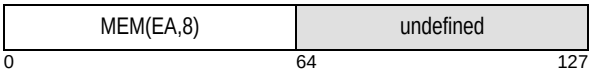
VSR Data Layout for lxsd

```
tgt = VSR[XT]
```



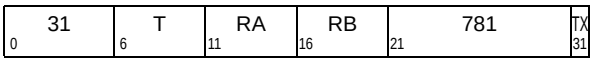
VSR Data Layout for lxsdx

```
tgt = VSR[XT]
```



Load VSX Scalar as Integer Byte & Zero Indexed X-form

lxsibzx XT,RA,RB




```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[32×TX+T].dword[0] ← EXTZ64(MEM(EA,1))
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×TX + T.

Let the effective address (EA) be sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

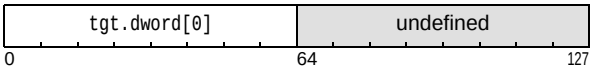
The unsigned integer in the byte in storage addressed by EA is placed in doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered:

None

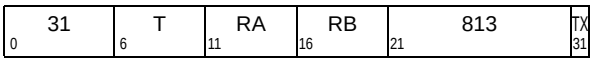
VSR Data Layout for lxsibzx

tgt = VSR[XT]



Load VSX Scalar as Integer Halfword & Zero Indexed X-form

lxsihzx XT,RA,RB




```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[32×TX+T].dword[0] ← EXTZ64(MEM(EA,2))
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×TX + T.

Let the effective address (EA) be sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

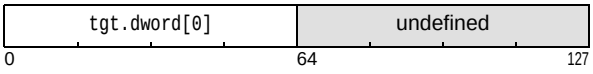
The unsigned integer in the halfword in storage addressed by EA is placed in doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered:

None

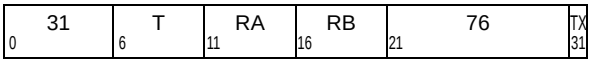
VSR Data Layout for lxsihzx

tgt = VSR[XT]



Load VSX Scalar as Integer Word Algebraic Indexed X-form

lxsiwax XT,RA,RB




```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[32×TX+T].dword[0] ← EXTS64(MEM(EA,4))
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×TX + T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data,
- the contents of the byte in storage at address EA+2 are placed into byte 2 of load_data, and
- the contents of the byte in storage at address EA+3 are placed into byte 3 of load_data.

When Little-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte 3 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte 2 of load_data,
- the contents of the byte in storage at address EA+2 are placed into byte 1 of load_data, and
- the contents of the byte in storage at address EA+3 are placed into byte 0 of load_data.

load_data is sign-extended to a doubleword and placed in doubleword element 0 of VSR[XT].

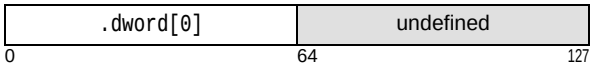
The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

None

VSR Data Layout for lxsiwax

tgt = VSR[XT]



Load VSX Scalar as Integer Word and Zero Indexed X-form

lxsiwzx XT,RA,RB

31	T	RA	RB	12	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[32×TX+T].dword[0] ← ExtendZero(MEM(EA,4))
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×TX + T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte 0 of load_data,

– the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data,

– the contents of the byte in storage at address EA+2 are placed into byte 2 of load_data, and

– the contents of the byte in storage at address EA+3 are placed into byte 3 of load_data.

When Little-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte 3 of load_data,

– the contents of the byte in storage at address EA+1 are placed into byte 2 of load_data,

– the contents of the byte in storage at address EA+2 are placed into byte 1 of load_data, and

– the contents of the byte in storage at address EA+3 are placed into byte 0 of load_data.

load_data is zero-extended to a doubleword and placed in doubleword element 0 of VSR[XT].

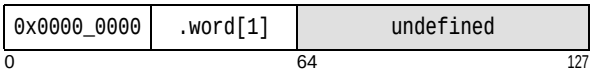
The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

None

VSR Data Layout for lxsiwzx

tgt = VSR[XT]



Load VSX Scalar Single DS-form

lxssp VRT,DS(RA)

57	VRT	RA	DS	3
0	6	11	16	30 31

```
if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + EXTS(DS||0b00)

VSR[VRT+32].dword[0] ← ConvertSPtoSP64(MEM(EA,4))
VSR[VRT+32].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value VRT + 32.

Let EA be the sum of the contents of GPR[RA], or 0 if RA=0, and the signed integer value DS||0b00.

When Big-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte 0 of load_data,

– the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data,

– the contents of the byte in storage at address EA+2 are placed into byte 2 of load_data, and

– the contents of the byte in storage at address EA+3 are placed into byte 3 of load_data.

When Little-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte 3 of load_data,

– the contents of the byte in storage at address EA+1 are placed into byte 2 of load_data,

– the contents of the byte in storage at address EA+2 are placed into byte 1 of load_data, and

– the contents of the byte in storage at address EA+3 are placed into byte 0 of load_data.

load_data, interpreted as a single-precision floating-point value, is placed into doubleword element 0 of VSR[VRT+32] in double-precision format.

The contents of doubleword element 1 of VSR[VRT+32] are undefined.

Special Registers Altered:

None

Load VSX Scalar Single-Precision Indexed X-form

lxsspx XT,RA,RB

31	T	RA	RB	524	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[VRT+32].dword[0] ← ConvertSPtoSP64(MEM(EA,4))
VSR[VRT+32].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×TX + T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte 0 of load_data,

– the contents of the byte in storage at address EA+1 are placed into byte 1 of load_data,

– the contents of the byte in storage at address EA+2 are placed into byte 2 of load_data, and

– the contents of the byte in storage at address EA+3 are placed into byte 3 of load_data.

When Little-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte 3 of load_data,

– the contents of the byte in storage at address EA+1 are placed into byte 2 of load_data,

– the contents of the byte in storage at address EA+2 are placed into byte 1 of load_data, and

– the contents of the byte in storage at address EA+3 are placed into byte 0 of load_data.

load_data, interpreted as a single-precision floating-point value, is placed in doubleword element 0 of VSR[XT] in double-precision format.

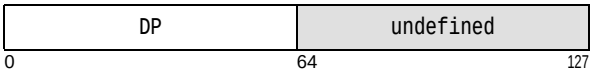
The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

None

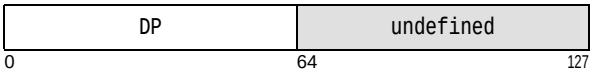
VSR Data Layout for lxssp

tgt = VSR[XT]



VSR Data Layout for lxsspx

tgt = VSR[XT]



*Load VSX Vector Byte*16 Indexed X-form*

lxvb16x XT,RA,RB

31	T	RA	RB	876	TX
0	6	11	16	21	31

```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

do i = 0 to 15
   VSR[32×TX+T].byte[i] ← MEM(EA+i, 1)
end
```

Let XT be the value 32×TX + T.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value from 0 to 15, do the following.
The contents of the byte in storage at address EA+i are placed into byte element i of VSR[XT],

Special Registers Altered:
None

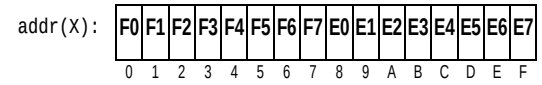
Programming Note

lxvd2x, ***lxvw4x***, ***lxvh8x***, ***lxvb16x***, and ***lxvx*** exhibit identical behavior in Big-Endian mode.

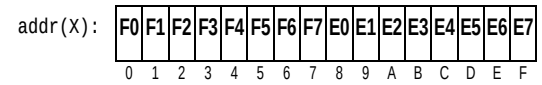
Example: Loading data using *Load VSX Vector Byte*16 Indexed*

```
char      X[]   = { 0xF0, 0xF1, 0xF2, 0xF3,
                    0xF4, 0xF5, 0xF6, 0xF7,
                    0xE0, 0xE1, 0xE2, 0xE3,
                    0xE4, 0xE5, 0xE6, 0xE7 };
```

Big-endian storage image of X

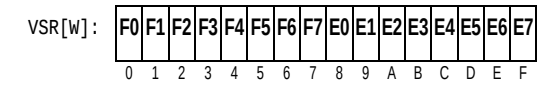


Little-endian storage image of X



Loading a vector of 16 byte elements from Big-Endian storage in VSR[XT] using ***lxvb16x***, retaining left-to-right element ordering.

```
# Assumptions
#   GPR[PX] = address of X

lxvb16x   xX,r0,rPX
```

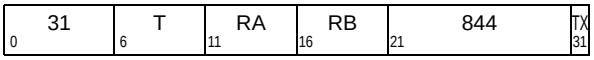


Loading a vector of 16 byte elements from Little-Endian storage in VSR[XT] using ***lxvb16x***, retaining left-to-right element ordering.

```
# Assumptions
#   GPR[PX] = address of X

lxvb16x   xX,r0,rPX
```

*Load VSX Vector Doubleword*2 Indexed X-form*

lxvd2x XT,RA,RB




```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[32×TX+T].dword[0] ← MEM(EA, 8)
VSR[32×TX+T].dword[1] ← MEM(EA+8, 8)
```

Let XT be the value 32×TX + T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value i from 0 to 1, do the following.

When Big-Endian byte ordering is employed, the contents of the doubleword in storage at address EA+8×i are placed into load_data in such an order that;

- the contents of the byte in storage at address EA+8×i are placed into byte element 0 of load_data,
- the contents of the byte in storage at address EA+8×i+1 are placed into byte element 1 of load_data, and so forth until
- the contents of the byte in storage at address EA+8×i+7 are placed into byte element 7 of load_data.

When Little-Endian byte ordering is employed, the contents of the doubleword in storage at address EA+8×i are placed into load_data in such an order that;

- the contents of the byte in storage at address EA+8×i are placed into byte element 7 of load_data,
- the contents of the byte in storage at address EA+8×i+1 are placed into byte element 6 of load_data, and so forth until
- the contents of the byte in storage at address EA+8×i+7 are placed into byte element 0 of load_data.

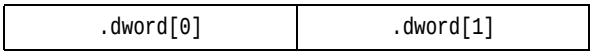
load_data is placed into doubleword element i of VSR[XT].

Special Registers Altered

None

VSR Data Layout for lxvd2x

tgt = VSR[XT]

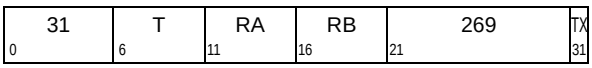


Programming Note

lxvd2x, ***lxvw4x***, ***lxvh8x***, ***lxvb16x***, and ***lxvx*** exhibit identical behavior in Big-Endian mode.

Load VSX Vector with Length X-form

lxvl XT,RA,RB




```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← (RA=0) ? 0 : GPR[RA]
nb ← EXTZ(GPR[RB].bit[0:7])
if nb>16 then nb ← 16

load_data ← 0x0000_0000_0000_0000_0000_0000_0000_0000

if MSR.LE = 0 then  // Big-Endian byte-ordering
   load_data.byte[0:nb-1]   ← MEM(EA,nb)
else               // Little-Endian byte-ordering
   load_data.byte[16-nb:15] ← MEM(EA,nb)

VSR[32×TX+T] ← load_data
```

Let XT be the value 32×TX + T.

Let the effective address (EA) be the contents of GPR[RA], or 0 if RA is equal to 0.

Let nb be the unsigned integer value in bits 0:7 of GPR[RB].

If nb is equal to 0, the storage access is not performed and the contents of VSR[XT] are set to 0.

Otherwise, when Big-Endian byte-ordering is employed, do the following.

If nb less than 16, the contents of the nb bytes in storage starting at address EA are placed into the leftmost nb bytes of VSR[XT], and the contents of the rightmost 16-nb bytes of VSR[XT] are set to 0x00.

Otherwise, the contents of the quadword in storage at address EA are placed into VSR[XT].

Otherwise, when Little-Endian byte ordering is employed, do the following.

If nb less than 16, the contents of the nb bytes in storage starting at address EA are placed into the rightmost nb bytes of VSR[XT] in byte-reversed order, and the contents of the leftmost 16-nb bytes of VSR[XT] are set to 0x00.

Otherwise, the contents of the quadword in storage at address EA are placed into VSR[XT] in byte-reversed order.

If the contents of bits 8:63 of GPR[RB] are not equal to 0, the results are boundedly undefined.

Special Registers Altered:

None

Example: Loading less than 16-byte data into VSR using lxvl

```
char        S[14] = "This is a TEST";
short       X[6]  = { 0xE0E1, 0xE2E3, 0xE4E5, 0xE6E7, 0xE8E9, 0xEAEB };
binary80    Z     = 0xF0F1F2F3F4F5F6F7F8F9
```

Loading less than 16-byte data from Big-Endian storage in VSR[XT] using ***lxvl***.

Big-endian storage image of S, X, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
addr(S)+0x0000:	"T"	"h"	"i"	"s"	" "	"i"	"s"	" "	"a"	" "	"T"	"E"	"S"	"T"	E0	E1
addr(S)+0x0010:	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	F0	F1	F2	F3	F4	F5
addr(S)+0x0020:	F6	F7	F8	F9	00	00	00	00	00	00	00	00	00	00	00	00

```
# Assumptions
#   GPR[NS] = 14 (length of S in # of bytes)
#   GPR[NX] = 12 (length of X in # of bytes)
#   GPR[NZ] = 10 (length of Z in # of bytes)
#   GPR[PS] = address of S

add    rPX,rPS,rNS      # address of X
add    rPZ,rPX,rNX      # address of Z
sldi   rLS,rNS,56
sldi   rLX,rNX,56
sldi   rLZ,rNZ,56
lxvl   xS,rPS,rLS
lxvl   xX,rPX,rLX
lxvl   xZ,rPZ,rLZ
```

VSR register image of S, X, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
VSR[S]:	"T"	"h"	"i"	"s"	" "	"i"	"s"	" "	"a"	" "	"T"	"E"	"S"	"T"	00	00
VSR[X]:	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	00	00	00	00
VSR[Z]:	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	00	00	00	00	00	00

Loading less than 16-byte data from Little-Endian storage in VSR[XT] using ***lxvl***.

Little-endian storage image of S, X, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
addr(S)+0x0000:	"T"	"h"	"i"	"s"	" "	"i"	"s"	" "	"a"	" "	"T"	"E"	"S"	"T"	E1	E0
addr(S)+0x0010:	E3	E2	E5	E4	E7	E6	E9	E8	EB	EA	F9	F8	F7	F6	F5	F4
addr(S)+0x0020:	F3	F2	F1	F0	00	00	00	00	00	00	00	00	00	00	00	00

```
# Assumptions
#   GPR[NS] = 14 (length of S in # of bytes)
#   GPR[NX] = 12 (length of X in # of bytes)
#   GPR[NZ] = 10 (length of Z in # of bytes)
#   GPR[PS] = address of S

add    rPX,rPS,rNS      # address of X
add    rPZ,rPX,rNX      # address of Z
sldi   rLS,rNS,56
sldi   rLX,rNX,56
sldi   rLZ,rNZ,56
lxvl   xS,rPS,rLS
lxvl   xX,rPX,rLX
lxvl   xZ,rPZ,rLZ
```

VSR register image of S, X, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
VSR[S]:	00	00	"T"	"S"	"E"	"T"	" "	"a"	" "	"s"	"i"	" "	"s"	"i"	"h"	"T"
VSR[X]:	00	00	00	00	EA	EB	E8	E9	E6	E7	E4	E5	E2	E3	E0	E1
VSR[Z]:	00	00	00	00	00	00	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9

Load VSX Vector Left-justified with Length X-form

lxvll XT,RA,RB

31	T	RA	RB	301	TX
0	6	11	16	21	31

```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← (RA=0) ? 0 : GPR[RA]
nb ← EXTZ(GPR[RB].bit[0:7])
if nb>16 then nb ← 16
if nb>0 then do i = 0 to nb-1
   VSR[32×TX+T].byte[i] ← MEM(EA+i,1)
end
if nb<16 then do i = nb to 15
   VSR[32×TX+T].byte[i] ← 0x00
end
```

Let XT be the value 32×TX + T.

Let the effective address (EA) be the contents of GPR[RA], or 0 if RA is equal to 0.

Let nb be the unsigned integer value in bits 0:7 of GPR[RB].

If nb is equal to 0, the storage access is not performed and the contents of VSR[XT] are set to 0.

Otherwise, do the following.

If nb less than 16, the contents of the nb bytes in storage starting at address EA are placed into the lefttmost nb bytes of VSR[XT], and the contents of the rightmost 16-nb bytes of VSR[XT] are set to 0x00.

Otherwise, the contents of the quadword in storage at address EA are placed into VSR[XT].

Data is loaded from storage into VSR[XT] in Big-Endian byte ordering (i.e., the byte in storage at address EA is placed into byte element 0 of VSR[XT], the byte in storage at address EA+1 is placed in byte element 1 of VSR[XT], and so forth).

If the contents of bits 8:63 of GPR[RB] are not equal to 0, the results are boundedly undefined.

Special Registers Altered:

None

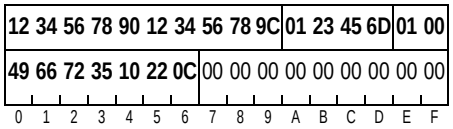
Example: Loading less than 16-byte left-justified data

```
decimal  X = +1234567890123456789;
decimal  Y = -123456;
decimal  Z = +1004966723510220;
```

Loading less than 16-byte data from storage in VSR[XT], left-justified, using ***lxvll***.

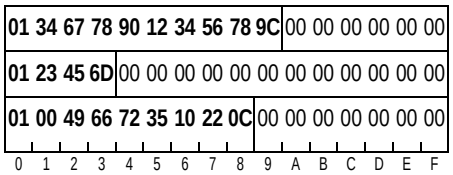
Initial state of VSRs X, Y, & Z

Big-endian & Little-Endian storage image of X, Y, & Z



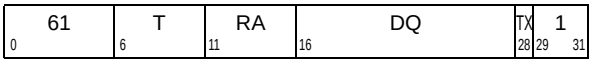

```
# Assumptions
#   GPR[NX] = 10 (length of X)
#   GPR[NY] =  4 (length of Y)
#   GPR[NZ] =  9 (length of Z)
#   GPR[PX] = address of X
#   GPR[PY] = address of Y = address of X + 10
#   GPR[PZ] = address of Z = address of X + 10 + 4

lxvll  xX,rPX,rNX
lxvll  xY,rPY,rNY
lxvll  xZ,rPZ,rNZ
```

Final state of VSRs X, Y, & Z



Load VSX Vector DQ-form

lxv XT,DQ(RA)




```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + EXTS(DQ||0b0000)

VSR[32×TX+T] ← MEM(EA,16)
```

Let XT be the value 32×TX + T.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the signed integer value DQ||0b0000.

When Big-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte element 0 of load_data,

– the contents of the byte in storage at address EA+1 are placed into byte element 1 of load_data, and so forth until

– the contents of the byte in storage at address EA+15 are placed into byte element 15 of load_data.

When Little-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte element 15 of load_data,

– the contents of the byte in storage at address EA+1 are placed into byte element 14 of load_data, and so forth until

– the contents of the byte in storage at address EA+15 are placed into byte element 0 of load_data.

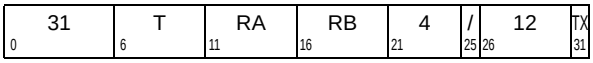
load_data is placed into VSR[XT].

Special Registers Altered

None

Load VSX Vector Indexed X-form

lxvx XT,RA,RB




```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[32×TX+T] ← MEM(EA,16)
```

Let XT be the value 32×TX + T.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte element 0 of load_data,

– the contents of the byte in storage at address EA+1 are placed into byte element 1 of load_data, and so forth until

– the contents of the byte in storage at address EA+15 are placed into byte element 15 of load_data.

When Little-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte element 15 of load_data,

– the contents of the byte in storage at address EA+1 are placed into byte element 14 of load_data, and so forth until

– the contents of the byte in storage at address EA+15 are placed into byte element 0 of load_data.

load_data is placed into VSR[XT].

Special Registers Altered:

None

Example: Loading data using *Load VSX Vector Indexed*

```
char    W[16] = { 0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7, 0xE0, 0xE1, 0xE2, 0xE3, 0xE4, 0xE5, 0xE6, 0xE7 };
short   X[8]  = { 0xF0F1, 0xF2F3, 0xF4F5, 0xF6F7, 0xE0E1, 0xE2E3, 0xE4E5, 0xE6E7 };
float   Y[4]  = { 0xF0F1_F2F3, 0xF4F5_F6F7, 0xE0E1_E2E3, 0xE4E5_E6E7 };
double  Z[2]  = { 0xF0F1_F2F3_F4F5_F6F7, 0xE0E1_E2E3_E4E5_E6E7 };
```

Loading 16 bytes of data from Big-Endian storage in VSR[XT] using ***lxvx***.

Big-endian storage image of W, X, Y, & Z

addr(W+0x0000):	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
addr(W+0x0010):	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
addr(W+0x0020):	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
addr(W+0x0030):	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

```
# Assumptions
#   GPR[PW] = address of W
#   GPR[PX] = address of X = GPR[PW] + 16
#   GPR[PY] = address of Y = GPR[PW] + 32
#   GPR[PZ] = address of Z = GPR[PW] + 48

lxvx   xW,r0,rPW
lxvx   xX,r0,rPX
lxvx   xY,r0,rPY
lxvx   xZ,r0,rPZ
```

Final state of VSRs W, X, Y, & Z

VSR[W]:	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
VSR[X]:	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
VSR[Y]:	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
VSR[Z]:	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Loading 16 bytes of data from Little-Endian storage in VSR[XT] using ***lxvx***.

Little-endian storage image of W, X, Y, & Z

addr(W+0x0000):	F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
addr(W+0x0010):	F1	F0	F3	F2	F5	F4	F7	F6	E1	E0	E3	E2	E5	E4	E7	E6
addr(W+0x0020):	F3	F2	F1	F0	F7	F6	F5	F4	E3	E2	E1	E0	E7	E6	E5	E4
addr(W+0x0030):	F7	F6	F5	F4	F3	F2	F1	F0	E7	E6	E5	E4	E3	E2	E1	E0
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

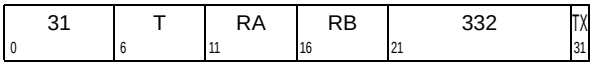
```
# Assumptions
#   GPR[PW] = address of W
#   GPR[PX] = address of X = GPR[PW] + 16
#   GPR[PY] = address of Y = GPR[PW] + 32
#   GPR[PZ] = address of Z = GPR[PW] + 48

lxvx   xW,r0,rPW
lxvx   xX,r0,rPX
lxvx   xY,r0,rPY
lxvx   xZ,r0,rPZ
```

Final state of VSRs W, X, Y, & Z

VSR[W]:	E7	E6	E5	E4	E3	E2	E1	E0	F7	F6	F5	F4	F3	F2	F1	F0
VSR[X]:	E6	E7	E4	E5	E2	E3	E0	E1	F6	F7	F4	F5	F2	F3	F0	F1
VSR[Y]:	E4	E5	E6	E7	E0	E1	E2	E3	F4	F5	F6	F7	F0	F1	F2	F3
VSR[Z]:	E0	E1	E2	E3	E4	E5	E6	E7	F0	F1	F2	F3	F4	F5	F6	F7
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Load VSX Vector Doubleword & Splat Indexed X-form

lxvdsx XT,RA,RB




```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

load_data ← MEM(EA, 8)

VSR[32×TX+T].dword[0] ← load_data
VSR[32×TX+T].dword[1] ← load_data
```

Let XT be the value 32×TX + T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte element 0 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte element 1 of load_data, and so forth until
- the contents of the byte in storage at address EA+7 are placed into byte element 7 of load_data.

When Little-Endian byte ordering is employed, the contents of the doubleword in storage at address EA are placed into load_data in such an order that;

- the contents of the byte in storage at address EA are placed into byte element 7 of load_data,
- the contents of the byte in storage at address EA+1 are placed into byte element 6 of load_data, and so forth until
- the contents of the byte in storage at address EA+7 are placed into byte element 0 of load_data.

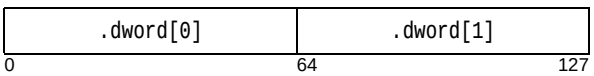
load_data is copied into each doubleword element of VSR[XT].

Special Registers Altered

None

VSR Data Layout for lxvdsx

tgt = VSR[XT]



*Load VSX Vector Halfword*8 Indexed X-form*

lxvh8x XT,RA,RB

31	T	RA	RB	812	TX
0	6	11	16	21	31

```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

do i = 0 to 7
   VSR[32×TX+T].hword[i] ← MEM(EA+2×i, 2)
end
```

Let XT be the value 32×TX + T.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value from 0 to 7, do the following.

When Big-Endian byte ordering is employed, the contents of the halfword in storage at address EA+2×i are placed into load_data in such an order that;

– the contents of the byte in storage at address EA+2×i are placed into byte element 0 of load_data,

– the contents of the byte in storage at address EA+2×i+1 are placed into byte element 1 of load_data.

When Little-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into VSR[XT] in such an order that;

– the contents of the byte in storage at address EA+2×i are placed into byte element 1 of load_data,

– the contents of the byte in storage at address EA+2×i+1 are placed into byte element 0 of load_data.

load_data is placed into halfword element i of VSR[XT].

Special Registers Altered:

None

Programming Note

lxvd2x, ***lxvw4x***, ***lxvh8x***, ***lxvb16x***, and ***lxvx*** exhibit identical behavior in Big-Endian mode.

Example: Loading data using *Load VSX Vector Halfword*8 Indexed*

```
short     X[] = { 0x0001, 0x1011, 0x2021, 0x3031,
                  0x4041, 0x5051, 0x6061, 0x7071 };
```

Big-endian storage image of X

Little-endian storage image of X

Loading a vector of 8 halfword elements from Big-Endian storage in VSR[XT] using ***lxvh8x***, retaining left-to-right element ordering.

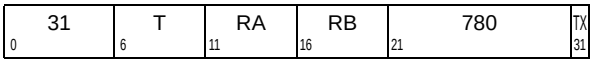
```
# Assumptions
#   GPR[PX] = address of X

lxvh8x   xX,r0,rPX
```

Loading a vector of 8 halfword elements from Little-Endian storage in VSR[XT] using ***lxvh8x***, retaining left-to-right element ordering.

```
# Assumptions
#   GPR[PX] = address of X

lxvh8x   xX,r0,rPX
```

*Load VSX Vector Word*4 Indexed X-form*

lxvw4x XT,RA,RB




```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

VSR[32×TX+T].word[0] ← MEM(EA, 4)
VSR[32×TX+T].word[1] ← MEM(EA+4, 4)
VSR[32×TX+T].word[2] ← MEM(EA+8, 4)
VSR[32×TX+T].word[3] ← MEM(EA+12, 4)
```

Let XT be the value 32×TX + T.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value i from 0 to 3, do the following.

When Big-Endian byte ordering is employed, the contents of the word in storage at address EA+4×i are placed into load_data in such an order that;

- the contents of the byte in storage at address EA+4×i are placed into byte element 0 of load_data,
- the contents of the byte in storage at address EA+4×i+1 are placed into byte element 1 of load_data,
- the contents of the byte in storage at address EA+4×i+2 are placed into byte element 2 of load_data, and
- the contents of the byte in storage at address EA+4×i+3 are placed into byte element 3 of load_data.

When Little-Endian byte ordering is employed, the contents of the word in storage at address EA+4×i are placed into word element i of VSR[XT] in such an order that;

- the contents of the byte in storage at address EA+4×i are placed into byte element 3 of load_data,
- the contents of the byte in storage at address EA+4×i+1 are placed into byte element 2 of load_data,
- the contents of the byte in storage at address EA+4×i+2 are placed into byte element 1 of load_data, and
- the contents of the byte in storage at address EA+4×i+3 are placed into byte element 0 of load_data.

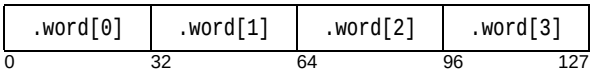
load_data is placed into word element i of VSR[XT].

Special Registers Altered

None

VSR Data Layout for lxvw4x

tgt = VSR[XT]



Programming Note

lxvd2x, ***lxvw4x***, ***lxvh8x***, ***lxvb16x***, and ***lxvx*** exhibit identical behavior in Big-Endian mode.

Load VSX Vector Word & Splat Indexed X-form

lxvwsx XT,RA,RB

31	T	RA	RB	364	TX
0	6	11	16	21	31

```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

load_data ← MEM(EA,4)

do i = 0 to 3
   VSR[32×TX+T].word[i] ← load_data
end
```

Let XT be the value 32×TX + T.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, the contents of the word in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte element 0 of load_data,

– the contents of the byte in storage at address EA+1 are placed into byte element 1 of load_data,

– the contents of the byte in storage at address EA+2 are placed into byte element 2 of load_data, and

– the contents of the byte in storage at address EA+3 are placed into byte element 3 of load_data.

When Little-Endian byte ordering is employed, the contents of the quadword in storage at address EA are placed into load_data in such an order that;

– the contents of the byte in storage at address EA are placed into byte element 3 of load_data,

– the contents of the byte in storage at address EA+1 are placed into byte element 2 of load_data,

– the contents of the byte in storage at address EA+2 are placed into byte element 1 of load_data, and

– the contents of the byte in storage at address EA+3 are placed into byte element 0 of load_data.

load_data is copied into each word element of VSR[XT].

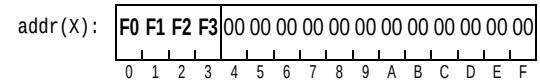
Special Registers Altered:

None

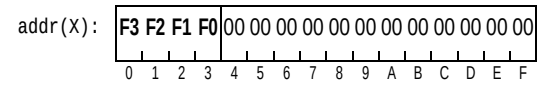
Example: Loading data using *Load VSX Vector Word & Splat Indexed*

```
int        X  = 0xF0F1_F2F3;
```

Big-endian storage image of X



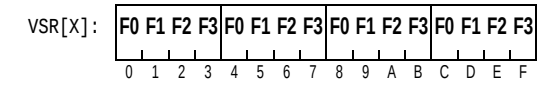
Little-endian storage image of X



Loading scalar word data from Big-Endian storage in VSR[XT] using ***lxvwsx***.

```
# Assumptions
#   GPR[PX] = address of X

lxvwsx   xX,r0,rPX
```

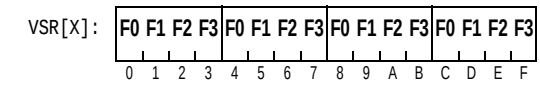
Final state of VSR X



Loading scalar word data from Little-Endian storage in VSR[XT] using ***lxvwsx***.

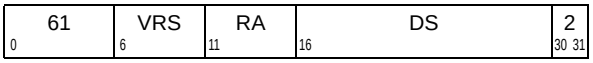
```
# Assumptions
#   GPR[PX] = address of X

lxvwsx   xX,r0,rPX
```

Final state of VSR X



Store VSX Scalar Doubleword DS-form

stxsd VRS,DS(RA)




```
if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + EXTS(DS||0b00)

MEM(EA,8)    ← VSR[VRS+32].dword[0]
```

Let XS be the value VRS + 32.

Let EA be the sum of the contents of GPR[RA], or 0 if RA=0, and the signed integer value DS<<2.

Let store_data be the contents of doubleword element 0 of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed in the doubleword in storage at address EA in such order that;

– byte 0 of store_data is placed into the byte in storage at address EA,

– byte 1 of store_data is placed into the byte in storage at address EA+1, and so forth until

– byte 7 of store_data is placed into the byte in storage at address EA+7.

When Little-Endian byte ordering is employed, store_data is placed in the doubleword in storage at address EA in such order that;

– the contents of byte 7 of doubleword element 0 of VSR[VRS+32] are placed into the byte in storage at address EA,

– the contents of byte 6 of doubleword element 0 of VSR[VRS+32] are placed into the byte in storage at address EA+1, and so forth until

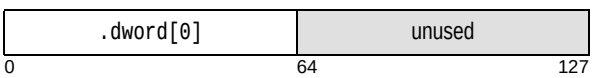
– the contents of byte 0 of doubleword element 0 of VSR[VRS+32] are placed into the byte in storage at address EA+7.

Special Registers Altered:

None

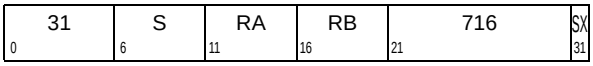
VSR Data Layout for stxsd

src = VSR[XS]



Store VSX Scalar Doubleword Indexed X-form

stxsdx XS,RA,RB




```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

MEM(EA,8)    ← VSR[XS].dword[0]
```

Let XS be the value 32×SX + S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

Let store_data be the contents of doubleword element 0 of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed in the doubleword in storage at address EA in such order that;

– byte 0 of store_data is placed into the byte in storage at address EA,

– byte 1 of store_data is placed into the byte in storage at address EA+1, and so forth until

– byte 7 of store_data is placed into the byte in storage at address EA+7.

When Little-Endian byte ordering is employed, store_data is placed in the doubleword in storage at address EA in such order that;

– byte 0 of store_data is placed into the byte in storage at address EA+7,

– byte 1 of store_data is placed into the byte in storage at address EA+6, and so forth until

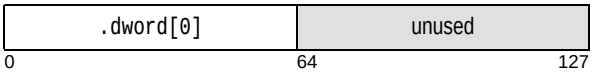
– byte 7 of store_data is placed into the byte in storage at address EA.

Special Registers Altered

None

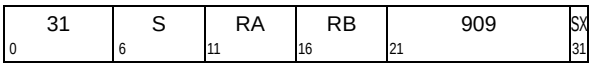
VSR Data Layout for stxsdx

src = VSR[XS]



Store VSX Scalar as Integer Byte Indexed X-form

stxsibx XS,RA,RB




```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

MEM(EA,1) ← VSR[32×SX+S].byte[7]
```

Let XS be the value 32×SX + S.

Let the effective address (EA) be sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

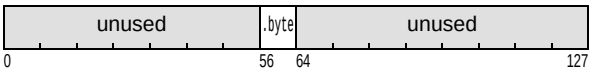
The contents of byte element 7 of VSR[XS] are placed into the byte in storage addressed by EA.

Special Registers Altered:

None

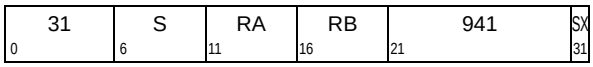
VSR Data Layout for stxsibx

src = VSR[XS]



Store VSX Scalar as Integer Halfword Indexed X-form

stxsihx XS,RA,RB




```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

MEM(EA,2) ← VSR[32×SX+S].hword[3]
```

Let XS be the value 32×SX + S.

Let the effective address (EA) be sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

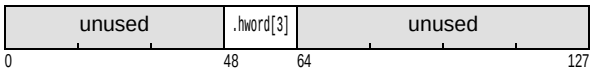
The contents of halfword element 3 of VSR[XS] are placed into the halfword in storage addressed by EA.

Special Registers Altered:

None

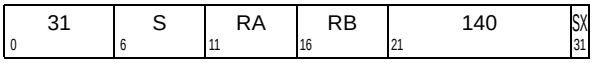
VSR Data Layout for stxsihx

src = VSR[XS]



Store VSX Scalar as Integer Word Indexed X-form

stxsiwx XS,RA,RB




```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ( (RA=0) ? 0 : GPR[RA] ) + GPR[RB]

MEM(EA,4) ← VSR[32×SX+S].word[1]
```

Let XS be the value 32×SX + S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

Let store_data be the contents of word element 1 of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed in the word in storage at address EA in such order that;

– byte 0 of store_data is placed into the byte in storage at address EA,

– byte 1 of store_data is placed into the byte in storage at address EA+1,

– byte 2 of store_data is placed into the byte in storage at address EA+2, and

– byte 3 of store_data is placed into the byte in storage at address EA+3.

When Little-Endian byte ordering is employed, store_data is placed in the word in storage at address EA in such order that;

– byte 0 of store_data is placed into the byte in storage at address EA+3,

– byte 1 of store_data is placed into the byte in storage at address EA+2,

– byte 2 of store_data is placed into the byte in storage at address EA+1, and

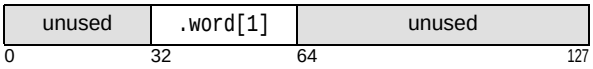
– byte 3 of store_data is placed into the byte in storage at address EA.

Special Registers Altered

None

VSR Data Layout for stxsiwx

src = VSR[XS]



Store VSX Scalar Single DS-form

stxssp VRS,DS(RA)

61	VRS	RA	DS	3
0	6	11	16	30 31

```
if MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + EXTS(DS||0b00)

MEM(EA,4) ← ConvertSP64toSP(VSR[VRS+32].dword[0])
```

Let XS be the value VRS + 32.

Let EA be the sum of the contents of GPR[RA], or 0 if RA=0, and the signed integer value DS||0b00.

Let store_data be the double-precision floating-point value in doubleword element 0 of VSR[XS] converted to single-precision format

When Big-Endian byte ordering is employed, store_data is placed in the word in storage at address EA in such order that;

– byte 0 of store_data is placed into the byte in storage at address EA,

– byte 1 of store_data is placed into the byte in storage at address EA+1,

– byte 2 of store_data is placed into the byte in storage at address EA+2, and

– byte 3 of store_data is placed into the byte in storage at address EA+3.

When Little-Endian byte ordering is employed, store_data is placed in the word in storage at address EA in such order that;

– byte 0 of store_data is placed into the byte in storage at address EA+3,

– byte 1 of store_data is placed into the byte in storage at address EA+2,

– byte 2 of store_data is placed into the byte in storage at address EA+1, and

– byte 3 of store_data is placed into the byte in storage at address EA.

Special Registers Altered:

None

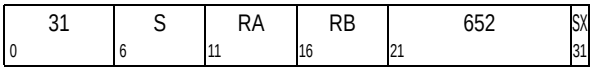
VSR Data Layout for stxssp

src = VSR[XS]

.dword[0]	unused
0	64 127

Store VSX Scalar Single-Precision Indexed X-form

stxsspx XS,RA,RB




```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

MEM(EA,4) ← ConvertSP64toSP(VSR[32×SX+S].dword[0])
```

Let XS be the value 32×SX + S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

Let store_data be the double-precision floating-point value in doubleword element 0 of VSR[XS] converted to single-precision format

When Big-Endian byte ordering is employed, store_data is placed in the word in storage at address EA in such order that;

– byte 0 of store_data is placed into the byte in storage at address EA,

– byte 1 of store_data is placed into the byte in storage at address EA+1,

– byte 2 of store_data is placed into the byte in storage at address EA+2, and

– byte 3 of store_data is placed into the byte in storage at address EA+3.

When Little-Endian byte ordering is employed, store_data is placed in the word in storage at address EA in such order that;

– byte 0 of store_data is placed into the byte in storage at address EA+3,

– byte 1 of store_data is placed into the byte in storage at address EA+2,

– byte 2 of store_data is placed into the byte in storage at address EA+1, and

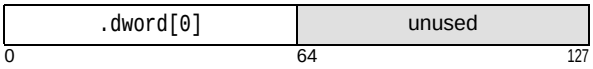
– byte 3 of store_data is placed into the byte in storage at address EA.

Special Registers Altered

None

VSR Data Layout for stxsspx

src = VSR[XS]



*Store VSX Vector Byte*16 Indexed X-form*

stxvb16x XS,RA,RB

31	S	RA	RB	1004	SX
0	6	11	16	21	31

```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

do i = 0 to 15
   MEM(EA+i,1) ← VSR[32×SX+S].byte[i]
end
```

Let XS be the value 32×SX + S.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value from 0 to 15, do the following.

The contents of byte element i of VSR[XS] are placed into the byte in storage at address EA+i.

Special Registers Altered:

None

Programming Note

stxvd2x, ***stxvw4x***, ***stxvh8x***, ***stxvb16x***, and ***stxvx*** exhibit identical behavior in Big-Endian mode.

Example: Storing data using *Store VSX Vector Byte*16 Indexed*

```
char       X[16];
```

VSR[X]:

F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Storing a vector of 16 byte elements from VSR[XS] into Big-Endian storage using ***sxvb16x***, retaining left-to-right element ordering.

```
# Assumptions
#   GPR[PX] = address of X

stxvb16x   xX,r0,rPX
```

Big-endian storage image of X

addr(X):

F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Loading a vector of 16 byte elements from Little-Endian storage in VSR[XT] using ***lxvb16x***, retaining left-to-right element ordering.

```
# Assumptions
#   GPR[PX] = address of X

stxvb16x   xX,r0,rPX
```

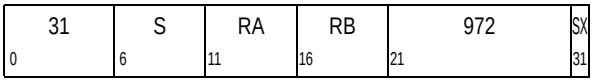
Little-endian storage image of X

addr(X):

F0	F1	F2	F3	F4	F5	F6	F7	E0	E1	E2	E3	E4	E5	E6	E7
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

*Store VSX Vector Doubleword*2 Indexed X-form*

stxvd2x XS,RA,RB




```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

MEM(EA,8)     ← VSR[32×SX+S].dword[0]
MEM(EA+8,8)   ← VSR[32×SX+S].dword[1]
```

Let XS be the value 32×SX + S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value i from 0 to 1, do the following.

Let store_data be the contents of doubleword element i of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed in the doubleword in storage at address EA+8×i in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA+8×i,
- byte 1 of store_data is placed into the byte in storage at address EA+8×i+1, and so forth until
- byte 7 of store_data is placed into the byte in storage at address EA+8×i+7.

When Little-Endian byte ordering is employed, store_data is placed in the doubleword in storage at address EA+8×i in such order that;

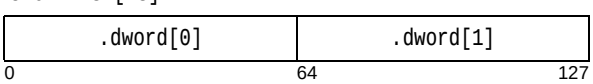
- byte 0 of store_data is placed into the byte in storage at address EA+8×i+7,
- byte 1 of store_data is placed into the byte in storage at address EA+8×i+6, and so forth until
- byte 7 of store_data is placed into the byte in storage at address EA+8×i.

Special Registers Altered

None

VSR Data Layout for stxvd2x

src = VSR[XS]



Programming Note

stxvd2x, ***stxvw4x***, ***stxvh8x***, ***stxvb16x***, and ***stxvx*** exhibit identical behavior in Big-Endian mode.

*Store VSX Vector Halfword*8 Indexed X-form*

stxvh8x XS,RA,RB

31	S	RA	RB	940	SX
0	6	11	16	21	31

```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

do i = 0 to 7
   MEM(EA+2×i,2) ← VSR[32×SX+S].hword[i]
end
```

Let XS be the value 32×SX + S.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value from 0 to 7, do the following.
The contents of byte element i of VSR[XS] are placed into the byte in storage at address EA+i.

For each integer value from 0 to 7, do the following.
When Big-Endian byte ordering is employed, the contents of halfword element i of VSR[XS] are placed into the halfword in storage at address EA+2×i in such an order that;

– the contents of byte sub-element 0 of halfword element i of VSR[XS] are placed into the byte in storage at address EA+2×i, and

– the contents of byte sub-element 1 of halfword element i of VSR[XS] are placed into the byte in storage at address EA+2×i+1.

When Little-Endian byte ordering is employed, the contents of halfword element i of VSR[XS] are placed into the halfword in storage at address EA+2×i in such an order that;

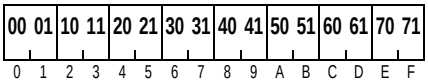
– the contents of byte sub-element 1 of halfword element i of VSR[XS] are placed into the byte in storage at address EA+2×i, and

– the contents of byte sub-element 0 of halfword element i of VSR[XS] are placed into the byte in storage at address EA+2×i+1.

Special Registers Altered:
None

Example: Storing data using *Store VSX Vector Halfword*8 Indexed*

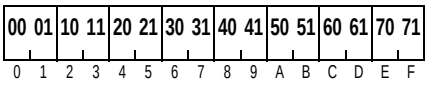
```
short     X[8];
```



Storing a vector of 8 halfword elements from VSR[X] into Big-Endian storage using ***stxvh8x***, retaining left-to-right element ordering.

```
# Assumptions
#   GPR[PX] = address of X

stxvh8x   xX,r0,rPX
```

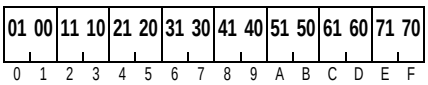
Big-endian storage image of X



Storing a vector of 8 halfword elements from VSR[X] into Little-Endian storage using ***stxvh8x***, retaining left-to-right element ordering.

```
# Assumptions
#   GPR[PX] = address of X

stxvh8x   xX,r0,rPX
```

Little-endian storage image of X

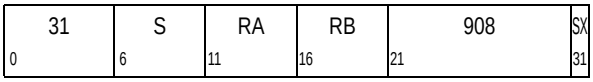


Programming Note

stxvd2x, ***stxvw4x***, ***stxvh8x***, ***stxvb16x***, and ***stxvx*** exhibit identical behavior in Big-Endian mode.

*Store VSX Vector Word*4 Indexed X-form*

stxvw4x XS,RA,RB




```
if MSR.VSX=0 then VSX_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

MEM(EA,4)     ← VSR[32×SX+S].word[0]
MEM(EA+4,4)   ← VSR[32×SX+S].word[1]
MEM(EA+8,4)   ← VSR[32×SX+S].word[2]
MEM(EA+12,4)  ← VSR[32×SX+S].word[3]
```

Let XS be the value 32×SX + S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

For each integer value i from 0 to 3, do the following.

Let store_data be the contents of word element i of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed in the word in storage at address EA+4×i in such order that;

- byte 0 of store_data is placed into the byte in storage at address EA+4×i,
- byte 1 of store_data is placed into the byte in storage at address EA+4×i+1, and so forth until
- byte 3 of store_data is placed into the byte in storage at address EA+4×i+3.

When Little-Endian byte ordering is employed, store_data is placed in the word in storage at address EA+4×i in such order that;

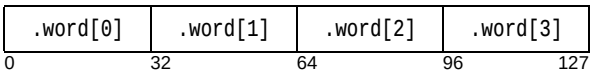
- byte 0 of store_data is placed into the byte in storage at address EA+4×i+3,
- byte 1 of store_data is placed into the byte in storage at address EA+4×i+2, and so forth until
- byte 3 of store_data is placed into the byte in storage at address EA+4×i.

Special Registers Altered

None

VSR Data Layout for stxvw4x

src = VSR[XS]

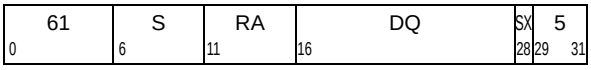


Programming Note

stxvd2x, ***stxvw4x***, ***stxvh8x***, ***stxvb16x***, and ***stxvx*** exhibit identical behavior in Big-Endian mode.

Store VSX Vector DQ-form

stxv XS,DQ(RA)




```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + EXTS(DQ||0b0000)
MEM(EA,16) ← VSR[32×SX+S]
```

Let XS be the value 32×SX + S.

Let EA be the sum of the contents of GPR[RA], or 0 if RA=0, and the signed integer value DQ<<4.

Let store_data be the contents of VSR[XS].

When Big-Endian byte ordering is employed, store_data is placed into the quadword in storage at address EA in such an order that;

– byte 0 of store_data is placed into the byte in storage at address EA,

– byte 1 of store_data is placed into the byte in storage at address EA+1, and so forth until

– byte 15 of store_data is placed into the byte in storage at address EA+15.

When Little-Endian byte ordering is employed, store_data is placed into the quadword in storage at address EA in such an order that;

– byte 15 of store_data is placed into the byte in storage at address EA,

– byte 14 of store_data is placed into the byte in storage at address EA+1, and so forth until

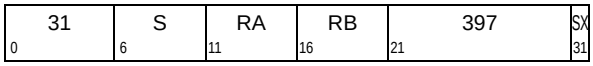
– byte 0 of store_data is placed into the byte in storage at address EA+15.

Special Registers Altered

None

Store VSX Vector with Length X-form

stxvl XS,RA,RB




```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← (RA=0) ? 0 : GPR[RA]

nb ← EXTZ(GPR[RB].bit[0:7])
if nb>16 then nb ← 16

if MSR.LE = 0 then // Big-Endian byte-ordering
   store_data ← VSR[32×SX+S].byte[0:nb-1]
else                    // Little-Endian byte ordering
   store_data ← VSR[32×SX+S].byte[16-nb:15]

MEM(EA,nb) ← store_data
```

Let XS be the value 32×SX + S.

Let the effective address (EA) be the contents of GPR[RA], or 0 if RA is equal to 0.

Let nb be the unsigned integer value in bits 0:7 of GPR[RB].

If nb is equal to 0, the storage access is not performed.

Otherwise, when Big-Endian byte-ordering is employed, do the following.

If nb less than 16, the contents of the leftmost nb bytes of VSR[XS] are placed in storage starting at address EA.

Otherwise, the contents of VSR[XS] are placed into the quadword in storage at address EA.

Otherwise, when Little-Endian byte ordering is employed, do the following.

If nb less than 16, the contents of the rightmost nb bytes of VSR[XS] are placed in storage starting at address EA in byte-reversed order.

Otherwise, the contents of VSR[XS] are placed into the quadword in storage at address EA in byte-reversed order.

If the contents of bits 8:63 of GPR[RB] are not equal to 0, the results are boundedly undefined.

Special Registers Altered:

None

Example: Storing less than 16-byte data from VSR using stxvl

```
char        S[14] = "This is a TEST";
short       X[6]  = { 0xE0E1, 0xE2E3, 0xE4E5, 0xE6E7, 0xE8E9, 0xEAEB };
binary80    Z     = 0xF0F1F2F3F4F5F6F7F8F9
```

Storing less than 16-byte data in VSR[XS] into Big-Endian storage using ***stxvl***.

```
# Assumptions
#   GPR[NS] = 14 (length of S in # of bytes)
#   GPR[NX] = 12 (length of X in # of bytes)
#   GPR[NZ] = 10 (length of Z in # of bytes)
#   GPR[PS] = address of S
```

VSR register image of S, X, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
VSR[S]:	"T"	"h"	"i"	"s"	" "	"i"	"s"	" "	"a"	" "	"T"	"E"	"S"	"T"	00	00
VSR[X]:	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	00	00	00	00
VSR[Z]:	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	00	00	00	00	00	00

```
add     rPX,rPS,rNS       # address of X
add     rPZ,rPX,rNX       # address of Z
sldi    rLS,rNS,56
sldi    rLX,rNX,56
sldi    rLZ,rNZ,56
stxvl   xS,rPS,rLS
stxvl   xX,rPX,rLX
stxvl   xZ,rPZ,rLZ
```

Final state of Big-Endian storage image of S, X, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
addr(S)+0x0000:	"T"	"h"	"i"	"s"	" "	"i"	"s"	" "	"a"	" "	"T"	"E"	"S"	"T"	E0	E1
addr(S)+0x0010:	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	F0	F1	F2	F3	F4	F5
addr(S)+0x0020:	F6	F7	F8	F9	00	00	00	00	00	00	00	00	00	00	00	00

Storing less than 16-byte data in VSR[XS] into Little-Endian storage using ***stxvl***.

```
# Assumptions
#   GPR[NS] = 14 (length of S in # of bytes)
#   GPR[NX] = 12 (length of X in # of bytes)
#   GPR[NZ] = 10 (length of Z in # of bytes)
#   GPR[PS] = address of S
```

VSR register image of S, X, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
VSR[S]:	00	00	"T"	"S"	"E"	"T"	" "	"a"	" "	"s"	"i"	" "	"s"	"i"	"h"	"T"
VSR[X]:	00	00	00	00	EA	EB	E8	E9	E6	E7	E4	E5	E2	E3	E0	E1
VSR[Z]:	00	00	00	00	00	00	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9

```
add     rPX,rPS,rNS       # address of X
add     rPZ,rPX,rNX       # address of Z
sldi    rLS,rNS,56
sldi    rLX,rNX,56
sldi    rLZ,rNZ,56
stxvl   xS,rPS,rLS
stxvl   xX,rPX,rLX
stxvl   xZ,rPZ,rLZ
```

Final state of Little-Endian storage image of S, X, & Z

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
addr(S)+0x0000:	"T"	"h"	"i"	"s"	" "	"i"	"s"	" "	"a"	" "	"T"	"E"	"S"	"T"	E1	E0
addr(S)+0x0010:	E3	E2	E5	E4	E7	E6	E9	E8	EB	EA	F9	F8	F7	F6	F5	F4
addr(S)+0x0020:	F3	F2	F1	F0	00	00	00	00	00	00	00	00	00	00	00	00

Store VSX Vector Left-justified with Length X-form

stxvll XS,RA,RB

31	S	RA	RB	429	SX
0	6	11	16	21	31

```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← (RA=0) ? 0 : GPR[RA]

nb ← EXTZ(GPR[RB].bit[0:7])
if nb>16 then nb ← 16

if nb>0 then do i = 0 to nb-1
   MEM(EA+i,1) ← VSR[32×SX+S].byte[i]
end
```

Let XS be the value 32×SX + S.

Let the effective address (EA) be the contents of GPR[RA], or 0 if RA is equal to 0.

Let nb be the unsigned integer value in bits 0:7 of GPR[RB].

If nb is equal to 0, the storage access is not performed.

Otherwise, do the following.

If nb less than 16, the contents of the leftmost nb bytes of VSR[XS] are placed in storage starting at address EA.

Otherwise, the contents of VSR[XS] are placed into the quadword in storage at address EA.

Data is stored from VSR[XS] into storage in Big-Endian byte ordering (i.e., the contents of byte element 0 of VSR[XS] are placed into the byte in storage at address EA, the contents of byte element 1 of VSR[XS] are placed into the byte in storage at address EA+1, and so forth).

If the contents of bits 8:63 of GPR[RB] are not equal to 0, the results are boundedly undefined.

Special Registers Altered:

None

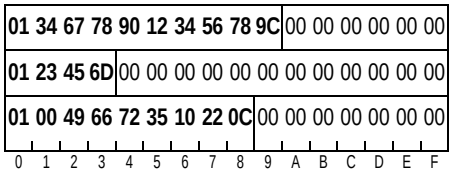
Example: Storing less than 16-byte left-justified data

```
decimal  X = +1234567890123456789;
decimal  Y = -123456;
decimal  Z = +1004966723510220;
```

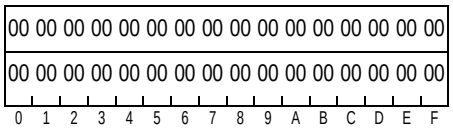
Storing less than 16-byte data, left-justified in VSR[XS], into storage using ***stxvll***.

```
# Assumptions
#   GPR[NX] = 10 (length of X)
#   GPR[NY] =  4 (length of Y)
#   GPR[NZ] =  9 (length of Z)
#   GPR[PX] = address of X
#   GPR[PY] = address of Y = address of X + 10
#   GPR[PZ] = address of Z = address of X + 10 + 4
```

VSRs X, Y, & Z

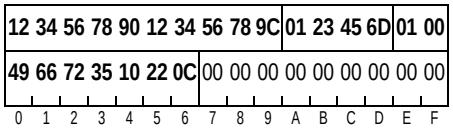


Initial state of Big-endian & Little-Endian storage image of X, Y, & Z



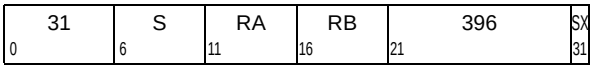

```
stxvll  xX,rPX,rNX
stxvll  xY,rPY,rNY
stxvll  xZ,rPZ,rNZ
```

Final state of Big-endian & Little-Endian storage image of X, Y, & Z



Store VSX Vector Indexed X-form

stxvx XS,RA,RB




```
if SX=0 & MSR.VSX=0 then VSX_Unavailable()
if SX=1 & MSR.VEC=0 then Vector_Unavailable()

EA ← ((RA=0) ? 0 : GPR[RA]) + GPR[RB]

MEM(EA,16) ← VSR[32×SX+S]
```

Let XS be the value 32×SX + S.

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA is equal to 0, and the contents of GPR[RB].

When Big-Endian byte ordering is employed, store_data is placed into the quadword in storage at address EA in such an order that;

– byte 0 of store_data is placed into the byte in storage at address EA,

– byte 1 of store_data is placed into the byte in storage at address EA+1, and so forth until

– byte 15 of store_data is placed into the byte in storage at address EA+15.

When Little-Endian byte ordering is employed, store_data is placed into the quadword in storage at address EA in such an order that;

– byte 15 of store_data is placed into the byte in storage at address EA,

– byte 14 of store_data is placed into the byte in storage at address EA+1, and so forth until

– byte 0 of store_data is placed into the byte in storage at address EA+15.

Special Registers Altered:

None

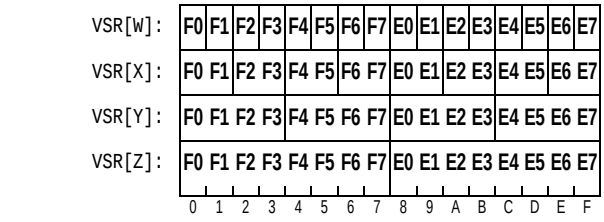
Programming Note

stxvd2x, ***stxvw4x***, ***stxvh8x***, ***stxvb16x***, and ***stxvx*** exhibit identical behavior in Big-Endian mode.

Example: Storing data using *Store VSX Vector Indexed*

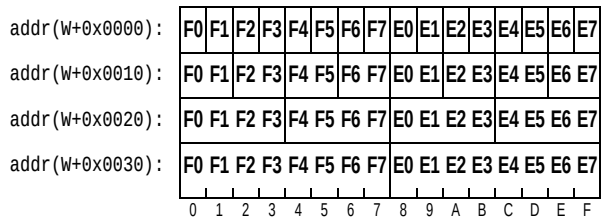
```
char    W[16] = { 0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7, 0xE0, 0xE1, 0xE2, 0xE3, 0xE4, 0xE5, 0xE6, 0xE7 };
short   X[8]  = { 0xF0F1, 0xF2F3, 0xF4F5, 0xF6F7, 0xE0E1, 0xE2E3, 0xE4E5, 0xE6E7 };
float   Y[4]  = { 0xF0F1_F2F3, 0xF4F5_F6F7, 0xE0E1_E2E3, 0xE4E5_E6E7 };
double  Z[2]  = { 0xF0F1_F2F3_F4F5_F6F7, 0xE0E1_E2E3_E4E5_E6E7 };
```

Storing 16 bytes of data into Big-Endian storage from VSR[XS] using ***stxvx***.

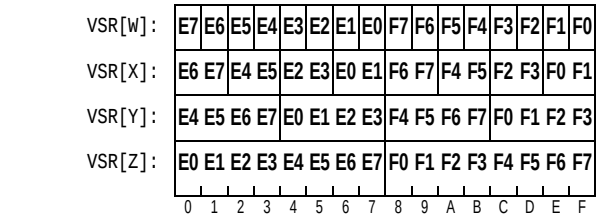



```
# Assumptions
#   GPR[PW] = address of W
#   GPR[PX] = address of X = GPR[PW] + 16
#   GPR[PY] = address of Y = GPR[PW] + 32
#   GPR[PZ] = address of Z = GPR[PW] + 48

stxvx   xW,r0,rPW
stxvx   xX,r0,rPX
stxvx   xY,r0,rPY
stxvx   xZ,r0,rPZ
```

Big-endian storage image of W, X, Y, & Z

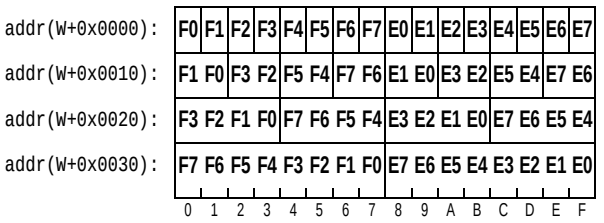


Storing 16 bytes of data into Little-Endian storage from VSR[XS] using ***stxvx***.



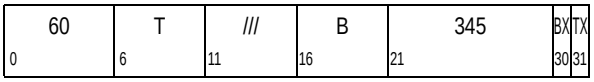

```
# Assumptions
#   GPR[PW] = address of W
#   GPR[PX] = address of X = GPR[PW] + 16
#   GPR[PY] = address of Y = GPR[PW] + 32
#   GPR[PZ] = address of Z = GPR[PW] + 48

stxvx   xW,r0,rPW
stxvx   xX,r0,rPX
stxvx   xY,r0,rPY
stxvx   xZ,r0,rPZ
```

Little-endian storage image of W, X, Y, & Z



VSX Scalar Absolute Double-Precision XX2-form

xsabsdp XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

result.bit[0]    ← 0b0
result.bit[1:63] ← VSR[32×BX+B].dword[0].bit[1:63]
VSR[32×TX+T].dword[0] ← result
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

The absolute value of the double-precision floating-point operand in doubleword element 0 of VSR[XB] is placed into doubleword element 0 of VSR[XT] in double-precision format.

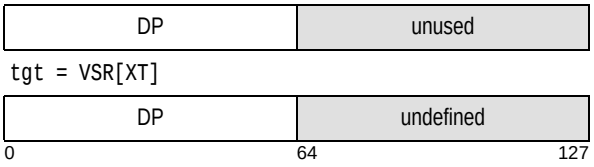
The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

None

VSR Data Layout for xsabsdp

src = VSR[XB]

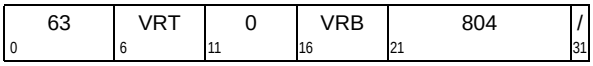


Programming Note

This instruction can be used to operate on a single-precision source operand.

VSX Scalar Absolute Quad-Precision X-form

xsabsqp VRT,VRB




```
if MSR.VSX=0 then VSX_Unavailable()

VSR[VRT+32] ← VSR[VRB+32] & 0x7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF
```

Let XT be the value VRT + 32.
Let XB be the value VRB + 32.

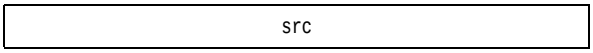
The absolute value of the quad-precision floating-point value in VSR[XB] is placed into VSR[XT].

Special Registers Altered:

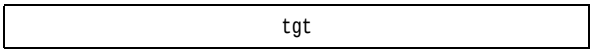
None

VSR Data Layout for xsabsqp

VSR[XB]



VSR[XT]



VSX Scalar Add Double-Precision XX3-form

xsadddp XT,XA,XB

60	T	A	B	32	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
reset_xflags()
src1         ← VSR[XA]{0:63}
src2         ← VSR[XB]{0:63}
v{0:inf}     ← AddDP(src1,src2)
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
vex_flag     ← VE & (vxsnan_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassSP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src2 is added[1] to src1, producing a sum having unbounded range and precision.

The sum is normalized[2].

See Table 49, “Actions for xsadddp,” on page 514.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI

VSR Data Layout for xsadddp

src1 = VSR[XA]

DP	unused

src2 = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← -Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← -Infinity	v ← src2	v ← -Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← -Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← -Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

`src1`	The double-precision floating-point value in doubleword element 0 of `VSR[XA]`.
`src2`	The double-precision floating-point value in doubleword element 0 of `VSR[XB]`.
`dQNaN`	Default quiet NaN (`0x7FF8_0000_0000_0000`).
`NZF`	Nonzero finite number.
`Rezd`	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
`A(x,y)`	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If `x = -y`, v is considered to be an exact-zero-difference result (`Rezd`).
`Q(x)`	Return a QNaN with the payload of x.
`v`	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 49.Actions for xsadddp

Range of v	Case	Rounding Mode				
		Round To Nearest (RTN)	Round Towards Zero (RTZ)	Round Towards +Infinity (RTP)	Round Towards –Infinity (RTM)	Round To Odd (RTO)
v is a QNaN	Special	r ← v	r ← v	r ← v	r ← v	r ← v
v = -Infinity	Special	r ← v	r ← v	r ← v	r ← v	r ← v
-Infinity < v [-(Nmax + 1ulp)	Overflow	q ← rnd(v) r ← -Infinity	q ← rnd(v) r ← -Nmax	q ← rnd(v) r ← -Nmax	q ← rnd(v) r ← -Infinity	q ← rnd(v) r ← -Nmax
-(Nmax + 1ulp) < v [-(Nmax + ½ulp)	Overflow	q ← rnd(v) r ← -Infinity	–	–	q ← rnd(v) r ← -Infinity	-
	Normal	–	r ← -Nmax	r ← -Nmax	–	r ← -Nmax
-(Nmax + ½ulp) < v < -Nmax	Overflow	–	–	–	q ← rnd(v) r ← -Infinity	-
	Normal	r ← -Nmax	r ← -Nmax	r ← -Nmax	–	r ← -Nmax
-Nmax [v [-Nmin	Normal	r ← rnd(v)	r ← rnd(v)	r ← rnd(v)	r ← rnd(v)	r ← rnd(v)
-Nmin < v < -Zero	Tiny	q ← rnd(v) r ← rnd(den(v))	q ← rnd(v) r ← rnd(den(v))	q ← rnd(v) r ← rnd(den(v))	q ← rnd(v) r ← rnd(den(v))	q ← rnd(v) r ← rnd(den(v))
v = -Zero	Special	r ← v	r ← v	r ← v	r ← v	r ← v
v = Rezd	Special	r ← +Zero	r ← +Zero	r ← +Zero	r ← -Zero	r ← +Zero (RN=RTN) r ← +Zero (RN=RTZ) r ← +Zero (RN=RTP) r ← -Zero (RN=RTM)
v = +Zero	Special	r ← v	r ← v	r ← v	r ← v	r ← v
+Zero < v < +Nmin	Tiny	q ← rnd(v) r ← rnd(den(v))	q ← rnd(v) r ← rnd(den(v))	q ← rnd(v) r ← rnd(den(v))	q ← rnd(v) r ← rnd(den(v))	q ← rnd(v) r ← rnd(den(v))
+Nmin [v [+Nmax	Normal	r ← rnd(v)	r ← rnd(v)	r ← rnd(v)	r ← rnd(v)	r ← rnd(v)
+Nmax < v < +(Nmax + ½ulp)	Overflow	–	–	q ← rnd(v) r ← +Infinity	–	-
	Normal	r ← +Nmax	r ← +Nmax	–	r ← +Nmax	r ← +Nmax
+(Nmax + ½ulp) [v < +(Nmax + 1ulp)	Overflow	q ← rnd(v) r ← +Infinity	–	q ← rnd(v) r ← +Infinity	–	-
	Normal	–	r ← +Nmax	–	r ← +Nmax	r ← +Nmax
+(Nmax + 1ulp) [v < +Infinity	Overflow	q ← rnd(v) r ← +Infinity	q ← rnd(v) r ← +Nmax	q ← rnd(v) r ← +Infinity	q ← rnd(v) r ← +Nmax	q ← rnd(v) r ← +Nmax
v = +Infinity	Special	r ← v	r ← v	r ← v	r ← v	r ← v

Explanation:

–	This situation cannot occur.
v	The precise intermediate result defined in the instruction having unbounded range and precision.
den(x)	The significand of x is shifted right by the amount of the difference between the target rounding precision Emin and the unbiased exponent of x. The unbiased exponent of the denormalized value is Emin. The significand of the denormalized value has unbounded significand precision. Emin = -16382 (quad-precision) Emin = -16382 (double-extended-precision) Emin = -1022 (double-precision) Emin = -126 (single-precision)
Rezd	Exact-zero-difference result. Applies only to add operations involving source operands having the same magnitude and different signs or subtract operations involving source operands having the same magnitude and same signs. Whether +Zero or -Zero is returned is controlled by the setting of the rounding mode in RN, even when the rounding mode is overridden to Round to Odd.
rnd(x)	The significand of x is rounded to the target rounding precision according to the rounding mode specified in FPSCR.RN. Exponent range of the rounded result is unbounded. See Section 7.3.2.6.
Nmax	Largest (in magnitude) representable normalized number in the target rounding precision format. Nmax = ± 2^{+16383} × 1.FFFFFFFFFFFFFFFFFFFFFFFFFFFF (quad-precision) Nmax = ± 2^{+16383} × 1.FFFFFFFFFFFFFFF000000000000 (double-extended-precision) Nmax = ± 2^{+1023} × 1.FFFFFFFFFFFFF0000000000000000 (double-precision) Nmax = ± 2^{+127} × 1.FFFFFF00000000000000000000000 (single-precision)
Nmin	Smallest (in magnitude) representable normalized number in the target rounding precision format. Nmin = ± 2^{-16382} × 1.00000000000000000000000000000000 (quad-precision) Nmin = ± 2^{-16382} × 1.00000000000000000000000000000000 (double-extended-precision) Nmin = ± 2^{-1022} × 1.00000000000000000000000000000000 (double-precision) Nmin = ± 2^{-126} × 1.00000000000000000000000000000000 (single-precision)
ulp	Least significant bit in the target precision format's significand (Unit in the Last Position).

Table 50.Scalar Floating-Point Intermediate Result Handling

Case	FPSCR.VE	FPSCR.OE	FPSCR.UE	FPSCR.ZE	FPSCR.XE	vxsnan_flag	vximz_flag	vxisi_flag	vxidi_flag	vxzdz_flag	vxsqrt_flag	zx_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Special	–	–	–	0	–	–	–	–	–	–	–	1	–	–	–	–	T(r), class_bfp(r), fi(0), fr(0), fx(ZX)
	–	–	–	1	–	–	–	–	–	–	–	1	–	–	–	–	fx(ZX), error()
	0	–	–	–	–	–	–	–	–	–	1	–	–	–	–	–	T(r), class_bfp(r), fi(0), fr(0), fx(VXSQRT)
	0	–	–	–	–	–	–	–	–	1	–	–	–	–	–	–	T(r), class_bfp(r), fi(0), fr(0), fx(VXZDZ)
	0	–	–	–	–	–	–	–	1	–	–	–	–	–	–	–	T(r), class_bfp(r), fi(0), fr(0), fx(VXIDI)
	0	–	–	–	–	–	–	1	–	–	–	–	–	–	–	–	T(r), class_bfp(r), fi(0), fr(0), fx(VXISI)
	0	–	–	–	–	0	1	–	–	–	–	–	–	–	–	–	T(r), class_bfp(r), fi(0), fr(0), fx(VXIMZ)
	0	–	–	–	–	1	0	–	–	–	–	–	–	–	–	–	T(r), class_bfp(r), fi(0), fr(0), fx(VXSNAN)
	0	–	–	–	–	1	0	–	–	–	–	–	–	–	–	–	T(r), class_bfp(r), fi(0), fr(0), fx(VXSNAN), fx(VXIMZ)
	1	–	–	–	–	–	–	–	–	–	1	–	–	–	–	–	fx(VXSQRT), error()
	1	–	–	–	–	–	–	–	–	1	–	–	–	–	–	–	fx(VXZDZ), error()
	1	–	–	–	–	–	–	–	1	–	–	–	–	–	–	–	fx(VXIDI), error()
	1	–	–	–	–	–	–	1	–	–	–	–	–	–	–	–	fx(VXISI), error()
	1	–	–	–	–	0	1	–	–	–	–	–	–	–	–	–	fx(VXIMZ), error()
	1	–	–	–	–	1	0	–	–	–	–	–	–	–	–	–	fx(VXSNAN), error()
	1	–	–	–	–	1	1	–	–	–	–	–	–	–	–	–	fx(VXSNAN), fx(VXIMZ), error()

Explanation:

–	The results do not depend on this condition.
T(x)	Places the result into the target VSR. For scalar single-precision and double-precision results VSR[XT].dword[0] ← bfp_CONVERT_TO_BFP64(r) VSR[XT].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU For scalar quad-precision results VSR[VRT+32] ← bfp_CONVERT_TO_BFP128(r)
class_bfp(x)	Sets FPSCR.FPRF to the sign and class of x. FPSCR.FPRF ← fprf_CLASS_BFP32(x) (single-precision) FPSCR.FPRF ← fprf_CLASS_BFP64(x) (double-precision) FPSCR.FPRF ← fprf_CLASS_BFP128(x) (quad-precision)
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
fi(x)	FPSCR.FI is set to the value x.
fr(x)	FPSCR.FR is set to the value x.
β	Wrap adjust $\beta = 2^{192}$ (single-precision) $\beta = 2^{1536}$ (double-precision) $\beta = 2^{24576}$ (quad-precision) See Table 7.4.3.2, “Action for OE=1,” on page 404 for trap-enabled Overflow exceptions. See Table 7.4.4.2, “Action for UE=1,” on page 409 for trap-enabled Underflow exceptions.
q	The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, significand rounded to the target rounding precision, unbounded exponent range.
r	The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, significand rounded to the target rounding precision, exponent bounded to the target rounding precision format exponent range.
error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.

Table 51.VSX Scalar Floating-Point Final Result

Case	FPSCR.VE	FPSCR.OE	FPSCR.UE	FPSCR.ZE	FPSCR.XE	vxsnan_flag	vximz_flag	vxisi_flag	vxidi_flag	vxzdz_flag	vxsqrt_flag	zx_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Normal	–	–	–	–	–	0	0	0	0	0	0	0	no	–	–	–	T(r), class_bfp(r), fi(0), fr(0)
	–	–	–	–	0	0	0	0	0	0	0	0	yes	no	–	–	T(r), class_bfp(r), fi(1), fr(0), fx(XX)
	–	–	–	–	0	0	0	0	0	0	0	0	yes	yes	–	–	T(r), class_bfp(r), fi(1), fr(1), fx(XX)
	–	–	–	–	1	0	0	0	0	0	0	0	yes	no	–	–	T(r), class_bfp(r), fi(1), fr(0), fx(XX), error()
	–	–	–	–	1	0	0	0	0	0	0	0	yes	yes	–	–	T(r), class_bfp(r), fi(1), fr(1), fx(XX), error()
Overflow	–	0	–	–	0	0	0	0	0	0	0	0	–	–	–	–	T(r), class_bfp(r), fi(1), fr(?), fx(OX), fx(XX)
	–	0	–	–	1	0	0	0	0	0	0	0	–	–	–	–	T(r), class_bfp(r), fi(1), fr(?), fx(OX), fx(XX), error()
	–	1	–	–	–	0	0	0	0	0	0	0	–	–	no	–	T(q÷β), class_bfp(q÷β), fi(0), fr(0), fx(OX), error()
	–	1	–	–	–	0	0	0	0	0	0	0	–	–	yes	no	T(q÷β), class_bfp(q÷β), fi(1), fr(0), fx(OX), fx(XX), error()
	–	1	–	–	–	0	0	0	0	0	0	0	–	–	yes	yes	T(q÷β), class_bfp(q÷β), fi(1), fr(1), fx(OX), fx(XX), error()
Tiny	–	–	0	–	–	0	0	0	0	0	0	0	no	–	–	–	T(r), class_bfp(r), fi(0), fr(0)
	–	–	0	–	0	0	0	0	0	0	0	0	yes	no	–	–	T(r), class_bfp(r), fi(1), fr(0), fx(UX), fx(XX)
	–	–	0	–	0	0	0	0	0	0	0	0	yes	yes	–	–	T(r), class_bfp(r), fi(1), fr(1), fx(UX), fx(XX)
	–	–	0	–	1	0	0	0	0	0	0	0	yes	no	–	–	T(r), class_bfp(r), fi(1), fr(0), fx(UX), fx(XX), error()
	–	–	0	–	1	0	0	0	0	0	0	0	yes	yes	–	–	T(r), class_bfp(r), fi(1), fr(1), fx(UX), fx(XX), error()
	–	–	1	–	–	0	0	0	0	0	0	0	–	–	no	–	T(q×β), class_bfp(q×β), fi(0), fr(0), fx(UX), error()
	–	–	1	–	–	0	0	0	0	0	0	0	–	–	yes	no	T(q×β), class_bfp(q×β), fi(1), fr(0), fx(UX), fx(XX), error()
	–	–	1	–	–	0	0	0	0	0	0	0	–	–	yes	yes	T(q×β), class_bfp(q×β), fi(1), fr(1), fx(UX), fx(XX), error()

Explanation:

–	The results do not depend on this condition.
T(x)	Places the result into the target VSR. For scalar single-precision and double-precision results VSR[XT].dword[0] ← bfp_CONVERT_TO_BFP64(r) VSR[XT].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU For scalar quad-precision results VSR[VRT+32] ← bfp_CONVERT_TO_BFP128(r)
class_bfp(x)	Sets FPSCR.FPRF to the sign and class of x. FPSCR.FPRF ← fprf_CLASS_BFP32(x) (single-precision) FPSCR.FPRF ← fprf_CLASS_BFP64(x) (double-precision) FPSCR.FPRF ← fprf_CLASS_BFP128(x) (quad-precision)
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
fi(x)	FPSCR.FI is set to the value x.
fr(x)	FPSCR.FR is set to the value x.
β	Wrap adjust $\beta = 2^{192}$ (single-precision) $\beta = 2^{1536}$ (double-precision) $\beta = 2^{24576}$ (quad-precision) See Table 7.4.3.2, “Action for OE=1,” on page 404 for trap-enabled Overflow exceptions. See Table 7.4.4.2, “Action for UE=1,” on page 409 for trap-enabled Underflow exceptions.
q	The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, significand rounded to the target rounding precision, unbounded exponent range.
r	The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, significand rounded to the target rounding precision, exponent bounded to the target rounding precision format exponent range.
error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.

Table 51.VSX Scalar Floating-Point Final Result (Continued)

VSX Scalar Add Single-Precision XX3-form

xsaddsp XT,XA,XB

60	T	A	B	0	AX	BX	TX
0	6	11	16	21	29	30	31

```
reset_xflags()

src1   ← VSR[32×AX+A].dword[0]
src2   ← VSR[32×BX+B].dword[0]
v      ← AddDP(src1,src2)
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[32×TX+T].dword[0] ← ConverSPtoDP(result)
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src2 is added[1] to src1, producing a sum having unbounded range and precision.

The sum is normalized[2].

See Table 52, “Actions for xsaddsp,” on page 519.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI

VSR Data Layout for xsaddsp

src1 = VSR[XA]

DP	unused

src2 = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

		src2							
		-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
src1	-Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	-NZF	v ← -Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	-Zero	v ← -Infinity	v ← src2	v ← -Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← -Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← -Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
	SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (0x7FF8_0000_0000_0000).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 52.Actions for xsaddsp

VSX Scalar Add Quad-Precision [*using round to Odd*] *X-form*

xsaddqp VRT,VRA,VRB (RO=0)
xsaddqpo VRT,VRA,VRB (RO=1)

63	VRT	VRA	VRB	4	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2   ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_ADD(src1, src2)
rnd    ← bfp_ROUND_TO_BFP128(RO, FPSCR.RN, v)
result ← bfp_CONVERT_TO_BFP128(rnd)

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(vxisi_flag)  then SetFX(FPSCR.VXISI)
if(ox_flag)     then SetFX(FPSCR.OX)
if(ux_flag)     then SetFX(FPSCR.UX)
if(xx_flag)     then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vxisi_flag
ex_flag  ← FPSCR.VE & vx_flag

if ex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1 or src2 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 and src2 are Infinity values having opposite signs, an Invalid Operation exception occurs and VXISI is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src2 is a Quiet NaN, the result is src2.

Otherwise, if src1 and src2 are Infinity values having opposite signs, the result is the default Quiet NaN[1].

Otherwise, do the following.

The normalized sum of src2 added to src1 is produced with unbounded significand precision and exponent range.

See Table 53, "Actions for xsaddqp[o]," on page 521.

If the intermediate result is *Tiny* (i.e., the unbiased exponent is less than -16382) and UE=0, the significand is shifted right N bits, where N is the difference between -16382 and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value -16382.

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. The intermediate result is rounded to quad-precision using the specified rounding mode.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

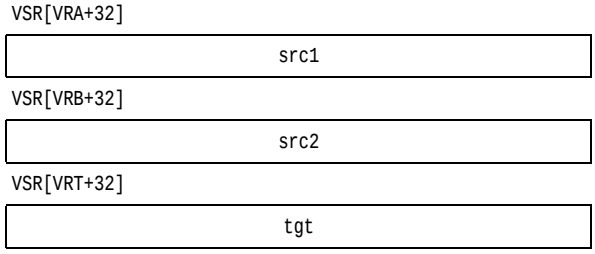
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, "VSX Scalar Floating-Point Final Result," on page 516.

Special Registers Altered:

FPRF FR FI
FX VXSNAN VXISI OX UX XX

VSR Data Layout for xsaddqp[o]



1. The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.

<table>
<tr><th></th><th></th><th colspan="8">src2</th></tr>
<tr><th></th><th></th><th>-Infinity</th><th>-NZF</th><th>-Zero</th><th>+Zero</th><th>+NZF</th><th>+Infinity</th><th>QNaN</th><th>SNaN</th></tr>
<tr><td rowspan="8">src1</td><td>-Infinity</td><td colspan="5" rowspan="2">v ← -Infinity</td><td>v ← dQNaN
vxisi_flag ← 1</td><td rowspan="8">v ← src2</td><td rowspan="7">v ← quiet(src2)
vxsnan_flag ← 1</td></tr>
<tr><td>-NZF</td><td>v ← add(src1,src2)</td><td colspan="2">v ← src1</td><td>v ← add(src1,src2)</td><td rowspan="6"></td></tr>
<tr><td>-Zero</td><td rowspan="2">v ← src2</td><td>v ← -Zero</td><td>v ←Rezd</td><td rowspan="2">v ← src2</td></tr>
<tr><td>+Zero</td><td>v ← Rezd</td><td>v ← +Zero</td></tr>
<tr><td>+NZF</td><td>v ← add(src1,src2)</td><td colspan="2">v ← src1</td><td>v ← add(src1,src2)</td></tr>
<tr><td>+Infinity</td><td>v ← dQNaN
vxisi_flag ← 1</td><td colspan="4">v ← +Infinity</td></tr>
<tr><td>QNaN</td><td colspan="7">v ← src1</td><td>v ← src1
vxsnan_flag ← 1</td></tr>
<tr><td>SNaN</td><td colspan="8">v ← quiet(src1)
vxsnan_flag ← 1</td></tr>
</table>

Explanation:

src1	The quad-precision floating-point value in VSR[VRA+32].
src2	The quad-precision floating-point value in VSR[VRB+32].
dQNaN	Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000_0000).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude and opposite signs).
add(x,y)	The floating-point value y is added[1] to the floating-point value x. Return the normalized[2] sum, having unbounded significand precision and exponent range. When x = -y, v is considered to be an exact-zero-difference result (Rezd).
quiet(x)	Convert x to the corresponding Quiet NaN by setting the most significant fraction bit to 1.
v	The intermediate result having unbounded significand precision and unbounded exponent range.

Table 53. Actions for xsaddqp[o]

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate difference.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSX Scalar Compare Exponents Double-Precision XX3-form

xscmpexpdp BF,XA,XB

60	BF	//	A	B	59	AX	BX	/
0	6	9	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← VSR[32×AX+A].dword[0]
src2 ← VSR[32×BX+B].dword[0]

src1.exponent ← EXTZ(src1.bit[1:11])
src2.exponent ← EXTZ(src2.bit[1:11])
src1.fraction ← src1.bit[12:63]
src2.fraction ← src2.bit[12:63]

src1.class.NaN ← (src1.exponent = 2047) & (src1.fraction != 0)
src2.class.NaN ← (src2.exponent = 2047) & (src2.fraction != 0)

lt_flag ← (src1.exponent < src2.exponent)
gt_flag ← (src1.exponent > src2.exponent)
eq_flag ← (src1.exponent = src2.exponent)
uo_flag ← src1.class.NaN | src2.class.NaN

CR.bit[4×BF+32] ← FPSCR.FL ← !uo_flag & lt_flag
CR.bit[4×BF+33] ← FPSCR.FG ← !uo_flag & gt_flag
CR.bit[4×BF+34] ← FPSCR.FE ← !uo_flag & eq_flag
CR.bit[4×BF+35] ← FPSCR.FU ←  uo_flag
```

Let XA be the sum 32×AX + A.
Let XB be the sum 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

The exponent of src1 is compared with the exponent of src2. The result of the compare is placed into FPCC and CR field BF.

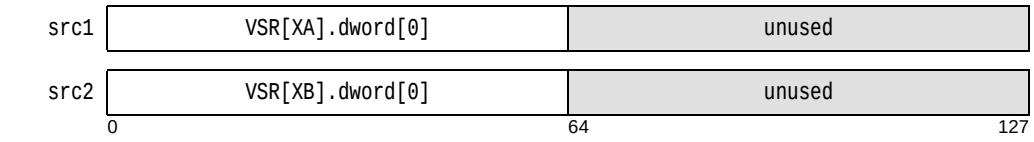
Special Registers Altered:

CR field BF
FPCC

Programming Note

This instruction can be used to operate on single-precision source operands.

VSR Data Layout for xscmpexpdp



VSX Scalar Compare Exponents Quad-Precision X-form

xscmpexpqp BF,VRA,VRB

63	BF	//	VRA	VRB	164	/
0	6	9	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_flags()

src1          ← VSR[VRA+32]
src2          ← VSR[VRB+32]

src1.exponent ← EXTZ(src1.bit[1:15])
src2.exponent ← EXTZ(src2.bit[1:15])
src1.fraction ← EXTZ(src1.bit[16:127])
src2.fraction ← EXTZ(src2.bit[16:127])

src1.class.NaN ← (src1.exponent = 32767) & (src1.fraction != 0)
src2.class.NaN ← (src2.exponent = 32767) & (src2.fraction != 0)

lt_flag ← (src1.exponent < src2.exponent)
gt_flag ← (src1.exponent > src2.exponent)
eq_flag ← (src1.exponent = src2.exponent)
uo_flag ← src1.class.NaN | src2.class.NaN

CR.bit[4×BF+32] ← FPSCR.FL ← !uo_flag & lt_flag
CR.bit[4×BF+33] ← FPSCR.FG ← !uo_flag & gt_flag
CR.bit[4×BF+34] ← FPSCR.FE ← !uo_flag & eq_flag
CR.bit[4×BF+35] ← FPSCR.FU ←  uo_flag
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

The exponent of src1 is compared with the exponent of src2 as unsigned integer values. The result of the compare is placed into FPCC and CR field BF.

Special Registers Altered:

CR field BF
FPCC

VSR Data Layout for xscmpexpqp

VSR[VRA+32]

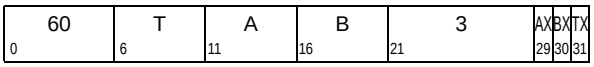
src1

VSR[VRB+32]

src2

VSX Scalar Compare Equal Double-Precision XX3-form

xscmpeqdp XT,XA,XB




```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])

vxsnan_flag ← (src1.class="SNaN") | (src2.class="SNaN")

vex_flag ← FPSCR.VE & vxsnan_flag

if(vxsnan_flag) SetFX(FPSCR.VXSNAN)

if (vex_flag=0) then do
   if bfp_COMPARE_EQ(src1, src2)=1 then
      VSR[32×TX+T].dword[0] ← 0xFFFF_FFFF_FFFF_FFFF
      VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   end
   else do
      VSR[32×TX+T].dword[0] ← 0x0000_0000_0000_0000
      VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   end
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.

Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.

src1 is compared to src2.

A NaN compared to any value, including itself, compares false for the predicate, equal.

The contents of doubleword 0 of VSR[XT] are set to 0xFFFF_FFFF_FFFF_FFFF if src1 is equal to src2, and are set to 0x0000_0000_0000_0000 otherwise.

The contents of doubleword 1 of VSR[XT] are set to 0x0000_0000_0000_0000.

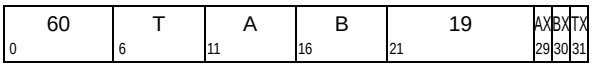
If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

Special Registers Altered:

FX VXSNAN

VSX Scalar Compare Greater Than or Equal Double-Precision XX3-form

xscmpgedp XT,XA,XB




```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])

if (src1.class="SNaN") | (src2.class="SNaN") then do
   vxsnan_flag ← 0b1
   if(FPSCR.VE=0) then vxvc_flag ← 0b1
end
else
   vxvc_flag ← (src1.class="QNaN") | (src2.class="QNaN")

vex_flag ← FPSCR.VE & (vxsnan_flag | vxvc_flag)

if (vxsnan_flag=1) SetFX(FPSCR.VXSNAN)
if (vxcv_flag=1)   SetFX(FPSCR.VXVC)

if (vex_flag=0) then do
   if bfp_COMPARE_GE(src1, src2)=1 then
      VSR[32×TX+T].dword[0] ← 0xFFFF_FFFF_FFFF_FFFF
      VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   end
   else do
      VSR[32×TX+T].dword[0] ← 0x0000_0000_0000_0000
      VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   end
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

src1 is compared to src2.

A NaN compared to any value, including itself, compares false for the predicate, greater than or equal.

The contents of doubleword 0 of VSR[XT] are set to 0xFFFF_FFFF_FFFF_FFFF if src1 is greater than or equal to src2, and are set to 0x0000_0000_0000_0000 otherwise.

The contents of doubleword 1 of VSR[XT] are set to 0x0000_0000_0000_0000.

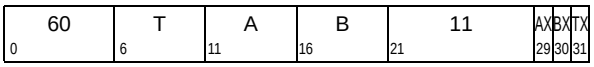
If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

Special Registers Altered:

FX VXSNAN VXVC

VSX Scalar Compare Greater Than Double-Precision XX3-form

xscmpgtdp XT,XA,XB




```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])

if (src1.class="SNaN") | (src2.class="SNaN") then do
   vxsnan_flag ← 0b1
   if(FPSCR.VE=0) then vxvc_flag ← 0b1
end
else
   vxvc_flag ← (src1.class="QNaN") | (src2.class="QNaN")

vex_flag ← FPSCR.VE & (vxsnan_flag | vxvc_flag)

if (vxsnan_flag=1) SetFX(FPSCR.VXSNAN)
if (vxcv_flag=1)   SetFX(FPSCR.VXVC)

if (vex_flag=0) then do
   if bfp_COMPARE_GT(src1, src2)=1 then
      VSR[32×TX+T].dword[0] ← 0xFFFF_FFFF_FFFF_FFFF
      VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   end
   else do
      VSR[32×TX+T].dword[0] ← 0x0000_0000_0000_0000
      VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
   end
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

src1 is compared to src2.

A NaN compared to any value, including itself, compares false for the predicate, greater than.

The contents of doubleword 0 of VSR[VRT] are set to 0xFFFF_FFFF_FFFF_FFFF if src1 is greater than src2, and are set to 0x0000_0000_0000_0000 otherwise.

The contents of doubleword 1 of VSR[VRT] are set to 0x0000_0000_0000_0000.

If a trap-enabled Invalid Operation occurs, VSR[VRT+32] is not modified.

Special Registers Altered:

FX VXSNAN VXVC

VSX Scalar Compare Ordered Double-Precision XX3-form

xscmpodp BF,XA,XB

60	BF	//	A	B	43	AX	BX	/
0	6	9	11	16	21	29	30	31

```
XA          ← AX || A
XB          ← BX || B
reset_xflags()
src1        ← VSR[XA]{0:63}
src2        ← VSR[XB]{0:63}

if( IsSNaN(src1) | IsSNaN(src2) ) then do
   vxsnan_flag ← 0b1
   if(VE=0) then vxvc_flag ← 0b1
end
else if( IsQNaN(src1) | IsQNaN(src2) ) then vxvc_flag = 0b1

FL          ← CompareLTDP(src1,src2)
FG          ← CompareGTDP(src1,src2)
FE          ← CompareEQDP(src1,src2)
FU          ← IsNAN(src1) | IsNAN(src2)
CR[BF]      ← FL || FG || FE || FU
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxvc_flag)   then SetFX(VXVC)
```

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is compared to src2.

Zeros of same or opposite signs compare equal.

Infinities of same signs compare equal.

See Table 54, “Actions for xscmpodp - Part 1: Compare Ordered,” on page 528.

The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, VXSNAN is set, and Invalid Operation is disabled (VE=0), VXVC is set. If neither operand is a Signaling NaN but at least one operand is a Quiet NaN, VXVC is set.

See Table 55, “Actions for xscmpodp - Part 2: Result,” on page 528.

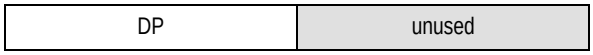
Special Registers Altered

CR field BF

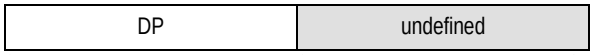
FPCC FX VXSNAN VXVC

VSR Data Layout for xscmpodp

src1 = VSR[XA]



src2 = VSR[XB]



0 64 127

Programming Note

This instruction can be used to operate on single-precision source operands.

src1 \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	cc←0b0010	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
–NZF	cc←0b0100	cc←C(src1,src2)	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b1000	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
–Zero	cc←0b0100	cc←0b0100	cc←0b0010	cc←0b0010	cc←0b1000	cc←0b1000	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
+Zero	cc←0b0100	cc←0b0100	cc←0b0010	cc←0b0010	cc←0b1000	cc←0b1000	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
+NZF	cc←0b0100	cc←0b0100	cc←0b0100	cc←0b0100	cc←C(src1,src2)	cc←0b1000	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
+Infinity	cc←0b0100	cc←0b0100	cc←0b0100	cc←0b0100	cc←0b0100	cc←0b0010	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
QNaN	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxvc_flag←1	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)
SNaN	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)	cc←0b0001 vxsnan_flag←1 vxvc_flag←(VE=0)

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
NZF	Nonzero finite number.
C(x,y)	The floating-point value x is compared to the floating-point value y, returning one of three 4-bit results. 0b1000 when x is greater than y 0b0100 when x is less than y 0b0010 when x is equal to y
cc	The 4-bit result compare code.

Table 54.Actions for xscmpodp - Part 1: Compare Ordered

VE	vxsnan_flag	vxvc_flag	Returned Results and Status Setting
–	0	0	FPCC←cc, CR[BF]←cc
0	0	1	FPCC←cc, CR[BF]←cc, fx(VXVC)
0	1	0	FPCC←cc, CR[BF]←cc, fx(VXSNAN)
0	1	1	FPCC←cc, CR[BF]←cc, fx(VXSNAN), fx(VXVC)
1	0	1	FPCC←cc, CR[BF]←cc, fx(VXVC), error()
1	1	–	FPCC←cc, CR[BF]←cc, fx(VXSNAN), error()

Explanation:

–	The results do not depend on this condition.
cc	The 4-bit result as defined in Table 54.
fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
FX	Floating-Point Summary Exception status flag, $FPSCR_{FX}$.
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$. See Section 7.4.1.
VXC	Floating-Point Invalid Operation Exception (Invalid Compare) status flag, $FPSCR_{VXVC}$. See Section 7.4.1.

Table 55.Actions for xscmpodp - Part 2: Result

VSX Scalar Compare Ordered Quad-Precision X-form

xscmpoqp BF,VRA,VRB

63	BF	//	VRA	VRB	132	/
0	6	9	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1 ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2 ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if( src1.class.SNaN | src2.class.SNaN ) then do
   vxsnan_flag ← 0b1
   if(FPSCR.VE=0) then vxvc_flag ← 0b1
end
else if( src1.class.QNaN | src2.class.QNaN ) then vxvc_flag ← 0b1

cc.bit[0] ← bfp_COMPARE_LT(src1,src2)
cc.bit[1] ← bfp_COMPARE_GT(src1,src2)
cc.bit[2] ← bfp_COMPARE_EQ(src1,src2)
cc.bit[3] ← src1.class.SNaN | src1.class.QNaN |
            src2.class.SNaN | src2.class.QNaN

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(vxvc_flag)   then SetFX(FPSCR.VXVC)

FPSCR.FPCC   ← cc
CR.field[BF] ← cc
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

src1 is compared to src2.

Zeros of same or opposite signs compare equal. Infinities of same signs compare equal.

Bit 0 of CR field BF and FL are set to indicate if src1 is less than src2.

Bit 1 of CR field BF and FG are set to indicate if src1 is greater than src2.

Bit 2 of CR field BF and FE are set to indicate if src1 is equal to src2.

Bit 3 of CR field BF and FU are set to indicate unordered (i.e., src1 or src2 is a NaN).

If either of the operands is a NaN, either quiet or signaling, CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, an Invalid Operation exception occurs and VXSNAN is set, and if Invalid Operation exceptions are disabled (VE=0), VXVC is set. If neither operand is a Signaling NaN but at least one operand is a Quiet NaN, an Invalid Operation exception occurs and VXVC is set.

Special Registers Altered:

CR field BF
FPCC FX VXSNAN VXVC

VSR Data Layout for xscmpoqp

VSR[VRA+32]

src1

VSR[VRB+32]

src2

VSX Scalar Compare Unordered Double-Precision XX3-form

xscmpudp BF,XA,XB

60	BF	//	A	B	35	AX	BX	/
0	6	9	11	16	21	29	30	31

```
XA     ← AX || A
XB     ← BX || B

reset_xflags()

src1   ← VSR[XA]{0:63}
src2   ← VSR[XB]{0:63}

if( IsSNaN(src1) | IsSNaN(src2) ) then vxsnan_flag ← 1

FL     ← CompareLTDP(src1,src2)
FG     ← CompareGTDP(src1,src2)
FE     ← CompareEQDP(src1,src2)
FU     ← IsNAN(src1) | IsNAN(src2)
CR[BF] ← FL || FG || FE || FU
if(vxsnan_flag) then SetFX(VXSNAN)
```

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is compared to src2.

Zeros of same or opposite signs compare equal equal.

Infinities of same signs compare equal.

See Table 56, “Actions for xscmpudp - Part 1: Compare Unordered,” on page 531.

The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, VXSNAN is set.

See Table 57, “Actions for xscmpudp - Part 2: Result,” on page 531.

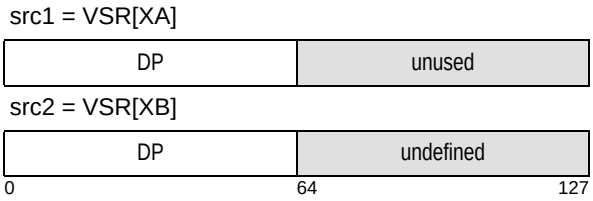
Special Registers Altered

CR[BF]
FPCC FX VXSNAN

Programming Note

This instruction can be used to operate on single-precision source operands.

VSR Data Layout for xscmpudp



src1 \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	cc = 0b0010	cc = 0b1000	cc = 0b1000	cc = 0b1000	cc = 0b1000	cc = 0b1000	cc = 0b0001	cc = 0b0001 vxsnan_flag = 1
–NZF	cc = 0b0100	cc = C(src1,src2)	cc = 0b1000	cc = 0b1000	cc = 0b1000	cc = 0b1000	cc = 0b0001	cc = 0b0001 vxsnan_flag = 1
–Zero	cc = 0b0100	cc = 0b0100	cc = 0b0010	cc = 0b0010	cc = 0b1000	cc = 0b1000	cc = 0b0001	cc = 0b0001 vxsnan_flag = 1
+Zero	cc = 0b0100	cc = 0b0100	cc = 0b0010	cc = 0b0010	cc = 0b1000	cc = 0b1000	cc = 0b0001	cc = 0b0001 vxsnan_flag = 1
+NZF	cc = 0b0100	cc = 0b0100	cc = 0b0100	cc = 0b0100	cc = C(src1,src2)	cc = 0b1000	cc = 0b0001	cc = 0b0001 vxsnan_flag = 1
+Infinity	cc = 0b0100	cc = 0b0100	cc = 0b0100	cc = 0b0100	cc = 0b0100	cc = 0b0010	cc = 0b0001	cc = 0b0001 vxsnan_flag = 1
QNaN	cc = 0b0001	cc = 0b0001	cc = 0b0001	cc = 0b0001	cc = 0b0001	cc = 0b0001	cc = 0b0001	cc = 0b0001 vxsnan_flag = 1
SNaN	cc = 0b0001 vxsnan_flag = 1	cc = 0b0001 vxsnan_flag = 1	cc = 0b0001 vxsnan_flag = 1	cc = 0b0001 vxsnan_flag = 1	cc = 0b0001 vxsnan_flag = 1	cc = 0b0001 vxsnan_flag = 1	cc = 0b0001 vxsnan_flag = 1	cc = 0b0001 vxsnan_flag = 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
NZF	Nonzero finite number.
C(x,y)	The floating-point value x is compared to the floating-point value y, returning one of three 4-bit results. 0b1000 when x is greater than y 0b0100 when x is less than y 0b0010 when x is equal to y
cc	The 4-bit result compare code.

Table 56.Actions for xscmpudp - Part 1: Compare Unordered

VE	vxsnan_flag	Returned Results and Status Setting
–	0	FPCC←cc, CR[BF]←cc
0	1	FPCC←cc, CR[BF]←cc, fx(VXSNAN)
1	1	FPCC←cc, CR[BF]←cc, fx(VXSNAN), error()

Explanation:

–	The results do not depend on this condition.
cc	The 4-bit result as defined in Table 56.
fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
FX	Floating-Point Summary Exception status flag, $FPSCR_{FX}$.
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$. See Section 7.4.1.

Table 57.Actions for xscmpudp - Part 2: Result

VSX Scalar Compare Unordered Quad-Precision X-form

xscmpuqp BF,VRA,VRB

63	BF	//	VRA	VRB	644	/
0	6	9	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1       ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2       ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

vxsnan_flag ← src1.class.SNaN | src2.class.SNaN

cc.bit[0]   ← bfp_COMPARE_LT(src1,src2)
cc.bit[1]   ← bfp_COMPARE_GT(src1,src2)
cc.bit[2]   ← bfp_COMPARE_EQ(src1,src2)
cc.bit[3]   ← src1.class.SNaN | src1.class.QNaN |
              src2.class.SNaN | src2.class.QNaN

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)

FPSCR.FPCC   ← cc
CR.field[BF] ← cc
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

src1 is compared to src2.

Zeros of same or opposite signs compare equal. Infinities of same signs compare equal.

Bit 0 of CR field BF and FL are set to indicate if src1 is less than src2.

Bit 1 of CR field BF and FG are set to indicate if src1 is greater than src2.

Bit 2 of CR field BF and FE are set to indicate if src1 is equal to src2.

Bit 3 of CR field BF and FU are set to indicate unordered (i.e., src1 or src2 is a NaN).

If either of the operands is a Signaling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

Special Registers Altered:

CR field BF
FPCC FX VXSNAN

VSR Data Layout for xscmpuqp

VSR[VRA+32]

src1

VSR[VRB+32]

src2

VSX Scalar Copy Sign Double-Precision XX3-form

xscpsgndp XT,XA,XB

60	T	A	B	176	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
result{0:63} ← VSR[XA]{0} || VSR[XB]{1:63}
VSR[XT]      ← result || 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Bit 0 of VSR[XT] is set to the contents of bit 0 of VSR[XA].

Bits 1:63 of VSR[XT] are set to the contents of bits 1:63 of VSR[XB].

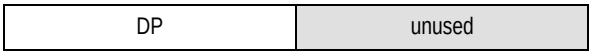
The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

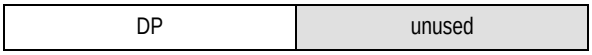
None

VSR Data Layout for xscpsgndp

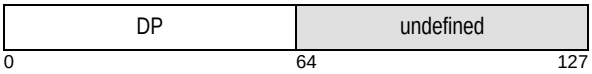
src1 = VSR[XA]



src2 = VSR[XB]



tgt = VSR[XT]



Programming Note

This instruction can be used to operate on single-precision source operands.

VSX Scalar Copy Sign Quad-Precision X-form

xscpsgnqp VRT,VRA,VRB

63	VRT	VRA	VRB	100	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1        ← VSR[VRA+32] & 0x8000_0000_0000_0000_0000_0000_0000_0000
src2        ← VSR[VRB+32] & 0x7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF

VSR[VRT+32] ← src1 | src2
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

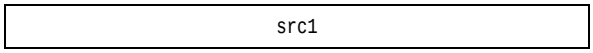
src2 is placed into VSR[VRT+32] with the sign of src1.

Special Registers Altered:

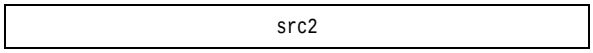
None

VSR Data Layout for xscpsgnqp

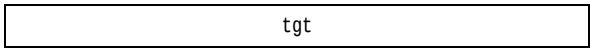
VSR[VRA+32]



VSR[VRB+32]



VSR[VRT+32]



VSX Scalar Convert with round Double-Precision to Half-Precision format XX2-form

xscvdphp XT,XB

60	T	17	B	347	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_flags()

src    ← bfp_CONVERT_FROM_BFP64(VSR[BX×32+B].dword[0])
rnd    ← bfp_ROUND_TO_BFP16(FPSCR.RN,src)
result ← bfp_CONVERT_TO_BFP16(rnd)

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(ox_flag) then SetFX(FPSCR.OX)
if(ux_flag) then SetFX(FPSCR.UX)
if(xx_flag) then SetFX(FPSCR.XX)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[TX×32+T].hword[0:2] ← 0x0000_0000_0000
   VSR[TX×32+T].hword[3]   ← result
   VSR[TX×32+T].dword[1]   ← 0xUUUU_UUUU_UUUU_UUUU
   FPSCR.FPRF              ← fprf_CLASS_BFP16(result)
end
FPSCR.FR ← (vex_flag=0) & inc_flag
FPSCR.FI ← (vex_flag=0) & xx_flag
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is an SNaN, the result is the half-precision representation of that SNaN converted to a QNaN.

Otherwise, if src is a QNaN, the result is the half-precision representation of that QNaN.

Otherwise, if src is an Infinity, the result is the half-precision representation of Infinity with the same sign as src.

Otherwise, if src is a Zero, the result is the half-precision representation of Zero with the same sign as src.

Otherwise, the result is the half-precision representation of src rounded to half-precision using the rounding mode specified by RN.

The result is zero-extended and placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in half-precision. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

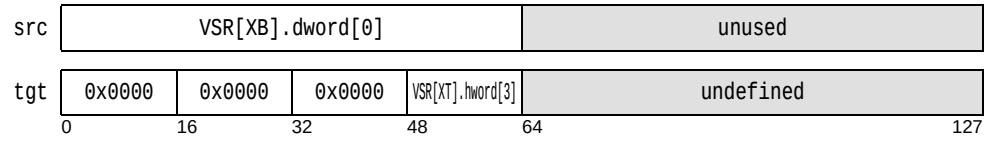
Special Registers Altered:

FPRF FR FI
FX VXSNAN OX UX XX

Programming Note

This instruction can be used to operate on a single-precision source operand.

VSR Data Layout for xscvdphp



VSX Scalar Convert Double-Precision to Quad-Precision format X-form

xscvdpqp VRT,VRB

63	VRT	22	VRB	836	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src ← bfp_CONVERT_FROM_BFP64(VSR[VRB+32].dword[0])

if src.class.SNaN then
   result ← bfp_CONVERT_TO_BFP128(bfp_QUIET(src))
else
   result ← bfp_CONVERT_TO_BFP128(src)

vxsnan_flag ← src.class.SNaN
if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← 0
FPSCR.FI ← 0
```

Let src be the floating-point value in doubleword element 0 of VSR[VRB+32] represented in double-precision format.

src is placed into VSR[VRT+32] in quad-precision format.

If src is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

FPRF is set to the class and sign of the result.

FR is set to 0. FI is set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[XT] and FPRF are not modified.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX VXSNAN

VSR Data Layout for xscvdpqp

VSR[VRB+32]

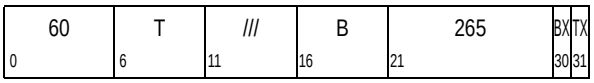
src.dword[0]	unused

VSR[VRT+32]

tgt

VSX Scalar Convert with round Double-Precision to Single-Precision format XX2-form

xscvdpsp XT,XB




```
reset_xflags()
src ← VSR[32×BX+B].dword[0]
result ← ConvertDPtoSP(src)
if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(xx_flag) then SetFX(FPSCR.XX)
if(ox_flag) then SetFX(FPSCR.OX)
if(ux_flag) then SetFX(FPSCR.UX)
vex_flag ← FPSCR.VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[32×TX+T].word[0] ← result
   VSR[32×TX+T].word[1] ← 0xUUUU_UUUU
   VSR[32×TX+T].word[2] ← 0xUUUU_UUUU
   VSR[32×TX+T].word[3] ← 0xUUUU_UUUU
   FPSCR.FPRF ← ClassSP(result)
   FPSCR.FR ← inc_flag
   FPSCR.FI ← xx_flag
end
else do
   FPSCR.FR ← 0b0
   FPSCR.FI ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is a SNaN, the result is src converted to a QNaN (i.e., bit 12 of src is set to 1). VXSNAN is set to 1.

Otherwise, if src is a QNaN, an Infinity, or a Zero, the result is src.

Otherwise, the result is src rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into word element 0 of VSR[XT] in single-precision format.

The contents of word elements 1, 2, and 3 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

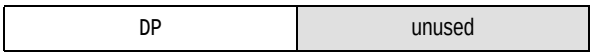
See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered

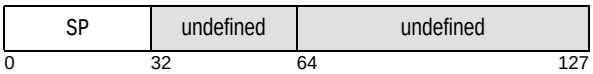
FPRF FR FI FX OX UX XX VXSNAN

VSR Data Layout for xscvdpsp

src = VSR[XB]



tgt = VSR[XT]

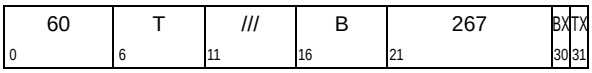


Programming Note

This instruction can be used to operate on a single-precision source operand.

VSX Scalar Convert Scalar Single-Precision to Vector Single-Precision format Non-signalling XX2-form

xscvdpspn XT,XB




```
reset_xflags()
src ← VSR[32×BX+B].dword[0]
result ← ConvertDPtoSP_NS(src)
VSR[32×TX+T].word[0] ← result
VSR[32×TX+T].word[1] ← 0xUUUU_UUUU
VSR[32×TX+T].word[2] ← 0xUUUU_UUUU
VSR[32×TX+T].word[3] ← 0xUUUU_UUUU
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the single-precision floating-point value in doubleword element 0 of VSR[XB] represented in double-precision format.

src is placed into word element 0 of VSR[XT] in single-precision format.

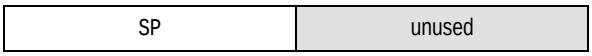
The contents of word elements 1, 2, and 3 of VSR[XT] are undefined.

Special Registers Altered

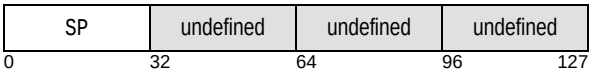
None

VSR Data Layout for xscvdpspn

src = VSR[XB]



tgt = VSR[XT]



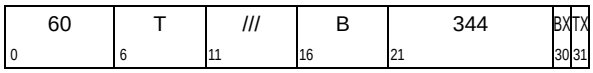
Programming Note

xscvdpsp should be used to convert a scalar double-precision value to vector single-precision format.

xscvdpspn should be used to convert a scalar single-precision value to vector single-precision format.

VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format XX2-form

xscvdpsxds XT,XB




```
XT          ← TX || T
XB          ← BX || B
reset_xflags()
result{0:63} ← ConvertDPtoSD(VSR[XB]{0:63})
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxcvi_flag)  then SetFX(VXCVI)
if(xx_flag)     then SetFX(XX)
vex_flag    ← VE & (vxsnan_flag | vxcvi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← 0bUUUUU
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is a NaN, the result is the value 0x8000_0000_0000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than $2^{63}-1$, the result is 0x7FFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{63}, the result is 0x8000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

If a trap-enabled invalid operation exception occurs,
- VSR[XT] and FPRF are not modified
- FR and FI are set to 0.

Otherwise,

- The result is placed into doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.
- FPRF is set to an undefined value.
- FR is set to indicate if the result was incremented when rounded.
- FI is set to indicate the result is inexact.

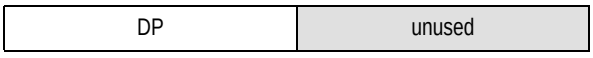
See Table 58.

Special Registers Altered

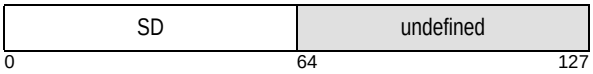
FPRF=0bUUUUU FR FI FX XX VXSNAN VXCVI

VSR Data Layout for xscvdpsxds

src = VSR[XB]



tgt = VSR[XT]



Programming Note

This instruction can be used to operate on a single-precision source operand.

Programming Note

xscvdpsxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xsrdpic*** which uses the rounding mode specified by RN.

	VE	XE	Inexact? (RoundToDPintegerTrunc(src) ≠ src)	Returned Results and Status Setting
src [Nmin-1	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), FR←0, FI←1, fx(XX)
		1	yes	T(Nmin), FR←0, FI←1, fx(XX), error()
src = Nmin	–	–	no	T(Nmin), FR←0, FI←0
Nmin < src < Nmax	–	–	no	T(ConvertDPtoSD(RoundToDPintegerTrunc(src))), FR←0, FI←0
		0	yes	T(ConvertDPtoSD(RoundToDPintegerTrunc(src))), FR←0, FI←1, fx(XX)
		1	yes	T(ConvertDPtoSD(RoundToDPintegerTrunc(src))), FR←0, FI←1, fx(XX), error()
src = Nmax	–	–	no	T(Nmax), FR←0, FI←0 Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), FR←0, FI←1, fx(XX)
		1	yes	T(Nmax), FR←0, FI←1, fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI), fx(VXSNAN)
	1	–	–	FR←0, FI←0, fx(VXCVI), fx(VXSNAN), error()

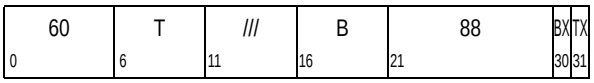
Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
Nmin	The smallest signed integer doubleword value, -2^{63} (0x8000_0000_0000_0000).
Nmax	The largest signed integer doubleword value, $2^{63}-1$ (0x7FFF_FFFF_FFFF_FFFF).
src	The double-precision floating-point value in doubleword element 0 of VSR[XB].
T(x)	The signed integer doubleword value x is placed in doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.

Table 58.Actions for xscvdpsxds

VSX Scalar Convert with round to zero Double-Precision to Signed Word format XX2-form

xscvdpsxws XT,XB




```
XT          ← TX || T
XB          ← BX || B
inc_flag    ← 0b0
reset_xflags()
result{0:31} ← ConvertDPtoSW(VSR[XB]{0:63})
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxcvi_flag)  then SetFX(VXCVI)
if(xx_flag)     then SetFX(XX)
vex_flag    ← VE & (vxsnan_flag | vxcvi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← 0xUUUU_UUUU || result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← 0bUUUUU
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is a NaN, the result is the value 0x8000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than $2^{31}-1$, the result is 0x7FFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{31}, the result is 0x8000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit signed-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

If a trap-enabled invalid operation exception occurs,

– VSR[XT] and FPRF are not modified

– FR and FI are set to 0.

Otherwise,

– The result is placed into word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

– FPRF is set to an undefined value.

– FR is set to indicate if the result was incremented when rounded.

– FI is set to indicate the result is inexact.

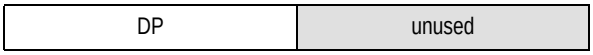
See Table 59.

Special Registers Altered

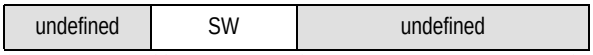
FPRF=0bUUUUU FR FI FX XX VXSNAN VXCVI

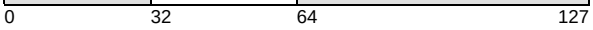
VSR Data Layout for xscvdpsxws

src = VSR[XB]



tgt = VSR[XT]





Programming Note

This instruction can be used to operate on a single-precision source operand.

Programming Note

xscvdpsxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xsrdpic*** which uses the rounding mode specified by RN.

	VE	XE	Inexact? (RoundToDPintegerTrunc(src) ≠ src)	Returned Results and Status Setting
src [Nmin-1	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), FR←0, FI←1, fx(XX)
		1	yes	T(Nmin), FR←0, FI←1, fx(XX), error()
src = Nmin	–	–	no	T(Nmin), FR←0, FI←0
Nmin < src < Nmax	–	–	no	T(ConvertDPtoSW(RoundToDPintegerTrunc(src))), FR←0, FI←0
		0	yes	T(ConvertDPtoSW(RoundToDPintegerTrunc(src))), FR←0, FI←1, fx(XX)
		1	yes	T(ConvertDPtoSW(RoundToDPintegerTrunc(src))), FR←0, FI←1, fx(XX), error()
src = Nmax	–	–	no	T(Nmax), FR←0, FI←0
Nmax < src < Nmax+1	–	0	yes	T(Nmax), FR←0, FI←1, fx(XX)
		1	yes	T(Nmax), FR←0, FI←1, fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI), fx(VXSNAN)
	1	–	–	FR←0, FI←0, fx(VXCVI), fx(VXSNAN), error()

Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
Nmin	The smallest signed integer word value, -2^{31} (0x8000_0000).
Nmax	The largest signed integer word value, $2^{31}-1$ (0x7FFF_FFFF).
src	The double-precision floating-point value in doubleword element 0 of VSR[XB].
T(x)	The signed integer word value x is placed in word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

Table 59.Actions for xscvdpsxws

VSX Scalar Convert with round to zero Double-Precision to Unsigned Doubleword format XX2-form

xscvdpuxds XT,XB

60	T	///	B	328	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
inc_flag    ← 0b0
reset_xflags()
result{0:63} ← ConvertDPtoUD(VSR[XB]{0:63})
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxcvi_flag)  then SetFX(VXCVI)
if(xx_flag)     then SetFX(XX)
vex_flag    ← VE & (vxsnan_flag | vxcvi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← 0bUUUUU
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is a NaN, the result is the value 0x0000_0000_0000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{64}-1, the result is 0xFFFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is 0x0000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

If a trap-enabled invalid operation exception occurs,

– VSR[XT] and FPRF are not modified

– FR and FI are set to 0.

Otherwise,

– The result is placed into doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.

– FPRF is set to an undefined value.

– FR is set to indicate if the result was incremented when rounded.

– FI is set to indicate the result is inexact.

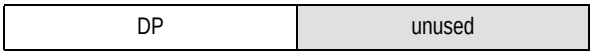
See Table 60.

Special Registers Altered

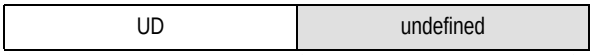
FPRF=0bUUUUU FR FI FX XX VXSNAN VXCVI

VSR Data Layout for xscvdpuxds

src = VSR[XB]



tgt = VSR[XT]



0 64 127

Programming Note

This instruction can be used to operate on a single-precision source operand.

Programming Note

xscvdpuxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xsrdpic*** which uses the rounding mode specified by RN.

	VE	XE	Inexact? (RoundToDPintegerTrunc(src) ≠ src)	Returned Results and Status Setting
src [Nmin-1	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), FR←0, FI←1, fx(XX)
		1	yes	T(Nmin), FR←0, FI←1, fx(XX), error()
src = Nmin	–	–	no	T(Nmin), FR←0, FI←0
Nmin < src < Nmax	–	–	no	T(ConvertDPtoUD(RoundToDPintegerTrunc(src))), FR←0, FI←0
		0	yes	T(ConvertDPtoUD(RoundToDPintegerTrunc(src))), FR←0, FI←1, fx(XX)
		1	yes	T(ConvertDPtoUD(RoundToDPintegerTrunc(src))), FR←0, FI←1, fx(XX), error()
src = Nmax	–	–	no	T(Nmax), FR←0, FI←0 Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), FR←0, FI←1, fx(XX)
		1	yes	T(Nmax), FR←0, FI←1, fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI), fx(VXSNAN)
	1	–	–	FR←0, FI←0, fx(VXCVI), fx(VXSNAN), error()

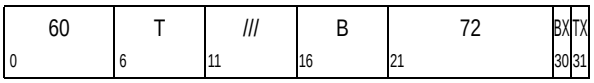
Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
Nmin	The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000).
Nmax	The largest unsigned integer doubleword value, 2^{64}-1 (0xFFFF_FFFF_FFFF_FFFF).
src	The double-precision floating-point value in doubleword element 0 of VSR[XB].
T(x)	The unsigned integer doubleword value x is placed in doubleword element 0 of VSR[XT]. The contents of doubleword element 1 of VSR[XT] are undefined.

Table 60.Actions for xscvdpuxds

VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format XX2-form

xscvdpuxws XT,XB




```
XT          ← TX || T
XB          ← BX || B
inc_flag    ← 0b0
reset_xflags()
result{0:31} ← ConvertDPtoUW(VSR[XB]{0:63})
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxcvi_flag)  then SetFX(VXCVI)
if(xx_flag)     then SetFX(XX)
vex_flag    ← VE & (vxsnan_flag | vxcvi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← 0xUUUU_UUUU || result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← 0bUUUUU
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src is a NaN, the result is the value 0x0000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than $2^{32}-1$, the result is 0xFFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is 0x0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

If a trap-enabled invalid operation exception occurs,

– VSR[XT] and FPRF are not modified

– FR and FI are set to 0.

Otherwise,

– The result is placed into word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

– FPRF is set to an undefined value.

– FR is set to indicate if the result was incremented when rounded.

– FI is set to indicate the result is inexact.

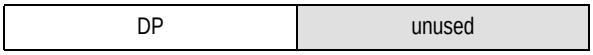
See Table 61.

Special Registers Altered

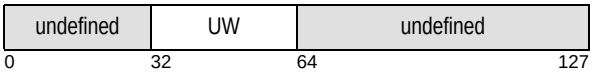
FPRF=0bUUUUU FR FI FX XX VXSNAN VXCVI

VSR Data Layout for xscvdpuxws

src = VSR[XB]



tgt = VSR[XT]



Programming Note

This instruction can be used to operate on a single-precision source operand.

Programming Note

xscvdpuxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xsrdpic*** which uses the rounding mode specified by RN.

	VE	XE	Inexact? (RoundToDPintegerTrunc(src) g src)	Returned Results and Status Setting
src [Nmin-1	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), FR←0, FI←1, fx(XX)
		1	yes	T(Nmin), FR←0, FI←1, fx(XX), error()
src = Nmin	–	–	no	T(Nmin), FR←0, FI←0
Nmin < src < Nmax	–	–	no	T(ConvertDPtoUW(RoundToDPintegerTrunc(src))), FR←0, FI←0
		0	yes	T(ConvertDPtoUW(RoundToDPintegerTrunc(src))), FR←0, FI←1, fx(XX)
		1	yes	T(ConvertDPtoUW(RoundToDPintegerTrunc(src))), FR←0, FI←1, fx(XX), error()
src = Nmax	–	–	no	T(Nmax), FR←0, FI←0
Nmax < src < Nmax+1	–	0	yes	T(Nmax), FR←0, FI←1, fx(XX)
		1	yes	T(Nmax), FR←0, FI←1, fx(XX), error()
src m Nmax+1	0	–	–	T(Nmax), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI)
	1	–	–	FR←0, FI←0, fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), FR←0, FI←0, fx(VXCVI), fx(VXSNAN)
	1	–	–	FR←0, FI←0, fx(VXCVI), fx(VXSNAN), error()

Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
Nmin	The smallest unsigned integer word value, 0 (0x0000_0000).
Nmax	The largest unsigned integer word value, 2^{32}-1 (0xFFFF_FFFF).
src	The double-precision floating-point value in doubleword element 0 of VSR[XB].
T(x)	The unsigned integer word value x is placed in word element 1 of VSR[XT]. The contents of word elements 0, 2, and 3 of VSR[XT] are undefined.

Table 61.Actions for xscvdpuxws

VSX Scalar Convert Half-Precision to Double-Precision format XX2-form

xscvhpdp XT,XB

60	T	16	B	347	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_flags()

src ← bfp_CONVERT_FROM_BFP16(VSR[BX×32+B].hword[3])

if src.class.SNaN=1 then
   result ← bfp_CONVERT_TO_BFP64(bfp_QUIET(src))
else
   result ← bfp_CONVERT_TO_BFP64(src)

vxsnan_flag ← src.class.SNaN
if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[TX×32+T].dword[0] ← result
   VSR[TX×32+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPSCR.FPRF            ← fprf_CLASS_BFP64(result)
end
FPSCR.FR ← 0
FPSCR.FI ← 0
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the half-precision floating-point value in the rightmost halfword of doubleword element 0 of VSR[XB].

If src is an SNaN, the result is the double-precision representation of that SNaN converted to a QNaN.

Otherwise, if src is a QNaN, the result is the double-precision representation of that QNaN.

Otherwise, if src is an Infinity, the result is the double-precision representation of Infinity with the same sign as src.

Otherwise, if src is a Zero, the result is the double-precision representation of Zero with the same sign as src.

Otherwise, if src is a denormal value, the result is the normalized double-precision representation of src.

Otherwise, the result is the double-precision representation of src.

The result is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in half-precision.

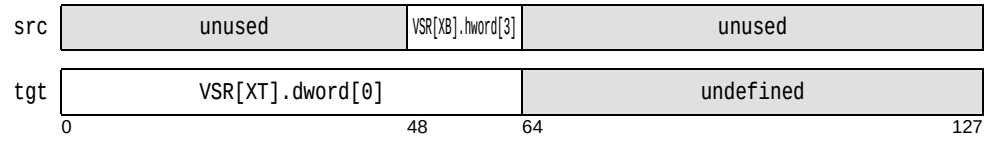
If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified.

FR is set to 0. FI is set to 0.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)
FX VXSNAN

VSR Data Layout for xscvhpdp



VSX Scalar Convert with round Quad-Precision to Double-Precision format [using round to Odd] X-form

xscvqpdp VRT,VRB (RO=0)
xscvqpdpo VRT,VRB (RO=1)

63	VRT	20	VRB	836	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
rnd    ← bfp_ROUND_TO_BFP64(RO,FPSCR.RN,src)
result ← bfp_CONVERT_TO_BFP64(rnd)

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(ox_flag)     then SetFX(FPSCR.OX)
if(ux_flag)     then SetFX(FPSCR.UX)
if(xx_flag)     then SetFX(FPSCR.XX)

vex_flag ← FPSCR.VE & vxsnan_flag

if vex_flag=0 then do
   VSR[VRT+32].dword[0] ← result
   VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF           ← fprf_CLASS_BFP64(result)
end
FPSCR.FR ← (vxsnan_flag=0) & inc_flag
FPSCR.FI ← (vxsnan_flag=0) & xx_flag
```

Let src be the quad-precision floating-point value in VSR[VRB+32].

If src is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src is a Signalling NaN, the result is the Quiet NaN corresponding to the Signalling NaN, with the significand truncated to the rounding precision.

Otherwise, if src is a Quiet NaN, then the result is src with the significand truncated to double-precision.

Otherwise, if src is an Infinity or a Zero, the result is src.

Otherwise, do the following.

If src is *Tiny* (i.e., the unbiased exponent is less than -1022) and UE=0, the significand is shifted right N bits, where N is the difference between -1022 and the unbiased exponent of src. The exponent of src is set to the value -1022.

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to double-precision (i.e., 11-bit exponent range and 53-bit significand precision) using the specified rounding mode.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element 0 of VSR[VRT+32] in double-precision format. The contents of doubleword element 1 of VSR[VRT+32] are set to 0.

FPRF is set to the class and sign of the result as represented in double-precision format. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered:

FPRF FR FI
FX VXSNAN OX UX XX

VSR Data Layout for xscvqpdp[o]

VSR[VRB+32]

src

VSR[VRT+32]

tgt.dword[0]	0x0000_0000_0000_0000

VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format X-form

xscvqpsdz VRT,VRB

63	VRT	25	VRB	836	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if src.class.QNaN | src.class.SNaN then do
   result ← 0x8000_0000_0000_0000
   vxsnan_flag ← src.class.SNaN
   vxcvi_flag ← 1
end
else if src.class.Infinity then do
   vxcvi_flag ← 1
   if src.sign = 0 then
      result ← 0x7FFF_FFFF_FFFF_FFFF
   else
      result ← 0x8000_0000_0000_0000
end
else if src.class.Zero then
   result ← 0x0000_0000_0000_0000
else do
   rnd ← bfp_ROUND_TO_INTEGER(0b001,src)
   if bfp_COMPARE_GT(rnd, +2^63-1) then do
      result ← 0x7FFF_FFFF_FFFF_FFFF
      vxcvi_flag ← 1
   end
   else if bfp_COMPARE_LT(rnd, -2^63) then do
      result ← 0x8000_0000_0000_0000
      vxcvi_flag ← 1
   end
   else do
      result ← bfp_CONVERT_TO_SI64(rnd)
      if(xx_flag) then SetFX(FPSCR.XX)
   end
end

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(vxcvi_flag)  then SetFX(FPSCR.VXCVI)

vx_flag ← vxsnan_flag | vxcvi_flag
ex_flag ← FPSCR.VE & vx_flag

if ex_flag=0 then do
   VSR[VRT+32].dword[0] ← result
   VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src be the quad-precision floating-point value in VSR[VRB+32].

If src is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN and VXCVI are set to 1.

If src is a Quiet NaN or an Infinity, an Invalid Operation exception occurs and VXCVI is set to 1.

If src is a NaN, the result is 0x8000_0000_0000_0000.

Otherwise, if src is a Zero, the result is 0x0000_0000_0000_0000.

Otherwise, if src is +Infinity, the result is 0x7FFF_FFFF_FFFF_FFFF.

Otherwise, if src is -Infinity, the result is 0x8000_0000_0000_0000.

Otherwise, do the following.

Let rnd be the value src truncated to a floating-point integer.

If rnd is greater than $+2^{63}-1$, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 0x7FFF_FFFF_FFFF_FFFF.

Otherwise, if rnd is less than -2^{63}, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 0x8000_0000_0000_0000.

Otherwise, the result is the value rnd, and an Inexact exception occurs if rnd is inexact (i.e., rnd is not equal to src).

The result is placed into doubleword element 0 of VSR[VRT+32] in signed integer format.

The contents of doubleword element 1 of VSR[VRT+32] are set to 0.

FPRF is set to undefined. FR is set to 0. FI is set to indicate if the rounded result is inexact.

If an Invalid Operation exception occurs, FR and FI are set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified.

See Table 58, “Actions for xscvdpsxds,” on page 539.

Special Registers Altered:

FPRF (undefined) FR FI FX VXSNAN VXCVI XX

VSR Data Layout for xscvqpsdz

VSR[VRB+32]

src

VSR[VRT+32]

tgt.dword[0]	0x0000_0000_0000_0000

	FPSCR.VE	FPSCR.XE	bfp_ROUND_TO_INTEGER(0b0001,src) ≠ src	Returned Results and Status Setting
src ≤ Nmin-1	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
	–	1	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmin	–	–	no	T(Nmin), fr(0), fi(0), fprf(0bUUUUU)
Nmin < src < Nmax	–	–	no	T(bfp_CONVERT_TO_SI64(trunc(src))), fr(0), fi(0), fprf(0bUUUUU)
	–	0	yes	T(bfp_CONVERT_TO_SI64(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
	–	1	yes	T(bfp_CONVERT_TO_SI64(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmax	–	–	no	T(Nmax), fr(0), fi(0), fprf(0bUUUUU)
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
	–	1	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNAN)
	1	–	–	fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()

Explanation:

T(x)	Places the value x into the target VSR. VSR[VRT+32].dword[0] ← x VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
Nmin	The smallest signed integer doubleword value, -2^{63} (0x8000_0000_0000_0000).
Nmax	The largest signed integer doubleword value, $2^{63}-1$ (0x7FFF_FFFF_FFFF_FFFF).
src	The quad-precision floating-point value in VSR[VRB+32].
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
fi(x)	FPSCR.FI is set to the value x.
fr(x)	FPSCR.FR is set to the value x.
fprf(x)	FPSCR.FPRF is set to the value x.
error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
trunc(x)	Return the floating-point value x truncated to a floating-point integer.

Table 62. Actions for xscvqpsdz

VSX Scalar Convert with round to zero Quad-Precision to Signed Word format X-form

xscvqpswz VRT,VRB

63	VRT	9	VRB	836	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if src.class.QNaN | src.class.SNaN then do
   result ← 0xFFFF_FFFF_8000_0000
   vxsnan_flag ← src.class.SNaN
   vxcvi_flag ← 1
end
else if src.class.Infinity then do
   vxcvi_flag ← 1
   if src.sign = 0 then
      result ← 0x0000_0000_7FFF_FFFF
   else
      result ← 0xFFFF_FFFF_8000_0000
end
else if src.class.Zero then
   result ← 0x0000_0000_0000_0000
else do
   rnd ← bfp_ROUND_TO_INTEGER(0b001,src)
   if bfp_COMPARE_GT(rnd, +2^31-1) then do
      result ← 0x0000_0000_7FFF_FFFF
      vxcvi_flag ← 1
   end
   else if bfp_COMPARE_LT(rnd, -2^31) then do
      result ← 0xFFFF_FFFF_8000_0000
      vxcvi_flag ← 1
   end
   else do
      result ← bfp_CONVERT_TO_SI64(rnd)
      if(xx_flag) then SetFX(FPSCR.XX)
   end
end

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(vxcvi_flag)  then SetFX(FPSCR.VXCVI)

vx_flag ← vxsnan_flag | vxcvi_flag
ex_flag ← FPSCR.VE & vx_flag

if ex_flag=0 then do
   VSR[VRT+32].dword[0] ← result
   VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← 0bUUUUU
end
FPSCR.FR ← 0
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src be the quad-precision floating-point value in VSR[VRB+32].

If src is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN and VXCVI are set to 1.

If src is a Quiet NaN or an Infinity, an Invalid Operation exception occurs and VXCVI is set to 1.

If src is a NaN, the result is 0xFFFF_FFFF_8000_0000.

Otherwise, if src is a Zero, the result is 0x0000_0000_0000_0000.

Otherwise, if src is a +Infinity, the result is 0x0000_0000_7FFF_FFFF.

Otherwise, if src is a -Infinity, the result is 0xFFFF_FFFF_8000_0000.

Otherwise, do the following.

Let rnd be the value src truncated to a floating-point integer.

If rnd is greater than $+2^{31}-1$, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 0x0000_0000_7FFF_FFFF.

Otherwise, if rnd is less than -2^{31}, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 0xFFFF_FFFF_8000_0000.

Otherwise, the result is the value rnd, and an Inexact exception occurs if rnd is inexact (i.e., rnd is not equal to src).

The result is placed into doubleword element 0 of VSR[VRT+32] in signed integer format.

The contents of doubleword element 1 of VSR[VRT+32] are set to 0.

FPRF is set to undefined. FR is set to 0. FI is set to indicate if the rounded result is inexact.

If an Invalid Operation exception occurs, FR and FI are set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified.

See Table 63, “Actions for xscvqpswz,” on page 551.

Special Registers Altered:

FPRF (undefined) FR (set to 0) FI
FX VXSNAN VXCVI XX

VSR Data Layout for xscvqpswz

VSR[VRB+32]

src

VSR[VRT+32]

tgt.dword[0]	0x0000_0000_0000_0000

	FPSCR.VE	FPSCR.XE	bfp_ROUND_TO_INTEGER(0b0001,src) ≠ src	Returned Results and Status Setting
src ≤ Nmin-1	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
	–	1	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmin	–	–	no	T(Nmin), fr(0), fi(0), fprf(0bUUUUU)
Nmin < src < Nmax	–	–	no	T(bfp_CONVERT_TO_SI64(trunc(src))), fr(0), fi(0), fprf(0bUUUUU)
	–	0	yes	T(bfp_CONVERT_TO_SI64(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
	–	1	yes	T(bfp_CONVERT_TO_SI64(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmax	–	–	no	T(Nmax), fr(0), fi(0), fprf(0bUUUUU)
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
	–	1	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNAN)
	1	–	–	fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()

Explanation:

T(x)	Places the value x into the target VSR. VSR[VRT+32].dword[0] ← x VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
Nmin	The smallest signed integer word value, -2^{31} (0xFFFF_FFFF_8000_0000).
Nmax	The largest signed integer word value, $2^{31}-1$ (0x0000_0000_7FFF_FFFF).
src	The quad-precision floating-point value in VSR[VRB+32].
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
fi(x)	FPSCR.FI is set to the value x.
fr(x)	FPSCR.FR is set to the value x.
fprf(x)	FPSCR.FPRF is set to the value x.
error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
trunc(x)	Return the floating-point value x truncated to a floating-point integer.

Table 63. Actions for xscvqpswz

VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format X-form

xscvqpudz VRT,VRB

63	VRT	17	VRB	836	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if src.class.QNaN | src.class.SNaN then do
   result ← 0x0000_0000_0000_0000
   vxsnan_flag ← src.class.SNaN
   vxcvi_flag ← 1
end
else if src.class.Infinity then do
   vxcvi_flag ← 1
   if src.sign = 0 then
      result ← 0xFFFF_FFFF_FFFF_FFFF
   else
      result ← 0x0000_0000_0000_0000
end
else if src.class.Zero then result ← 0x0000_0000_0000_0000
else do
   rnd ← bfp_ROUND_TO_INTEGER(0b001,src)
   if bfp_COMPARE_GT(rnd, +2^64-1) then do
      result ← 0xFFFF_FFFF_FFFF_FFFF
      vxcvi_flag ← 1
   end
   else if bfp_COMPARE_LT(rnd, 0) then do
      result ← 0x0000_0000_0000_0000
      vxcvi_flag ← 1
   end
   else do
      result ← bfp_CONVERT_TO_UI64(rnd)
      if(xx_flag) then SetFX(FPSCR.XX)
   end
end

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(vxcvi_flag)  then SetFX(FPSCR.VXCVI)

vx_flag ← vxsnan_flag | vxcvi_flag
ex_flag ← FPSCR.VE & vx_flag

if ex_flag=0 then do
   VSR[VRT+32].dword[0] ← result
   VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← 0bUUUUU
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src be the quad-precision floating-point value in VSR[VRB+32].

If src is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN and VXCVI are set to 1.

If src is a Quiet NaN or an Infinity, an Invalid Operation exception occurs and VXCVI is set to 1.

If src is a NaN, the result is 0x0000_0000_0000_0000.

Otherwise, if src is a Zero, the result is 0x0000_0000_0000_0000.

Otherwise, if src is a positive Infinity, the result is 0xFFFF_FFFF_FFFF_FFFF.

Otherwise, if src is a negative Infinity, the result is 0x0000_0000_0000_0000.

Otherwise, do the following.

Let rnd be the value src truncated to a floating-point integer.

If rnd is greater than $+2^{64}-1$, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 0xFFFF_FFFF_FFFF_FFFF.

Otherwise, if rnd is less than 0, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 0x0000_0000_0000_0000.

Otherwise, the result is the value rnd, and an Inexact exception occurs if rnd is inexact (i.e., rnd is not equal to src).

The result is placed into doubleword element 0 of VSR[VRT+32] in unsigned integer format.

The contents of doubleword element 1 of VSR[VRT+32] are set to 0.

FPRF is set to undefined. FR is set to 0. FI is set to indicate if the rounded result is inexact.

If an Invalid Operation exception occurs, FR and FI are set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified.

See Table 64, “Actions for xscvqpudz,” on page 553.

Special Registers Altered:

FPRF (undefined) FR (set to 0) FI
FX VXSNAN VXCVI XX

VSR Data Layout for xscvqpudz

VSR[VRB+32]

src

VSR[VRT+32]

tgt.dword[0]	0x0000_0000_0000_0000

	FPSCR.VE	FPSCR.XE	bfp_ROUND_TO_INTEGER(0b0001,src) ≠ src	Returned Results and Status Setting
src ≤ Nmin-1	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
	–	1	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmin	–	–	no	T(Nmin), fr(0), fi(0), fprf(0bUUUUU)
Nmin < src < Nmax	–	–	no	T(bfp_CONVERT_TO_UI64(trunc(src))), fr(0), fi(0), fprf(0bUUUUU)
	–	0	yes	T(bfp_CONVERT_TO_UI64(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
	–	1	yes	T(bfp_CONVERT_TO_UI64(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmax	–	–	no	T(Nmax), fr(0), fi(0), fprf(0bUUUUU)
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
	–	1	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNAN)
	1	–	–	fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()

Explanation:

T(x)	Places the value x into the target VSR. VSR[VRT+32].dword[0] ← x VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
Nmin	The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000).
Nmax	The largest unsigned integer doubleword value, 2^{64}-1 (0xFFFF_FFFF_FFFF_FFFF).
src	The quad-precision floating-point value in VSR[VRB+32].
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
fi(x)	FPSCR.FI is set to the value x.
fr(x)	FPSCR.FR is set to the value x.
fprf(x)	FPSCR.FPRF is set to the value x.
error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
trunc(x)	Return the floating-point value x truncated to a floating-point integer.

Table 64. Actions for xscvqpudz

VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format X-form

xscvqpuwz VRT,VRB

63	VRT	1	VRB	836	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if src.class.QNaN | src.class.SNaN then do
   result ← 0x0000_0000
   vxsnan_flag ← src.class.SNaN
   vxcvi_flag ← 1
end
else if src.class.Infinity then do
   vxcvi_flag ← 1
   if src.sign = 0 then
      result ← 0x0000_0000_FFFF_FFFF
   else
      result ← 0x0000_0000_0000_0000
end
else if src.class.Zero then
   result ← 0x0000_0000
else do
   rnd ← bfp_ROUND_TO_INTEGER(0b001,src)
   if bfp_COMPARE_GT(rnd, +2^32-1) then do
      result ← 0x0000_0000_FFFF_FFFF
      vxcvi_flag ← 1
   end
   else if bfp_COMPARE_LT(rnd, bfp_ZERO) then do
      result ← 0x0000_0000_0000_0000
      vxcvi_flag ← 1
   end
   else do
      result ← bfp_CONVERT_TO_UI64(rnd)
      if(xx_flag) then SetFX(FPSCR.XX)
   end
end

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(vxcvi_flag)  then SetFX(FPSCR.VXCVI)

vx_flag ← vxsnan_flag | vxcvi_flag
ex_flag ← FPSCR.VE & vx_flag

if ex_flag=0 then do
   VSR[VRT+32].dword[0] ← result
   VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
   FPSCR.FPRF ← 0bUUUUU
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src be the quad-precision floating-point value in VSR[VRB+32].

If src is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN and VXCVI are set to 1.

If src is a Quiet NaN or an Infinity, an Invalid Operation exception occurs and VXCVI is set to 1.

If src is a NaN, the result is 0x0000_0000_0000_0000.

Otherwise, if src is a Zero, the result is 0x0000_0000_0000_0000.

Otherwise, if src is a positive Infinity, the result is 0x0000_0000_FFFF_FFFF.

Otherwise, do the following.

Let rnd be the value src truncated to a floating-point integer.

If rnd is greater than $+2^{32}-1$, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 0x0000_0000_FFFF_FFFF.

Otherwise, if rnd is less than 0, an Invalid Operation exception occurs, VXCVI is set to 1, and the result is 0x0000_0000_0000_0000.

Otherwise, the result is the value rnd, and an Inexact exception occurs if rnd is inexact (i.e., rnd is not equal to src).

The result is placed into doubleword element 0 of VSR[VRT+32] in unsigned integer format.

The contents of doubleword element 1 of VSR[VRT+32] are set to 0.

FPRF is set to undefined. FR is set to 0. FI is set to indicate if the rounded result is inexact.

If an Invalid Operation exception occurs, FR and FI are set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified.

See Table 65, “Actions for xscvqpuwz,” on page 555.

Special Registers Altered:

FPRF (undefined) FR (set to 0) FI
FX VXSNAN VXCVI XX

VSR Data Layout for xscvqpuwz

VSR[VRB+32]

src

VSR[VRT+32]

tgt.dword[0]	0x0000_0000_0000_0000

	FPSCR.VE	FPSCR.XE	bfp_ROUND_TO_INTEGER(0b0001,src) ≠ src	Returned Results and Status Setting
src ≤ Nmin-1	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
	–	1	yes	T(Nmin), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmin	–	–	no	T(Nmin), fr(0), fi(0), fprf(0bUUUUU)
Nmin < src < Nmax	–	–	no	T(bfp_CONVERT_TO_UI64(trunc(src))), fr(0), fi(0), fprf(0bUUUUU)
	–	0	yes	T(bfp_CONVERT_TO_UI64(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
	–	1	yes	T(bfp_CONVERT_TO_UI64(trunc(src))), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src = Nmax	–	–	no	T(Nmax), fr(0), fi(0), fprf(0bUUUUU)
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX)
	–	1	yes	T(Nmax), fr(0), fi(1), fprf(0bUUUUU), fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI)
	1	–	–	fr(0), fi(0), fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fr(0), fi(0), fprf(0bUUUUU), fx(VXCVI), fx(VXSNAN)
	1	–	–	fr(0), fi(0), fx(VXCVI), fx(VXSNAN), error()

Explanation:

T(x)	Places the value x into the target VSR. VSR[VRT+32].dword[0] ← x VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
Nmin	The smallest unsigned integer word value, 0 (0x0000_0000_0000_0000).
Nmax	The largest unsigned integer word value, 2^{32}-1 (0x0000_0000_FFFF_FFFF).
src	The quad-precision floating-point value in VSR[VRB+32].
fx(x)	FPSCR.FX is set to 1 if FPSCR.x=0. FPSCR.x is set to 1.
fi(x)	FPSCR.FI is set to the value x.
fr(x)	FPSCR.FR is set to the value x.
fprf(x)	FPSCR.FPRF is set to the value x.
error()	The system error handler is invoked for the trap-enabled exception if MSR.FE0 and MSR.FE1 are set to any mode other than the ignore-exception mode.
trunc(x)	Return the floating-point value x truncated to a floating-point integer.

Table 65. Actions for xscvqpuwz

VSX Scalar Convert Signed Doubleword to Quad-Precision format X-form

xscvsdqp VRT,VRB

63	VRT	10	VRB	836	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src    ← bfp_CONVERT_FROM_SI64(VSR[VRB+32].dword[0])
result ← bfp_CONVERT_TO_BFP128(src)

VSR[VRT+32] ← result
FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
FPSCR.FR    ← 0
FPSCR.FI    ← 0
```

Let src be the signed integer value in doubleword element 0 of VSR[VRB+32].

src is placed into VSR[VRT+32] in quad-precision floating-point format.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

Special Registers Altered:

FPRF FR (set to 0) FI (set to 0)

VSR Data Layout for xscvsdqp

VSR[VRB+32]

src.dword[0]	unused

VSR[VRT+32]

tgt

VSX Scalar Convert Single-Precision to Double-Precision format XX2-form

xscvspdp XT,XB

60	T	///	B	329	BX	TX
0	6	11	16	21	30	31

```
reset_xflags()

src ← VSR[32×BX+B].word[0]
result ← ConvertVectorSPtoScalarSP(src)
if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
vex_flag ← FPSCR.VE & vxsnan_flag
FPSCR.FR ← 0b0
FPSCR.FI ← 0b0
if( ~vex_flag ) then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPSCR.FPRF ← ClassDP(result)
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the single-precision floating-point value in word element 0 of VSR[XB].

If src is a SNaN, the result is src, converted to a QNaN (i.e., bit 9 of src set to 1). VXSNAN is set to 1.

Otherwise, the result is src.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

If a trap-enabled invalid operation exception occurs, VSR[XT] is not modified, FPRF is not modified, FR is set to 0, and FI is set to 0.

Special Registers Altered

FPRF FR=0b0 FI=0b0 FX VXSNAN

VSR Data Layout for xscvspdp

src = VSR[XB]

.word[0]	unused	unused

tgt = VSR[XT]

.dword[0]	undefined

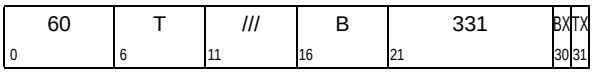
0 32 64 127

Programming Note

xscvspdp can be used to convert a single-precision value in single-precision format to double-precision format for use by Floating-Point scalar single-precision operations.

VSX Scalar Convert Single-Precision to Double-Precision format Non-signalling XX2-form

xscvspdpn XT,XB




```
reset_xflags()
src ← VSR[32×BX+B].word[0]
result ← ConvertSPtoDP_NS(src)
VSR[32×TX+T].dword[0] ← result
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the single-precision floating-point value in word element 0 of VSR[XB].

src is placed into doubleword element 0 of VSR[XT] in double-precision format.

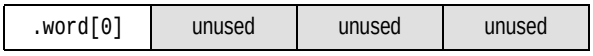
The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

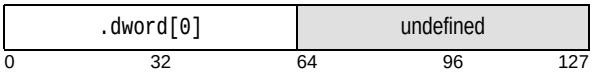
None

VSR Data Layout for xscvspdpn

src = VSR[XB]



tgt = VSR[XT]



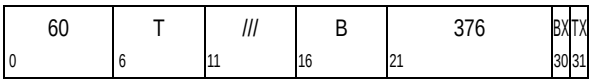
Programming Note

xscvspdp should be used to convert a vector single-precision floating-point value to scalar double-precision format.

xscvspdpn should be used to convert a vector single-precision floating-point value to scalar single-precision format.

VSX Scalar Convert with round Signed Doubleword to Double-Precision format XX2-form

xscvsxddp XT,XB




```
reset_xflags()

src    ← ConvertSDtoFP(VSR[32×BX+B].dword[0])
result ← RoundToDP(RN,src)
VSR[32×TX+T].dword[0] ← result
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU

if(xx_flag) then SetFX(XX)

FPRF        ← ClassDP(result)
FR          ← inc_flag
FI          ← xx_flag
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the signed integer value in doubleword element 0 of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

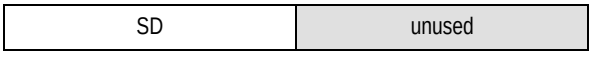
FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

Special Registers Altered

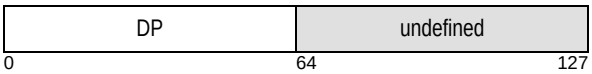
FPRF FR FI FX XX

VSR Data Layout for xscvsxddp

src = VSR[XB]

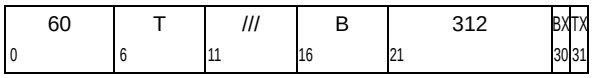


tgt = VSR[XT]



VSX Scalar Convert with round Signed Doubleword to Single-Precision format XX2-form

xscvsxdsp XT,XB




```
reset_xflags()

src    ← ConvertSDtoDP(VSR[32×BX+B].dword[0])
result ← RoundToSP(RN,src)
VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result)
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU

if(xx_flag) then SetFX(XX)

FPRF ← ClassSP(result)
FR   ← inc_flag
FI   ← xx_flag
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the two's-complement integer value in doubleword element 0 of VSR[XB].

src is converted to floating-point format, and rounded to single-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

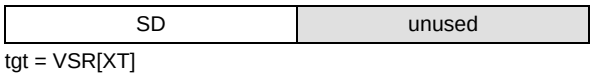
FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

Special Registers Altered

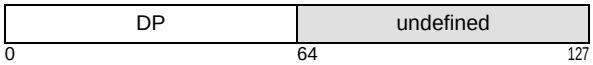
FPRF FR FI FX XX

VSR Data Layout for xscvsxdsp

src = VSR[XB]

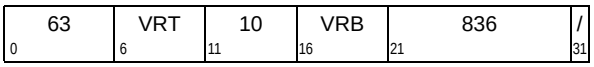


tgt = VSR[XT]



VSX Scalar Convert Signed Doubleword to Quad-Precision format X-form

xscvsdqp VRT,VRB




```
if MSR.VSX=0 then VSX_Unavailable()

src    ← bfp_CONVERT_FROM_SI64(VSR[VRB+32].dword[0])
result ← bfp_CONVERT_TO_BFP128(src)

VSR[VRT+32] ← result
FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
FPSCR.FR    ← 0
FPSCR.FI    ← 0
```

Let src be the signed integer value in doubleword element 0 of VSR[VRB+32].

src is placed into VSR[VRT+32] in quad-precision floating-point format.

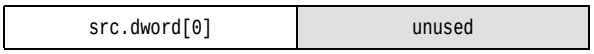
FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

Special Registers Altered:

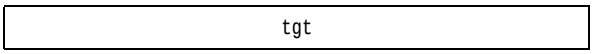
FPRF FR (set to 0) FI (set to 0)

VSR Data Layout for xscvsdqp

VSR[VRB+32]

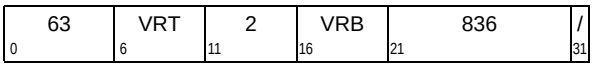


VSR[VRT+32]



VSX Scalar Convert Unsigned Doubleword to Quad-Precision format X-form

xscvudqp VRT,VRB




```
if MSR.VSX=0 then VSX_Unavailable()

src    ← bfp_CONVERT_FROM_UI64(VSR[VRB+32].dword[0])
result ← bfp_CONVERT_TO_BFP128(src)

VSR[VRT+32] ← result
FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
FPSCR.FR    ← 0
FPSCR.FI    ← 0
```

Let src be the unsigned integer value in doubleword element 0 of VSR[VRB+32].

src is placed into VSR[VRT+32] in quad-precision floating-point format.

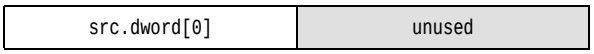
FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

Special Registers Altered:

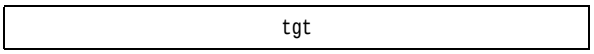
FPRF FR (set to 0) FI (set to 0)

VSR Data Layout for xscvudqp

VSR[VRB+32]

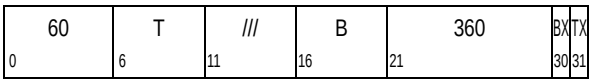


VSR[VRT+32]



VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format XX2-form

xscvuxddp XT,XB




```
reset_xflags()

src    ← ConvertUDtoFP(VSR[32×BX+B].dword[0])
result ← RoundToDP(RN,src)
VSR[32×TX+T].dword[0] ← result
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU

if(xx_flag) then SetFX(XX)

FPRF ← ClassDP(result)
FR   ← inc_flag
FI   ← xx_flag
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the unsigned integer value in doubleword element 0 of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

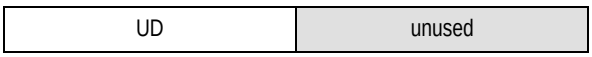
FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

Special Registers Altered

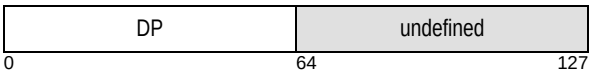
FPRF FR FI FX XX

VSR Data Layout for xscvuxddp

src = VSR[XB]

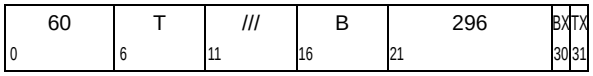


tgt = VSR[XT]



VSX Scalar Convert with round Unsigned Doubleword to Single-Precision XX2-form

xscvuxdsp XT,XB




```
reset_xflags()

src    ← ConvertUDtoDP(VSR[32×BX+B].dword[0])
result ← RoundToSP(RN,src)
VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result)
VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU

if(xx_flag) then SetFX(XX)

FPRF ← ClassSP(result)
FR   ← inc_flag
FI   ← xx_flag
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the unsigned-integer value in doubleword element 0 of VSR[XB].

src is converted to floating-point format, and rounded to single-precision using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

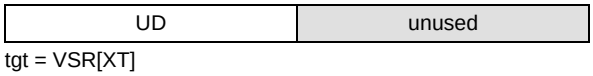
FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

Special Registers Altered

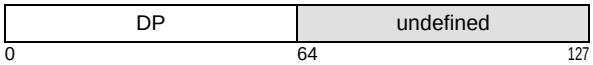
FPRF FR FI FX XX

VSR Data Layout for xscvuxdsp

src = VSR[XB]



tgt = VSR[XT]



VSX Scalar Divide Double-Precision XX3-form

xsdivdp XT,XA,XB

60	T	A	B	56	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
reset_xflags()
src1         ← VSR[XA]{0:63}
src2         ← VSR[XB]{0:63}
v{0:inf}     ← DivideFP(src1,src2)
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxidi_flag)  then SetFX(VXIDI)
if(vxzdz_flag)  then SetFX(VXZDZ)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
if(zx_flag)     then SetFX(ZX)
vex_flag     ← VE & (vxsnan_flag | vxidi_flag | vxzdz_flag)
zex_flag     ← ZE & zx_flag

if( ~vex_flag & ~zex_flag ) then do
   VSR[XT] = result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    = ClassDP(result)
   FR      = inc_flag
   FI      = xx_flag
end
else do
   FR      = 0b0
   FI      = 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is divided[1] by src2, producing a quotient having unbounded range and precision.

The quotient is normalized[2].

See *Actions for xsdivdp* (p. 563).

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered

FPRF FR FI FX OX UX ZX XX
VXSNAN VXIDI VXZDZ

VSR Data Layout for xsdivdp

src1 = VSR[XA]

DP	unused

src2 = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

1. Floating-point division is based on exponent subtraction and division of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← dQNaN vxidi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Zero	v ← D(src1,src2)	v ← +Infinity zx_flag ← 1	v ← –Infinity zx_flag ← 1	v ← D(src1,src2)	v ← –Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← +Zero	v ← +Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← –Zero	v ← –Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← –Zero	v ← –Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← +Zero	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Zero	v ← D(src1,src2)	v ← –Infinity zx_flag ← 1	v ← +Infinity zx_flag ← 1	v ← D(src1,src2)	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxidi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
D(x,y)	Return the normalized quotient of floating-point value x divided by floating-point value y, having unbounded range and precision.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 66.Actions for xsdivdp

VSX Scalar Divide Quad-Precision [*using round to Odd*] *X-form*

xsdivqp VRT,VRA,VRB (RO=0)
xsdivqpo VRT,VRA,VRB (RO=1)

63	VRT	VRA	VRB	548	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2   ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_DIVIDE(src1, src2)
rnd    ← bfp_ROUND_TO_BFP128(RO, FPSCR.RN, v)
result ← bfp_CONVERT_TO_BFP128(rnd)

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(vxidi_flag)  then SetFX(FPSCR.VXIDI)
if(vxzdz_flag)  then SetFX(FPSCR.VXZDZ)
if(ox_flag)     then SetFX(FPSCR.OX)
if(ux_flag)     then SetFX(FPSCR.UX)
if(zx_flag)     then SetFX(FPSCR.ZX)
if(xx_flag)     then SetFX(FPSCR.XX)

vx_flag ← vxsnan_flag | vxidi_flag | vxzdz_flag
ex_flag ← (FPSCR.VE & vx_flag) | (FPSCR.ZE & zx_flag)

if ex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & (zx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & (zx_flag=0) & xx_flag
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1 or src2 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1

If src1 and src2 are Infinity values, an Invalid Operation exception occurs and VXIDI is set to 1.

If src1 and src2 are Zero values, an Invalid Operation exception occurs and VXZDZ is set to 1.

If src1 is a finite value and src2 is a Zero value, an Zero Divide exception occurs and ZX is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src2 is a Quiet NaN, the result is src2.

Otherwise, if src1 and src2 are Infinity values, or if src1 and src2 are Zero values, the result is the default Quiet NaN[1].

Otherwise, if src1 is a non-zero value and src2 is a Zero value, the result is an Infinity.

Otherwise, do the following.

The normalized quotient of src1 divided by src2 is produced with unbounded significand precision and exponent range.

See Table 67, “Actions for xsdivqp[o],” on page 565.

If the intermediate result is *Tiny* (i.e., the unbiased exponent is less than -16382) and UE=0, the significand is shifted right N bits, where N is the difference between -16382 and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value -16382.

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

If a trap-disabled Zero Divide exception occurs, FR and FI are set to 0.

If a trap-enabled Invalid Operation exception or a trap-enabled Zero Divide exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

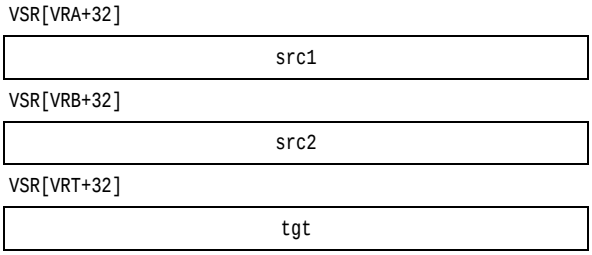
See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered:

FPRF FR FI
FX VXSNAN VXIDI VXZDZ OX UX ZX XX

1. The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.

VSR Data Layout for xsdivqp[o]



src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← dQNaN vxidi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← -Infinity	v ← -Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← quiet(src2) vxsnan_flag ← 1
-NZF		v ← Div(src1,src2)	v ← +Infinity zx_flag ← 1	v ← -Infinity zx_flag ← 1	v ← Div(src1,src2)			
-Zero	v ← +Zero		v ← dQNaN vxzdz_flag ← 1		v ← -Zero			
+Zero	v ← -Zero				v ← +Zero			
+NZF		v ← Div(src1,src2)	v ← -Infinity zx_flag ← 1	v ← +Infinity zx_flag ← 1	v ← Div(src1,src2)			
+Infinity	v ← dQNaN vxidi_flag ← 1	v ← -Infinity	v ← -Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxidi_flag ← 1		
QNaN	v ← src1							v ← src1 vxsnan_flag ← 1
SNaN	v ← quiet(src1) vxsnan_flag ← 1							

Explanation:

src1	The quad-precision floating-point value in VSR[VRA+32].
src2	The quad-precision floating-point value in VSR[VRB+32].
dQNaN	Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000_0000).
NZF	Nonzero finite number.
Div(x,y)	The floating-point value x is divided[1] by floating-point value y. Return the normalized[2] quotient, having unbounded range and precision.
quiet(x)	Convert x to the corresponding Quiet NaN.
v	The intermediate result having unbounded significand precision and unbounded exponent range.

Table 67. Actions for xsdivqp[o]

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then subtracted or added as appropriate, depending on the signs of the operands, to form an intermediate difference. All 64 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSX Scalar Divide Single-Precision XX3-form

xsdivsp XT,XA,XB

60	T	A	B	24	AX	BX	TX
0	6	11	16	21	29	30	31

```
reset_xflags()

src1   ← VSR[32×AX+A].dword[0]
src2   ← VSR[32×BX+B].dword[0]
v      ← DivideDP(src1,src2)
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vxidi_flag)  then SetFX(VXIDI)
if(vxzdz_flag)  then SetFX(VXZDZ)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
if(zx_flag)     then SetFX(ZX)

vex_flag ← VE & (vxsnan_flag|vxidi_flag|vxzdz_flag)
zex_flag ← ZE & zx_flag

if( ~vex_flag & ~zex_flag ) then do
   VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is divided[1] by src2, producing a quotient having unbounded range and precision.

The quotient is normalized[2].

See Table 68, “Actions for xsdivsp,” on page 567.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.
If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.
See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered

FPRF FR FI FX OX UX ZX XX
VXSNAN VXIDI VXZDZ

VSR Data Layout for xsdivsp

src1 = VSR[XA]

DP	unused

src2 = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

1. Floating-point division is based on exponent subtraction and division of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← dQNaN vxidi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Zero	v ← D(src1,src2)	v ← +Infinity zx_flag ← 1	v ← –Infinity zx_flag ← 1	v ← D(src1,src2)	v ← –Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← +Zero	v ← +Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← –Zero	v ← –Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← –Zero	v ← –Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← +Zero	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Zero	v ← D(src1,src2)	v ← –Infinity zx_flag ← 1	v ← +Infinity zx_flag ← 1	v ← D(src1,src2)	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxidi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
D(x,y)	Return the normalized quotient of floating-point value x divided by floating-point value y, having unbounded range and precision.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 68.Actions for xsdivsp

VSX Scalar Insert Exponent Double-Precision X-form

xsiexpdp XT,RA,RB

60	T	RA	RB	918	TX
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← GPR[RA]
src2 ← GPR[RB]

VSR[32×TX+T].dword[0].bit[0]     ← src1.bit[0]
VSR[32×TX+T].dword[0].bit[1:11]  ← src2.bit[53:63]
VSR[32×TX+T].dword[0].bit[12:63] ← src1.bit[12:63]
VSR[32×TX+T].dword[1]            ← 0xUUUU_UUUU_UUUU_UUUUU
```

Let XT be the sum 32×TX + T.

Let src1 be the unsigned integer value in GPR[RA].
Let src2 be the unsigned integer value in GPR[RB].

The contents of bit 0 of src1 are placed into bit 0 of VSR[XT].

The contents of bits 53:63 of src2 are placed into bits 1:11 of VSR[XT].

The contents of bits 12:63 of src1 are placed into bits 12:63 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

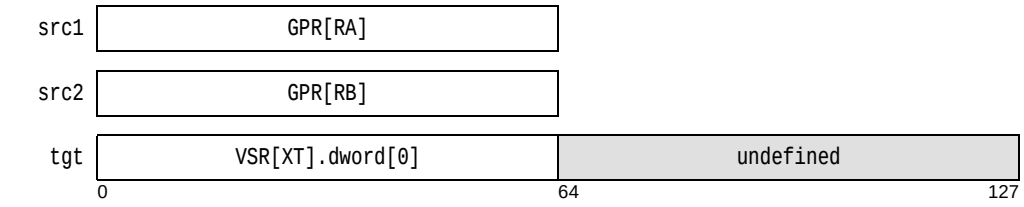
Special Registers Altered:

None

Programming Note

This instruction can be used to produce a single-precision result.

VSR Data Layout for xsiexpdp



VSX Scalar Insert Exponent Quad-Precision X-form

xsiexpqp VRT,VRA,VRB

63	VRT	VRA	VRB	868	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[VRT+32].bit[0]      ← VSR[VRA+32].bit[0]
VSR[VRT+32].bit[1:15]   ← VSR[VRB+32].dword[0].bit[49:63]
VSR[VRT+32].bit[16:127] ← VSR[VRA+32].bit[16:127]
```

The contents of bit 0 of VSR[VRA+32] are placed into bit 0 of VSR[VRT+32].

The contents of bit 49:63 of doubleword element 0 of VSR[VRB+32] are placed into bits 1:15 of VSR[VRT+32].

The contents of bit 16:127 of VSR[VRA+32] are placed into bits 16:127 of VSR[VRT+32].

Special Registers Altered:

None

VSR Data Layout for xsiexpqp

VSR[VRA+32]

src1

VSR[VRB+32]

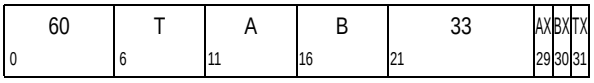
src2.dword[0]	unused

VSR[VRT+32]

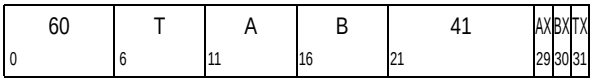
tgt

VSX Scalar Multiply-Add Double-Precision XX3-form

xsmaddadp XT,XA,XB



xsmaddmdp XT,XA,XB




```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
reset_xflags()
src1        ← VSR[XA]{0:63}
src2        ← "xsmaddadp" ? VSR[XT]{0:63} : VSR[XB]{0:63}
src3        ← "xsmaddadp" ? VSR[XB]{0:63} : VSR[XT]{0:63}
v{0:inf}    ← MultiplyAddFP(src1,src3,src2)
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
vex_flag    ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

For ***xsmaddadp***, do the following.
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsmaddmdp***, do the following.
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 69.

src2 is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 69.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, "VSX Scalar Floating-Point Final Result," on page 516.

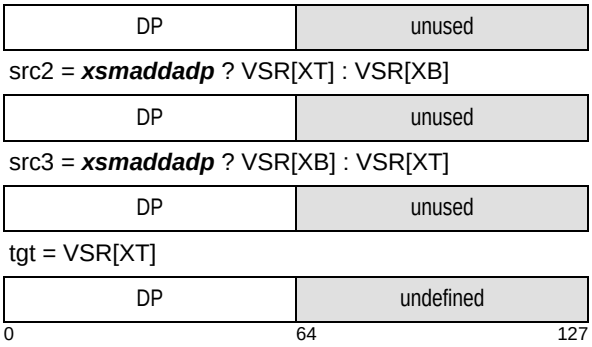
Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xsmadd(a|m)dp

src1 = VSR[XA]



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← –Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← –Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← –Infinity	v ← src2	v ← –Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← –Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsmaddadp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsmaddmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsmaddadp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsmaddmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
	Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 69.Actions for xsmadd(a|m)dp

VSX Scalar Multiply-Add Single-Precision XX3-form

xsmaddasp XT,XA,XB

60	T	A	B	1	AX	BX	TX
0	6	11	16	21	29	30	31

xsmaddmsp XT,XA,XB

60	T	A	B	9	AX	BX	TX
0	6	11	16	21	29	30	31

```
reset_xflags()

if “xsmaddasp” then do
   src1 ← VSR[32×AX+A].dword[0]
   src2 ← VSR[32×TX+T].dword[0]
   src3 ← VSR[32×BX+B].dword[0]
end
if “xsmaddmsp” then do
   src1 ← VSR[32×AX+A].dword[0]
   src2 ← VSR[32×BX+B].dword[0]
   src3 ← VSR[32×TX+T].dword[0]
end

v      ← MultiplyAddDP(src1,src3,src2)
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For ***xsmaddasp***, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsmaddmsp***, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 70, “Actions for xsmadd(a|m)sp,” on page 575.

src2 is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 70, “Actions for xsmadd(a|m)sp,” on page 575.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

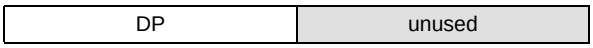
Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI VXIMZ

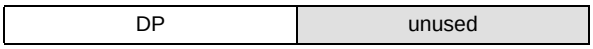
VSR Data Layout for xsmadd(a|m)sp

src1 = VSR[XA]

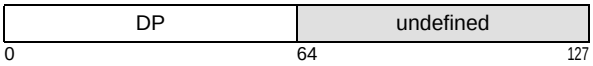
src2 = ***xsmaddasp*** ? VSR[XT] : VSR[XB]



src3 = ***xsmaddasp*** ? VSR[XB] : VSR[XT]



tgt = VSR[XT]



Part 1: Multiply

src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← –Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← –Infinity	p ← M(src1,src3)	p ← –Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add

p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–Zero	v ← –Infinity	v ← src2	v ← –Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← –Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsmaddasp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsmaddmsp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsmaddasp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsmaddmsp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 70.Actions for xsmadd(a|m)sp

VSX Scalar Multiply-Add Quad-Precision [using round to Odd] X-form

xsmaddqp	VRT,VRA,VRB	(RO=0)
xsmaddqpo	VRT,VRA,VRB	(RO=1)

63	VRT	VRA	VRB	388	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2   ← bfp_CONVERT_FROM_BFP128(VSR[VRT+32])
src3   ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_MULTIPLY_ADD(src1, src3, src2)
rnd    ← bfp_ROUND_TO_BFP128(RO, FPSCR.RN, v)
result ← bfp_CONVERT_TO_BFP128(rnd)

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(vximz_flag)  then SetFX(FPSCR.VXIMZ)
if(vxisi_flag)  then SetFX(FPSCR.VXISI)
if(ox_flag)     then SetFX(FPSCR.OX)
if(ux_flag)     then SetFX(FPSCR.UX)
if(xx_flag)     then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag | vxisi_flag
ex_flag  ← FPSCR.VE & vx_flag

if ex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRT+32] represented in quad-precision format.

Let src3 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1, src2, or src3 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 is an Infinity value and src3 is a Zero value, or if src1 is a Zero value and src3 is an Infinity value, an Invalid Operation exception occurs and VXIMZ is set to 1.

If src2 and the product of src1 and src3 are Infinity values having opposite signs, an Invalid Operation exception occurs and VXISI is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src2 is a Quiet NaN, the result is src2.

Otherwise, if src3 is a Signalling NaN, the result is the Quiet NaN corresponding to src3.

Otherwise, if src3 is a Quiet NaN, the result is src3.

Otherwise, if src1 is an Infinity value and src3 is a Zero value, or if src1 is a Zero value and src3 is an Infinity value, the result is the default Quiet NaN[1].

Otherwise, if the product of src1 and src3, and src2 are Infinity values having opposite signs, the result is the default Quiet NaN.

Otherwise, do the following.

src1 is multiplied by src3, producing a product having unbounded significand precision and exponent range.

See part 1 of Table 69. "Actions for xsmadd(a|m)dp".

src2 is added to the product, producing a sum having unbounded range and precision.

See part 2 of Table 69. "Actions for xsmadd(a|m)dp".

If the intermediate result is *Tiny* (i.e., the unbiased exponent is less than -16382) and UE=0, the significand is shifted right N bits, where N is the difference between -16382 and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value -16382.

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into VSR[VRT+32] in quad-precision format.

1. The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

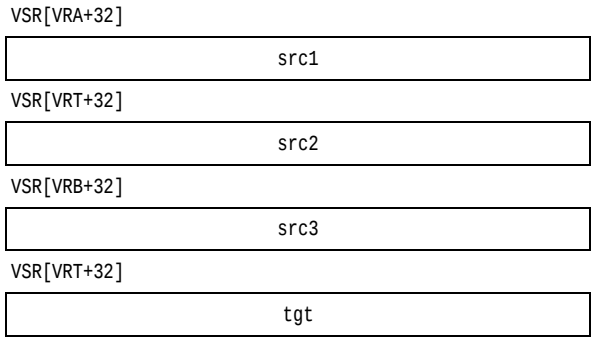
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered:

FPRF FR FI
FX VXSNAN VXIMZ VXISI OX UX XX

VSR Data Layout for xsmaddqp[o]



Part 1: Multiply

src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	p ← +Infinity		p ← dQNaN vximz_flag ← 1			p ← -Infinity	p ← src3	p ← quiet(src3) vxsnan_flag ← 1
–NZF		p ← mul(src1,src3)			p ← mul(src1,src3)			
–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero		p ← -Zero		p ← dQNaN vximz_flag ← 1		
+Zero		p ← -Zero		p ← +Zero				
+NZF		p ← mul(src1,src3)			p ← mul(src1,src3)			
+Infinity	p ← -Infinity		p ← dQNaN vximz_flag ← 1			p ← +Infinity		
QNaN	p ← src1							p ← src1 vxsnan_flag ← 1
SNaN	p ← quiet(src1) vxsnan_flag ← 1							

Part 2: Add

p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	v ← -Infinity					v ← dQNaN vxisi_flag ← 1	v ← src2	v ← quiet(src2) vxsnan_flag ← 1
–NZF		v ← add(p,src2)	v ← p		v ← add(p,src2)			
–Zero		v ← src2	v ← -Zero	v ← Rezd	v ← src2			
+Zero			v ← Rezd	v ← +Zero				
+NZF		v ← add(p,src2)	v ← p		v ← add(p,src2)			
+Infinity	v ← dQNaN vxisi_flag ← 1					v ← +Infinity		
QNaN & src1 is a NaN	v ← p							v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN							v ← src2	v ← quiet(src2) vxsnan_flag ← 1

Explanation:

src1	The quad-precision floating-point value in VSR[VRA+32].
src2	The quad-precision floating-point value in VSR[VRT+32].
src3	The quad-precision floating-point value in VSR[VRB+32].
dQNaN	Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000_0000).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
quiet(x)	Return a QNaN with the payload of x.
add(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
mul(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 71.Actions for xsmaddqp[o]

VSX Scalar Maximum Double-Precision XX3-form

xsmaxdp XT,XA,XB

60	T	A	B	160	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
reset_xflags()
src1        ← VSR[XA]{0:63}
src2        ← VSR[XB]{0:63}
result{0:63} ← MaximumDP(src1,src2)
if(vxsnan_flag) then SetFX(VXSNAN)
vex_flag    ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src1 is greater than src2, src1 is placed into doubleword element 0 of VSR[XT]. Otherwise, src2 is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

The maximum of +0 and –0 is +0. The maximum of a QNaN and any value is that value. The maximum of any value and an SNaN is that SNaN converted to a QNaN.

FPRF, FR and FI are not modified.

If a trap-enabled invalid operation exception occurs, VSR[XT] is not modified.

See Table 72.

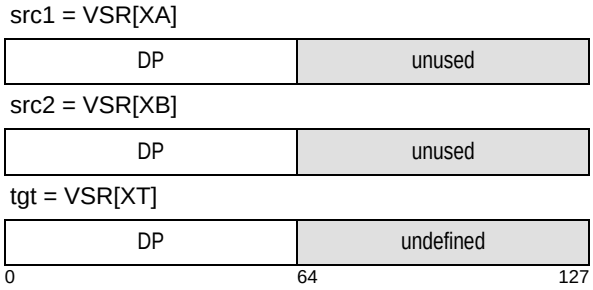
Special Registers Altered

FX VXSNAN

Programming Note

This instruction can be used to operate on single-precision source operands.

VSR Data Layout for xsmaxdp



		src2							
		–Infinity	**–NZF**	**–Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**–Infinity**	T(src1)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–NZF	T(src1)	T(M(src1,src2))	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–Zero	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Zero	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+NZF	T(src1)	T(src1)	T(src1)	T(src1)	T(M(src1,src2))	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Infinity	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

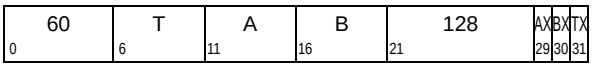
Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XT].
NZF	Nonzero finite number.
Q(x)	Return a QNaN with the payload of x.
M(x,y)	Return the greater of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$. If VE=1, update of VSR[XT] is suppressed.

Table 72.Actions for xsmaxdp

VSX Scalar Maximum Type-C Double-Precision XX3-form

xsmaxcdp XT,XA,XB




```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])

vxsnan_flag ← (src1.class="SNaN") | (src2.class="SNaN")

if (src1.type="SNaN") | (src1.type="QNaN") |
   (src2.type="SNaN") | (src2.type="QNaN") then
   result ← VSR[32×BX+B].dword[0]

else if bfp_COMPARE_GT(src1,src2) then
   result ← VSR[32×AX+A].dword[0]

else
   result ← VSR[32×BX+B].dword[0]

vex_flag ← FPSCR.VE & vxsnan_flag

if (vxsnan_flag=1) then SetFX(VXSNAN)

if (vex_flag=0) then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.

If either src1 or src2 is a NaN, result is src2.

Otherwise, if src1 is greater than src2, result is src1.

Otherwise, result is src2.

The contents of doubleword 0 of VSR[XT] are set to the value result.

The contents of doubleword 1 of VSR[XT] are undefined.

If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

Special Registers Altered:

FX VXSNAN

		src2							
		–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
src1	–Infinity	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	–NZF	T(src1)	T(M(src1,src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	–Zero	T(src1)	T(src1)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	+Zero	T(src1)	T(src1)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	+NZF	T(src1)	T(src1)	T(src1)	T(src1)	T(M(src1,src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	+Infinity	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	SNaN	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XT].
NZF	Nonzero finite number.
M(x,y)	Return the greater of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, VXSNAN. If VE=1, update of VSR[XT] is suppressed.

Table 73.Actions for xsmaxcdp

VSX Scalar Maximum Type-J Double-Precision XX3-form

xsmaxjdp XT,XA,XB

60	T	A	B	144	AX	BX	TX
0	6	11	16	21	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])

vxsnan_flag ← (src1.class="SNaN") | (src2.class="SNaN")

if (src1.type="SNaN") | (src1.type="QNaN") then
   result ← VSR[32×AX+A].dword[0]

else if (src2.type="SNaN") | (src2.type="QNaN") then
   result ← VSR[32×BX+B].dword[0]

else if (src1.type="Zero") & (src2.type="Zero") then
   if (src1.sign=0) | (src2.sign=0) then
      result ← 0x0000_0000_0000_0000  // +Zero
   else
      result ← 0x8000_0000_0000_0000  // -Zero

else if bfp_COMPARE_GT(src1,src2) then
   result ← VSR[32×AX+A].dword[0]

else
   result ← VSR[32×BX+B].dword[0]

vex_flag ← FPSCR.VE & vxsnan_flag

if (vxsnan_flag=1) then SetFX(FPSCR.VXSNAN)

if(vex_flag=0) then do
   VSR[32×TX+T].dword[0] ← bfp64_CONVERT_FROM_BFP(result)
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.

If src1 is a NaN, result is src1.

Otherwise, if src2 is a NaN, result is src2.

Otherwise, if src1 is a Zero and src2 is a Zero and either src1 or src2 is a +Zero, the result is +Zero.

Otherwise, if src1 is a -Zero and src2 is a -Zero, the result is -Zero.

Otherwise, if src1 is greater than src2, result is src1.

Otherwise, result is src2.

The contents of doubleword 0 of VSR[XT] are set to the value result.

The contents of doubleword 1 of VSR[XT] are undefined.

If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

Special Registers Altered:

FX VXSNAN

		src2							
		–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
src1	–Infinity	T(-INF)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	–NZF	T(src1)	T(M(src1,src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	–Zero	T(src1)	T(src1)	T(-Zero)	T(+Zero)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	+Zero	T(src1)	T(src1)	T(+Zero)	T(+Zero)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	+NZF	T(src1)	T(src1)	T(src1)	T(src1)	T(M(src1,src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	+Infinity	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(+INF)	T(src2)	T(src2) fx(VXSNAN)
	QNaN	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XT].
NZF	Nonzero finite number.
M(x,y)	Return the greater of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, VXSNAN. If VE=1, update of VSR[XT] is suppressed.

Table 74.Actions for xsmaxjdp

VSX Scalar Minimum Double-Precision XX3-form

xsmindp XT,XA,XB

60	T	A	B	168	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
reset_xflags()
src1        ← VSR[XA]{0:63}
src2        ← VSR[XB]{0:63}
result{0:63} ← MinimumDP(src1,src2)
if(vxsnan_flag) then SetFX(VXSNAN)
vex_flag    ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

If src1 is less than src2, src1 is placed into doubleword element 0 of VSR[XT] in double-precision format. Otherwise, src2 is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

The minimum of +0 and –0 is –0. The minimum of a QNaN and any value is that value. The minimum of any value and an SNaN is that SNaN converted to a QNaN.

FPRF, FR and FI are not modified.

If a trap-enabled invalid operation exception occurs, VSR[XT] is not modified.

See Table 75.

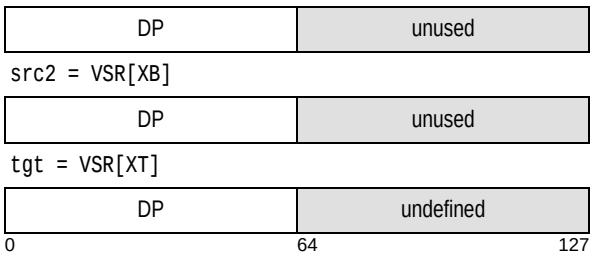
Special Registers Altered

FX VXSNAN

Programming Note

This instruction can be used to operate on single-precision source operands.

VSR Data Layout for xsmindp



		src2							
		–Infinity	**–NZF**	**–Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**–Infinity**	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–NZF	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–Zero	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Zero	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+NZF	T(src2)	T(src2)	T(src2)	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Infinity	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

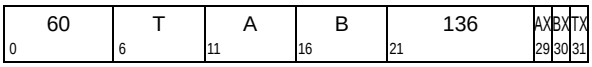
Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XT].
NZF	Nonzero finite number.
Q(x)	Return a QNaN with the payload of x.
M(x,y)	Return the lesser of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element i (i∈{0,1}) of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$. If VE=1, update of VSR[XT] is suppressed.

Table 75.Actions for xvmindp

VSX Scalar Minimum Type-C Double-Precision XX3-form

xsmincdp XT,XA,XB




```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])

vxsnan_flag ← (src1.class="SNaN") | (src2.class="SNaN")

if (src1.type="SNaN") | (src1.type="QNaN") |
   (src2.type="SNaN") | (src2.type="QNaN") then
   result ← VSR[32×BX+B].dword[0]

else if bfp_COMPARE_LT(src1,src2) then
   result ← VSR[32×AX+A].dword[0]
else
   result ← VSR[32×BX+B].dword[0]

vex_flag ← FPSCR.VE & vxsnan_flag

if (vxsnan_flag=1) then SetFX(VXSNAN)

if (vex_flag=0) then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.

If either src1 or src2 is a NaN, result is src2.

Otherwise, if src1 is less than src2, result is src1.

Otherwise, result is src2.

The contents of doubleword 0 of VSR[XT] are set to the value result.

The contents of doubleword 1 of VSR[XT] are undefined.

If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

Special Registers Altered:

FX VXSNAN

		src2							
		–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
src1	–Infinity	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	–NZF	T(src2)	T(M(src1,src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	–Zero	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	+Zero	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	+NZF	T(src2)	T(src2)	T(src2)	T(src2)	T(M(src1,src2)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	+Infinity	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2) fx(VXSNAN)
	SNaN	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)	T(src2) fx(VXSNAN)

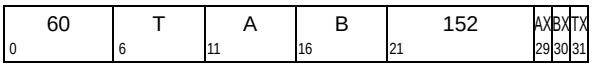
Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XT].
NZF	Nonzero finite number.
M(x,y)	Return the lesser of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, VXSNAN. If VE=1, update of VSR[XT] is suppressed.

Table 76.Actions for xsmincdp

VSX Scalar Minimum Type-J Double-Precision XX3-form

xsminjdp XT,XA,XB




```
if MSR.VSX=0 then VSX_Unavailable()

src1 ← bfp_CONVERT_FROM_BFP64(VSR[32×AX+A].dword[0])
src2 ← bfp_CONVERT_FROM_BFP64(VSR[32×BX+B].dword[0])

vxsnan_flag ← (src1.type=”SNaN”) | (src2.type=”SNaN”)

if (src1.type=”SNaN”) | (src1.type=”QNaN”) then
   result ← VSR[32×AX+A].dword[0]

else if (src2.type=”SNaN”) | (src2.type=”QNaN”) then
   result ← VSR[32×BX+B].dword[0]

else if (src1.type=”Zero”) & (src2.type=”Zero”) then
   if (src1.sign=1) | (src2.sign=1) then
      result ← 0x8000_0000_0000_0000  // -Zero
   else
      result ← 0x0000_0000_0000_0000  // +Zero

else if bfp_COMPARE_LT(src1,src2) then? src1 : src2
   result ← VSR[32×AX+A].dword[0]

else
   result ← VSR[32×BX+B].dword[0]

if (vxsnan_flag=1) then SetFX(FPSCR.VXSNAN)

vex_flag ← FPSCR.VE & vxsnan_flag

if(vex_flag=0) then do
   VSR[32×TX+T].dword[0] ← result
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword 0 of VSR[XB].

If src1 or src2 is a SNaN, an Invalid Operation exception occurs.

If src1 is a NaN, result is src1.

Otherwise, if src2 is a NaN, result is src2.

Otherwise, if src1 is a Zero and src2 is a Zero and either src1 or src2 is a -Zero, the result is -Zero.

Otherwise, if src1 is a +Zero and src2 is a +Zero, the result is +Zero.

Otherwise, if src1 is less than src2, result is src1.

Otherwise, result is src2.

The contents of doubleword 0 of VSR[XT] are set to the value result.

The contents of doubleword 1 of VSR[XT] are undefined.

If a trap-enabled Invalid Operation occurs, VSR[XT] is not modified.

Special Registers Altered:

FX VXSNAN

		src2							
		–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
src1	–Infinity	T(-INF)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	–NZF	T(src2)	T(M(src1,src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	–Zero	T(src2)	T(src2)	T(-Zero)	T(-Zero)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	+Zero	T(src2)	T(src2)	T(-Zero)	T(+Zero)	T(src1)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	+NZF	T(src2)	T(src2)	T(src2)	T(src2)	T(M(src1,src2)	T(src1)	T(src2)	T(src2) fx(VXSNAN)
	+Infinity	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(+INF)	T(src2)	T(src2) fx(VXSNAN)
	QNaN	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)	T(src1) fx(VXSNAN)

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XT].
NZF	Nonzero finite number.
M(x,y)	Return the greater of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element 0 of VSR[XT] in double-precision format. The contents of doubleword element 1 of VSR[XT] are undefined. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, VXSNAN. If VE=1, update of VSR[XT] is suppressed.

Table 77.Actions for xsminjdp

VSX Scalar Multiply-Subtract Double-Precision XX3-form

xsmsubadp XT,XA,XB

60	T	A	B	49	AX	BX	TX
0	6	11	16	21	29	30	31

xsmsubmdp XT,XA,XB

60	T	A	B	57	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
reset_xflags()
src1         ← VSR[XA]{0:63}
src2         ← VSR[XT]{0:63}
src3         ← VSR[XB]{0:63}
src2         ← "xsmsubadp" ? VSR[XT]{0:63} : VSR[XB]{0:63}
src3         ← "xsmsubadp" ? VSR[XB]{0:63} : VSR[XT]{0:63}
v{0:inf}     ← MultiplyAddDP(src1,src3,NegateDP(src2))
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
vex_flag     ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For ***xsmsubadp***, do the following.

– Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
– Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
– Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsmsubmdp***, do the following.

– Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
– Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
– Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 78.

src2 is negated and added[2] to the product, producing a sum having unbounded range and precision.

The result, having unbounded range and precision, is normalized[3].

See part 2 of Table 78.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, "VSX Scalar Floating-Point Final Result," on page 516.

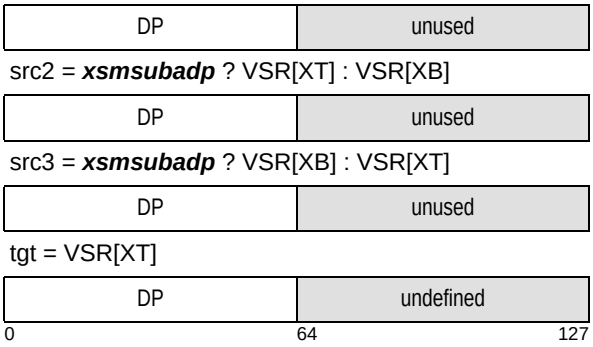
Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xsmsub(a|m)dp

src1 = VSR[XA]



Part 1: Multiply	src3							
src1	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← –Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← –Infinity	p ← M(src1,src3)	p ← –Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract	src2							
p	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← –Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← –src2	v ← +Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsmsubadp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsmsubmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsmsubadp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsmsubmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 78.Actions for xsmsub(a|m)dp

VSX Scalar Multiply-Subtract Single-Precision XX3-form

xsmsubasp XT,XA,XB

60	T	A	B	17	AX	BX	TX
0	6	11	16	21	29	30	31

xsmsubmsp XT,XA,XB

60	T	A	B	25	AX	BX	TX
0	6	11	16	21	29	30	31

```
reset_xflags()

if “xsmsubasp” then do
   src1 ← VSR[32×AX+A].dword[0]
   src2 ← VSR[32×TX+T].dword[0]
   src3 ← VSR[32×BX+B].dword[0]
end
if “xsmsubmsp” then do
   src1 ← VSR[32×AX+A].dword[0]
   src2 ← VSR[32×BX+B].dword[0]
   src3 ← VSR[32×TX+T].dword[0]
end

v      ← MultiplyAddDP(src1,src3,NegateDP(src2))
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For ***xsmsubasp***, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsmsubmsp***, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 79, “Actions for xsmsub(a|m)sp”.

src2 is negated and added[2] to the product, producing a sum having unbounded range and precision.

The result, having unbounded range and precision, is normalized[3].

See part 2 of Table 79, “Actions for xsmsub(a|m)sp”.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

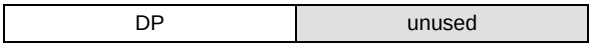
1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

Special Registers Altered

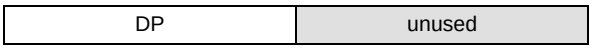
FPRF FR FI FX OX UX XX
VXSNAN VXISI VXIMZ

VSR Data Layout for xsmsub(a|m)sp

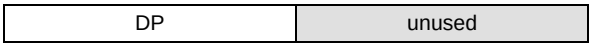
src1 = VSR[XA]



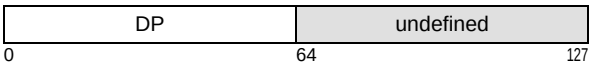
src2 = ***xsmsubasp*** ? VSR[XT] : VSR[XB]



src3 = ***xsmsubasp*** ? VSR[XB] : VSR[XT]



tgt = VSR[XT]



Part 1: Multiply

src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← –Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← –Infinity	p ← M(src1,src3)	p ← –Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract

p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← –Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← –src2	v ← +Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

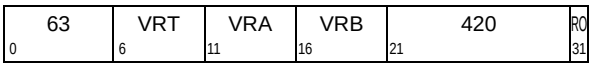
Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsmsubasp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsmsubmsp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsmsubasp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsmsubmsp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 79.Actions for xsmsub(a|m)sp

VSX Scalar Multiply-Subtract Quad-Precision [using round to Odd] X-form

xsmsubqp VRT,VRA,VRB (RO=0)
xsmsubqpo VRT,VRA,VRB (RO=1)




```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2   ← bfp_CONVERT_FROM_BFP128(VSR[VRT+32])
src3   ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_MULTIPLY_ADD(src1, src3, bfp_NEGATE(src2))
rnd    ← bfp_ROUND_TO_BFP128(RO, FPSCR.RN, v)
result ← bfp_CONVERT_TO_BFP128(rnd)

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(vximz_flag)  then SetFX(FPSCR.VXIMZ)
if(vxisi_flag)  then SetFX(FPSCR.VXISI)
if(ox_flag)     then SetFX(FPSCR.OX)
if(ux_flag)     then SetFX(FPSCR.UX)
if(xx_flag)     then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag | vxisi_flag
ex_flag  ← FPSCR.VE & vx_flag

if ex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRT+32] represented in quad-precision format.

Let src3 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1, src2, or src3 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 is an Infinity value and src3 is a Zero value, or if src1 is a Zero value and src3 is an Infinity value, an Invalid Operation exception occurs and VXIMZ is set to 1.

If src2 and the product of src1 and src3 are Infinity values having same signs, an Invalid Operation exception occurs and VXISI is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src2 is a Quiet NaN, the result is src2.

Otherwise, if src3 is a Signalling NaN, the result is the Quiet NaN corresponding to src3.

Otherwise, if src3 is a Quiet NaN, the result is src3.

Otherwise, if src1 is an Infinity value and src3 is a Zero value, or if src1 is a Zero value and src3 is an Infinity value, the result is the default Quiet NaN[1].

Otherwise, if the product of src1 and src3, and src2 are Infinity values having same signs, the result is the default Quiet NaN.

Otherwise, do the following.

src1 is multiplied by src3, producing a product having unbounded significand precision and exponent range.

See part 1 of Table 80. "Actions for xsmsubqp[o]".

src2 is negated and added to the product, producing a sum having unbounded range and precision.

See part 2 of Table 80. "Actions for xsmsubqp[o]".

If the intermediate result is *Tiny* (i.e., the unbiased exponent is less than -16382) and UE=0, the significand is shifted right N bits, where N is the difference between -16382 and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value -16382.

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

1. The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.

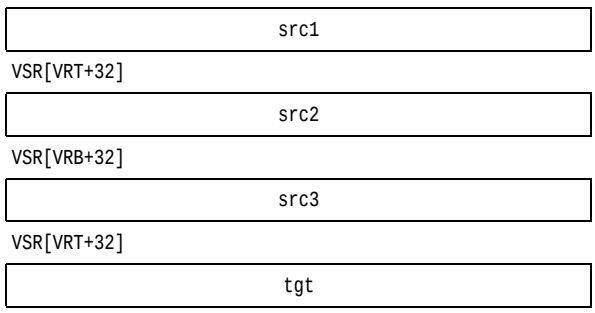
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered:

FPRF FR FI
FX VXSNAN VXIMZ VXISI OX UX XX

VSR Data Layout for xsmsubqp[o]



Part 1: Multiply

src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	p ← +Infinity		p ← dQNaN vximz_flag ← 1			p ← -Infinity	p ← src3	p ← quiet(src3) vxsnan_flag ← 1
–NZF		p ← mul(src1,src3)			p ← mul(src1,src3)			
–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero		p ← -Zero		p ← dQNaN vximz_flag ← 1		
+Zero		p ← -Zero		p ← +Zero				
+NZF		p ← mul(src1,src3)			p ← mul(src1,src3)			
+Infinity	p ← -Infinity		p ← dQNaN vximz_flag ← 1			p ← +Infinity		
QNaN	p ← src1							p ← src1 vxsnan_flag ← 1
SNaN	p ← quiet(src1) vxsnan_flag ← 1							

Part 2: Subtract

p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	v ← dQNaN vxisi_flag ← 1					v ← –Infinity	v ← src2	v ← quiet(src2) vxsnan_flag ← 1
–NZF		v ← sub(p,src2)	v ← p		v ← sub(p,src2)			
–Zero		v ← –src2	v ← Rezd	v ← –Zero	v ← –src2			
+Zero			v ← +Zero	v ← Rezd				
+NZF		v ← sub(p,src2)	v ← p		v ← sub(p,src2)			
+Infinity	v ← +Infinity					v ← dQNaN vxisi_flag ← 1		
QNaN & src1 is a NaN	v ← p							v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN							v ← src2	v ← quiet(src2) vxsnan_flag ← 1

Explanation:

src1	The quad-precision floating-point value in VSR[VRA+32].
src2	The quad-precision floating-point value in VSR[VRT+32].
src3	The quad-precision floating-point value in VSR[VRB+32].
dQNaN	Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
quiet(x)	Return a QNaN with the payload of x.
sub(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
mul(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 80.Actions for xsmsubqp[o]

VSX Scalar Multiply Double-Precision XX3-form

xsmuldp XT,XA,XB

60	T	A	B	48	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
reset_xflags()
src1        ← VSR[XA]{0:63}
src2        ← VSR[XB]{0:63}
v{0:inf}    ← MultiplyFP(src1,src2)
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
vex_flag    ← VE & (vxsnan_flag | vximz_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is multiplied[1] by src2, producing a product having unbounded range and precision.

The product is normalized[2].

See Table 81.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXIMZ

VSR Data Layout for xsmuldp

src1 = VSR[XA]

DP	unused

src2 = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Infinity	v ← M(src1,src2)	v ← +Zero	v ← –Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← dQNaN vximz_flag ← 1	v ← +Zero	v ← +Zero	v ← –Zero	v ← –Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← dQNaN vximz_flag ← 1	v ← –Zero	v ← –Zero	v ← +Zero	v ← +Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Infinity	v ← M(src1,src2)	v ← –Zero	v ← +Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← –Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 81.Actions for xsmuldp

VSX Scalar Multiply Quad-Precision [using round to Odd] X-form

xsmulqp	VRT,VRA,VRB	(RO=0)
xsmulqpo	VRT,VRA,VRB	(RO=1)

63	VRT	VRA	VRB	36	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2   ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_MULTIPLY(src1, src2)
rnd    ← bfp_ROUND_TO_BFP128(RO, FPSCR.RN, v)
result ← bfp_CONVERT_TO_BFP128(rnd)

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(vximz_flag) then SetFX(FPSCR.VXIMZ)
if(ox_flag) then SetFX(FPSCR.OX)
if(ux_flag) then SetFX(FPSCR.UX)
if(xx_flag) then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag
ex_flag  ← FPSCR.VE & vx_flag

if ex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1 or src2 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 is an Infinity value and src2 is a Zero value, or if src1 is a Zero value and src2 is an Infinity value, an Invalid Operation exception occurs and VXIMZ is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src2 is a Quiet NaN, the result is src2.

Otherwise, if src1 is an Infinity value and src2 is a Zero value, or if src1 is a Zero value and src2 is an Infinity value, the result is the default Quiet NaN[1].

Otherwise, do the following.

The normalized product of src1 multiplied by src2 is produced with unbounded significand precision and exponent range.

See Table 82. "Actions for xsmulqp[o]".

If the intermediate result is *Tiny* (i.e., the unbiased exponent is less than -16382) and UE=0, the significand is shifted right N bits, where N is the difference between -16382 and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value -16382.

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

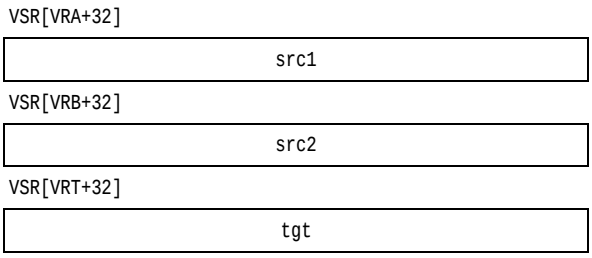
See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered:

FPRF FR FI FX VXSNAN VXIMZ OX UX XX

1. The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.

VSR Data Layout for xsmulqp[o]



src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← +Infinity		v ← dQNaN vximz_flag ← 1			v ← -Infinity	v ← src2	v ← quiet(src2) vxsnan_flag ← 1
-NZF		v ← mul(src1,src2)			v ← mul(src1,src2)			
-Zero	v ← dQNaN vximz_flag ← 1		v ← +Zero	v ← -Zero		v ← dQNaN vximz_flag ← 1		
+Zero			v ← -Zero	v ← +Zero				
+NZF		v ← mul(src1,src2)			v ← mul(src1,src2)			
+Infinity	v ← -Infinity		v ← dQNaN vximz_flag ← 1			v ← +Infinity		
QNaN	v ← src1							v ← src1 vxsnan_flag ← 1
SNaN	v ← quiet(src1) vxsnan_flag ← 1							

Explanation:

src1	The quad-precision floating-point value in VSR[VRA+32].
src2	The quad-precision floating-point value in VSR[VRB+32].
dQNaN	Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000).
NZF	Nonzero finite number.
mul(x,y)	The floating-point value x is multiplied[1] by the floating-point value y. Return the normalized product, having unbounded significand precision and exponent range.
quiet(x)	Convert x to the corresponding Quiet NaN.
v	The intermediate result having unbounded significand precision and unbounded exponent range.

Table 82. Actions for xsmulqp[o]

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.

VSX Scalar Multiply Single-Precision XX3-form

xsmulsp XT,XA,XB

60	T	A	B	16	AX	BX	TX
0	6	11	16	21	29	30	31

```
reset_xflags()

src1 ← VSR[32×AX+A].dword[0]
src2 ← VSR[32×BX+B].dword[0]

v      ← MultiplyDP(src1,src2)
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vximz_flag)

if( ~vex_flag ) then do
   VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src1 is multiplied[1] by src2, producing a product having unbounded range and precision.

The product is normalized[2].

See Table 83, “Actions for xsmulsp,” on page 605.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

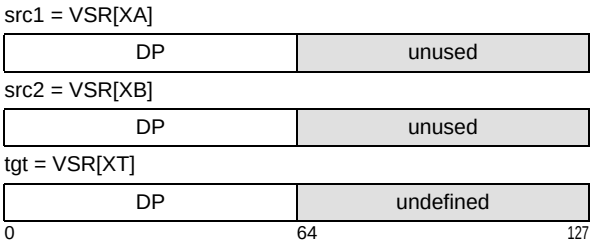
If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXIMZ

VSR Data Layout for xsmulsp



1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Infinity	v ← M(src1,src2)	v ← +Zero	v ← –Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← dQNaN vximz_flag ← 1	v ← +Zero	v ← +Zero	v ← –Zero	v ← –Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← dQNaN vximz_flag ← 1	v ← –Zero	v ← –Zero	v ← +Zero	v ← +Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Infinity	v ← M(src1,src2)	v ← –Zero	v ← +Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← –Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

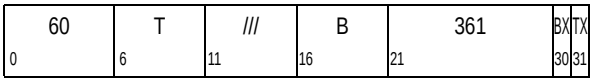
Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 83.Actions for xsmulsp

VSX Scalar Negative Absolute Double-Precision XX2-form

xsnabsdp XT,XB




```
XT          ← TX || T
XB          ← BX || B
result{0:63} ← 0b1 || VSR[XB]{1:63}
VSR[XT]     ← result || 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

The contents of doubleword element 0 of VSR[XB], with bit 0 set to 1, is placed into doubleword element 0 of VSR[XT].

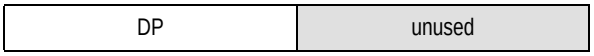
The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

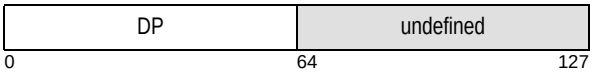
None

VSR Data Layout for xsnabsdp

src = VSR[XB]



tgt = VSR[XT]

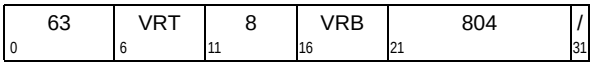


Programming Note

This instruction can be used to operate on a single-precision source operand.

VSX Scalar Negative Absolute Quad-Precision X-form

xsnabsqp VRT,VRB




```
if MSR.VSX=0 then VSX_Unavailable()

VSR[VRT+32] ← VSR[VRB+32] | 0x8000_0000_0000_0000_0000_0000_0000_0000
```

Let src be the floating-point value in VSR[VRB+32] represented in quad-precision format.

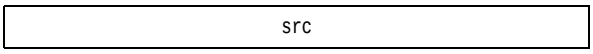
The negative absolute value of src is placed into VSR[VRT+32] in quad-precision format.

Special Registers Altered:

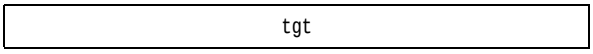
None

VSR Data Layout for xsnabsqp

VSR[VRB+32]



VSR[VRT+32]



VSX Scalar Negate Double-Precision XX2-form

xsnegdp XT,XB

60	T	///	B	377	BX	TX
0	6	11	16	21	30	31

```
XT            ← TX || T
XB            ← BX || B
result{0:63} ← ~VSR[XB]{0} || VSR[XB]{1:63}
VSR[XT]       ← result || 0xUUUU_UUUU_UUUU_UUUU
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

The contents of doubleword element 0 of VSR[XB], with bit 0 complemented, is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

Special Registers Altered

None

VSR Data Layout for xsnegdp

src = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

Programming Note

This instruction can be used to operate on a single-precision source operand.

VSX Scalar Negate Quad-Precision X-form

xsnegqp VRT,VRB

63	VRT	16	VRB	804	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

VSR[VRT+32] ← VSR[VRB+32] ^ 0x8000_0000_0000_0000_0000_0000_0000_0000
```

Let src be the floating-point value in VSR[VRB+32] represented in quad-precision format.

src is negated and placed into VSR[VRT+32] in quad-precision format.

Special Registers Altered:

None

VSR Data Layout for xsnegqp

VSR[VRB+32]

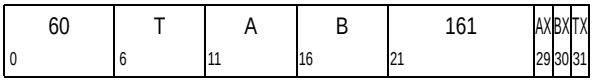
src

VSR[VRT+32]

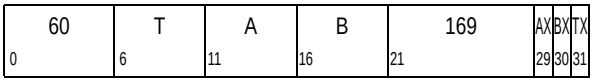
tgt

VSX Scalar Negative Multiply-Add Double-Precision XX3-form

xsnmaddadp XT,XA,XB



xsnmaddmdp XT,XA,XB




```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
reset_xflags()
src1         ← VSR[XA]{0:63}
src2         ← "xsnmaddadp" ? VSR[XT]{0:63} : VSR[XB]{0:63}
src3         ← "xsnmaddadp" ? VSR[XB]{0:63} : VSR[XT]{0:63}
v{0:inf}     ← MultiplyAddDP(src1,src3,src2)
result{0:63} ← NegateDP(RoundToDP(RN,v))
if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
vex_flag     ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0
   FI      ← 0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

For ***xsnmaddadp***, do the following.
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsnmaddmdp***, do the following.
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 84.

src2 is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 84.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is negated and placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 85, "Scalar Floating-Point Final Result with Negation," on page 611.

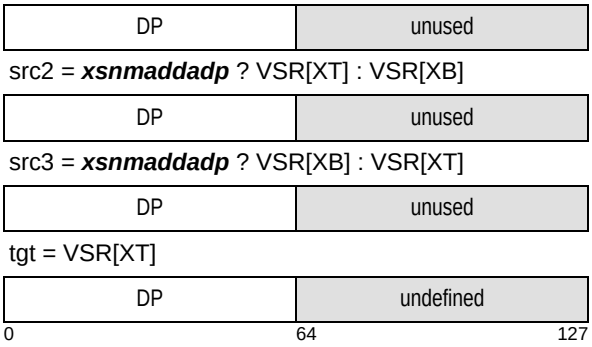
Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xsnmadd(a|m)dp

src1 = VSR[XA]



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← –Infinity	v ← src2	v ← –Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← –Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsnmaddadp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsnmaddmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsnmaddadp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsnmaddmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
	Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 84.Actions for xsnmadd(a|m)dp

Case	VE	OE	UE	ZE	XE	vxsnan_flag	vximz_flag	vxisi_flag	Is r inexact? (r g v)	Is r incremented? (|r| > |v|)	Is q inexact? (q g v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Special	–	–	–	–	–	0	0	0	–	–	–	–	T(N(r)), FPRF←ClassFP(r), FI←0, FR←0
	0	–	–	–	–	–	–	1	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0, fx(VXISI)
	0	–	–	–	–	0	1	–	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0, fx(VXIMZ)
	0	–	–	–	–	1	0	–	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0, fx(VXSNAN)
	0	–	–	–	–	1	1	–	–	–	–	–	T(r), FPRF←ClassFP(r), FI←0, FR←0, fx(VXSNAN), fx(VXIMZ)
	1	–	–	–	–	–	–	1	–	–	–	–	fx(VXISI), error()
	1	–	–	–	–	0	1	–	–	–	–	–	fx(VXIMZ), error()
	1	–	–	–	–	1	0	–	–	–	–	–	fx(VXSNAN), error()
	1	–	–	–	–	1	1	–	–	–	–	–	fx(VXSNAN), fx(VXIMZ), error()
Normal	–	–	–	–	–	–	–	–	no	–	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←0, FR←0
	–	–	–	–	0	–	–	–	yes	no	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←0, fx(XX)
	–	–	–	–	0	–	–	–	yes	yes	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←1, fx(XX)
	–	–	–	–	1	–	–	–	yes	no	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←0, fx(XX), error()
	–	–	–	–	1	–	–	–	yes	yes	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←1, fx(XX), error()
Overflow	–	0	–	–	0	–	–	–	–	–	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←?, fx(OX), fx(XX)
	–	0	–	–	1	–	–	–	–	–	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←?, fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	no	–	T(N(q)÷β), FPRF←ClassFP(N(q)÷β), FI←0, FR←0, fx(OX), error()
	–	1	–	–	–	–	–	–	–	–	yes	no	T(N(q)÷β), FPRF←ClassFP(N(q)÷β), FI←1, FR←0, fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	yes	yes	T(N(q)÷β), FPRF←ClassFP(N(q)÷β), FI←1, FR←1, fx(OX), fx(XX), error()

Explanation:

–	The results do not depend on this condition.
ClassFP(x)	Classifies the floating-point value x as defined in Table 2, “Floating-Point Result Flags,” on page 371.
fx(x)	FX is set to 1 if x=0. x is set to 1.
β	Wrap adjust, where $\beta = 2^{1536}$ for double-precision and $\beta = 2^{192}$ for single-precision.
q	The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
FI	Floating-Point Fraction Inexact status flag, $FPSCR_{FI}$. This status flag is nonsticky.
FR	Floating-Point Fraction Rounded status flag, $FPSCR_{FR}$.
OX	Floating-Point Overflow Exception status flag, $FPSCR_{OX}$.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
N(x)	The value x is is negated by complementing the sign bit of x.
T(x)	The value x is placed in element 0 of VSR[XT] in the target precision format. The contents of the remaining element(s) of VSR[XT] are undefined.
UX	Floating-Point Underflow Exception status flag, $FPSCR_{UX}$
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$.
VXIMZ	Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, $FPSCR_{VXIMZ}$.
VXISI	Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, $FPSCR_{VXISI}$.
XX	Float-Point Inexact Exception status flag, $FPSCR_{XX}$. The flag is a sticky version of $FPSCR_{FI}$. When $FPSCR_{FI}$ is set to a new value, the new value of $FPSCR_{XX}$ is set to the result of ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.

Table 85.Scalar Floating-Point Final Result with Negation

Case	VE	OE	UE	ZE	XE	vxsnan_flag	vximz_flag	vxisi_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Tiny	–	–	0	–	–	–	–	–	no	–	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←0, FR←0
	–	–	0	–	0	–	–	–	yes	no	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←0, fx(UX), fx(XX)
	–	–	0	–	0	–	–	–	yes	yes	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←1, fx(UX), fx(XX)
	–	–	0	–	1	–	–	–	yes	no	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←0, fx(UX), fx(XX), error()
	–	–	0	–	1	–	–	–	yes	yes	–	–	T(N(r)), FPRF←ClassFP(N(r)), FI←1, FR←1, fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	yes	–	no	–	T(N(q)×β), FPRF←ClassFP(N(q)×β), FI←0, FR←0, fx(UX), error()
	–	–	1	–	–	–	–	–	yes	–	yes	no	T(N(q)×β), FPRF←ClassFP(N(q)×β), FI←1, FR←0, fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	yes	–	yes	yes	T(N(q)×β), FPRF←ClassFP(N(q)×β), FI←1, FR←1, fx(UX), fx(XX), error()

Explanation:

–	The results do not depend on this condition.
ClassFP(x)	Classifies the floating-point value x as defined in Table 2, “Floating-Point Result Flags,” on page 371.
fx(x)	FX is set to 1 if x=0. x is set to 1.
β	Wrap adjust, where $\beta = 2^{1536}$ for double-precision and $\beta = 2^{192}$ for single-precision.
q	The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
FI	Floating-Point Fraction Inexact status flag, $FPSCR_{FI}$. This status flag is nonsticky.
FR	Floating-Point Fraction Rounded status flag, $FPSCR_{FR}$.
OX	Floating-Point Overflow Exception status flag, $FPSCR_{OX}$.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode.
N(x)	The value x is is negated by complementing the sign bit of x.
T(x)	The value x is placed in element 0 of VSR[XT] in the target precision format. The contents of the remaining element(s) of VSR[XT] are undefined.
UX	Floating-Point Underflow Exception status flag, $FPSCR_{UX}$
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$.
VXIMZ	Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, $FPSCR_{VXIMZ}$.
VXISI	Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, $FPSCR_{VXISI}$.
XX	Float-Point Inexact Exception status flag, $FPSCR_{XX}$. The flag is a sticky version of $FPSCR_{FI}$. When $FPSCR_{FI}$ is set to a new value, the new value of $FPSCR_{XX}$ is set to the result of ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.

Table 85.Scalar Floating-Point Final Result with Negation (Continued)

VSX Scalar Negative Multiply-Add Single-Precision XX3-form

xsnmaddasp XT,XA,XB

60	T	A	B	129	AX	BX	TX
0	6	11	16	21	29	30	31

xsnmaddmsp XT,XA,XB

60	T	A	B	137	AX	BX	TX
0	6	11	16	21	29	30	31

```
reset_xflags()

if “xsnmaddasp” then do
   src1 ← VSR[32×AX+A].dword[0]
   src2 ← VSR[32×TX+T].dword[0]
   src3 ← VSR[32×BX+B].dword[0]
end
if “xsnmaddmsp” then do
   src1 ← VSR[32×AX+A].dword[0]
   src2 ← VSR[32×BX+B].dword[0]
   src3 ← VSR[32×TX+T].dword[0]
end

v      ← MultiplyAddDP(src1,src3,src2)
result ← NegateSP(RoundToSP(RN,v))

if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[32×TX+T].dword[0] ← ConvertToSP(result)
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For ***xsnmaddasp***, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsnmaddmsp***, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 86, “Actions for xsnmadd(a|m)sp,” on page 615.

src2 is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 86, “Actions for xsnmadd(a|m)sp,” on page 615.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is negated and placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 85, “Scalar Floating-Point Final Result with Negation,” on page 611.

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

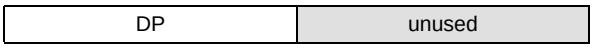
Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI VXIMZ

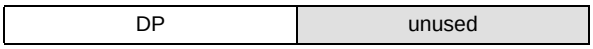
VSR Data Layout for xsnmadd(a|m)sp

src1 = VSR[XA]

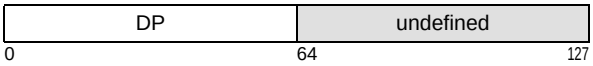
src2 = ***xsnmadda***(***dp***|***sp***) ? VSR[XT] : VSR[XB]



src3 = ***xsnmadda***(***dp***|***sp***) ? VSR[XB] : VSR[XT]



tgt = VSR[XT]



Part 1: Multiply

src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–NZF	p ← +Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← –Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add

p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–Zero	v ← –Infinity	v ← src2	v ← –Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← –Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsnmaddasp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsnmaddmsp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsnmaddasp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsnmaddmsp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 86.Actions for xsnmadd(a|m)sp

VSX Scalar Negative Multiply-Add Quad-Precision [*using round to Odd*] *X-form*

xsnmaddqp	VRT,VRA,VRB	(RO=0)
xsnmaddqpo	VRT,VRA,VRB	(RO=1)

63	VRT	VRA	VRB	452	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2   ← bfp_CONVERT_FROM_BFP128(VSR[VRT+32])
src3   ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_MULTIPLY_ADD(src1,src3,src2)
rnd    ← bfp_NEGATE(bfp_ROUND_TO_BFP128(RO, FPSCR.RN, v))
result ← bfp_CONVERT_TO_BFP128(rnd)

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(vximz_flag) then SetFX(FPSCR.VXIMZ)
if(vxisi_flag)  then SetFX(FPSCR.VXISI)
if(ox_flag)     then SetFX(FPSCR.OX)
if(ux_flag)     then SetFX(FPSCR.UX)
if(xx_flag)     then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag | vxisi_flag
ex_flag  ← FPSCR.VE & vx_flag

if ex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRT+32] represented in quad-precision format.

Let src3 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1, src2, or src3 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 is an Infinity value and src3 is a Zero value, or if src1 is a Zero value and src3 is an Infinity value, an Invalid Operation exception occurs and VXIMZ is set to 1.

If src2 and the product of src1 and src3 are Infinity values having opposite signs, an Invalid Operation exception occurs and VXISI is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src2 is a Quiet NaN, the result is src2.

Otherwise, if src3 is a Signalling NaN, the result is the Quiet NaN corresponding to src3.

Otherwise, if src3 is a Quiet NaN, the result is src3.

Otherwise, if src1 is an Infinity value and src3 is a Zero value, or if src1 is a Zero value and src3 is an Infinity value, the result is the default Quiet NaN[1].

Otherwise, if the product of src1 and src3, and src2 are Infinity values having opposite signs, the result is the default Quiet NaN.

Otherwise, do the following.

src1 is multiplied by src3, producing a product having unbounded significand precision and exponent range.

See part 1 of Table 69. "Actions for xsmadd(a|m)dp".

src2 is added to the product, producing a sum having unbounded range and precision.

See part 2 of Table 69. "Actions for xsmadd(a|m)dp".

If the intermediate result is *Tiny* (i.e., the unbiased exponent is less than -16382) and UE=0, the significand is shifted right N bits, where N is the difference between -16382 and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value -16382.

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is negated and placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

1. The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.

If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

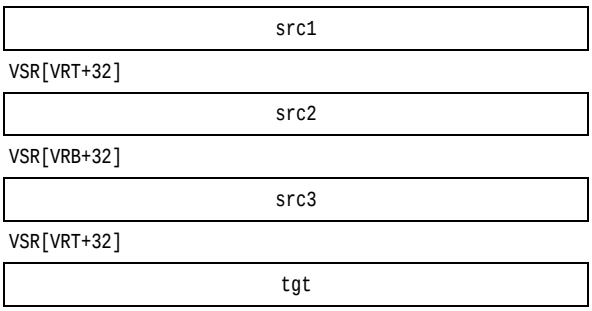
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered:

FPRF FR FI
FX VXSNAN VXIMZ VXISI OX UX XX

VSR Data Layout for xsnmaddqp[o]



Part 1: Multiply

src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	p ← +Infinity		p ← dQNaN vximz_flag ← 1			p ← -Infinity	p ← src3	p ← quiet(src3) vxsnan_flag ← 1
–NZF		p ← Mul(src1,src3)			p ← Mul(src1,src3)			
–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero		p ← -Zero		p ← dQNaN vximz_flag ← 1		
+Zero		p ← -Zero		p ← +Zero				
+NZF		p ← Mul(src1,src3)			p ← Mul(src1,src3)			
+Infinity	p ← -Infinity		p ← dQNaN vximz_flag ← 1			p ← +Infinity		
QNaN	p ← src1							p ← src1 vxsnan_flag ← 1
SNaN	p ← quiet(src1) vxsnan_flag ← 1							

Part 2: Add

p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	v ← -Infinity					v ← dQNaN vxisi_flag ← 1	v ← src2	v ← quiet(src2) vxsnan_flag ← 1
–NZF		v ← Add(p,src2)	v ← p		v ← Add(p,src2)			
–Zero	v ← src2		v ← -Zero	v ← Rezd	v ← src2			
+Zero			v ← Rezd	v ← +Zero				
+NZF		v ← Add(p,src2)	v ← p		v ← Add(p,src2)			
+Infinity	v ← dQNaN vxisi_flag ← 1					v ← +Infinity		
QNaN & src1 is a NaN	v ← p							v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN							v ← src2	v ← quiet(src2) vxsnan_flag ← 1

Explanation:

src1	The quad-precision floating-point value in VSR[VRA+32].
src2	The quad-precision floating-point value in VSR[VRT+32].
src3	The quad-precision floating-point value in VSR[VRB+32].
dQNaN	Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000_0000).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
quiet(x)	Return a QNaN with the payload of x.
Add(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
Mul(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 87.Actions for xsnmaddqp[o]

VSX Scalar Negative Multiply-Subtract Double-Precision XX3-form

xsnmsubadp XT,XA,XB

60	T	A	B	177	AX	BX	TX
0	6	11	16	21	29	30	31

xsnmsubmdp XT,XA,XB

60	T	A	B	185	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
reset_xflags()
src1         ← VSR[XA]{0:63}
src2         ← VSR[XT]{0:63}
src3         ← VSR[XB]{0:63}
src2         ← "xsnmsubadp" ? VSR[XT]{0:63} : VSR[XB]{0:63}
src3         ← "xsnmsubadp" ? VSR[XB]{0:63} : VSR[XT]{0:63}
v{0:inf}     ← MultiplyAddDP(src1,src3,NegateDP(src2))
result{0:63} ← NegateDP(RoundToDP(RN,v))
if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
vex_flag     ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

For ***xsnmsubadp***, do the following.
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsnmsubmdp***, do the following.
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 88.

src2 is negated and added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 88.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is negated and placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 85, "Scalar Floating-Point Final Result with Negation," on page 611.

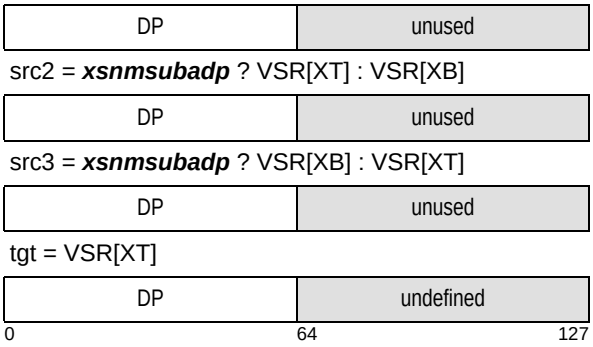
Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xsnmsub(a|m)dp

src1 = VSR[XA]



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← –Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← –src2	v ← +Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	For ***xsnmsubadp***, the double-precision floating-point value in doubleword element 0 of VSR[XT]. For ***xsnmsubmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XB].
src3	For ***xsnmsubadp***, the double-precision floating-point value in doubleword element 0 of VSR[XB]. For ***xsnmsubmdp***, the double-precision floating-point value in doubleword element 0 of VSR[XT].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 88.Actions for xsnmsub(a|m)dp

VSX Scalar Negative Multiply-Subtract Single-Precision XX3-form

xsnmsubasp XT,XA,XB

60	T	A	B	145	AX	BX	TX
0	6	11	16	21	29	30	31

xsnmsubmsp XT,XA,XB

60	T	A	B	153	AX	BX	TX
0	6	11	16	21	29	30	31

```
reset_xflags()

if "xsnmsubasp" then do
   src1 ← VSR[32×AX+A].dword[0]
   src2 ← VSR[32×TX+T].dword[0]
   src3 ← VSR[32×BX+B].dword[0]
end
if "xsnmsubmsp" then do
   src1 ← VSR[32×AX+A].dword[0]
   src2 ← VSR[32×BX+B].dword[0]
   src3 ← VSR[32×TX+T].dword[0]
end

v      ← MultiplyAddDP(src1,src3,NegateDP(src2)))
result ← NegateSP(RoundToSP(RN,v))

if(vxsnan_flag) then SetFX(VXSNAN)
if(vximz_flag)  then SetFX(VXIMZ)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vximz_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For ***xsnmsubasp***, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XT].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

For ***xsnmsubmsp***, do the following.
- Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].
- Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].
- Let src3 be the double-precision floating-point value in doubleword element 0 of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 89, "Actions for xsnmsub(a|m)sp," on page 624.

src2 is negated and added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 89, "Actions for xsnmsub(a|m)sp," on page 624.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is negated and placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 85, "Scalar Floating-Point Final Result with Negation," on page 611.

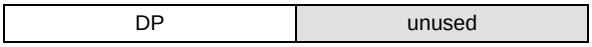
1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

Special Registers Altered

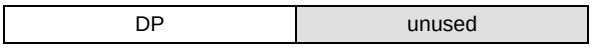
FPRF FR FI FX OX UX XX
VXSNAN VXISI VXIMZ

VSR Data Layout for xsnmsub(a|m)sp

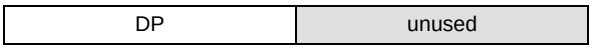
src1 = VSR[XA]



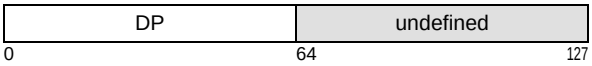
src2 = ***xsnmsubasp*** ? VSR[XT] : VSR[XB]



src3 = ***xsnmsubasp*** ? VSR[XB] : VSR[XT]



tgt = VSR[XT]



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← –Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← –src2	v ← +Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in VSR[XA].dword[0].
src2	For ***xsnmsubasp***, the double-precision floating-point value in VSR[XT].dword[0]. For ***xsnmsubmsp***, the double-precision floating-point value in VSR[XB].dword[0].
src3	For ***xsnmsubasp***, the double-precision floating-point value in VSR[XB].dword[0]. For ***xsnmsubmsp***, the double-precision floating-point value in VSR[XT].dword[0].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 89.Actions for xsnmsub(a|m)sp

VSX Scalar Negative Multiply-Subtract Quad-Precision [using round to Odd] X-form

xsnmsubqp VRT,VRA,VRB (RO=0)
xsnmsubqpo VRT,VRA,VRB (RO=1)

63	VRT	VRA	VRB	484	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2   ← bfp_CONVERT_FROM_BFP128(VSR[VRT+32])
src3   ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_MULTIPLY_ADD(src1, src3, bfp_NEGATE(src2))
rnd    ← bfp_NEGATE(bfp_ROUND_TO_BFP128(RO, FPSCR.RN, v))
result ← bfp_CONVERT_TO_BFP128(rnd)

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(vximz_flag)  then SetFX(FPSCR.VXIMZ)
if(vxisi_flag)  then SetFX(FPSCR.VXISI)
if(ox_flag)     then SetFX(FPSCR.OX)
if(ux_flag)     then SetFX(FPSCR.UX)
if(xx_flag)     then SetFX(FPSCR.XX)

vx_flag  ← vxsnan_flag | vximz_flag | vxisi_flag
ex_flag  ← FPSCR.VE & vx_flag

if ex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRT+32] represented in quad-precision format.

Let src3 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1, src2, or src3 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 is an Infinity value and src3 is a Zero value, or if src1 is a Zero value and src3 is an Infinity value, an Invalid Operation exception occurs and VXIMZ is set to 1.

If src2 and the product of src1 and src3 are Infinity values having same signs, an Invalid Operation exception occurs and VXISI is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src2 is a Quiet NaN, the result is src2.

Otherwise, if src3 is a Signalling NaN, the result is the Quiet NaN corresponding to src3.

Otherwise, if src3 is a Quiet NaN, the result is src3.

Otherwise, if src1 is an Infinity value and src3 is a Zero value, or if src1 is a Zero value and src3 is an Infinity value, the result is the default Quiet NaN[1].

Otherwise, if the product of src1 and src3, and src2 are Infinity values having same signs, the result is the default Quiet NaN.

Otherwise, do the following.

src1 is multiplied by src3, producing a product having unbounded significand precision and exponent range.

See part 1 of Table 80. "Actions for xsmsubqp[o]".

src2 is negated and added to the product, producing a sum having unbounded range and precision.

See part 2 of Table 80. "Actions for xsmsubqp[o]".

If the intermediate result is *Tiny* (i.e., the unbiased exponent is less than -16382) and UE=0, the significand is shifted right N bits, where N is the difference between -16382 and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value -16382.

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is negated and placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FR and FI are set to 0.

1. The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.

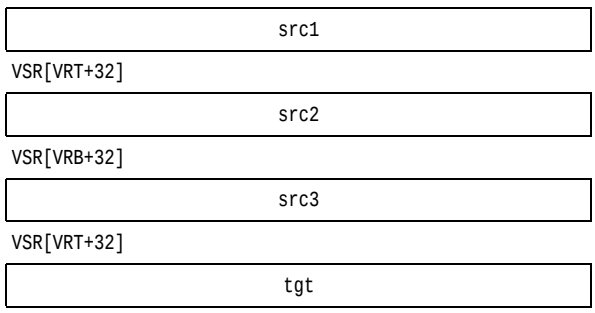
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered:

FPRF FR FI
FX VXSNAN VXIMZ VXISI OX UX XX

VSR Data Layout for xsnmsubqp[o]



Part 1: Multiply

src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	p ← +Infinity		p ← dQNaN vximz_flag ← 1			p ← -Infinity	p ← src3	p ← quiet(src3) vxsnan_flag ← 1
–NZF		p ← Mul(src1,src3)	p ← +Zero	p ← -Zero	p ← Mul(src1,src3)			
–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero			p ← -Zero	p ← dQNaN vximz_flag ← 1		
+Zero		p ← -Zero	p ← -Zero	p ← +Zero	p ← +Zero			
+NZF		p ← Mul(src1,src3)			p ← Mul(src1,src3)			
+Infinity	p ← -Infinity		p ← dQNaN vximz_flag ← 1			p ← +Infinity		
QNaN	p ← src1							p ← src1 vxsnan_flag ← 1
SNaN	p ← quiet(src1) vxsnan_flag ← 1							

Part 2: Subtract

p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	v ← dQNaN vxisi_flag ← 1					v ← -Infinity	v ← src2	v ← quiet(src2) vxsnan_flag ← 1
–NZF		v ← sub(p,src2)	v ← p		v ← sub(p,src2)			
–Zero		v ← -src2	v ← Rezd	v ← -Zero	v ← -src2			
+Zero			v ← +Zero	v ← Rezd				
+NZF		v ← sub(p,src2)	v ← p		v ← sub(p,src2)			
+Infinity	v ← +Infinity					v ← dQNaN vxisi_flag ← 1		
QNaN & src1 is a NaN	v ← p							v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN							v ← src2	v ← quiet(src2) vxsnan_flag ← 1

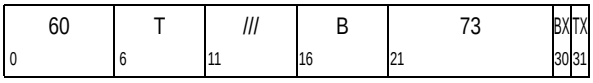
Explanation:

src1	The quad-precision floating-point value in VSR[VRA+32].
src2	The quad-precision floating-point value in VSR[VRT+32].
src3	The quad-precision floating-point value in VSR[VRB+32].
dQNaN	Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
quiet(x)	Return a QNaN with the payload of x.
sub(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
Mul(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 90.Actions for xsnmsubqp[o]

VSX Scalar Round to Double-Precision Integer using round to Nearest Away XX2-form

xsrdpi XT,XB




```
XT          ← TX || T
XB          ← BX || B
reset_xflags()
result{0:63} ← RoundToDPIntegerNearAway(VSR[XB]{0:63})
if(vxsnan_flag) then SetFX(VXSNAN)
FR          ← 0b0
FI          ← 0b0
vex_flag    ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassFP(result)
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to an integer using the rounding mode Round to Nearest Away.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

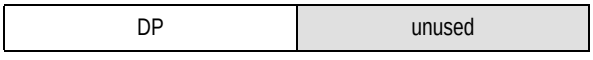
If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

Special Registers Altered

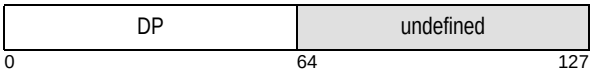
FPRF FR=0b0 FI=0b0 FX VXSNAN

VSR Data Layout for xsrdpi

src = VSR[XB]



tgt = VSR[XT]

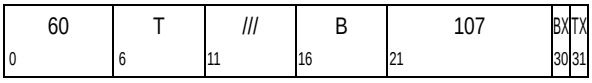


Programming Note

This instruction can be used to operate on a single-precision source operand.

VSX Scalar Round to Double-Precision Integer exact using Current rounding mode XX2-form

xsrdpic XT,XB




```
XT          ← TX || T
XB          ← BX || B
reset_xflags()
src         ← VSR[XB]{0:63}
if(RN=0b00) then result{0:63} ← RoundToDPIntegerNearEven(src)
if(RN=0b01) then result{0:63} ← RoundToDPIntegerTrunc(src)
if(RN=0b10) then result{0:63} ← RoundToDPIntegerCeil(src)
if(RN=0b11) then result{0:63} ← RoundToDPIntegerFloor(src)
if(vxsnan_flag) then SetFX(VXSNAN)
if(xx_flag)     then SetFX(XX)
vex_flag    ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to an integer using the rounding mode specified by RN.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

Special Registers Altered

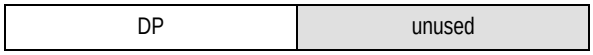
FPRF FR FI FX XX VXSNAN

Programming Note

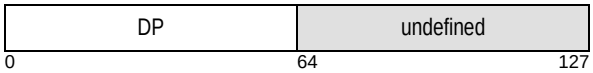
This instruction can be used to operate on a single-precision source operand.

VSR Data Layout for xsrdpic

src = VSR[XB]

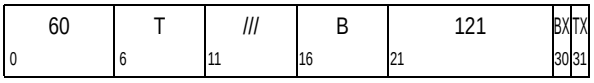


tgt = VSR[XT]



VSX Scalar Round to Double-Precision Integer using round toward -Infinity XX2-form

xsrdpim XT,XB




```
XT          ← TX || T
XB          ← BX || B
reset_xflags()
result{0:63} ← RoundToDPIntegerFloor(VSR[XB]{0:63})
if(vxsnan_flag) then SetFX(VXSNAN)
FR          ← 0b0
FI          ← 0b0
vex_flag    ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to an integer using the rounding mode Round toward -Infinity.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

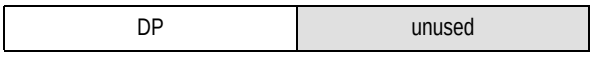
If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

Special Registers Altered

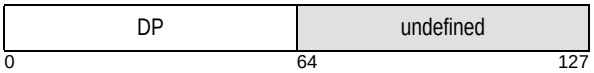
FPRF FR=0b0 FI=0b0 FX VXSNAN

VSR Data Layout for xsrdpim

src = VSR[XB]



tgt = VSR[XT]

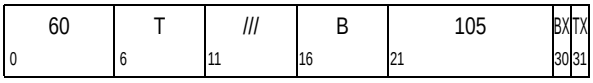


Programming Note

This instruction can be used to operate on a single-precision source operand.

VSX Scalar Round to Double-Precision Integer using round toward +Infinity XX2-form

xsrdpip XT,XB




```
XT          ← TX || T
XB          ← BX || B
reset_xflags()
result{0:63} ← RoundToDPIntegerCeil(VSR[XB]{0:63})
if(vxsnan_flag) then SetFX(VXSNAN)
FR          ← 0b0
FI          ← 0b0
vex_flag    ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to an integer using the rounding mode Round toward +Infinity.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

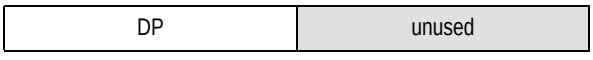
If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

Special Registers Altered

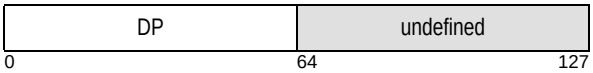
FPRF FR=0b0 FI=0b0 FX VXSNAN

VSR Data Layout for xsrdpip

src = VSR[XB]



tgt = VSR[XT]

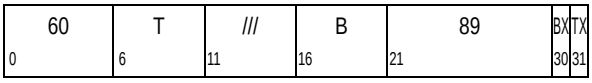


Programming Note

This instruction can be used to operate on a single-precision source operand.

VSX Scalar Round to Double-Precision Integer using round toward Zero XX2-form

xsrdpiz XT,XB




```
XT          ← TX || T
XB          ← BX || B
reset_xflags()
result{0:63} ← RoundToDPIntegerTrunc(VSR[XB]{0:63})
if(vxsnan_flag) then SetFX(VXSNAN)
FR          ← 0b0
FI          ← 0b0
vex_flag    ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to an integer using the rounding mode Round toward Zero.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to 0. FI is set to 0.

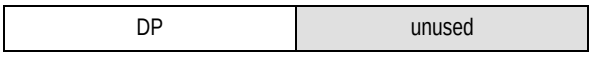
If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

Special Registers Altered

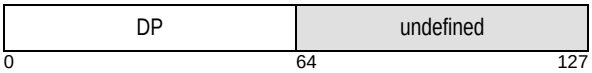
FPRF FR=0b0 FI=0b0 FX VXSNAN

VSR Data Layout for xsrdpiz

src = VSR[XB]



tgt = VSR[XT]



Programming Note

This instruction can be used to operate on a single-precision source operand.

VSX Scalar Reciprocal Estimate Double-Precision XX2-form

xsredp XT,XB

60	T	///	B	90	BX	TX
0	6	11	16	21	30	31

```
XT           ← TX || T
XB           ← BX || B
reset_xflags()
v{0:inf}     ← ReciprocalEstimateDP(VSR[XB]{0:63})
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(zx_flag)     then SetFX(ZX)
vex_flag     ← VE & vxsnan_flag
zex_flag     ← ZE & zx_flag

if( ~vex_flag & ~zex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← 0bU
   FI      ← 0bU
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

A double-precision floating-point estimate of the reciprocal of src is placed into doubleword element 0 of VSR[XT] in double-precision format.

Unless the reciprocal of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of src. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\text{src}}}{\frac{1}{\text{src}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
–Infinity	–Zero	None
–Zero	–Infinity[1]	ZX
+Zero	+Infinity[1]	ZX
+Infinity	+Zero	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

1. No result if ZE=1.
2. No result if VE=1.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FPRF FR=0bU FI=0bU FX OX UX XX=0bU VXSNAN

VSR Data Layout for xsredp

src = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

VSX Scalar Reciprocal Estimate Single-Precision XX2-form

xsresp XT,XB

60	T	///	B	26	BX	TX
0	6	11	16	21	30	31

```
reset_xflags()

src    ← VSR[32×BX+B].dword[0]
v      ← ReciprocalEstimateDP(src)
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(0bU)         then SetFX(XX)
if(zx_flag)     then SetFX(ZX)

vex_flag ← VE & vxsnan_flag
zex_flag ← ZE & zx_flag

if( ~vex_flag & ~zex_flag ) then do
   VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← 0bU
   FI   ← 0bU
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

A single-precision floating-point estimate of the reciprocal of src is placed into doubleword element 0 of VSR[XT] in double-precision format.

Unless the reciprocal of src would be a zero, an infinity, the result of a trap-disabled Overflow exception, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of src. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\text{src}}}{\frac{1}{\text{src}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
–Infinity	–Zero	None
–Zero	–Infinity[1]	ZX
+Zero	+Infinity[1]	ZX
+Infinity	+Zero	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

1. No result if ZE=1.
2. No result if VE=1.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FPRF FR=0bU FI=0bU FX OX UX ZX XX=0bU VXSNAN

VSR Data Layout for xsresp

src = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

VSX Scalar Round to Quad-Precision Integer [*with Inexact*] *Z23-form*

xsrqpi	R,VRT,VRB,RMC	(EX=0)
xsrqpix	R,VRT,VRB,RMC	(EX=1)

63	VRT	///	R	VRB	RMC	5	EX
0	6	11	15	16	21	23	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

if R=0 then do
   if RMC=0b00 then           // Round to Nearest Away
      rmode ← 0b100
   if RMC=0b11 then do
      if FPSCR.RN=0b00 then   // Round to Nearest Even
         rmode ← 0b000
      if FPSCR.RN=0b01 then   // Round towards Zero
         rmode ← 0b001
      if FPSCR.RN=0b10 then   // Round towards +Infinity
         rmode ← 0b010
      if FPSCR.RN=0b11 then   // Round towards -Infinity
         rmode ← 0b011
   end
end
else do // R=1
   if RMC=0b00 then           // Round to Nearest Even
      rmode ← 0b000
   if RMC=0b01 then           // Round towards Zero
      rmode ← 0b001
   if RMC=0b10 then           // Round towards +Infinity
      rmode ← 0b010
   if RMC=0b11 then           // Round towards -Infinity
      rmode ← 0b011
end

src ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])

if src.class.SNaN then do
   result ← bfp_CONVERT_TO_BFP128(bfp_QUIET(src))
   vxsnan_flag ← 1
end
else if src.class.QNaN     |
        src.class.Infinity |
        src.class.Zero  then
   result ← bfp_CONVERT_TO_BFP128(src)
else do
   rnd    ← bfp_ROUND_TO_INTEGER(rmode, src)
   result ← bfp_CONVERT_TO_BFP128(rnd)
end

if(vxsnan_flag)  then SetFX(FPSCR.VXSNAN)
if(xx_flag & EX) then SetFX(FPSCR.XX)

ex_flag  ← FPSCR.VE & vxsnan_flag

if ex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← EX & (vxsnan_flag=0) & inc_flag
FPSCR.FI ← EX & (vxsnan_flag=0) & xx_flag
```

Let R and RMC specify the rounding mode as follows.

R	RMC	FPSCR.RN	Rounding Mode
0	00	–	Round to Nearest Away
0	01	–	reserved
0	10	–	reserved
0	11	00	Round to Nearest Even
0	11	01	Round towards Zero
0	11	10	Round towards +Infinity
0	11	11	Round towards -Infinity
1	00	–	Round to Nearest Even
1	01	–	Round towards Zero
1	10	–	Round towards +Infinity
1	11	–	Round towards -Infinity

Let src be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If src is a Signalling NaN, an Invalid Operation exception occurs, VXSNAN is set to 1, and the result is the Quiet NaN corresponding to the Signalling NaN.

Otherwise, if src is a Quiet NaN, an Infinity, or a Zero, then the result is src.

Otherwise, src is rounded to an integer using the rounding mode rmode.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result.

For ***xsrqpi***, FR is set to 0, FI is set to 0, and XX is not set by an Inexact exception.

For ***xsrqpix***, FR is set to indicate if the result was incremented when rounded, FI is set to indicate the result is inexact, and XX is set by an Inexact exception.

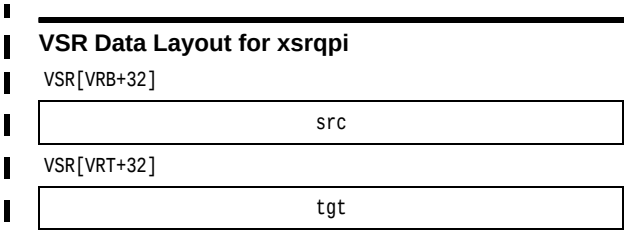
If a trap-disabled Invalid Operation exception occurs, FPRF is set to an undefined value.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified.

Special Registers Altered:

FPRF VXSNAN FX
FR (set to 0) FI (set to 0) . . . . . (if ***xsrqpi***)
FR FI XX . . . . . (if ***xsrqpix***)

VSR Data Layout for xsrqpi



VSX Scalar Round Quad-Precision to Double-Extended Precision Z23-form

xsrqpxp R,VRT,VRB,RMC

63	VRT	///	R	VRB	RMC	37	/
0	6	11	15	16	21	23	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

if R=0 then do
   if RMC=0b00 then           // Round to Nearest Away
      rmode ← 0b100
   if RMC=0b11 then do
      if FPSCR.RN=0b00 then   // Round to Nearest Even
         rmode ← 0b000
      if FPSCR.RN=0b01 then   // Round towards Zero
         rmode ← 0b001
      if FPSCR.RN=0b10 then   // Round towards +Infinity
         rmode ← 0b010
      if FPSCR.RN=0b11 then   // Round towards -Infinity
         rmode ← 0b011
   end
end
else do // R=1
   if RMC=0b00 then           // Round to Nearest Even
      rmode ← 0b000
   if RMC=0b01 then           // Round towards Zero
      rmode ← 0b001
   if RMC=0b10 then           // Round towards +Infinity
      rmode ← 0b010
   if RMC=0b11 then           // Round towards -Infinity
      rmode ← 0b011
end

src    ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
rnd    ← bfp_ROUND_TO_BFP80(rmode,src)
result ← bfp_CONVERT_TO_BFP128(rnd)

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(ox_flag)     then SetFX(FPSCR.OX)
if(ux_flag)     then SetFX(FPSCR.UX)
if(xx_flag)     then SetFX(FPSCR.XX)

ex_flag  ← FPSCR.VE & vxsnan_flag

if ex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vxsnan_flag=0) & inc_flag
FPSCR.FI ← (vxsnan_flag=0) & xx_flag
```

Let R and RMC specify the rounding mode as follows.

R	RMC	FPSCR.RN	Rounding Mode
0	00	–	Round to Nearest Away
0	01	–	reserved
0	10	–	reserved
0	11	00	Round to Nearest Even
0	11	01	Round to Zero
0	11	10	Round to +Infinity
0	11	11	Round to -Infinity
1	00	–	Round to Nearest Even
1	01	–	Round to Zero
1	10	–	Round to +Infinity
1	11	–	Round to -Infinity

Let src be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If src is a Signalling NaN, an Invalid Operation exception occurs, VXSNAN is set to 1, and the result is the Quiet NaN corresponding to the Signalling NaN, with the significand truncated to double-extended-precision.

Otherwise, if src is a Quiet NaN, then the result is src with the significand truncated to double-extended-precision.

Otherwise, if src is an Infinity or a Zero, the result is src.

Otherwise, src is rounded to double-extended precision (i.e., 15-bit exponent range and 64-bit significand precision) using the specified rounding mode.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FPRF is set to an undefined value, and FR and FI are set to 0.

If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered:

FPRF FR FI FX VXSNAN OX UX XX

VSR Data Layout for xsrqpxp

VSR[VRB+32]

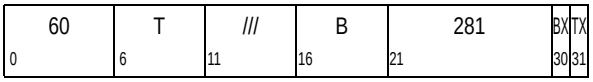
src

VSR[VRT+32]

tgt

VSX Scalar Round to Single-Precision XX2-form

xsrsp XT,XB




```
reset_xflags()

src    ← VSR[32×BX+B].dword[0]
result ← RoundToSP(RN,src)

if(vxsnan_flag) then SetFX(VXSNAN)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & vxsnan_flag

if( ~vex_flag ) then do
   VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format.

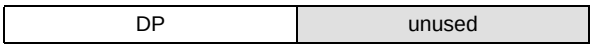
If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified.

Special Registers Altered

FPRF FR FI FX OX UX XX VXSNAN

VSR Data Layout for xsrsp

src = VSR[XB]



tgt = VSR[XT]

VSX Scalar Reciprocal Square Root Estimate Double-Precision XX2-form

xsrsqrtedp XT,XB

60	T	///	B	74	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
reset_xflags()
v{0:inf}    ← ReciprocalSquareRootEstimateDP(VSR[XB]{0:63})
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxsqrt_flag) then SetFX(VXSQRT)
if(zx_flag)     then SetFX(ZX)
vex_flag    ← VE & (vxsnan_flag | vxsqrt_flag)
zex_flag    ← ZE & zx_flag

if( ~vex_flag & ~zex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← 0bU
   FI      ← 0bU
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

A double-precision floating-point estimate of the reciprocal square root of src is placed into doubleword element 0 of VSR[XT] in double-precision format.

Unless the reciprocal of the square root of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of src. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\sqrt{\text{src}}}}{\frac{1}{\sqrt{\text{src}}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
–Infinity	QNaN[1]	VXSQRT
–Finite	QNaN[1]	VXSQRT
–Zero	–Infinity[2]	ZX
+Zero	+Infinity[2]	ZX
+Infinity	+Zero	None
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

1. No result if VE=1.
2. No result if ZE=1.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FPRF FR=0bU FI=0bU FX XX=0bU VXSNAN VXSQRT

VSR Data Layout for xsrsqrtedp

src = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

VSX Scalar Reciprocal Square Root Estimate Single-Precision XX2-form

xsrsqrtesp XT,XB

60	T	///	B	10	BX	TX
0	6	11	16	21	30	31

```
reset_xflags()

src    ← VSR[32×BX+B].dword[0]
v      ← ReciprocalSquareRootEstimateDP(src)
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vxsqrt_flag) then SetFX(VXSQRT)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(0bU)         then SetFX(XX)
if(zx_flag)     then SetFX(ZX)

vex_flag ← VE & (vxsnan_flag | vxsqrt_flag)
zex_flag ← ZE & zx_flag

if( ~vex_flag & ~zex_flag ) then do
   VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← 0bU
   FI   ← 0bU
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

A single-precision floating-point estimate of the reciprocal square root of src is placed into doubleword element 0 of VSR[XT] in double-precision format.

Unless the reciprocal of the square root of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of src. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\sqrt{\text{src}}}}{\frac{1}{\sqrt{\text{src}}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
–Infinity	QNaN[1]	VXSQRT
–Finite	QNaN[1]	VXSQRT
–Zero	–Infinity[2]	ZX
+Zero	+Infinity[2]	ZX
+Infinity	+Zero	None
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

1. No result if VE=1.
2. No result if ZE=1.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to an undefined value. FI is set to an undefined value.

If a trap-enabled invalid operation exception or a trap-enabled zero divide exception occurs, VSR[XT] and FPRF are not modified.

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FPRF FR=0bU FI=0bU FX OX UX ZX XX=0bU VXSNAN VXSQRT

VSR Data Layout for xsrsqrtesp

src = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

VSX Scalar Square Root Double-Precision XX2-form

xssqrtdp XT,XB

60	T	///	B	75	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
reset_xflags()
v{0:inf}    ← SquareRootFP(VSR[XB]{0:63})
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxsqrt_flag) then SetFX(VXSQRT)
if(xx_flag)     then SetFX(XX)
vex_flag    ← VE & (vxsnan_flag | vxsqrt_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

The unbounded-precision square root of src is produced.

See Table 91.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

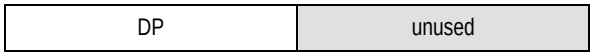
See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered

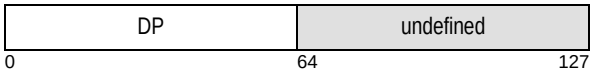
FPRF FR FI FX XX VXSNAN VXSQRT

VSR Data Layout for xssqrtdp

src = VSR[XB]



tgt = VSR[XT]



src							
-Infinity	**-NZF**	**-Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
v ← dQNaN vxsqrt_flag ← 1	v ← dQNaN vxsqrt_flag ← 1	v ← +Zero	v ← +Zero	v ← SQRT(src)	v ← +Infinity	v ← src	v ← Q(src) vxsnan_flag ← 1

Explanation:

src	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (0x7FF8_0000_0000_0000).
NZF	Nonzero finite number.
SQRT(x)	The unbounded-precision square root of the floating-point value x.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 91.Actions for xssqrtdp

VSX Scalar Square Root Quad-Precision [using round to Odd] X-form

xssqrtqp VRT,VRB (RO=0)
xssqrtqpo VRT,VRB (RO=1)

63	VRT	27	VRB	804	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src    ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_SQUARE_ROOT(src)
rnd    ← bfp_ROUND_TO_BFP128(RO,FPSCR.RN,v)
result ← bfp_CONVERT_TO_BFP128(rnd)

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(vxsqrt_flag) then SetFX(FPSCR.VXSQRT)
if(xx_flag)     then SetFX(FPSCR.XX)

vx_flag ← vxsnan_flag | vxsqrt_flag
ex_flag ← FPSCR.VE & vx_flag

if ex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If src is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src is a negative, non-zero value, an Invalid Operation exception occurs and VXSQRT is set to 1.

If src is a Signalling NaN, the result is the Quiet NaN corresponding to src.

Otherwise, if src is a Quiet NaN, the result is src.

Otherwise, if src is a negative value, the result is the default Quiet NaN[1].

Otherwise, do the following.

The normalized square root of src is produced with unbounded significand precision and exponent range.

See Table 92, “Actions for xssqrtqp[o],” on page 643.

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.

See Section 7.3.2.6, “Rounding” on page 381 for a description of rounding modes.

If there is loss of precision, an Inexact exception occurs.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FPRF is set to an undefined value, and FR and FI are set to 0.

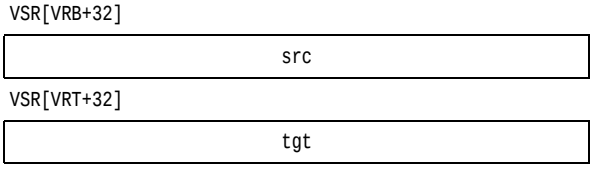
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered:

FPRF FR FI FX VXSNAN VXSQRT XX

VSR Data Layout for xssqrtqp[o]



1. The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.

src							
-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
v ← dQNaN vxsqrt_flag ← 1	v ← dQNaN vxsqrt_flag ← 1	v ← +Zero	v ← +Zero	v ← sqrt(src)	v ← +Infinity	v ← src	v ← quiet(src) vxsnan_flag ← 1

Explanation:

src	The quad-precision floating-point value in VSR[VRB+32].
dQNaN	Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000_0000).
NZF	Nonzero finite number.
sqrt(x)	Return the normalized[1] square root of floating-point value x, having unbounded significand precision and exponent range.
quiet(x)	Convert x to the corresponding Quiet NaN.
v	The intermediate result having unbounded significand precision and unbounded exponent range.

Table 92. Actions for xssqrtqp[o]

1. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSX Scalar Square Root Single-Precision XX2-form

xssqrtsp XT,XB

60	T	///	B	11	BX	TX
0	6	11	16	21	30	31

```
reset_xflags()

src    ← VSR[32×BX+B].dword[0]
v      ← SquareRootDP(src)
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vxsqrt_flag) then SetFX(VXSQRT)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vxsqrt_flag)

if( ~vex_flag ) then do
   VSR[32×TX+T].dword[0] ← ConvertToDP(result)
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

The unbounded-precision square root of src is produced.

See Table 91.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element 0 of VSR[XT] in double-precision format.

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

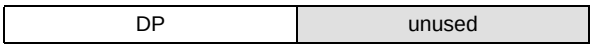
See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXSQRT

VSR Data Layout for xssqrtsp

src = VSR[XB]



tgt = VSR[XT]

0 64 127

src							
-Infinity	**-NZF**	**-Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
v ← dQNaN vxsqrt_flag ← 1	v ← dQNaN vxsqrt_flag ← 1	v ← +Zero	v ← +Zero	v ← SQRT(src)	v ← +Infinity	v ← src	v ← Q(src) vxsnan_flag ← 1

Explanation:

src	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (0x7FF8_0000_0000_0000).
NZF	Nonzero finite number.
SQRT(x)	The unbounded-precision and exponent range square root of the floating-point value x.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 93.Actions for xssqrtsp

VSX Scalar Subtract Double-Precision XX3-form

xssubdp XT,XA,XB

60	T	A	B	40	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT           ← TX || T
XA           ← AX || A
XB           ← BX || B
reset_xflags()
src1         ← VSR[XA]{0:63}
src2         ← VSR[XB]{0:63}
v{0:inf}     ← AddDP(src1,NegateDP(src2))
result{0:63} ← RoundToDP(RN,v)
if(vxsnan_flag) then SetFX(VXSNAN)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)
vex_flag     ← VE & (vxsnan_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[XT] ← result || 0xUUUU_UUUU_UUUU_UUUU
   FPRF    ← ClassDP(result)
   FR      ← inc_flag
   FI      ← xx_flag
end
else do
   FR      ← 0b0
   FI      ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src2 is negated and added[1] to src1, producing a sum having unbounded range and precision.

See Table 94.

The sum is normalized[2].

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI

VSR Data Layout for xssubdp

src1 = VSR[XA]

DP	unused

src2 = VSR[XB]

DP	unused

tgt = VSR[XT]

DP	undefined

0 64 127

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← +Infinity	v ← –src2	v ← –Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← +Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
S(x,y)	The floating-point value y is negated and then added to the floating-point value x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 94.Actions for xssubdp

VSX Scalar Subtract Quad-Precision [*using round to Odd*] *X-form*

xssubqp VRT,VRA,VRB (RO=0)
xssubqpo VRT,VRA,VRB (RO=1)

63	VRT	VRA	VRB	516	RO
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_xflags()

src1   ← bfp_CONVERT_FROM_BFP128(VSR[VRA+32])
src2   ← bfp_CONVERT_FROM_BFP128(VSR[VRB+32])
v      ← bfp_ADD(src1, bfp_NEGATE(src2))
rnd    ← bfp_ROUND_TO_BFP128(RO,FPSCR.RN,v)
result ← bfp_CONVERT_TO_BFP128(rnd)

if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
if(vxisi_flag)  then SetFX(FPSCR.VXISI)
if(ox_flag)     then SetFX(FPSCR.OX)
if(ux_flag)     then SetFX(FPSCR.UX)
if(xx_flag)     then SetFX(FPSCR.XX)

vx_flag ← vxsnan_flag | vxisi_flag
ex_flag ← FPSCR.VE & vx_flag

if ex_flag=0 then do
   VSR[VRT+32] ← result
   FPSCR.FPRF  ← fprf_CLASS_BFP128(result)
end
FPSCR.FR ← (vx_flag=0) & inc_flag
FPSCR.FI ← (vx_flag=0) & xx_flag
```

Let src1 be the floating-point value in VSR[VRA+32] represented in quad-precision format.

Let src2 be the floating-point value in VSR[VRB+32] represented in quad-precision format.

If either src1 or src2 is a Signalling NaN, an Invalid Operation exception occurs and VXSNAN is set to 1.

If src1 and src2 are Infinity values having same signs, an Invalid Operation exception occurs and VXISI is set to 1.

If src1 is a Signalling NaN, the result is the Quiet NaN corresponding to src1.

Otherwise, if src1 is a Quiet NaN, the result is src1.

Otherwise, if src2 is a Signalling NaN, the result is the Quiet NaN corresponding to src2.

Otherwise, if src2 is a Quiet NaN, the result is src2.

Otherwise, if src1 and src2 are Infinity values having same signs, the result is the default Quiet NaN[1].

Otherwise, do the following.

The normalized sum of the negation of src2 added to src1 is produced with unbounded significand precision and exponent range.

See Table 95, "Actions for xssubqp[o]," on page 648.

If the intermediate result is *Tiny* (i.e., the unbiased exponent is less than -16382) and UE=0, the significand is shifted right N bits, where N is the difference between -16382 and the unbiased exponent of the intermediate result. The exponent of the intermediate result is set to the value -16382.

If RO=1, let the rounding mode be Round to Odd. Otherwise, let the rounding mode be specified by RN. Unless the result is an Infinity or a Zero, the intermediate result is rounded to quad-precision using the specified rounding mode.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is placed into VSR[VRT+32] in quad-precision format.

FPRF is set to the class and sign of the result. FR is set to indicate if the rounded result was incremented. FI is set to indicate the result is inexact.

If a trap-disabled Invalid Operation exception occurs, FPRF is set to an undefined value, and FR and FI are set to 0.

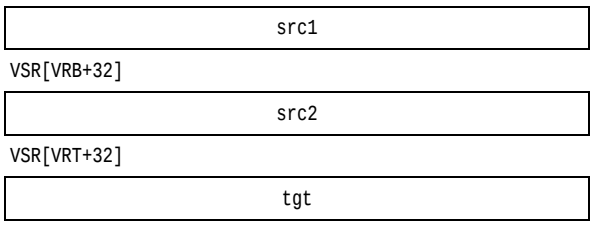
If a trap-enabled Invalid Operation exception occurs, VSR[VRT+32] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, "VSX Scalar Floating-Point Final Result," on page 516.

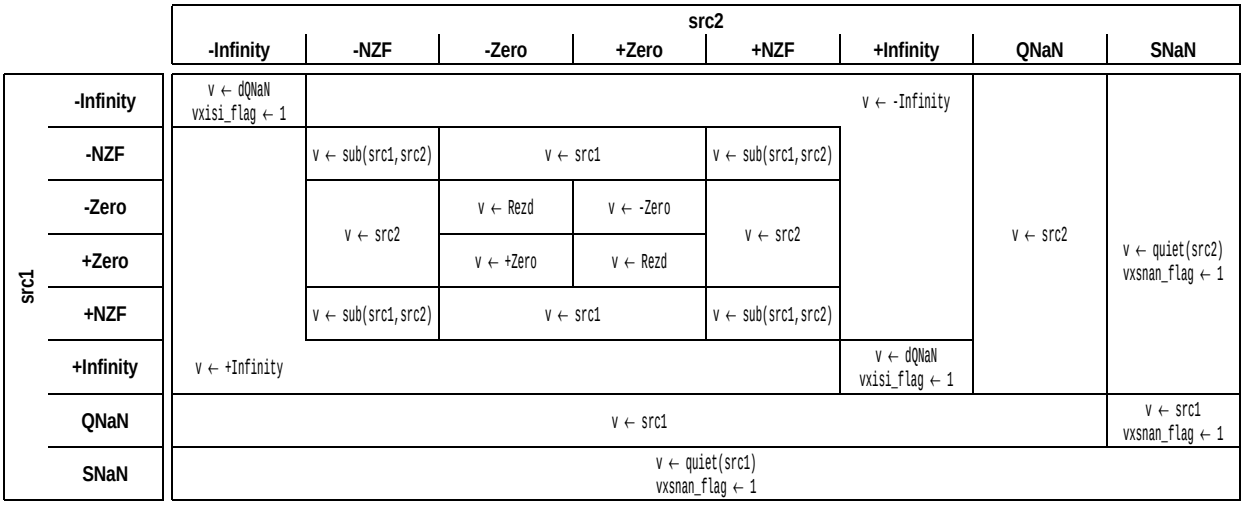
Special Registers Altered:

FPRF FR FI FX VXSNAN VXISI OX UX XX

VSR Data Layout for xssubqp[o]



1. The quad-precision default Quiet NaN is the value, 0x7FFF_8000_0000_0000_0000_0000_0000.



Explanation:

src1	The quad-precision floating-point value in VSR[VRA+32].
src2	The quad-precision floating-point value in VSR[VRB+32].
dQNaN	Default quiet NaN (0x7FFF_8000_0000_0000_0000_0000_0000).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (subtraction of two finite numbers having same magnitude and signs).
sub(x,y)	Return the normalized difference of floating-point value x and floating-point value y, having unbounded significand precision and exponent range.
	Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
quiet(x)	Convert x to the corresponding Quiet NaN.
v	The intermediate result having unbounded significand precision and unbounded exponent range.

Table 95. Actions for xssubqp[o]

VSX Scalar Subtract Single-Precision XX3-form

xssubsp XT,XA,XB

60	T	A	B	8	AX	BX	TX
0	6	11	16	21	30	30	31

```
reset_xflags()

src1   ← VSR[32×AX+A].dword[0]
src2   ← VSR[32×BX+B].dword[0]
v      ← AddDP(src1,NegateDP(src2))
result ← RoundToSP(RN,v)

if(vxsnan_flag) then SetFX(VXSNAN)
if(vxisi_flag)  then SetFX(VXISI)
if(ox_flag)     then SetFX(OX)
if(ux_flag)     then SetFX(UX)
if(xx_flag)     then SetFX(XX)

vex_flag ← VE & (vxsnan_flag | vxisi_flag)

if( ~vex_flag ) then do
   VSR[32×TX+T].dword[0] ← ConvertSPtoSP64(result)
   VSR[32×TX+T].dword[1] ← 0xUUUU_UUUU_UUUU_UUUU
   FPRF ← ClassSP(result)
   FR   ← inc_flag
   FI   ← xx_flag
end
else do
   FR   ← 0b0
   FI   ← 0b0
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

src2 is negated and added[1] to src1, producing the sum, v, having unbounded range and precision.

See Table 96, “Actions for xssubsp,” on page 650.

v is normalized[2] and rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element 0 of VSR[XT].

The contents of doubleword element 1 of VSR[XT] are undefined.

FPRF is set to the class and sign of the result as represented in single-precision format. FR is set to indicate if the result was incremented when rounded. FI is set to indicate the result is inexact.

If a trap-enabled invalid operation exception occurs, VSR[XT] and FPRF are not modified, and FR and FI are set to 0.

See Table 51, “VSX Scalar Floating-Point Final Result,” on page 516.

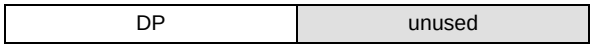
Special Registers Altered

FPRF FR FI FX OX UX XX
VXSNAN VXISI

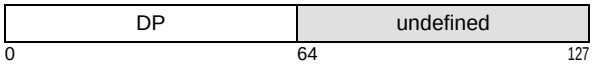
VSR Data Layout for xssubsp

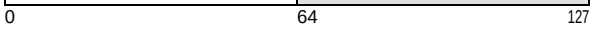
src1 = VSR[XA]

src2 = VSR[XB]



tgt = VSR[XT]





1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← +Infinity	v ← –src2	v ← –Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← +Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element 0 of VSR[XA].
src2	The double-precision floating-point value in doubleword element 0 of VSR[XB].
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
S(x,y)	The floating-point value y is negated and then added to the floating-point value x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 96.Actions for xssubsp

VSX Scalar Test for software Divide Double-Precision XX3-form

xstdivdp BF,XA,XB

60	BF	//	A	B	61	AX	BX	/
0	6	9	11	16	21	29	30	31

```
XA          ← AX || A
XB          ← BX || B
src1        ← VSR[XA]{0:63}
src2        ← VSR[XB]{0:63}
e_a         ← VSR[XA]{1:11} - 1023
e_b         ← VSR[XB]{1:11} - 1023
fe_flag     ← IsNaN(src1) | IsInf(src1) |
              IsNaN(src2) | IsInf(src2) | IsZero(src2) |
              ( e_b <= -1022 ) |
              ( e_b >=  1021 ) |
              ( !IsZero(src1) & ( (e_a - e_b) >=  1023 ) ) |
              ( !IsZero(src1) & ( (e_a - e_b) <= -1021 ) ) |
              ( !IsZero(src1) & ( e_a <= -970 ) )
fg_flag     ← IsInf(src1) | IsInf(src2) |
              IsZero(src2) | IsDen(src2)
fl_flag     ← xsredp_error() <= 2^-14
CR[BF]      ← 0b1 || fg_flag || fe_flag || 0b0
```

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let src1 be the double-precision floating-point value in doubleword element 0 of VSR[XA].

Let src2 be the double-precision floating-point value in doubleword element 0 of VSR[XB].

Let e_a be the unbiased exponent of src1.
Let e_b be the unbiased exponent of src2.

fe_flag is set to 1 for any of the following conditions.
- src1 is a NaN or an infinity.
- src2 is a zero, a NaN, or an infinity.
- e_b is less than or equal to -1022.
- e_b is greater than or equal to 1021.
- src1 is not a zero and the difference, e_a - e_b, is greater than or equal to 1023.
- src1 is not a zero and the difference, e_a - e_b, is less than or equal to -1021.
- src1 is not a zero and e_a is less than or equal to -970

Otherwise fe_flag is set to 0.

fg_flag is set to 1 for any of the following conditions.
- src1 is an infinity.
- src2 is a zero, an infinity, or a denormalized value.

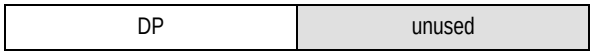
Otherwise fg_flag is set to 0.

CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

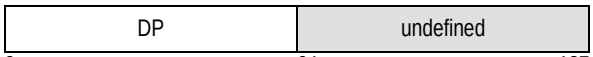
Special Registers Altered

VSR Data Layout for xstdivdp

src1 = VSR[XA]



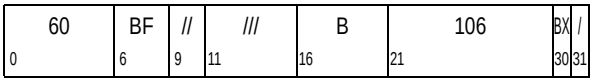
src2 = VSR[XB]



0 64 127

VSX Scalar Test for software Square Root Double-Precision XX2-form

xstsqrtdp BF,XB




```
XB        ← BX || B
src       ← VSR[XB]{0:63}
e_b       ← VSR[XB]{1:11} - 1023
fe_flag   ← IsNaN(src) | IsInf(src) | IsZero(src) |
              IsNeg(src) | ( e_b <= -970 )
fg_flag   ← IsInf(src) | IsZero(src) | IsDen(src)
fl_flag   ← xsrsqrtedp_error() <= 2^-14
CR[BF]    ← 0b1 || fg_flag || fe_flag || 0b0
```

Let XB be the value 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

Let e_b be the unbiased exponent of src.

fe_flag is set to 1 for any of the following conditions.
- src is a zero, a NaN, an infinity, or a negative value.
- e_b is less than or equal to -970

Otherwise fe_flag is set to 0.

fg_flag is set to 1 for any of the following conditions.
- src is a zero, an infinity, or a denormalized value.

Otherwise fg_flag is set to 0.

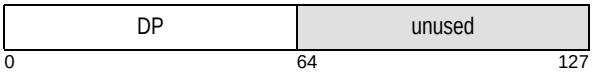
CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered

CR[BF]

VSR Data Layout for xstsqrtdp

src = VSR[XB]



VSX Scalar Test Data Class Double-Precision XX2-form

xststdcdp BF,XB,DCMX

60	BF	DCMX	B	362	BX	/
0	6	9	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src            ← VSR[32×BX+B].dword[0]
exponent       ← src.bit[1:11]
fraction       ← src.bit[12:63]

class.Infinity ← (exponent = 0x7FF) & (fraction  = 0)
class.NaN      ← (exponent = 0x7FF) & (fraction != 0)
class.Zero     ← (exponent = 0x000) & (fraction  = 0)
class.Denormal ← (exponent = 0x000) & (fraction != 0)

match          ← (DCMX.bit[0] & class.NaN)             |
                 (DCMX.bit[1] & class.Infinity & !sign) |
                 (DCMX.bit[2] & class.Infinity &  sign) |
                 (DCMX.bit[3] & class.Zero     & !sign) |
                 (DCMX.bit[4] & class.Zero     &  sign) |
                 (DCMX.bit[5] & class.Denormal & !sign) |
                 (DCMX.bit[6] & class.Denormal &  sign)

CR.bit[4×BF]   ← FPSCR.FL ← src.sign
CR.bit[4×BF+1] ← FPSCR.FG ← 0b0
CR.bit[4×BF+2] ← FPSCR.FE ← match
CR.bit[4×BF+3] ← FPSCR.FU ← 0b0
```

Let XB be the sum 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

Bit 0 of CR field BF and bit 0 of FPCC are set to the sign bit of src.

Bit 1 of CR field BF and bit 1 of FPCC are set to 0b0.

Bit 2 of CR field BF and bit 2 of FPCC are set to indicate whether the data class of src, as represented in double-precision format, matches any of the data classes specified by DCMX (Data Class Mask).

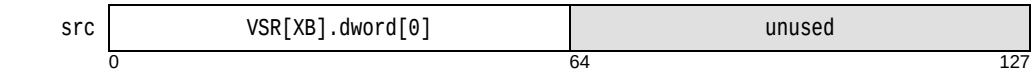
DCMX bit	Data Class
0	NaN
1	+Infinity
2	-Infinity
3	+Zero
4	-Zero
5	+Denormal
6	-Denormal

Bit 3 of CR field BF and bit 3 of FPCC are set to 0b0.

Special Registers Altered:

CR field BF
FPCC

VSR Data Layout for xststdcdp



VSX Scalar Test Data Class Quad-Precision X-form

xststdcqp BF,VRB,DCMX

63	BF	DCMX	VRB	708	/
0	6	9	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src            ← VSR[VRB+32]
exponent       ← src.bit[1:15]
fraction       ← src.bit[16:127]

class.Infinity ← (exponent = 0x7FFF) & (fraction  = 0)
class.NaN      ← (exponent = 0x7FFF) & (fraction != 0)
class.Zero     ← (exponent = 0x0000) & (fraction  = 0)
class.Denormal ← (exponent = 0x0000) & (fraction != 0)

match          ← (DCMX.bit[0] & class.NaN)             |
                 (DCMX.bit[1] & class.Infinity & !sign) |
                 (DCMX.bit[2] & class.Infinity &  sign) |
                 (DCMX.bit[3] & class.Zero     & !sign) |
                 (DCMX.bit[4] & class.Zero     &  sign) |
                 (DCMX.bit[5] & class.Denormal & !sign) |
                 (DCMX.bit[6] & class.Denormal &  sign)

CR.bit[4×BF]   ← FPSCR.FL ← src.sign
CR.bit[4×BF+1] ← FPSCR.FG ← 0b0
CR.bit[4×BF+2] ← FPSCR.FE ← match
CR.bit[4×BF+3] ← FPSCR.FU ← 0b0
```

Let src be the quad-precision floating-point value in VSR[VRB+32].

Let the DCMX (Data Class Mask) field specify one or more of the 7 possible data classes, where each bit corresponds to a specific data class.

DCM bit	Data Class
0	NaN
1	+Infinity
2	-Infinity
3	+Zero
4	-Zero
5	+Denormal
6	-Denormal

Bit 0 of CR field BF and bit 0 of FPCC are set to the sign of src.

Bit 1 of CR field BF and bit 1 of FPCC are set to 0b0.

Bit 2 of CR field BF and bit 2 of FPCC are set to indicate whether the data class of src, as represented in quad-precision format, matches any of the data classes specified by DCM.

Bit 3 of CR field BF and bit 3 of FPCC are set to 0b0.

Special Registers Altered:

CR field BF
FPCC

VSR Data Layout for xststdcqp

VSR[VRB+32]

src

VSX Scalar Test Data Class Single-Precision XX2-form

xststdcsp BF,XB,DCMX

60	BF	DCMX	B	298	BX	/
0	6	9	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src            ← VSR[32×BX+B].dword[0]
exponent       ← src.bit[1:11]
fraction       ← src.bit[12:63]

class.Infinity ← (exponent = 0x7FF) & (fraction  = 0)
class.NaN      ← (exponent = 0x7FF) & (fraction != 0)
class.Zero     ← (exponent = 0x000) & (fraction  = 0)
class.Denormal ← (exponent = 0x000) & (fraction != 0) |
                 (exponent > 0x000) & (exponent < 0x381)

match          ← (DCMX.bit[0] & class.NaN)             |
                 (DCMX.bit[1] & class.Infinity & !sign) |
                 (DCMX.bit[2] & class.Infinity &  sign) |
                 (DCMX.bit[3] & class.Zero     & !sign) |
                 (DCMX.bit[4] & class.Zero     &  sign) |
                 (DCMX.bit[5] & class.Denormal & !sign) |
                 (DCMX.bit[6] & class.Denormal &  sign)

not_SP_value   ← (src != Convert_SPtoDP(Convert_DPtoSP(src)))

CR.bit[4×BF]   ← FPSCR.FL ← src.sign
CR.bit[4×BF+1] ← FPSCR.FG ← 0b0
CR.bit[4×BF+2] ← FPSCR.FE ← match
CR.bit[4×BF+3] ← FPSCR.FU ← not_SP_value
```

Let XB be the sum 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

Bit 0 of CR field BF and bit 0 of FPCC are set to the sign bit of src.

Bit 1 of CR field BF and bit 1 of FPCC are set to 0b0.

Bit 2 of CR field BF and bit 2 of FPCC are set to indicate whether the data class of src, as represented in single-precision format, matches any of the data classes specified by DCMX (Data Class Mask).

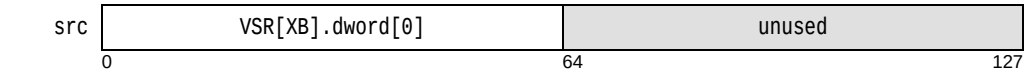
DCMX bit	Data Class
0	NaN
1	+Infinity
2	-Infinity
3	+Zero
4	-Zero
5	+Denormal
6	-Denormal

Bit 3 of CR field BF and bit 3 of FPCC are set to indicate if src is not representable in single-precision format.

Special Registers Altered:

CR field BF
FPCC

VSR Data Layout for xststdcdp



VSX Scalar Extract Exponent Double-Precision XX2-form

xsxexpdp RT,XB

60	RT	0	B	347	BX	/
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

src ← VSR[32×BX+B].dword[0]

GPR[RT] ← (src >> 52) & 0x0000_0000_0000_07FF
```

Let XB be the sum 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

The value of the exponent field in src is placed into GPR[RT] in unsigned integer format.

Special Registers Altered:

None

Programming Note

This instruction can be used to operate on a single-precision source operand.

VSX Scalar Extract Exponent Quad-Precision X-form

xsxexpqp VRT,VRB

63	VRT	2	VRB	804	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src ← VSR[VRB+32]
VSR[VRT+32].dword[0] ← EXTZ64(src.bit[1:15]), 64)
VSR[VRT+32].dword[1] ← 0x0000_0000_0000_0000
```

Let src be the quad-precision floating-point value in VSR[VRB+32].

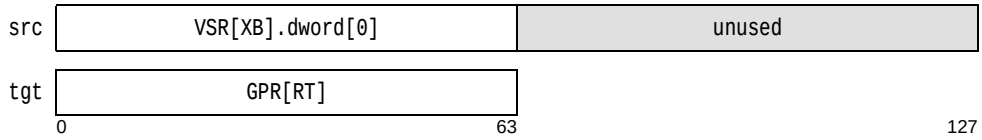
The contents of the exponent field of src (bits 1:15) are zero-extended and placed into doubleword 0 of VSR[VRT+32].

The contents of doubleword 1 of VSR[VRT+32] are set to 0.

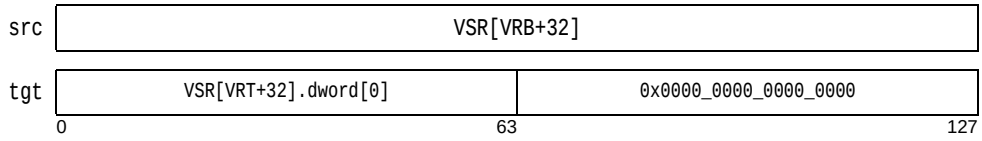
Special Registers Altered:

None

VSR Data Layout for xsxexpdp



VSR Data Layout for xsxexpdp



VSX Scalar Extract Significand Double-Precision XX2-form

xsxsigdp RT,XB

60	RT	1	B	347	BX	/
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

exponent ← VSR[32×BX+B].bit[1:11]
fraction ← EXTZ64(VSR[32×BX+B].bit[12:63])

if (exponent != 0) & (exponent != 2047) then
   significand ← fraction | 0x0010_0000_0000_0000
else
   significand ← fraction

GPR[RT] ← significand
```

Let XB be the sum 32×BX + B.

Let src be the double-precision floating-point value in doubleword element 0 of VSR[XB].

The significand of src is placed into GPR[RT] in unsigned integer format. If src is a normal value, the implicit leading bit is set to 1.

Special Registers Altered:

None

Programming Note

This instruction can be used to operate on a single-precision source operand.

VSX Scalar Extract Significand Quad-Precision X-form

xsxsigqp VRT,VRB

63	VRT	18	VRB	804	/
0	6	11	16	21	31

```
if MSR.VSX=0 then VSX_Unavailable()

src      ← VSR[VRB+32]
exponent ← EXTZ(src.bit[1:15])
fraction ← EXTZ128(src.bit[16:127])

if (exponent != 0) &  (exponent != 32767) then
   VSR[VRT+32] ← fraction | 0x0001_0000_0000_0000_0000_0000_0000_0000
else
   VSR[VRT+32] ← fraction
```

Let src be the quad-precision floating-point value in VSR[VRB+32].

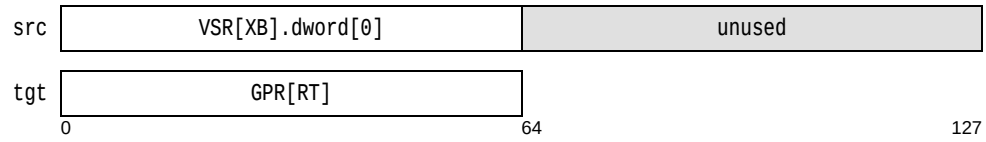
The significand of src is placed into VSR[VRT+32].

If the value of the exponent field of src is equal to 0b000_0000_0000_0000 (i.e., Zero or Denormal value) or 0b111_1111_1111 (i.e., Infinity or NaN), 0b0 is placed into bit 15 of VSR[VRT+32]. Otherwise (i.e., Normal value), 0b1 is placed into bit 15 of VSR[VRT+32]. The contents of bits 0:14 of VSR[VRT+32] are set to 0.

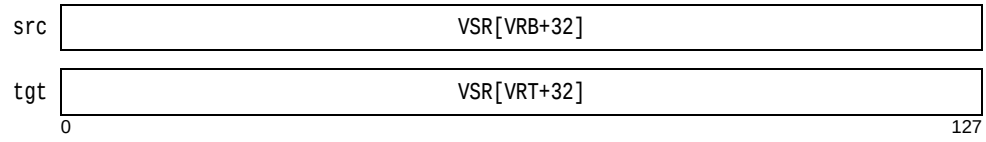
Special Registers Altered:

None

VSR Data Layout for xsxsigdp

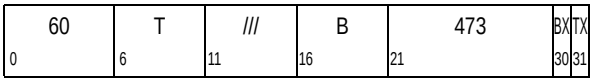


VSR Data Layout for xsxsigqp



VSX Vector Absolute Value Double-Precision XX2-form

xvabsdp XT,XB




```
XT ← TX || T
XB ← BX || B

do i=0 to 127 by 64
   VSR[XT]{i:i+63} ← 0b0 || VSR[XB]{i+1:i+63}
end
```

Let XT be the value 32×TX + T.

Let XB be the value 32×BX + B.

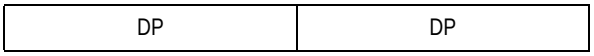
For each vector element i from 0 to 1, do the following.
The contents of doubleword element i of VSR[XB], with bit 0 set to 0, is placed into doubleword element i of VSR[XT].

Special Registers Altered

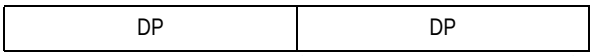
None

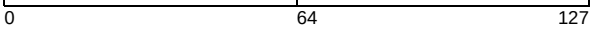
VSR Data Layout for xvabsdp

src = VSR[XB]



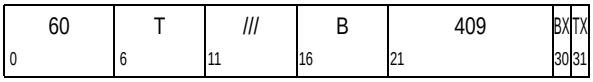
tgt = VSR[XT]





VSX Vector Absolute Value Single-Precision XX2-form

xvabssp XT,XB




```
XT ← TX || T
XB ← BX || B

do i=0 to 127 by 32
   VSR[XT]{i:i+31} ← 0b0 || VSR[XB]{i+1:i+31}
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

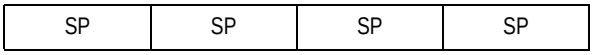
For each vector element i from 0 to 3, do the following.
The contents of word element i of VSR[XB], with bit 0 set to 0, is placed into word element i of VSR[XT].

Special Registers Altered

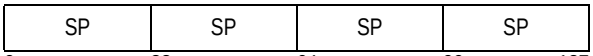
None

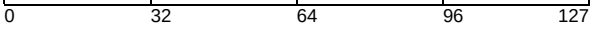
VSR Data Layout for xvabssp

src = VSR[XB]



tgt = VSR[XT]





VSX Vector Add Double-Precision XX3-form

xvadddp XT,XA,XB

60	T	A	B	96	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1           ← VSR[XA]{i:i+63}
   src2           ← VSR[XB]{i:i+63}
   v{0:inf}       ← AddDP(src1,src2)
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src2 is added[1] to src1, producing a sum having unbounded range and precision.

The sum is normalized[2].

See Table 97.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 98, “Vector Floating-Point Final Result,” on page 661.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX XX VXSNAN VXISI

VSR Data Layout for xvadddp

src1 = VSR[XA]

DP	DP

src2 = VSR[XB]

DP	DP

tgt = VSR[XT]

DP	DP

0 64 127

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← -Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← -Infinity	v ← src2	v ← -Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← -Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← -Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 97.Actions for xvadddp (element i)

Case	VE	OE	UE	ZE	XE	vxsnan_flag	vximz_flag	vxisi_flag	vxidi_flag	vxzdz_flag	vxsqrt_flag	zx_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Special	–	–	–	–	–	0	0	0	0	0	0	0	–	–	–	–	T(r)
	–	–	–	0	–	–	–	–	–	–	–	1	–	–	–	–	T(r), fx(ZX)
	–	–	–	1	–	–	–	–	–	–	–	1	–	–	–	–	fx(ZX), error()
	0	–	–	–	–	–	–	–	–	–	1	–	–	–	–	–	T(r), fx(VXSQRT)
	0	–	–	–	–	–	–	–	–	1	–	–	–	–	–	–	T(r), fx(VXZDZ)
	0	–	–	–	–	–	–	–	1	–	–	–	–	–	–	–	T(r), fx(VXIDI)
	0	–	–	–	–	–	–	1	–	–	–	–	–	–	–	–	T(r), fx(VXISI)
	0	–	–	–	–	0	1	–	–	–	–	–	–	–	–	–	T(r), fx(VXIMZ)
	0	–	–	–	–	1	0	–	–	–	–	–	–	–	–	–	T(r), fx(VXSNAN)
	0	–	–	–	–	1	1	–	–	–	–	–	–	–	–	–	T(r), fx(VXSNAN), fx(VXIMZ)
	1	–	–	–	–	–	–	–	–	–	1	–	–	–	–	–	T(r), fx(VXSQRT)
	1	–	–	–	–	–	–	–	–	1	–	–	–	–	–	–	fx(VXZDZ), error()
	1	–	–	–	–	–	–	–	1	–	–	–	–	–	–	–	fx(VXIDI), error()
	1	–	–	–	–	–	–	1	–	–	–	–	–	–	–	–	fx(VXISI), error()
	1	–	–	–	–	0	1	–	–	–	–	–	–	–	–	–	fx(VXIMZ), error()
	1	–	–	–	–	1	0	–	–	–	–	–	–	–	–	–	fx(VXSNAN), error()
	1	–	–	–	–	1	1	–	–	–	–	–	–	–	–	–	fx(VXSNAN), fx(VXIMZ), error()
Normal	–	–	–	–	–	–	–	–	–	–	–	–	no	–	–	–	T(r)
	–	–	–	–	0	–	–	–	–	–	–	–	yes	no	–	–	T(r), fx(XX)
	–	–	–	–	0	–	–	–	–	–	–	–	yes	yes	–	–	T(r), fx(XX)
	–	–	–	–	1	–	–	–	–	–	–	–	yes	no	–	–	T(r), fx(XX), error()
	–	–	–	–	1	–	–	–	–	–	–	–	yes	yes	–	–	T(r), fx(XX), error()

Explanation:

–	The results do not depend on this condition.
fx(x)	FX is set to 1 if x=0. x is set to 1.
q	The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
OX	Floating-Point Overflow Exception status flag, $FPSCR_{OX}$.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.
T(x)	The value x is placed in element i of VSR[XT] in the target precision format (where i ∈ {0,1} for results with 64-bit elements, and i ∈ {0,1,3,4}) for results with 32-bit elements).
UX	Floating-Point Underflow Exception status flag, $FPSCR_{UX}$
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$.
VXSQRT	Floating-Point Invalid Operation Exception (Invalid Square Root) status flag, $FPSCR_{VXSQRT}$.
VXIDI	Floating-Point Invalid Operation Exception (Infinity ÷ Infinity) status flag, $FPSCR_{VXIDI}$.
VXIMZ	Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, $FPSCR_{VXIMZ}$.
VXISI	Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, $FPSCR_{VXISI}$.
VXZDZ	Floating-Point Invalid Operation Exception (Zero ÷ Zero) status flag, $FPSCR_{VXZDZ}$.
XX	Float-Point Inexact Exception status flag, $FPSCR_{XX}$. The flag is a sticky version of $FPSCR_{FI}$. When $FPSCR_{FI}$ is set to a new value, the new value of $FPSCR_{XX}$ is set to the result of ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.
ZX	Floating-Point Zero Divide Exception status flag, $FPSCR_{ZX}$.

Table 98.Vector Floating-Point Final Result

Case	VE	OE	UE	ZE	XE	vxsnan_flag	vximz_flag	vxisi_flag	vxidi_flag	vxzdz_flag	vxsqrt_flag	zx_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Overflow	–	0	–	–	0	–	–	–	–	–	–	–	–	–	–	–	T(r), fx(OX), fx(XX)
	–	0	–	–	1	–	–	–	–	–	–	–	–	–	–	–	T(r), fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	–	–	–	–	no	–	fx(OX), error()
	–	1	–	–	–	–	–	–	–	–	–	–	–	–	yes	no	fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	–	–	–	–	yes	yes	fx(OX), fx(XX), error()
Tiny	–	–	0	–	–	–	–	–	–	–	–	–	no	–	–	–	T(r)
	–	–	0	–	0	–	–	–	–	–	–	–	yes	no	–	–	T(r), fx(UX), fx(XX)
	–	–	0	–	0	–	–	–	–	–	–	–	yes	yes	–	–	T(r), fx(UX), fx(XX)
	–	–	0	–	1	–	–	–	–	–	–	–	yes	no	–	–	T(r), fx(UX), fx(XX), error()
	–	–	0	–	1	–	–	–	–	–	–	–	yes	yes	–	–	T(r), fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	–	–	–	–	yes	–	no	–	fx(UX), error()
	–	–	1	–	–	–	–	–	–	–	–	–	yes	–	yes	no	fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	–	–	–	–	yes	–	yes	yes	fx(UX), fx(XX), error()

Explanation:

–	The results do not depend on this condition.
fx(x)	FX is set to 1 if x=0. x is set to 1.
q	The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
OX	Floating-Point Overflow Exception status flag, $FPSCR_{OX}$.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.
T(x)	The value x is placed in element i of VSR[XT] in the target precision format (where i ∈ {0,1} for results with 64-bit elements, and i ∈ {0,1,3,4}) for results with 32-bit elements).
UX	Floating-Point Underflow Exception status flag, $FPSCR_{UX}$
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$.
VXSQRT	Floating-Point Invalid Operation Exception (Invalid Square Root) status flag, $FPSCR_{VXSQRT}$.
VXIDI	Floating-Point Invalid Operation Exception (Infinity ÷ Infinity) status flag, $FPSCR_{VXIDI}$.
VXIMZ	Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, $FPSCR_{VXIMZ}$.
VXISI	Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, $FPSCR_{VXISI}$.
VXZDZ	Floating-Point Invalid Operation Exception (Zero ÷ Zero) status flag, $FPSCR_{VXZDZ}$.
XX	Float-Point Inexact Exception status flag, $FPSCR_{XX}$. The flag is a sticky version of $FPSCR_{FI}$. When $FPSCR_{FI}$ is set to a new value, the new value of $FPSCR_{XX}$ is set to the result of ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.
ZX	Floating-Point Zero Divide Exception status flag, $FPSCR_{ZX}$.

Table 98.Vector Floating-Point Final Result (Continued)

VSX Vector Add Single-Precision XX3-form

xvaddsp XT,XA,XB

60	T	A	B	64	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1           ← VSR[XA]{i:i+31}
   src2           ← VSR[XB]{i:i+31}
   v{0:inf}       ← AddSP(src1,src2)
   result{i:i+31} ← RoundToSP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

src2 is added[1] to src1, producing a sum having unbounded range and precision.

The sum is normalized[2].

See Table 99.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 98, “Vector Floating-Point Final Result,” on page 661.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX XX VXSNAN VXISI

VSR Data Layout for xvaddsp

src1 = VSR[XA]

SP	SP	SP	SP

src2 = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← -Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← -Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← -Infinity	v ← src2	v ← -Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← -Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← -Infinity	v ← A(src1,src2)	v ← src1	v ← src1	v ← A(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

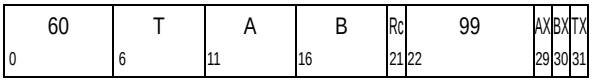
Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (0x7FC0_0000).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 99.Actions for xvaddsp (element i)

VSX Vector Compare Equal To Double-Precision XX3-form

xvcmpeqdp XT,XA,XB (Rc=0)
xvcmpeqdp. XT,XA,XB (Rc=1)




```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0
all_false   ← 0b1
all_true    ← 0b1

do i←0 to 127 by 64
   reset_xflags()
   src1        ← VSR[XA]{i:i+63}
   src2        ← VSR[XB]{i:i+63}
   vxsnan_flag ← IsSNaN(src1) | IsSNaN(src2)

   if( CompareEQDP(src1,src2) ) then
      result{i:i+63} ← 0xFFFF_FFFF_FFFF_FFFF
      all_false      ← 0b0
   end
   else do
      result{i:i+63} ← 0x0000_0000_0000_0000
      all_true       ← 0b0
   end
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

if(Rc=1) then do
   if( !vex_flag ) then
      CR[6] ← all_true || 0b0 || all_false || 0b0
   else
      CR[6] ← 0bUUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is compared to src2.

The contents of doubleword element i of VSR[XT] are set to all 1s if src1 is equal to src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] is set to 0.

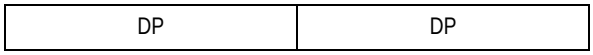
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR[6] are undefined if Rc is equal to 1.

Special Registers Altered

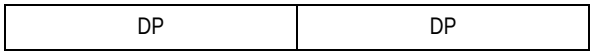
CR[6] . . . . . (if Rc=1)
FX VXSNAN

VSR Data Layout for xvcmpeqdp[.]

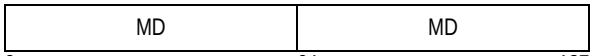
src1 = VSR[XA]



src2 = VSR[XB]



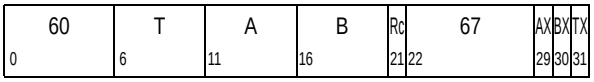
tgt = VSR[XT]



0 64 127

VSX Vector Compare Equal To Single-Precision XX3-form

xvcmpeqsp XT,XA,XB (Rc=0)
xvcmpeqsp. XT,XA,XB (Rc=1)




```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0
all_false   ← 0b1
all_true    ← 0b1

do i=0 to 127 by 32
   reset_xflags()
   src1        ← VSR[XA]{i:i+31}
   src2        ← VSR[XB]{i:i+31}
   vxsnan_flag ← IsSNaN(src1) | IsSNaN(src2)

   if( CompareEQSP(src1,src2) ) then
      result{i:i+31} ← 0xFFFF_FFFF
      all_false      ← 0b0
   end
   else do
      result{i:i+31} ← 0x0000_0000
      all_true       ← 0b0
   end
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

if(Rc=1) then do
   if( !vex_flag ) then
      CR[6] ← all_true || 0b0 || all_false || 0b0
   else
      CR[6] ← 0bUUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

src1 is compared to src2.

The contents of word element i of VSR[XT] are set to all 1s if src1 is equal to src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] is set to 0.

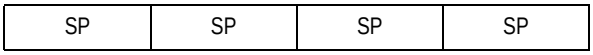
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR[6] are undefined if Rc is equal to 1.

Special Registers Altered

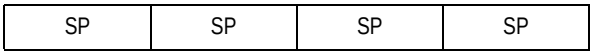
CR[6] . . . . . (if Rc=1)
FX VXSNAN

VSR Data Layout for xvcmpeqsp[.]

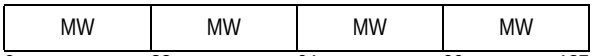
src1 = VSR[XA]



src2 = VSR[XB]



tgt = VSR[XT]



0 32 64 96 127

VSX Vector Compare Greater Than or Equal To Double-Precision XX3-form

xvcmpgedp XT,XA,XB (Rc=0)
xvcmpgedp. XT,XA,XB (Rc=1)

60	T	A	B	Rc	115	AX	BX	TX
0	6	11	16	21	22	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0
all_false   ← 0b1
all_true    ← 0b1

do i=0 to 127 by 64
   reset_xflags()
   src1        ← VSR[XA]{i:i+63}
   src2        ← VSR[XB]{i:i+63}

   if( IsSNaN(src1) | IsSNaN(src2) ) then do
      vxsnan_flag ← 0b1
      if(VE=0) then vxvc_flag ← 0b1
   end
   else vxvc_flag ← IsQNaN(src1) | IsQNaN(src2)

   if( CompareGEDP(src1,src2) ) then
      result{i:i+63} ← 0xFFFF_FFFF_FFFF_FFFF
      all_false      ← 0b0
   end
   else do
      result{i:i+63} ← 0x0000_0000_0000_0000
      all_true       ← 0b0
   end
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxvc_flag)   then SetFX(VXVC)
   ex_flag ← ex_flag | (VE & vxsnan_flag)
   ex_flag ← ex_flag | (VE & vxvc_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

if(Rc=1) then do
   if( !vex_flag ) then
      CR[6] ← all_true || 0b0 || all_false || 0b0
   else
      CR[6] ← 0bUUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is compared to src2.

The contents of doubleword element i of VSR[XT] are set to all 1s if src1 is greater than or equal to the double-precision floating-point operand in doubleword element i of VSR[XB]src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR[6] are undefined if Rc is equal to 1.

Special Registers Altered

CR[6] . . . . . (if Rc=1)
FX VXSNAN VXVC

VSR Data Layout for xvcmpgedp[.]

src1 = VSR[XA]

DP	DP

src2 = VSR[XB]

DP	DP

tgt = VSR[XT]

MD	MD

0 64 127

VSX Vector Compare Greater Than or Equal To Single-Precision XX3-form

xvcmpgesp XT,XA,XB (Rc=0)
xvcmpgesp. XT,XA,XB (Rc=1)

60	T	A	B	Rc	83	AX	BX	TX
0	6	11	16	21	22	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0
all_false   ← 0b1
all_true    ← 0b1

do i=0 to 127 by 32
   reset_xflags()
   src1        ← VSR[XA]{i:i+31}
   src2        ← VSR[XB]{i:i+31}

   if( IsSNaN(src1) | IsSNaN(src2) ) then do
      vxsnan_flag ← 0b1
      if(VE=0) then vxvc_flag ← 0b1
   end
   else vxvc_flag ← IsQNaN(src1) | IsQNaN(src2)

   if( CompareGESP(src1,src2) ) then
      result{i:i+31} ← 0xFFFF_FFFF
      all_false      ← 0b0
   end
   else do
      result{i:i+31} ← 0x0000_0000
      all_true       ← 0b0
   end
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxvc_flag)   then SetFX(VXVC)
   ex_flag ← ex_flag | (VE & vxsnan_flag)
   ex_flag ← ex_flag | (VE & vxvc_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

if(Rc=1) then do
   if( !vex_flag ) then
      CR[6] ← all_true || 0b0 || all_false || 0b0
   else
      CR[6] ← 0bUUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

src1 is compared to src2.

The contents of word element i of VSR[XT] are set to all 1s if src1 is greater than or equal to src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return true for that element.

Two infinity inputs of same signs return true for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] is set to 0.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR[6] are undefined if Rc is equal to 1.

Special Registers Altered

CR[6] . . . . . (if Rc=1)
FX VXSNAN VXVC

VSR Data Layout for xvcmpgesp[.]

src1 = VSR[XA]

SP	SP	SP	SP

src2 = VSR[XB]

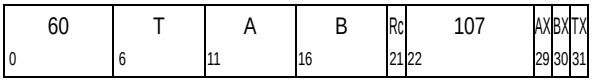
SP	SP	SP	SP

tgt = VSR[XT]

MW	MW	MW	MW
0	32	64	96	127

VSX Vector Compare Greater Than Double-Precision XX3-form

xvcmpgtdp XT,XA,XB (Rc=0)
xvcmpgtdp. XT,XA,XB (Rc=1)




```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0
all_false   ← 0b1
all_true    ← 0b1

do i=0 to 127 by 64
   reset_xflags()
   src1        ← VSR[XA]{i:i+63}
   src2        ← VSR[XB]{i:i+63}

   if( IsSNaN(src1) | IsSNaN(src2) ) then do
      vxsnan_flag ← 0b1
      if(VE=0) then vxvc_flag ← 0b1
   end
   else vxvc_flag ← IsQNaN(src1) | IsQNaN(src2)

   if( CompareGTDP(src1,src2) ) then do
      result{i:i+63} ← 0xFFFF_FFFF_FFFF_FFFF
      all_false      ← 0b0
   end
   else do
      result{i:i+63} ← 0x0000_0000_0000_0000
      all_true       ← 0b0
   end
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxvc_flag)   then SetFX(VXVC)
   ex_flag ← ex_flag | (VE & vxsnan_flag)
   ex_flag ← ex_flag | (VE & vxvc_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

if(Rc=1) then do
   if( !vex_flag ) then
      CR[6] ← all_true || 0b0 || all_false || 0b0
   else
      CR[6] ← 0bUUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is compared to src2.

The contents of doubleword element i of VSR[XT] are set to all 1s if src1 is greater than src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return false for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] is set to 0.

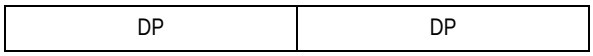
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR[6] are undefined if Rc is equal to 1.

Special Registers Altered

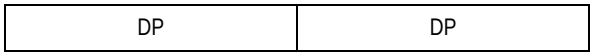
CR[6] . . . . . (if Rc=1)
FX VXSNAN VXVC

VSR Data Layout for xvcmpgtdp[.]

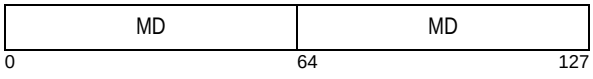
src1 = VSR[XA]



src2 = VSR[XB]

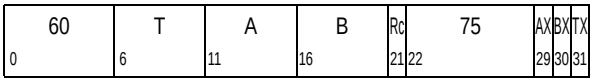


tgt = VSR[XT]



VSX Vector Compare Greater Than Single-Precision XX3-form

xvcmpgtsp XT,XA,XB (Rc=0)
xvcmpgtsp. XT,XA,XB (Rc=1)




```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0
all_false   ← 0b1
all_true    ← 0b1

do i=0 to 127 by 32
   reset_xflags()
   src1        ← VSR[XA]{i:i+31}
   src2        ← VSR[XB]{i:i+31}

   if( IsSNaN(src1) | IsSNaN(src2) ) then do
      vxsnan_flag ← 0b1
      if(VE=0) then vxvc_flag ← 0b1
   end
   else vxvc_flag ← IsQNaN(src1) | IsQNaN(src2)

   if( CompareGTSP(src1,src2) ) then do
      result{i:i+31} ← 0xFFFF_FFFF
      all_false      ← 0b0
   end
   else do
      result{i:i+31} ← 0x0000_0000
      all_true       ← 0b0
   end
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxvc_flag)   then SetFX(VXVC)
   ex_flag ← ex_flag | (VE & vxsnan_flag)
   ex_flag ← ex_flag | (VE & vxvc_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result

if(Rc=1) then do
   if( !vex_flag ) then
      CR[6] ← all_true || 0b0 || all_false || 0b0
   else
      CR[6] ← 0bUUUU
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

src1 is compared to src2.

The contents of word element i of VSR[XT] are set to all 1s if src1 is greater than src2, and is set to all 0s otherwise.

A NaN input causes the comparison to return false for that element.

Two zero inputs of same or different signs return false for that element.

If Rc=1, CR Field 6 is set as follows.

- Bit 0 of CR[6] is set to indicate all vector elements compared true.
- Bit 1 of CR[6] is set to 0.
- Bit 2 of CR[6] is set to indicate all vector elements compared false.
- Bit 3 of CR[6] is set to 0.

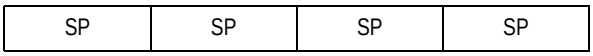
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT] and the contents of CR[6] are undefined if Rc is equal to 1.

Special Registers Altered

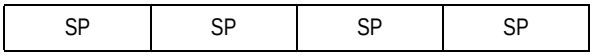
CR[6] . . . . . (if Rc=1)
FX VXSNAN VXVC

VSR Data Layout for xvcmpgtsp[.]

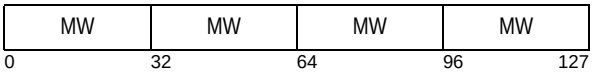
src1 = VSR[XA]



src2 = VSR[XB]

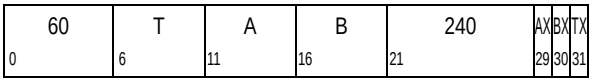


tgt = VSR[XT]



VSX Vector Copy Sign Double-Precision XX3-form

xvcpsgndp XT,XA,XB




```
XT ← TX || T
XA ← AX || A
XB ← BX || B

do i=0 to 127 by 64
   VSR[XT]{i:i+63} ← VSR[XA]{i} || VSR[XB]{i+1:i+63}
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.
The contents of bit 0 of doubleword element i of VSR[XA] are concatenated with the contents of bits 1:63 of doubleword element i of VSR[XB] and placed into doubleword element i of VSR[XT].

Special Registers Altered

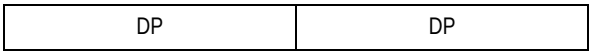
None

Extended Mnemonic		Equivalent To	
xvmovdp	XT,XB	xvcpsgndp	XT,XB,XB

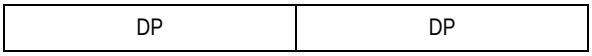
Table 100:

VSR Data Layout for xvcpsgndp

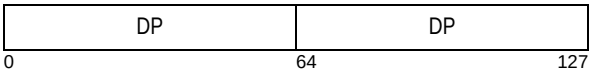
src1 = VSR[XA]

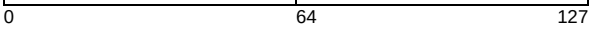


src2 = VSR[XB]



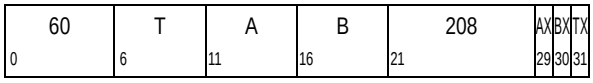
tgt = VSR[XT]





VSX Vector Copy Sign Single-Precision XX3-form

xvcpsgnsp XT,XA,XB




```
XT ← TX || T
XA ← AX || A
XB ← BX || B

do i=0 to 127 by 32
   VSR[XT]{i:i+31} ← VSR[XA]{i} || VSR[XB]{i+1:i+31}
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.
The contents of bit 0 of word element i of VSR[XA] are concatenated with the contents of bits 1:31 of word element i of VSR[XB] and placed into word element i of VSR[XT].

Special Registers Altered

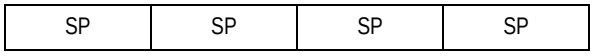
None

Extended Mnemonic		Equivalent To	
xvmovsp	XT,XB	xvcpsgnsp	XT,XB,XB

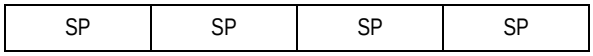
Table 101:

VSR Data Layout for xvcpsgnsp

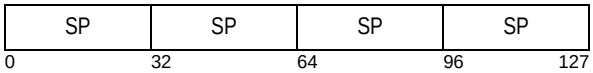
src1 = VSR[XA]

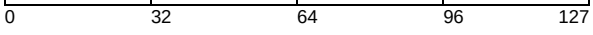


src2 = VSR[XB]



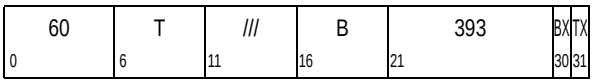
tgt = VSR[XT]





VSX Vector Convert with round Double-Precision to Single-Precision format XX2-form

xvcvdpsp XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src              ← VSR[XB]{i:i+63}
   result{i:i+31}   ← RoundToSP(RN,src)
   result{i+32:i+63} ← 0xUUUU_UUUU
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag          ← ex_flag | (VE & vxsnan_flag)
   ex_flag          ← ex_flag | (OE & ox_flag)
   ex_flag          ← ex_flag | (UE & ux_flag)
   ex_flag          ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to single-precision using the rounding mode specified by RN.

The result is placed into bits 0:31 of doubleword element i of VSR[XT] in single-precision format.

The contents of bits 32:63 of doubleword element i of VSR[XT] are undefined.

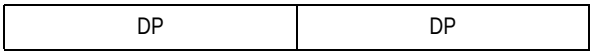
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

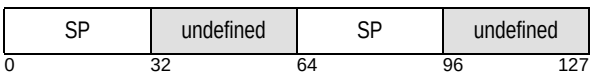
FX OX UX XX VXSNAN

VSR Data Layout for xvcvdpsp

src = VSR[XB]



tgt = VSR[XT]



VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format XX2-form

xvcvdpsxds XT,XB

60	T	///	B	472	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   result{i:i+63} ← ConvertDPtoSD(VSR[XB]{i:i+63})
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxcvi_flag)  then SetFX(VXCVI)
   if(xx_flag)     then SetFX(XX)
   ex_flag       ← ex_flag | (VE & vxsnan_flag)
                           | (VE & vxcvi_flag)
                           | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

If src is a NaN, the result is the value 0x8000_0000_0000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than $2^{63}-1$, the result is 0x7FFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{63}, the result is 0x8000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

The result is placed into doubleword element i of VSR[XT].

See Table 102.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX XX VXSNAN VXCVI

VSR Data Layout for xvcvdpsxds

src = VSR[XB]

DP	DP

tgt = VSR[XT]

SD	SD
0	64 127

Programming Note

xvcvdpsxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrdpic*** which uses the rounding mode specified by the RN.

	VE	XE	Inexact? (RoundToDPintegerTrunc(src) g src)	Returned Results and Status Setting
src [Nmin-1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(ConvertDPtoSD(RoundToDPintegerTrunc(src)))
		0	yes	T(ConvertDPtoSD(RoundToDPintegerTrunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src m Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), fx(VXSNAN), error()

Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
Nmin	The smallest signed integer doubleword value, -2^{63} (0x8000_0000_0000_0000).
Nmax	The largest signed integer doubleword value, $2^{63}-1$ (0x7FFF_FFFF_FFFF_FFFF).
src	The double-precision floating-point value in doubleword element i of VSR[XB] (where i C {0,1}).
T(x)	The signed integer doubleword value x is placed in doubleword element i of VSR[XT] (where i C {0,1}).

Table 102.Actions for xvcvdpsxds

VSX Vector Convert with round to zero Double-Precision to Signed Word format XX2-form

xvcvdpsxws XT,XB

60	T	///	B	216	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   result{i:i+31}    ← ConvertDPtoSW(VSR[XB]{i:i+63})
   result{i+32:i+63} ← 0xUUUU_UUUU
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxcvi_flag)  then SetFX(VXCVI)
   if(xx_flag)     then SetFX(XX)
   ex_flag       ← ex_flag | (VE & vxsnan_flag)
                           | (VE & vxcvi_flag)
                           | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

If src is a NaN, the result is the value 0x8000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than $2^{31}-1$, the result is 0x7FFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{31}, the result is 0x8000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit signed-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

The result is placed into bits 0:31 of doubleword element i of VSR[XT].

The contents of bits 32:63 of doubleword element 1 of VSR[XT] are undefined.

See Table 103.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX XX VXSNAN VXCVI

VSR Data Layout for xvcvdpsxws

src = VSR[XB]

DP	DP

tgt = VSR[XT]

SW	undefined	SW	undefined
0	32	64	96	127

Programming Note

xvcvdpsxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrdpic*** which uses the rounding mode specified by RN.

	VE	XE	Inexact? (RoundToDPintegerTrunc(src) g src)	Returned Results and Status Setting
src [Nmin-1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(ConvertDPtoSW(RoundToDPintegerTrunc(src)))
		0	yes	T(ConvertDPtoSW(RoundToDPintegerTrunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax)
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	T(Nmax), fx(XX), error()
src m Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), fx(VXSNAN), error()

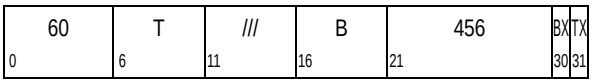
Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
Nmin	The smallest signed integer word value, -2^{31}(0x8000_0000).
Nmax	The largest signed integer word value, $2^{31}-1$ (0x7FFF_FFFF).
src	The double-precision floating-point value in doubleword element i of VSR[XB] (where i C {0,1}).
T(x)	The signed integer word value x is placed in word element i of VSR[XT] (where i C {0,2}).

Table 103.Actions for xvcvdpsxws

VSX Vector Convert with round to zero Double-Precision to Unsigned Doubleword format XX2-form

xvcvdpuxds XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   result{i:i+63} ← ConvertDPtoUD(VSR[XB]{i:i+63})
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxcvi_flag)  then SetFX(VXCVI)
   if(xx_flag)     then SetFX(XX)
   ex_flag       ← ex_flag | (VE & vxsnan_flag)
                           | (VE & vxcvi_flag)
                           | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

If src is a NaN, the result is the value 0x0000_0000_0000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{64}-1, the result is 0xFFFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is 0x0000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

The result is placed into doubleword element i of VSR[XT].

See Table 104.

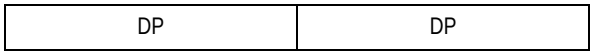
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

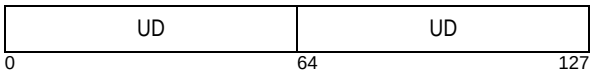
FX XX VXSNAN VXCVI

VSR Data Layout for xvcvdpuxds

src = VSR[XB]



tgt = VSR[XT]



Programming Note

xvcvdpuxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrdpic*** which uses the rounding mode specified by the RN.

	VE	XE	Inexact? (RoundToDPintegerTrunc(src) g src)	Returned Results and Status Setting
src [Nmin-1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(ConvertDPtoUD(RoundToDPintegerTrunc(src)))
		0	yes	T(ConvertDPtoUD(RoundToDPintegerTrunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in DP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	T(Nmax), fx(XX), error()
src m Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), fx(VXSNAN), error()

Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
Nmin	The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000).
Nmax	The largest unsigned integer doubleword value, 2^{64}-1 (0xFFFF_FFFF_FFFF_FFFF).
src	The double-precision floating-point value in doubleword element i VSR[XB] (where i C {0,1}).
T(x)	The unsigned integer doubleword value x is placed in doubleword element i of VSR[XT] (where i C {0,1}).

Table 104.Actions for xvcvdpuxds

VSX Vector Convert with round to zero Double-Precision to Unsigned Word format XX2-form

xvcvdpuxws XT,XB

60	T	///	B	200	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   result{i:i+31}    ← ConvertDPtoUW(VSR[XB]{i:i+63})
   result{i+32:i+63} ← 0xUUUU_UUUU
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxcvi_flag)  then SetFX(VXCVI)
   if(xx_flag)     then SetFX(XX)
   ex_flag       ← ex_flag | (VE & vxsnan_flag)
                           | (VE & vxcvi_flag)
                           | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

If src is a NaN, the result is the value 0x8000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than $2^{32}-1$, the result is 0xFFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is 0x0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

The result is placed into bits 0:31 of doubleword element i of VSR[XT].

The contents of bits 32:63 of doubleword element i of VSR[XT] are undefined.

See Table 105.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX XX VXSNAN VXCVI

VSR Data Layout for xvcvdpuxws

src = VSR[XB]

DP	DP

tgt = VSR[XT]

UW	undefined	UW	undefined
0	32	64	96	127

Programming Note

xvcvdpuxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Double-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrdpic*** which uses the rounding mode specified by RN.

	VE	XE	Inexact? (RoundToDPintegerTrunc(src) g src)	Returned Results and Status Setting
src [Nmin-1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(ConvertDPtoUW(RoundToDPintegerTrunc(src)))
		0	yes	T(ConvertDPtoUW(RoundToDPintegerTrunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax)
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src m Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), fx(VXSNAN), error()

Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
Nmin	The smallest unsigned integer word value, 0 (0x0000_0000).
Nmax	The largest unsigned integer word value, 2^{32}-1 (0xFFFF_FFFF).
src	The double-precision floating-point value in doubleword element i of VSR[XB] (where i C {0,1}).
T(x)	The unsigned integer word value x is placed in word element i of VSR[XT] (where i C {0,2}).

Table 105.Actions for xvcvdpuxws

VSX Vector Convert Half-Precision to Single-Precision format XX2-form

xvcvhpsp XT,XB

60	T	24	B	475	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_flags()

do i = 0 to 3
   src ← bfp_CONVERT_FROM_BFP16(VSR[BX×32+B].word[i].hword[1])

   if src.class.SNaN=1 then
      result.word[i] ← bfp_CONVERT_TO_BFP32(bfp_QUIET(src))
   else
      result.word[i] ← bfp_CONVERT_TO_BFP32(src)

   vxsnan_flag ← src.class.SNaN
   if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
end

if ex_flag=0 then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src be the half-precision floating-point value in the rightmost halfword of word element i of VSR[XB].

If src is an SNaN, the result is the single-precision representation of that SNaN converted to a QNaN.

Otherwise, if src is a QNaN, the result is the single-precision representation of that QNaN.

Otherwise, if src is an Infinity, the result is the single-precision representation of Infinity with the same sign as src.

Otherwise, if src is a Zero, the result is the single-precision representation of Zero with the same sign as src.

Otherwise, if src is a denormal value, the result is the normalized single-precision representation of src.

Otherwise, the result is the single-precision representation of src.

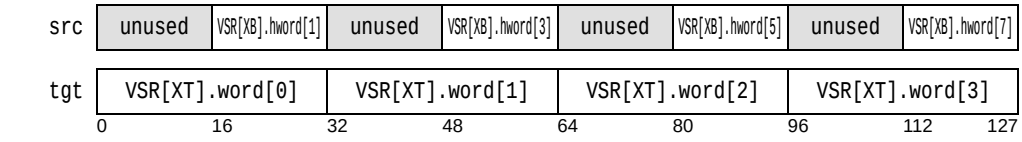
The result is placed into word element i of VSR[XT].

If a trap-enabled exception occurs, VSR[XT] is not modified.

Special Registers Altered:

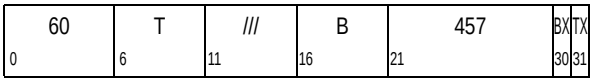
FX VXSNAN

VSR Data Layout for xvcvhpsp



VSX Vector Convert Single-Precision to Double-Precision format XX2-form

xvcvspdp XT,XB




```
XT              ← TX || T
XB              ← BX || B
ex_flag         ← 0b0
do i=0 to 127 by 64
   reset_xflags()
   result{i:i+63} ← ConvertSPtoDP(VSR[XB]{i:i+31})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the single-precision floating-point operand in bits 0:31 of doubleword element i of VSR[XB].

src is placed into doubleword element i of VSR[XT] in double-precison format.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

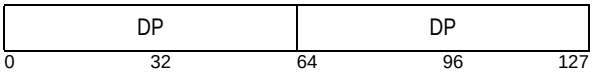
Special Registers Altered

FX VXSNAN

VSR Data Layout for xvcvspdp

src = VSR[XB]

tgt = VSR[XT]



VSX Vector Convert with round Single-Precision to Half-Precision format XX2-form

xvcvsphp XT,XB

60	T	25	B	475	BX	TX
0	6	11	16	21	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

reset_flags()

do i = 0 to 3
   src                ← bfp_CONVERT_FROM_BFP32(VSR[BX×32+B].word[i])
   rnd                ← bfp_ROUND_TO_BFP16(FPSCR.RN,rnd)
   result.hword[2×i]  ← 0x0000
   result.hword[2×i+1] ← bfp_CONVERT_TO_BFP16(rnd)

   if(vxsnan_flag) then SetFX(FPSCR.VXSNAN)
   if(ox_flag) then SetFX(FPSCR.OX)
   if(ux_flag) then SetFX(FPSCR.UX)
   if(xx_flag) then SetFX(FPSCR.XX)

   ex_flag ← ex_flag | (FPSCR.VE & vxsnan_flag)
                     | (FPSCR.OE & ox_flag)
                     | (FPSCR.UE & ux_flag)
                     | (FPSCR.XE & xx_flag)
end

if(ex_flag=0) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src be the single-precision floating-point value in word element i of VSR[XB].

If src is an SNaN, the result is the half-precision representation of that SNaN converted to a QNaN.

Otherwise, if src is a QNaN, the result is the half-precision representation of that QNaN.

Otherwise, if src is an Infinity, the result is the half-precision representation of Infinity with the same sign as src.

Otherwise, if src is a Zero, the result is the half-precision representation of Zero with the same sign as src.

Otherwise, the result is the half-precision representation of src rounded to half-precision using the rounding mode specified by RN.

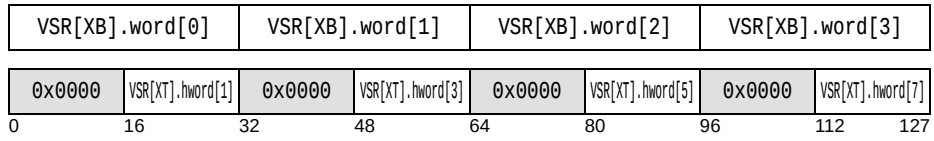
The result is zero-extended and placed into word element i of VSR[XT].

If a trap-enabled exception occurs, VSR[XT] is not modified.

Special Registers Altered:

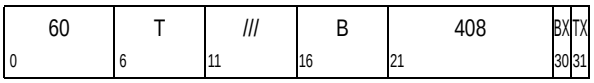
FX VXSNAN OX UX XX

VSR Data Layout for xvcvsphp



VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format XX2-form

xvcvspsxds XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   result{i:i+63} ← ConvertSPtoSD(VSR[XB]{i:i+31})
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxcvi_flag)  then SetFX(VXCVI)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
                            | (VE & vxcvi_flag)
                            | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the single-precision floating-point operand in word element i×2 of VSR[XB].

If src is a NaN, the result is the value 0x8000_0000_0000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than $2^{63}-1$, the result is 0x7FFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{63}, the result is 0x8000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit signed-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

The result is placed into doubleword element i of VSR[XT].

See Table 105.

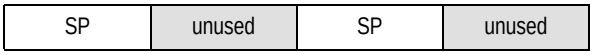
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

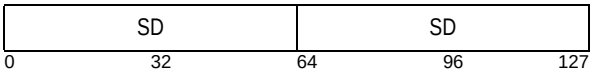
FX XX VXSNAN VXCVI

VSR Data Layout for xvcvspsxds

src = VSR[XB]



tgt = VSR[XT]



Programming Note

xvcvspsxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Single-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrspic*** which uses the rounding mode specified by RN.

	VE	XE	Inexact? (RoundToSPintegerTrunc(src) ≠ src)	Returned Results and Status Setting
src [Nmin-1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(ConvertSPtoSD(RoundToSPintegerTrunc(src)))
		0	yes	T(ConvertSPtoSD(RoundToSPintegerTrunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src ≥ Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), fx(VXSNAN), error()

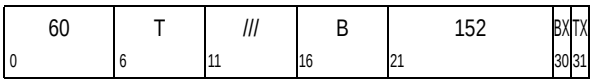
Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
Nmin	The smallest signed integer doubleword value, -2^{63} (0x8000_0000_0000_0000).
Nmax	The largest signed integer doubleword value, $2^{63}-1$ (0x7FFF_FFFF_FFFF_FFFF).
src	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,2}).
T(x)	The signed integer doubleword value x is placed in doubleword element i of VSR[XT] (where i ∈ {0,1}).

Table 106.Actions for xvcvspsxds

VSX Vector Convert with round to zero Single-Precision to Signed Word format XX2-form

xvcvspsxws XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   result{i:i+31} ← ConvertSPtoSW(VSR[XB]{i:i+31})
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxcvi_flag)  then SetFX(VXCVI)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
                            | (VE & vxcvi_flag)
                            | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

If src is a NaN, the result is the value 0x8000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than $2^{31}-1$, the result is 0x7FFF_FFFF, and VXCVI is set to 1.

Otherwise, if the rounded value is less than -2^{31}, the result is 0x8000_0000, and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit signed-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

The result is placed into word element i of VSR[XT].

See Table 105.

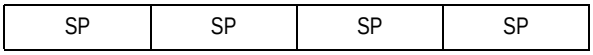
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

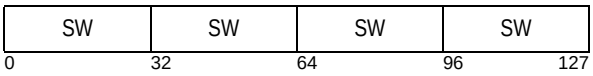
FX XX VXSNAN VXCVI

VSR Data Layout for xvcvspsxws

src = VSR[XB]



tgt = VSR[XT]



Programming Note

xvcvspsxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Single-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrspic*** which uses the rounding mode specified by RN.

	VE	XE	Inexact? (RoundToSPintegerTrunc(src) g src)	Returned Results and Status Setting
src [Nmin-1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(ConvertSPtoSW(RoundToSPintegerTrunc(src)))
		0	yes	T(ConvertSPtoSW(RoundToSPintegerTrunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src m Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), fx(VXSNAN), error()

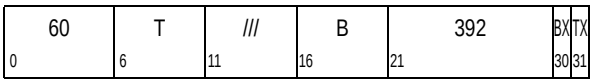
Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
Nmin	The smallest signed integer word value, -2^{31} (0x8000_0000).
Nmax	The largest signed integer word value, $2^{31}-1$ (0x7FFF_FFFF).
src	The single-precision floating-point value in word element i of VSR[XB] (where i C {0,1,2,3}).
T(x)	The signed integer word value x is placed in word element i of VSR[XT] (where i C {0,1,2,3}).

Table 107.Actions for xvcvspsxws

VSX Vector Convert with round to zero Single-Precision to Unsigned Doubleword format XX2-form

xvcvspuxds XT,XB




```
XT      ← TX || T
XB      ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   result{i:i+63} ← ConvertSPtoUD(VSR[XB]{i:i+31})
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxcvi_flag)  then SetFX(VXCVI)
   if(xx_flag)     then SetFX(XX)
   ex_flag       ← ex_flag | (VE & vxsnan_flag)
                           | (VE & vxcvi_flag)
                           | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the single-precision floating-point operand in word element i×2 of VSR[XB].

If src is a NaN, the result is the value 0x0000_0000_0000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than 2^{64}-1, the result is 0xFFFF_FFFF_FFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is 0x0000_0000_0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 64-bit unsigned-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

The result is placed into doubleword element i of VSR[XT].

See Table 105.

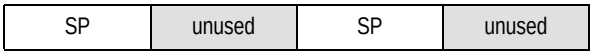
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

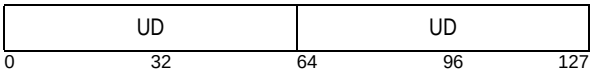
FX XX VXSNAN VXCVI

VSR Data Layout for xvcvspuxds

src = VSR[XB]



tgt = VSR[XT]



Programming Note

xvcvspuxds rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Single-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrspic*** which uses the rounding mode specified by RN.

	VE	XE	Inexact? (RoundToSPintegerTrunc(src) g src)	Returned Results and Status Setting
src [Nmin-1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(ConvertSPtoUD(RoundToSPintegerTrunc(src)))
		0	yes	T(ConvertSPtoUD(RoundToSPintegerTrunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src m Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), fx(VXSNAN), error()

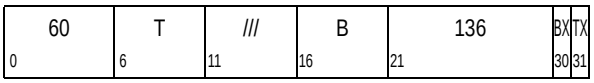
Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
Nmin	The smallest unsigned integer doubleword value, 0 (0x0000_0000_0000_0000).
Nmax	The largest unsigned integer doubleword value, 2^{64}-1 (0xFFFF_FFFF_FFFF_FFFF).
src	The single-precision floating-point value in word element i of VSR[XB] (where i C {0,2}).
T(x)	The unsigned integer doubleword value x is placed in doubleword element i of VSR[XT] (where i C {0,1}).

Table 108.Actions for xvcvspuxds

VSX Vector Convert with round to zero Single-Precision to Unsigned Word format XX2-form

xvcvspuxws XT,XB




```
XT      ← TX || T
XB      ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   result{i:i+31} ← ConvertSPtoUW(VSR[XB]{i:i+31})
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxcvi_flag)  then SetFX(VXCVI)
   if(xx_flag)     then SetFX(XX)
   ex_flag       ← ex_flag | (VE & vxsnan_flag)
                           | (VE & vxcvi_flag)
                           | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

If src is a NaN, the result is the value 0x0000_0000 and VXCVI is set to 1. If src is an SNaN, VXSNAN is also set to 1.

Otherwise, src is rounded to a floating-point integer using the rounding mode Round Toward Zero.

If the rounded value is greater than $2^{32}-1$, the result is 0xFFFF_FFFF and VXCVI is set to 1.

Otherwise, if the rounded value is less than 0, the result is 0x0000_0000 and VXCVI is set to 1.

Otherwise, the result is the rounded value converted to 32-bit unsigned-integer format, and if the result is inexact (i.e., not equal to src), XX is set to 1.

The result is placed into word element i of VSR[XT].

See Table 105.

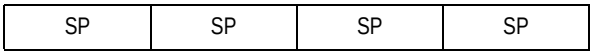
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

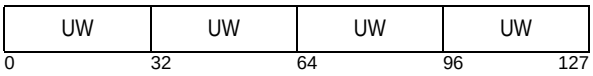
FX XX VXSNAN VXCVI

VSR Data Layout for xvcvspuxws

src = VSR[XB]



tgt = VSR[XT]



Programming Note

xvcvspuxws rounds using Round towards Zero rounding mode. For other rounding modes, software must use a *Round to Single-Precision Integer* instruction that corresponds to the desired rounding mode, including ***xvrspic*** which uses the rounding mode specified by RN.

	VE	XE	Inexact? (RoundToSPintegerTrunc(src) g src)	Returned Results and Status Setting
src [Nmin-1	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
Nmin-1 < src < Nmin	–	0	yes	T(Nmin), fx(XX)
		1	yes	fx(XX), error()
src = Nmin	–	–	no	T(Nmin)
Nmin < src < Nmax	–	–	no	T(ConvertSPtoUW(RoundToSPintegerTrunc(src)))
		0	yes	T(ConvertSPtoUW(RoundToSPintegerTrunc(src))), fx(XX)
		1	yes	fx(XX), error()
src = Nmax	–	–	no	T(Nmax) Note: This case cannot occur as Nmax is not representable in SP format but is included here for completeness.
Nmax < src < Nmax+1	–	0	yes	T(Nmax), fx(XX)
		1	yes	fx(XX), error()
src m Nmax+1	0	–	–	T(Nmax), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a QNaN	0	–	–	T(Nmin), fx(VXCVI)
	1	–	–	fx(VXCVI), error()
src is a SNaN	0	–	–	T(Nmin), fx(VXCVI), fx(VXSNAN)
	1	–	–	fx(VXCVI), fx(VXSNAN), error()

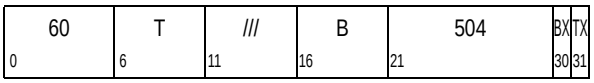
Explanation:

fx(x)	FX is set to 1 if x=0. x is set to 1.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of VSR[XT] is suppressed.
Nmin	The smallest unsigned integer word value, 0 (0x0000_0000).
Nmax	The largest unsigned integer word value, 2^{32}-1 (0xFFFF_FFFF).
src	The single-precision floating-point value in word element i of VSR[XB] (where i C {0,1,2,3}).
T(x)	The unsigned integer word value x is placed in word element i of VSR[XT] (where i C {0,1,2,3}).

Table 109.Actions for xvcvspuxws

VSX Vector Convert with round Signed Doubleword to Double-Precision format XX2-form

xvcvsxddp XT,XB




```
XT      ← TX || T
XB      ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   v{0:inf}       ← ConvertSDtoFP(VSR[XB]{i:i+63})
   result{i:i+63} ← RoundToDP(RN,v)
   if(xx_flag) then SetFX(XX)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the signed integer in doubleword element i of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by RN.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

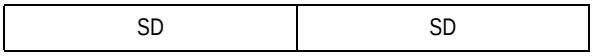
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

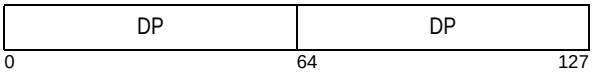
FX XX

VSR Data Layout for xvcvsxddp

src = VSR[XB]

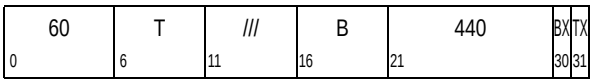


tgt = VSR[XT]



VSX Vector Convert with round Signed Doubleword to Single-Precision format XX2-form

xvcvsxdsp XT,XB




```
XT      ← TX || T
XB      ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   v{0:inf}          ← ConvertSDtoFP(VSR[XB]{i:i+63})
   result{i:i+31}    ← RoundToSP(RN,v)
   result{i+32:i+63} ← 0xUUUU_UUUU
   if(xx_flag) then SetFX(XX)
   ex_flag           ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the signed integer in doubleword element i of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by RN.

The result is placed into bits 0:31 of doubleword element i of VSR[XT] in single-precision format.

The contents of bits 32:63 of doubleword element i of VSR[XT] are undefined.

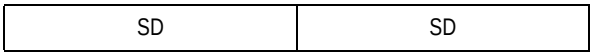
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

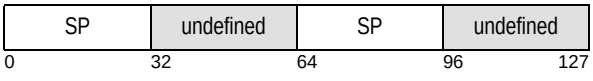
FX XX

VSR Data Layout for xvcvsxdsp

src = VSR[XB]

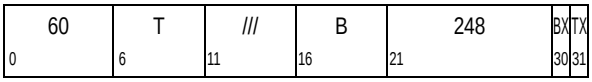


tgt = VSR[XT]



VSX Vector Convert Signed Word to Double-Precision format XX2-form

xvcvsxwdp XT,XB




```
do i = 0 to 1
   src ← bfp_CONVERT_FROM_SI32(VSR[32×BX+B].dword[i].word[0])
   VSR[32×TX+T].dword[i] ← bfp64_CONVERT_FROM_BFP(src)
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the signed integer value in bits 0:31 of doubleword element i of VSR[XB].

src is placed into doubleword element i of VSR[XT] in double-precision format.

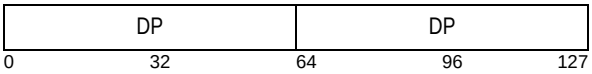
Special Registers Altered

None

VSR Data Layout for xvcvsxwdp

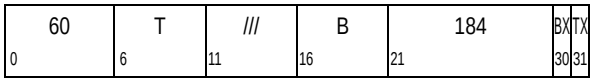
src = VSR[XB]

tgt = VSR[XT]



VSX Vector Convert with round Signed Word to Single-Precision format XX2-form

xvcvsxwsp XT,XB




```
ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()
   v{0:inf}       ← ConvertSWtoFP(VSR[32×BX+B].word[i])
   result.word[i] ← RoundToSP(RN,v)
   if(xx_flag) then SetFX(XX)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if(ex_flag=0) then VSR[32×TX+T] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src be the signed integer in word element i of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by RN.

The result is placed into word element i of VSR[XT] in single-precision format.

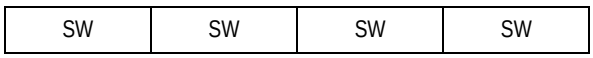
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

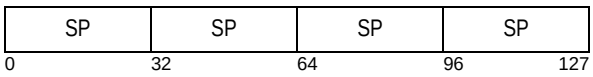
FX XX

VSR Data Layout for xvcvsxwsp

src = VSR[XB]

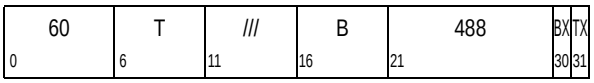


tgt = VSR[XT]



VSX Vector Convert with round Unsigned Doubleword to Double-Precision format XX2-form

xvcvuxddp XT,XB




```
XT      ← TX || T
XB      ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   v{0:inf}       ← ConvertUDtoFP(VSR[XB]{i:i+63})
   result{i:i+63} ← RoundToDP(RN,v)
   if(xx_flag) then SetFX(XX)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the unsigned integer in doubleword element i of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to double-precision using the rounding mode specified by RN.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

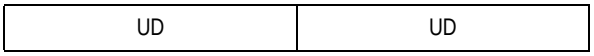
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

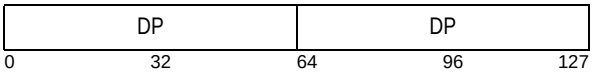
FX XX

VSR Data Layout for xvcvuxddp

src = VSR[XB]

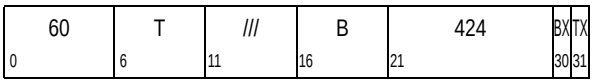


tgt = VSR[XT]



VSX Vector Convert with round Unsigned Doubleword to Single-Precision format XX2-form

xvcvuxdsp XT,XB




```
XT      ← TX || T
XB      ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   v{0:inf}          ← ConvertUDtoFP(VSR[XB]{i:i+63})
   result{i:i+31}    ← RoundToSP(RN,v)
   result{i+32:i+63} ← 0xUUUU_UUUU
   if(xx_flag) then SetFX(XX)
   ex_flag           ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the unsigned integer in doubleword element i of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by RN.

The result is placed into bits 0:31 of doubleword element i of VSR[XT] in single-precision format.

The contents of bits 32:63 of doubleword element i of VSR[XT] are undefined.

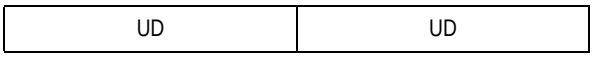
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

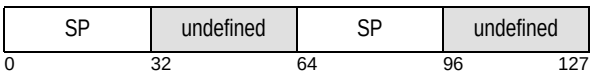
FX XX

VSR Data Layout for xvcvuxdsp

src = VSR[XB]

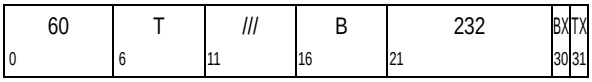


tgt = VSR[XT]



VSX Vector Convert Unsigned Word to Double-Precision format XX2-form

xvcvuxwdp XT,XB




```
do i = 0 to 1
   src ← bfp_CONVERT_FROM_UI32(VSR[32×BX+B].dword[i].word[0])
   VSR[32×TX+T].dword[i] ← bfp64_CONVERT_FROM_BFP(src)
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the unsigned integer value in bits 0:31 of doubleword element i of VSR[XB].

src is placed into doubleword element i of VSR[XT] in double-precision format.

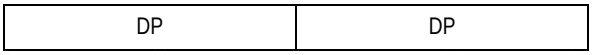
Special Registers Altered

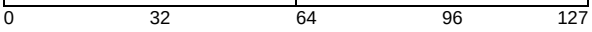
None

VSR Data Layout for xvcvuxwdp

src = VSR[XB]

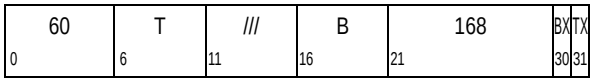
tgt = VSR[XT]





VSX Vector Convert with round Unsigned Word to Single-Precision format XX2-form

xvcvuxwsp XT,XB




```
XT      ← TX || T
XB      ← BX || B
ex_flag ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   v{0:inf}       ← ConvertUWtoFP(VSR[XB]{i:i+31})
   result{i:i+31} ← RoundToSP(RN,v)
   if(xx_flag) then SetFX(XX)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src be the unsigned integer value in word element i of VSR[XB].

src is converted to an unbounded-precision floating-point value and rounded to single-precision using the rounding mode specified by RN.

The result is placed into word element i of VSR[XT] in single-precision format.

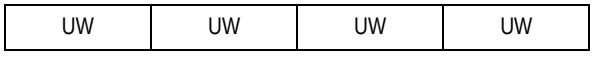
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

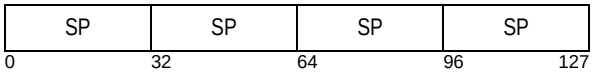
FX XX

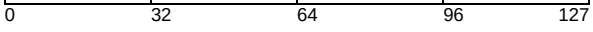
VSR Data Layout for xvcvuxwsp

src = VSR[XB]



tgt = VSR[XT]





VSX Vector Divide Double-Precision XX3-form

xvdivdp XT,XA,XB

60	T	A	B	120	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1           ← VSR[XA]{i:i+63}
   src2           ← VSR[XB]{i:i+63}
   v{0:inf}       ← DivideDP(src1,src2)
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxidi_flag)  then SetFX(VXIDI)
   if(vxisi_flag)  then SetFX(VXZDZ)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   if(zx_flag)     then SetFX(ZX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxidi_flag)
   ex_flag        ← ex_flag | (VE & vxzdz_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (ZE & zx_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is divided[1] by src2, producing a quotient having unbounded range and precision.

The quotient is normalized[2].

See Table 110.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 98, “Vector Floating-Point Final Result,” on page 661.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX ZX XX
VXSNAN VXIDI VXZDZ

VSR Data Layout for xvdivdp

src1 = VSR[XA]

DP	DP

src2 = VSR[XB]

DP	DP

tgt = VSR[XT]

DP	DP

0 64 127

1. Floating-point division is based on exponent subtraction and division of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← dQNaN vxidi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Zero	v ← D(src1,src2)	v ← +Infinity zx_flag ← 1	v ← –Infinity zx_flag ← 1	v ← D(src1,src2)	v ← –Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← +Zero	v ← +Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← –Zero	v ← –Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← –Zero	v ← –Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← +Zero	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Zero	v ← D(src1,src2)	v ← –Infinity zx_flag ← 1	v ← +Infinity zx_flag ← 1	v ← D(src1,src2)	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxidi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
D(x,y)	Return the normalized quotient of floating-point value x divided by floating-point value y, having unbounded range and precision.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 110.Actions for xvdivdp (element i)

VSX Vector Divide Single-Precision XX3-form

xvdivsp XT,XA,XB

60	T	A	B	88	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1           ← VSR[XA]{i:i+31}
   src2           ← VSR[XB]{i:i+31}
   v{0:inf}       ← DivideSP(src1,src2)
   result{i:i+31} ← RoundToSP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxidi_flag)  then SetFX(VXIDI)
   if(vxisi_flag)  then SetFX(VXZDZ)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   if(zx_flag)     then SetFX(ZX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxidi_flag)
   ex_flag        ← ex_flag | (VE & vxzdz_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (ZE & zx_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

src1 is divided[1] by src2, producing a quotient having unbounded range and precision.

The quotient is normalized[2].

See Table 111.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 98, “Vector Floating-Point Final Result,” on page 661.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX ZX XX
VXSNAN VXIDI VXZDZ

VSR Data Layout for xvdivsp

src1 = VSR[XA]

SP	SP	SP	SP

src2 = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

1. Floating-point division is based on exponent subtraction and division of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← dQNaN vxidi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Zero	v ← D(src1,src2)	v ← +Infinity zx_flag ← 1	v ← –Infinity zx_flag ← 1	v ← D(src1,src2)	v ← –Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← +Zero	v ← +Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← –Zero	v ← –Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← –Zero	v ← –Zero	v ← dQNaN vxzdz_flag ← 1	v ← dQNaN vxzdz_flag ← 1	v ← +Zero	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Zero	v ← D(src1,src2)	v ← –Infinity zx_flag ← 1	v ← +Infinity zx_flag ← 1	v ← D(src1,src2)	v ← +Zero	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxidi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxidi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

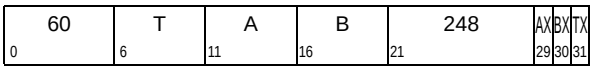
Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (`0x7FC0_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
D(x,y)	Return the normalized quotient of floating-point value x divided by floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 111.Actions for xvdivsp (element i)

VSX Vector Insert Exponent Double-Precision XX3-form

xviexpdp XT,XA,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 1
   src1 ← VSR[32×AX+A].dword[i]
   src2 ← VSR[32×BX+B].dword[i]

   VSR[32×TX+T].dword[i].bit[0]     ← src1.bit[0]
   VSR[32×TX+T].dword[i].bit[1:11]  ← src2.bit[53:63]
   VSR[32×TX+T].dword[i].bit[12:63] ← src1.bit[12:63]
end
```

Let XT be the sum 32×TX + T.
Let XA be the sum 32×AX + A.
Let XB be the sum 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src1 be the unsigned integer value in doubleword element i of VSR[XA].

Let src2 be the unsigned integer value in doubleword element i of VSR[XB].

The contents of bits 0 of src1 are placed into bit 0 of doubleword element i of VSR[XT].

The contents of bits 53:63 of src2 are placed into bits 1:11 of doubleword element i of VSR[XT].

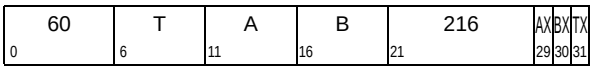
The contents of bits 12:63 of src1 are placed into bits 12:63 of doubleword element i of VSR[XT].

Special Registers Altered:

None

VSX Vector Insert Exponent Single-Precision XX3-form

xviexpsp XT,XA,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 3
   src1 ← VSR[32×AX+A].word[i]
   src2 ← VSR[32×BX+B].word[i]

   VSR[32×TX+T].word[i].bit[0]    ← src1.bit[0]
   VSR[32×TX+T].word[i].bit[1:8]  ← src2.bit[24:31]
   VSR[32×TX+T].word[i].bit[9:31] ← src1.bit[9:31]
end
```

Let XT be the sum 32×TX + T.
Let XA be the sum 32×AX + A.
Let XB be the sum 32×BX + B.

For each integer value i from 0 to 3, do the following.

Let src1 be the unsigned integer value in word element i of VSR[XA].

Let src2 be the unsigned integer value in word element i of VSR[XB].

The contents of bits 0 of src1 are placed into bit 0 of word element i of VSR[XT].

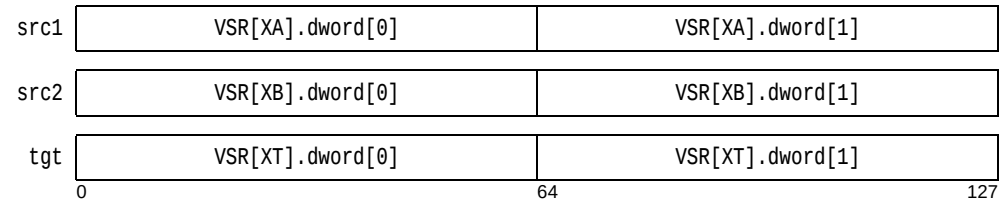
The contents of bits 24:31 of src2 are placed into bits 1:8 of word element i of VSR[XT].

The contents of bits 9:31 of src1 are placed into bits 9:31 of word element i of VSR[XT].

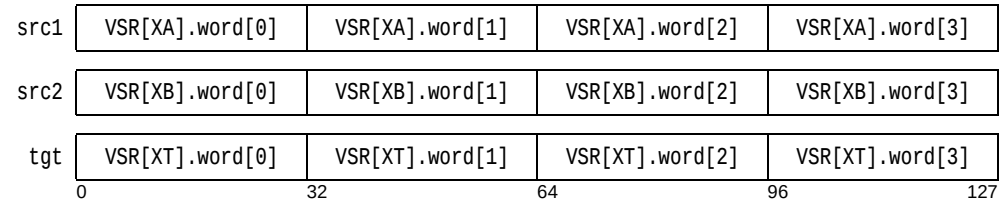
Special Registers Altered:

None

VSR Data Layout for xviexpdp



VSR Data Layout for xviexpsp



VSX Vector Multiply-Add Double-Precision XX3-form

xvmaddadp XT,XA,XB

60	T	A	B	97	AX	BX	TX
0	6	11	16	21	29	30	31

xvmaddmdp XT,XA,XB

60	T	A	B	105	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1 ← VSR[XA]{i:i+63}
   src2 ← "xvmaddadp" ? VSR[XT]{i:i+63} : VSR[XB]{i:i+63}
   src3 ← "xvmaddadp" ? VSR[XB]{i:i+63} : VSR[XT]{i:i+63}
   v{0:inf}       ← MultiplyAddDP(src1,src3,src2)
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

For ***xvmaddadp***, do the following.

– Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
– Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XT].
– Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XB].

For ***xvmaddmdp***, do the following.

– Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
– Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].
– Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 112.

src2 is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 112.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 98, "Vector Floating-Point Final Result," on page 661.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

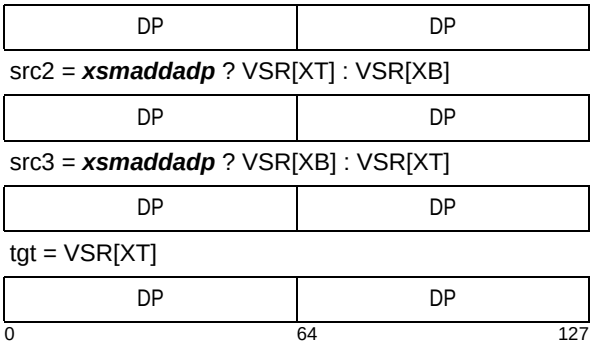
Special Registers Altered

FX OX UX XX VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvmadd(a|m)dp

src1 = VSR[XA]



Part 1: Multiply

src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← –Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+NZF	p ← –Infinity	p ← M(src1,src3)	p ← –Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add

p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
–Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
–Zero	v ← –Infinity	v ← src2	v ← –Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← –Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	For ***xvmaddadp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}). For ***xvmaddmdp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
src3	For ***xvmaddadp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}). For ***xvmaddmdp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
	Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 112.Actions for xvmadd(a|m)dp

VSX Vector Multiply-Add Single-Precision XX3-form

xvmaddasp XT,XA,XB

60	T	A	B	65	AX	BX	TX
0	6	11	16	21	29	30	31

xvmaddmsp XT,XA,XB

60	T	A	B	73	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT       ← TX || T
XA       ← AX || A
XB       ← BX || B
ex_flag  ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1 ← VSR[XA]{i:i+31}
   src2 ← "xvmaddasp" ? VSR[XT]{i:i+31} : VSR[XB]{i:i+31}
   src3 ← "xvmaddasp" ? VSR[XB]{i:i+31} : VSR[XT]{i:i+31}
   v{0:inf}       ← MultiplyAddSP(src1,src3,src2)
   result{i:i+63} ← RoundToSP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

For ***xvmaddasp***, do the following.
- Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
- Let src2 be the single-precision floating-point operand in word element i of VSR[XT].
- Let src3 be the single-precision floating-point operand in word element i of VSR[XB].

For ***xvmaddmsp***, do the following.
- Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
- Let src2 be the single-precision floating-point operand in word element i of VSR[XB].
- Let src3 be the single-precision floating-point operand in word element i of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 113.

src2 is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 113.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 98, "Vector Floating-Point Final Result," on page 661.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

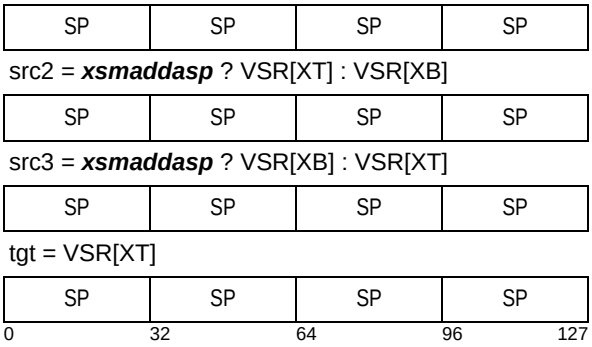
Special Registers Altered

FX OX UX XX VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvmadd(a|m)sp

src1 = VSR[XA]



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← –Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← –Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← –Infinity	v ← src2	v ← –Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← –Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	For ***xvmaddasp***, the single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}). For ***xvmaddmsp***, the single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
src3	For ***xvmaddasp***, the single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}). For ***xvmaddmsp***, the single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (`0x7FC0_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
	Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 113.Actions for xvmadd(a|m)sp

VSX Vector Maximum Double-Precision XX3-form

xvmaxdp XT,XA,XB

60	T	A	B	224	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT       ← TX || T
XA       ← AX || A
XB       ← BX || B
ex_flag  ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1           ← VSR[XA]{i:i+63}
   src2           ← VSR[XB]{i:i+63}
   result{i:i+63} ← MaximumDP(src1,src2)
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

If src1 is greater than src2, src1 is placed into doubleword element i of VSR[XT] in double-precision format. Otherwise, src2 is placed into doubleword element i of VSR[XT] in double-precision format.

The maximum of +0 and –0 is +0. The maximum of a QNaN and any value is that value. The maximum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

See Table 114.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX VXSNAN

VSR Data Layout for xvmaxdp

src1 = VSR[XA]

DP	DP

src2 = VSR[XB]

DP	DP

tgt = VSR[XT]

DP	DP

0 64 127

		src2							
		–Infinity	**–NZF**	**–Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**–Infinity**	T(src1)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–NZF	T(src1)	T(M(src1,src2))	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–Zero	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Zero	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+NZF	T(src1)	T(src1)	T(src1)	T(src1)	T(M(src1,src2))	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Infinity	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	The double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}).
NZF	Nonzero finite number.
Q(x)	Return a QNaN with the payload of x.
M(x,y)	Return the greater of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element i (i∈{0,1}) of VSR[XT] in double-precision format. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-point Invalid Operation Exception (SNaN). If VE=1, update of VSR[XT] is suppressed.

Table 114.Actions for xvmaxdp

VSX Vector Maximum Single-Precision XX3-form

xvmaxsp XT,XA,XB

60	T	A	B	192	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1           ← VSR[XA]{i:i+31}
   src2           ← VSR[XB]{i:i+31}
   result{i:i+63} ← MaximumSP(src1,src2)
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

If src1 is greater than src2, src1 is placed into word element i of VSR[XT] in single-precision format. Otherwise, src2 is placed into word element i of VSR[XT] in single-precision format.

The maximum of +0 and –0 is +0. The maximum of a QNaN and any value is that value. The maximum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

See Table 115.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX VXSNAN

VSR Data Layout for xvmaxsp

src1 = VSR[XA]

SP	SP	SP	SP

src2 = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

		src2							
		–Infinity	**–NZF**	**–Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**–Infinity**	T(src1)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–NZF	T(src1)	T(M(src1,src2))	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–Zero	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Zero	T(src1)	T(src1)	T(src1)	T(src1)	T(src2)	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+NZF	T(src1)	T(src1)	T(src1)	T(src1)	T(M(src1,src2))	T(src2)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Infinity	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

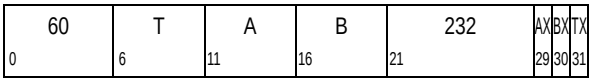
Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	The single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
NZF	Nonzero finite number.
Q(x)	Return a QNaN with the payload of x.
M(x,y)	Return the greater of floating-point value x and floating-point value y.
T(x)	The value x is placed in word element i (i∈{0,1,2,3}) of VSR[XT] in single-precision format. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-point Invalid Operation Exception (SNaN). If VE=1, update of VSR[XT] is suppressed.

Table 115.Actions for xvmaxsp

VSX Vector Minimum Double-Precision XX3-form

xvmindp XT,XA,XB




```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1           ← VSR[XA]{i:i+63}
   src2           ← VSR[XB]{i:i+63}
   result{i:i+63} ← MinimumDP(src1,src2)
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

If src1 is less than src2, src1 is placed into doubleword element i of VSR[XT] in double-precision format. Otherwise, src2 is placed into doubleword element i of VSR[XT] in double-precision format.

The minimum of +0 and –0 is –0. The minimum of a QNaN and any value is that value. The minimum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

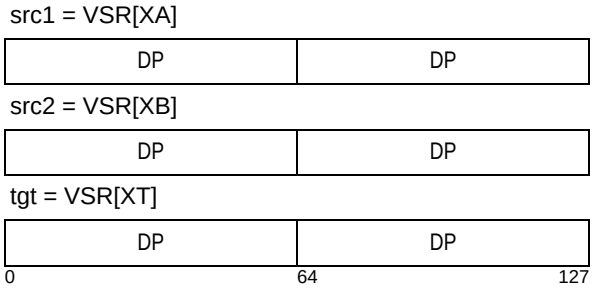
See Table 116.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX VXSNAN

VSR Data Layout for xvmindp



		src2							
		–Infinity	**–NZF**	**–Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**–Infinity**	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–NZF	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–Zero	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Zero	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+NZF	T(src2)	T(src2)	T(src2)	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Infinity	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	The double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}).
NZF	Nonzero finite number.
Q(x)	Return a QNaN with the payload of x.
M(x,y)	Return the lesser of floating-point value x and floating-point value y.
T(x)	The value x is placed in doubleword element i (i∈{0,1}) of VSR[XT] in double-precision format. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-point Invalid Operation Exception (SNaN). If VE=1, update of VSR[XT] is suppressed.

Table 116.Actions for xvmindp

VSX Vector Minimum Single-Precision XX3-form

xvminsp XT,XA,XB

60	T	A	B	200	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1           ← VSR[XA]{i:i+31}
   src2           ← VSR[XB]{i:i+31}
   result{i:i+31} ← MinimumSP(src1,src2)
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

If src1 is less than src2, src1 is placed into word element i of VSR[XT] in single-precision format. Otherwise, src2 is placed into word element i of VSR[XT] in single-precision format.

The minimum of +0 and –0 is –0. The minimum of a QNaN and any value is that value. The minimum of any value and an SNaN when VE=0 is that SNaN converted to a QNaN.

See Table 117.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX VXSNAN

VSR Data Layout for xvminsp

src1 = VSR[XA]

SP	SP	SP	SP

src2 = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

		src2							
		–Infinity	**–NZF**	**–Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
src1	**–Infinity**	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–NZF	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	–Zero	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Zero	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+NZF	T(src2)	T(src2)	T(src2)	T(src2)	T(M(src1,src2))	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	+Infinity	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1)	T(Q(src2)) fx(VXSNAN)
	QNaN	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src2)	T(src1)	T(src1) fx(VXSNAN)
	SNaN	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)	T(Q(src1)) fx(VXSNAN)

Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	The single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
NZF	Nonzero finite number.
Q(x)	Return a QNaN with the payload of x.
M(x,y)	Return the lesser of floating-point value x and floating-point value y.
T(x)	The value x is placed in word element i (i∈{0,1,2,3}) of VSR[XT] in single-precision format. FPRF, FR and FI are not modified.
fx(x)	If x is equal to 0, FX is set to 1. x is set to 1.
VXSNAN	Floating-point Invalid Operation Exception (SNaN). If VE=1, update of VSR[XT] is suppressed.

Table 117.Actions for xvminsp

VSX Vector Multiply-Subtract Double-Precision XX3-form

xvmsubadp XT,XA,XB

60	T	A	B	113	AX	BX	TX
0	6	11	16	21	29	30	31

xvmsubmdp XT,XA,XB

60	T	A	B	121	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1 ← VSR[XA]{i:i+63}
   src2 ← "xvmsubadp" ? VSR[XT]{i:i+63} : VSR[XB]{i:i+63}
   src3 ← "xvmsubadp" ? VSR[XB]{i:i+63} : VSR[XT]{i:i+63}
   v{0:inf}       ← MultiplyAddDP(src1,src3,NegateDP(src2))
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

For ***xvmsubadp***, do the following.
- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XT].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XB].

For ***xvmsubmdp***, do the following.
- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 118.

src2 is negated and added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 118.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 98, "Vector Floating-Point Final Result," on page 661.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

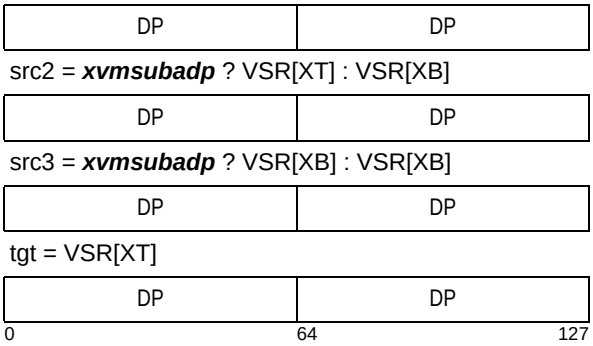
Special Registers Altered

FX OX UX XX VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvmsub(a|m)dp

src1 = VSR[XA]



Part 1: Multiply		src3							
		–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
src1	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← –Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← –Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract		src2							
		–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
p	–Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← +Infinity	v ← –src2	v ← –Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← +Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

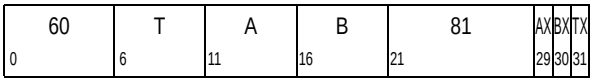
Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	For ***xvmsubadp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}). For ***xvmsubmdp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
src3	For ***xvmsubadp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}). For ***xvmsubmdp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision.
	Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

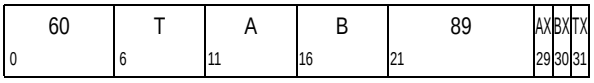
Table 118.Actions for xvmsub(a|m)dp

VSX Vector Multiply-Subtract Single-Precision XX3-form

xvmsubasp XT,XA,XB



xvmsubmsp XT,XA,XB




```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1           ← VSR[XA]{i:i+31}
   src2 ← "xvmsubasp" ? VSR[XT]{i:i+31} : VSR[XB]{i:i+31}
   src3 ← "xvmsubasp" ? VSR[XB]{i:i+31} : VSR[XT]{i:i+31}
   v{0:inf}       ← MultiplyAddSP(src1,src3,NegateSP(src2))
   result{i:i+31} ← RoundToSP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

For ***xvmsubasp***, do the following.
- Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
- Let src2 be the single-precision floating-point operand in word element i of VSR[XT].
- Let src3 be the single-precision floating-point operand in word element i of VSR[XB].

For ***xvmsubmsp***, do the following.
- Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
- Let src2 be the single-precision floating-point operand in word element i of VSR[XB].
- Let src3 be the single-precision floating-point operand in word element i of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 119.

src2 is negated and added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 119.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 98, "Vector Floating-Point Final Result," on page 661.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

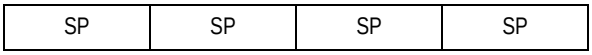
Special Registers Altered

FX OX UX XX VXSNAN VXISI VXIMZ

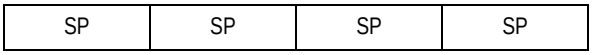
1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvmsub(a|m)sp

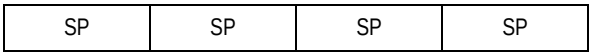
src1 = VSR[XA]



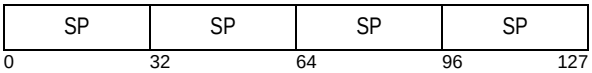
src2 = ***xvmsubasp*** ? VSR[XT] : VSR[XB]



src3 = ***xvmsubasp*** ? VSR[XB] : VSR[XT]



tgt = VSR[XT]



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← +Zero	p ← –Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← –Zero	p ← +Zero	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← +Infinity	v ← –src2	v ← –Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← +Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	For ***xvmsubasp***, the single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}). For ***xvmsubmsp***, the single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
src3	For ***xvmsubasp***, the single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}). For ***xvmsubmsp***, the single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (`0x7FC0_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 119.Actions for xvmsub(a|m)sp

VSX Vector Multiply Double-Precision XX3-form

xvmuldp XT,XA,XB

60	T	A	B	112	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1           ← VSR[XA]{i:i+63}
   src3           ← VSR[XB]{i:i+63}
   v{0:inf}       ← MultiplyDP(src1,src3)
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src1 is multiplied[1] by src2, producing a product having unbounded range and precision.

The product is normalized[2].

See Table 120.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 98, “Vector Floating-Point Final Result,” on page 661.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX XX VXSNAN VXIMZ

VSR Data Layout for xvmuldp

src1 = VSR[XA]

DP	DP

src2 = VSR[XB]

DP	DP

tgt = VSR[XT]

DP	DP

0 64 127

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Infinity	v ← M(src1,src2)	v ← +Zero	v ← –Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← dQNaN vximz_flag ← 1	v ← +Zero	v ← +Zero	v ← –Zero	v ← –Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← dQNaN vximz_flag ← 1	v ← –Zero	v ← –Zero	v ← +Zero	v ← +Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Infinity	v ← M(src1,src2)	v ← –Zero	v ← +Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← –Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 120.Actions for xvmuldp

VSX Vector Multiply Single-Precision XX3-form

xvmulsp XT,XA,XB

60	T	A	B	80	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1           ← VSR[XA]{i:i+31}
   src3           ← VSR[XB]{i:i+31}
   v{0:inf}       ← MultiplySP(src1,src3)
   result{i:i+31} ← RoundToSP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

src1 is multiplied[1] by src2, producing a product having unbounded range and precision.

The product is normalized[2].

See Table 121.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 98, “Vector Floating-Point Final Result,” on page 661.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX XX VXSNAN VXIMZ

VSR Data Layout for xvmulsp

src1 = VSR[XA]

SP	SP	SP	SP

src2 = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Infinity	v ← M(src1,src2)	v ← +Zero	v ← –Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← dQNaN vximz_flag ← 1	v ← +Zero	v ← +Zero	v ← –Zero	v ← –Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← dQNaN vximz_flag ← 1	v ← –Zero	v ← –Zero	v ← +Zero	v ← +Zero	v ← dQNaN vximz_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← –Infinity	v ← M(src1,src2)	v ← –Zero	v ← +Zero	v ← M(src1,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← –Infinity	v ← +Infinity	v ← dQNaN vximz_flag ← 1	v ← dQNaN vximz_flag ← 1	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

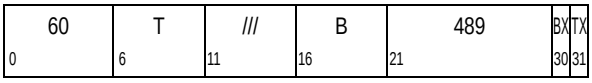
Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (`0x7FC0_0000`).
NZF	Nonzero finite number.
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 121.Actions for xvmulsp

VSX Vector Negative Absolute Double-Precision XX2-form

xvnabsdp XT,XB




```
XT ← TX || T
XB ← BX || B

do i=0 to 127 by 64
   VSR[XT]{i:i+63} ← 0b1 || VSR[XB]{i+1:i+63}
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

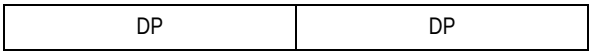
For each vector element i from 0 to 1, do the following.
The contents of doubleword element i of VSR[XB], with bit 0 set to 1, is placed into doubleword element i of VSR[XT].

Special Registers Altered

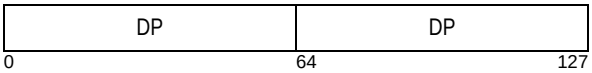
None

VSR Data Layout for xvnabsdp

src = VSR[XB]

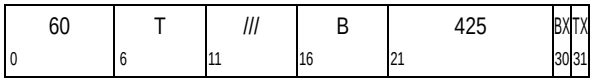


tgt = VSR[XT]



VSX Vector Negative Absolute Single-Precision XX2-form

xvnabssp XT,XB




```
XT ← TX || T
XB ← BX || B

do i=0 to 127 by 32
   VSR[XT]{i:i+31} ← 0b1 || VSR[XB]{i+1:i+31}
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

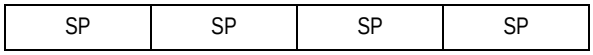
For each vector element i from 0 to 3, do the following.
The contents of word element i of VSR[XB], with bit 0 set to 1, is placed into word element i of VSR[XT].

Special Registers Altered

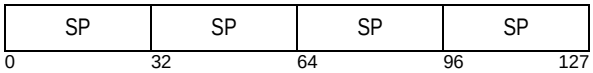
None

VSR Data Layout for xvnabssp

src = VSR[XB]

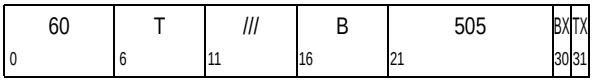


tgt = VSR[XT]



VSX Vector Negate Double-Precision XX2-form

xvnegdp XT,XB




```
XT ← TX || T
XB ← BX || B

do i=0 to 127 by 64
   VSR[XT]{i:i+63} ← ~VSR[XB]{i} || VSR[XB]{i+1:i+63}
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

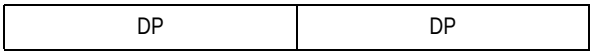
For each vector element i from 0 to 1, do the following.
The contents of doubleword element i of VSR[XB], with bit 0 complemented, is placed into doubleword element i of VSR[XT].

Special Registers Altered

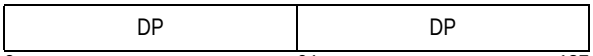
None

VSR Data Layout for xvnegdp

src = VSR[XB]

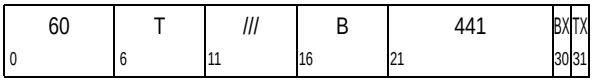


tgt = VSR[XT]



VSX Vector Negate Single-Precision XX2-form

xvnegsp XT,XB




```
XT ← TX || T
XB ← BX || B

do i=0 to 127 by 32
   VSR[XT]{i:i+31} ← ~VSR[XB]{i} || VSR[XB]{i+1:i+31}
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

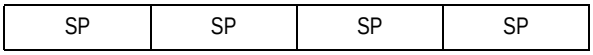
For each vector element i from 0 to 3, do the following.
The contents of word element i of VSR[XB], with bit 0 complemented, is placed into word element i of VSR[XT].

Special Registers Altered

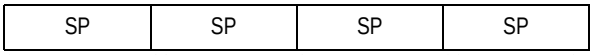
None

VSR Data Layout for xvnegsp

src = VSR[XB]



tgt = VSR[XT]



VSX Vector Negative Multiply-Add Double-Precision XX3-form

xvnmaddadp XT,XA,XB

60	T	A	B	225	AX	BX	TX
0	6	11	16	21	29	30	31

xvnmaddmdp XT,XA,XB

60	T	A	B	233	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1 ← VSR[XA]{i:i+63}
   src2 ← "xvnmaddadp" ? VSR[XT]{i:i+63} : VSR[XB]{i:i+63}
   src3 ← "xvnmaddadp" ? VSR[XB]{i:i+63} : VSR[XT]{i:i+63}
   v{0:inf}       ← MultiplyAddDP(src1,src3,src2)
   result{i:i+63} ← NegateDP(RoundToDP(RN,v))
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

For ***xvnmaddadp***, do the following.

- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XT].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XB].

For ***xvnmaddmdp***, do the following.

- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 122.

src2 is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 122.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is negated and placed into doubleword element i of VSR[XT] in double-precision format.

See Table 123, "Vector Floating-Point Final Result with Negation," on page 730.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

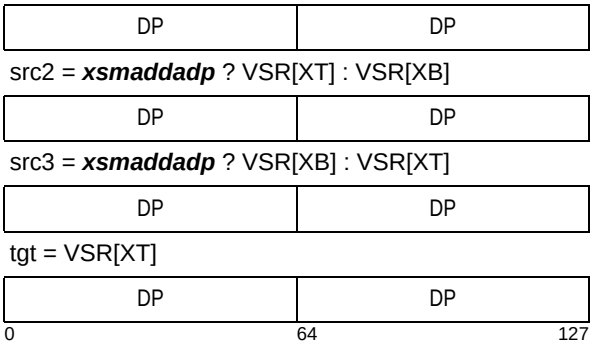
Special Registers Altered

FX OX UX XX VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvnmadd(a|m)dp

src1 = VSR[XA]



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← –Infinity	v ← src2	v ← –Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← –Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	For ***xvnmaddadp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}). For ***xvnmaddmdp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
src3	For ***xvnmaddadp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}). For ***xvnmaddmdp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
	Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 122.Actions for xvnmadd(a|m)dp

Case	VE	OE	UE	ZE	XE	vxsnan_flag	vximz_flag	vxisi_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Special	–	–	–	–	–	0	0	0	–	–	–	–	T(N(r))
	0	–	–	–	–	–	–	1	–	–	–	–	T(r), fx(VXISI)
	0	–	–	–	–	0	1	–	–	–	–	–	T(r), fx(VXIMZ)
	0	–	–	–	–	1	0	–	–	–	–	–	T(r), fx(VXSNAN)
	0	–	–	–	–	1	1	–	–	–	–	–	T(r), fx(VXSNAN), fx(VXIMZ)
	1	–	–	–	–	–	–	1	–	–	–	–	fx(VXISI), error()
	1	–	–	–	–	0	1	–	–	–	–	–	fx(VXIMZ), error()
	1	–	–	–	–	1	0	–	–	–	–	–	fx(VXSNAN), error()
	1	–	–	–	–	1	1	–	–	–	–	–	fx(VXSNAN), fx(VXIMZ), error()
Normal	–	–	–	–	–	–	–	–	no	–	–	–	T(N(r))
	–	–	–	–	0	–	–	–	yes	no	–	–	T(N(r)), fx(XX)
	–	–	–	–	0	–	–	–	yes	yes	–	–	T(N(r)), fx(XX)
	–	–	–	–	1	–	–	–	yes	no	–	–	T(N(r)), fx(XX), error()
	–	–	–	–	1	–	–	–	yes	yes	–	–	T(N(r)), fx(XX), error()
Overflow	–	0	–	–	0	–	–	–	–	–	–	–	T(N(r)), fx(OX), fx(XX)
	–	0	–	–	1	–	–	–	–	–	–	–	T(N(r)), fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	no	–	fx(OX), error()
	–	1	–	–	–	–	–	–	–	–	yes	no	fx(OX), fx(XX), error()
	–	1	–	–	–	–	–	–	–	–	yes	yes	fx(OX), fx(XX), error()

Explanation:

–	The results do not depend on this condition.
fx(x)	FX is set to 1 if x=0. x is set to 1.
q	The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
FI	Floating-Point Fraction Inexact status flag, $FPSCR_{FI}$. This status flag is nonsticky.
FR	Floating-Point Fraction Rounded status flag, $FPSCR_{FR}$.
OX	Floating-Point Overflow Exception status flag, $FPSCR_{OX}$.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.
N(x)	The value x is is negated by complementing the sign bit of x.
T(x)	The value x is placed in element i of VSR[XT] in the target precision format (where i ∈ {0,1} for results with 64-bit elements, and i ∈ {0,1,3,4}) for results with 32-bit elements).
UX	Floating-Point Underflow Exception status flag, $FPSCR_{UX}$
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$.
VXIMZ	Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, $FPSCR_{VXIMZ}$.
VXISI	Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, $FPSCR_{VXISI}$.
XX	Float-Point Inexact Exception status flag, $FPSCR_{XX}$. The flag is a sticky version of $FPSCR_{FI}$. When $FPSCR_{FI}$ is set to a new value, the new value of $FPSCR_{XX}$ is set to the result of ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.

Table 123.Vector Floating-Point Final Result with Negation

Case	VE	OE	UE	ZE	XE	vxsnan_flag	vximz_flag	vxisi_flag	Is r inexact? (r ≠ v)	Is r incremented? (|r| > |v|)	Is q inexact? (q ≠ v)	Is q incremented? (|q| > |v|)	Returned Results and Status Setting
Tiny	–	–	0	–	–	–	–	–	no	–	–	–	T(N(r))
	–	–	0	–	0	–	–	–	yes	no	–	–	T(N(r)), fx(UX), fx(XX)
	–	–	0	–	0	–	–	–	yes	yes	–	–	T(N(r)), fx(UX), fx(XX)
	–	–	0	–	1	–	–	–	yes	no	–	–	T(N(r)), fx(UX), fx(XX), error()
	–	–	0	–	1	–	–	–	yes	yes	–	–	T(N(r)), fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	yes	–	no	–	fx(UX), error()
	–	–	1	–	–	–	–	–	yes	–	yes	no	fx(UX), fx(XX), error()
	–	–	1	–	–	–	–	–	yes	–	yes	yes	fx(UX), fx(XX), error()

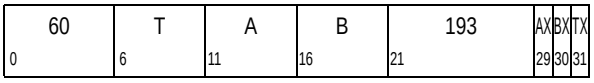
Explanation:

–	The results do not depend on this condition.
fx(x)	FX is set to 1 if x=0. x is set to 1.
q	The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, signficand rounded to the target precision, unbounded exponent range.
r	The value defined in Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515, signficand rounded to the target precision, bounded exponent range.
v	The precise intermediate result defined in the instruction having unbounded signficand precision, unbounded exponent range.
FI	Floating-Point Fraction Inexact status flag, $FPSCR_{FI}$. This status flag is nonsticky.
FR	Floating-Point Fraction Rounded status flag, $FPSCR_{FR}$.
OX	Floating-Point Overflow Exception status flag, $FPSCR_{OX}$.
error()	The system error handler is invoked for the trap-enabled exception if the FE0 and FE1 bits in the Machine State Register are set to any mode other than the ignore-exception mode. Update of the target VSR is suppressed for all vector elements.
N(x)	The value x is is negated by complementing the sign bit of x.
T(x)	The value x is placed in element i of VSR[XT] in the target precision format (where i ∈ {0,1} for results with 64-bit elements, and i ∈ {0,1,3,4}) for results with 32-bit elements).
UX	Floating-Point Underflow Exception status flag, $FPSCR_{UX}$
VXSNAN	Floating-Point Invalid Operation Exception (SNaN) status flag, $FPSCR_{VXSNAN}$.
VXIMZ	Floating-Point Invalid Operation Exception (Infinity × Zero) status flag, $FPSCR_{VXIMZ}$.
VXISI	Floating-Point Invalid Operation Exception (Infinity – Infinity) status flag, $FPSCR_{VXISI}$.
XX	Float-Point Inexact Exception status flag, $FPSCR_{XX}$. The flag is a sticky version of $FPSCR_{FI}$. When $FPSCR_{FI}$ is set to a new value, the new value of $FPSCR_{XX}$ is set to the result of ORing the old value of $FPSCR_{XX}$ with the new value of $FPSCR_{FI}$.

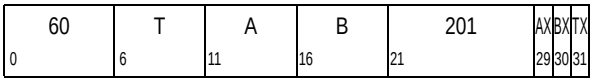
Table 123.Vector Floating-Point Final Result with Negation (Continued)

VSX Vector Negative Multiply-Add Single-Precision XX3-form

xvnmaddasp XT,XA,XB



xvnmaddmsp XT,XA,XB




```
XT       ← TX || T
XA       ← AX || A
XB       ← BX || B
ex_flag  ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1          ← VSR[XA]{i:i+31}
   src2 ← "xvnmaddasp" ? VSR[XT]{i:i+31} : VSR[XB]{i:i+31}
   src3 ← "xvnmaddasp" ? VSR[XB]{i:i+31} : VSR[XT]{i:i+31}
   v{0:inf}      ← MultiplyAddSP(src1,src3,src2)
   result{i:i+31} ← NegateSP(RoundToSP(RN,v))
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag       ← ex_flag | (VE & vxsnan_flag)
   ex_flag       ← ex_flag | (VE & vximz_flag)
   ex_flag       ← ex_flag | (VE & vxisi_flag)
   ex_flag       ← ex_flag | (OE & ox_flag)
   ex_flag       ← ex_flag | (UE & ux_flag)
   ex_flag       ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

For ***xvnmaddasp***, do the following.
- Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
- Let src2 be the single-precision floating-point operand in word element i of VSR[XT].
- Let src3 be the single-precision floating-point operand in word element i of VSR[XB].

For ***xvnmaddmsp***, do the following.
- Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
- Let src2 be the single-precision floating-point operand in word element i of VSR[XB].
- Let src3 be the single-precision floating-point operand in word element i of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 124.

src2 is added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 124.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is negated and placed into word element i of VSR[XT] in single-precision format.

See Table 123, "Vector Floating-Point Final Result with Negation," on page 730.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

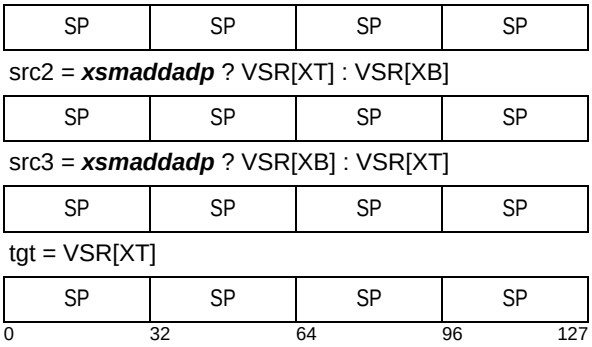
Special Registers Altered

FX OX UX XX VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvnmadd(a|m)sp

src1 = VSR[XA]



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Add	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← –Infinity	v ← src2	v ← –Zero	v ← Rezd	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← –Infinity	v ← src2	v ← Rezd	v ← +Zero	v ← src2	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← –Infinity	v ← A(p,src2)	v ← p	v ← p	v ← A(p,src2)	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← dQNaN vxisi_flag ← 1	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	For ***xvnmaddasp***, the single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}). For ***xvnmaddmsp***, the single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
src3	For ***xvnmaddasp***, the single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}). For ***xvnmaddmsp***, the single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (`0x7FC0_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
A(x,y)	Return the normalized sum of floating-point value x and floating-point value y, having unbounded range and precision.
	Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 124.Actions for xvnmadd(a|m)sp

VSX Vector Negative Multiply-Subtract Double-Precision XX3-form

xvnmsubadp XT,XA,XB

60	T	A	B	241	AX	BX	TX
0	6	11	16	21	29	30	31

xvnmsubmdp XT,XA,XB

60	T	A	B	249	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1 ← VSR[XA]{i:i+63}
   src2 ← "xvmsubadp" ? VSR[XT]{i:i+63} : VSR[XB]{i:i+63}
   src3 ← "xvmsubadp" ? VSR[XB]{i:i+63} : VSR[XT]{i:i+63}
   v{0:inf}       ← MultiplyAddDP(src1,src3,NegateDP(src2))
   result{i:i+63} ← NegateDP(RoundToDP(RN,v))
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

For ***xvmsubadp***, do the following.
- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XT].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XB].

For ***xvmsubmdp***, do the following.
- Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].
- Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].
- Let src3 be the double-precision floating-point operand in doubleword element i of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 125.

src2 is negated and added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 125.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is negated and placed into doubleword element i of VSR[XT] in double-precision format.

See Table 123, "Vector Floating-Point Final Result with Negation," on page 730.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

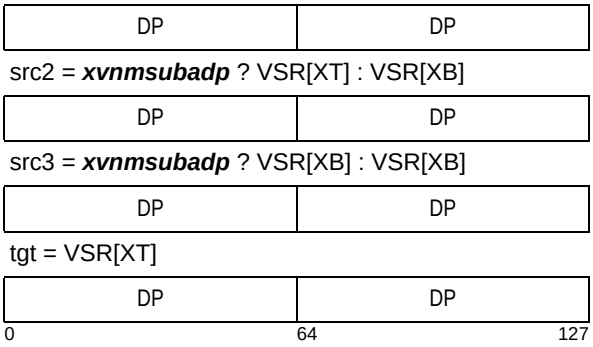
Special Registers Altered

FX OX UX XX VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvnmsub(a|m)dp

src1 = VSR[XA]



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← +Infinity	v ← –src2	v ← –Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← +Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	For ***xvnmsubadp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}). For ***xvnmsubmdp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
src3	For ***xvnmsubadp***, the double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}). For ***xvnmsubmdp***, the double-precision floating-point value in doubleword element i of VSR[XT] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 125.Actions for xvnmsub(a|m)dp

VSX Vector Negative Multiply-Subtract Single-Precision XX3-form

xvnmsubasp XT,XA,XB

60	T	A	B	209	AX	BX	TX
0	6	11	16	21	29	30	31

xvnmsubmsp XT,XA,XB

60	T	A	B	217	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1 ← VSR[XA]{i:i+31}
   src2 ← "xvnmsubasp" ? VSR[XT]{i:i+31} : VSR[XB]{i:i+31}
   src3 ← "xvnmsubasp" ? VSR[XB]{i:i+31} : VSR[XT]{i:i+31}
   v{0:inf}       ← MultiplyAddSP(src1,src3,NegateSP(src2))
   result{i:i+31} ← NegateSP(RoundToSP(RN,v))
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vximz_flag)  then SetFX(VXIMZ)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vximz_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

For ***xvnmsubasp***, do the following.
- Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
- Let src2 be the single-precision floating-point operand in word element i of VSR[XT].
- Let src3 be the single-precision floating-point operand in word element i of VSR[XB].

For ***xvnmsubmsp***, do the following.
- Let src1 be the single-precision floating-point operand in word element i of VSR[XA].
- Let src2 be the single-precision floating-point operand in word element i of VSR[XB].
- Let src3 be the single-precision floating-point operand in word element i of VSR[XT].

src1 is multiplied[1] by src3, producing a product having unbounded range and precision.

See part 1 of Table 126.

src2 is negated and added[2] to the product, producing a sum having unbounded range and precision.

The sum is normalized[3].

See part 2 of Table 126.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is negated and placed into word element i of VSR[XT] in single-precision format.

See Table 123, "Vector Floating-Point Final Result with Negation," on page 730.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

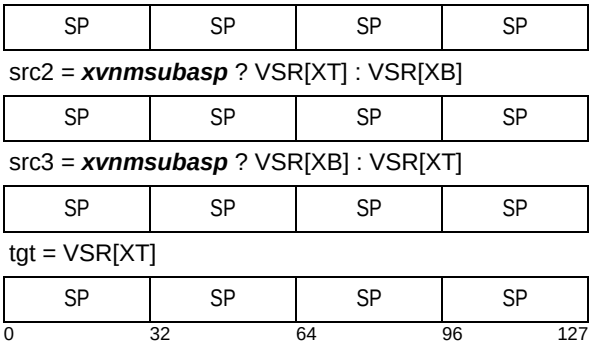
Special Registers Altered

FX OX UX XX VXSNAN VXISI VXIMZ

1. Floating-point multiplication is based on exponent addition and multiplication of the significands.
2. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
3. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

VSR Data Layout for xvnmsub(a|m)sp

src1 = VSR[XA]



Part 1: Multiply	src1 \ src3	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	p ← +Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← –Infinity	p ← –Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–NZF	p ← +Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	–Zero	p ← dQNaN vximz_flag ← 1	p ← +Zero	p ← +Zero	p ← –Zero	p ← –Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Zero	p ← dQNaN vximz_flag ← 1	p ← –Zero	p ← –Zero	p ← +Zero	p ← +Zero	p ← dQNaN vximz_flag ← 1	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+NZF	p ← –Infinity	p ← M(src1,src3)	p ← src1	p ← src1	p ← M(src1,src3)	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	+Infinity	p ← –Infinity	p ← +Infinity	p ← dQNaN vximz_flag ← 1	p ← dQNaN vximz_flag ← 1	p ← +Infinity	p ← +Infinity	p ← src3	p ← Q(src3) vxsnan_flag ← 1
	QNaN	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1	p ← src1 vxsnan_flag ← 1
	SNaN	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1	p ← Q(src1) vxsnan_flag ← 1

Part 2: Subtract	p \ src2	–Infinity	–NZF	–Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
	–Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	–Zero	v ← +Infinity	v ← –src2	v ← –Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← +Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+NZF	v ← +Infinity	v ← S(p,src2)	v ← p	v ← p	v ← S(p,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
	QNaN & src1 is a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p vxsnan_flag ← 1
	QNaN & src1 not a NaN	v ← p	v ← p	v ← p	v ← p	v ← p	v ← p	v ← src2	v ← Q(src2) vxsnan_flag ← 1

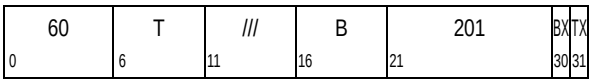
Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	The single-precision floating-point value in word element i of VSR[XT] (where i ∈ {0,1,2,3}).
src3	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (`0x7FC0_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs). Can also occur with two nonzero finite number source operands.
Q(x)	Return a QNaN with the payload of x.
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
M(x,y)	Return the normalized product of floating-point value x and floating-point value y, having unbounded range and precision.
p	The intermediate product having unbounded range and precision.
v	The intermediate result having unbounded range and precision.

Table 126.Actions for xvnmsub(a|m)sp

VSX Vector Round to Double-Precision Integer using round to Nearest Away XX2-form

xvrdpi XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   result{i:i+63} ← RoundToDPIntegerNearAway(VSR[XB]{i:i+63})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to an integer using the rounding mode Round to Nearest Away.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

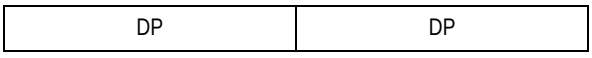
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

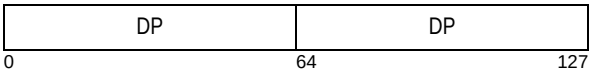
FX VXSNAN

VSR Data Layout for xvrdpi

src = VSR[XB]



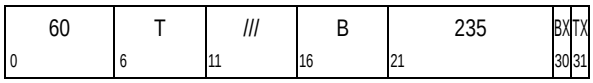
tgt = VSR[XT]



0 64 127

VSX Vector Round to Double-Precision Integer Exact using Current rounding mode XX2-form

xvrdpic XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src{0:63} ← VSR[XB]{i:i+63}
   if(RN=0b00) then
      result{i:i+63} ← RoundToDPIntegerNearEven(src)
   if(RN=0b01) then
      result{i:i+63} ← RoundToDPIntegerTrunc(src)
   if(RN=0b10) then
      result{i:i+63} ← RoundToDPIntegerCeil(src)
   if(RN=0b11) then
      result{i:i+63} ← RoundToDPIntegerFloor(src)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to an integer using the rounding mode specified by RN.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

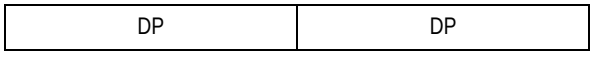
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

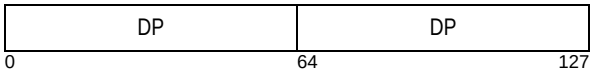
FX XX VXSNAN

VSR Data Layout for xvrdpic

src = VSR[XB]



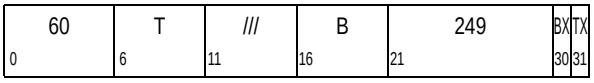
tgt = VSR[XT]



0 64 127

VSX Vector Round to Double-Precision Integer using round toward -Infinity XX2-form

xvrdpim XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   result{i:i+63} ← RoundToDPIntegerFloor(VSR[XB]{i:i+63})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward -Infinity.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

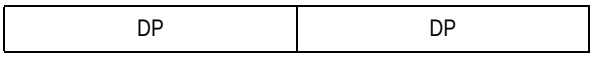
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

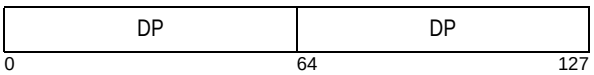
FX VXSNAN

VSR Data Layout for xvrdpim

src = VSR[XB]

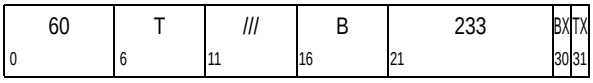


tgt = VSR[XT]



VSX Vector Round to Double-Precision Integer using round toward +Infinity XX2-form

xvrdpip XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   result{i:i+63} ← RoundToDPIntegerCeil(VSR[XB]{i:i+63})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward +Infinity.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

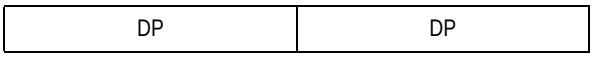
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

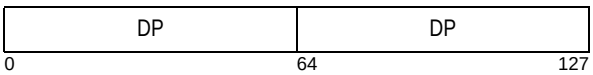
FX VXSNAN

VSR Data Layout for xvrdpip

src = VSR[XB]

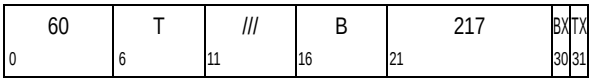


tgt = VSR[XT]



VSX Vector Round to Double-Precision Integer using round toward Zero XX2-form

xvrdpiz XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   result{i:i+63} ← RoundToDPIntegerTrunc(VSR[XB]{i:i+63})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward Zero.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

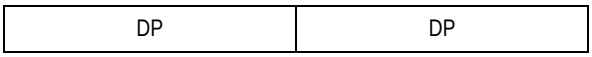
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

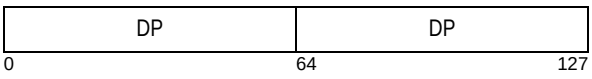
FX VXSNAN

VSR Data Layout for xvrdpiz

src = VSR[XB]



tgt = VSR[XT]



VSX Vector Reciprocal Estimate Double-Precision XX2-form

xvredp XT,XB

60	T	///	B	218	BX	TX
0	6	11	16	21	30	31

```
XT       ← TX || T
XB       ← BX || B
ex_flag  ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   v{0:inf}       ← ReciprocalEstimateDP(VSR[XB]{i:i+63})
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(zx_flag)     then SetFX(ZX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (ZE & zx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

A double-precision floating-point estimate of the reciprocal of src is placed into doubleword element i of VSR[XT] in double-precision format.

Unless the reciprocal of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of src. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\text{src}}}{\frac{1}{\text{src}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
–Infinity	–Zero	None
–Zero	–Infinity[1]	ZX
+Zero	+Infinity[1]	ZX
+Infinity	+Zero	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

1. No result if ZE=1.
2. No result if VE=1.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FX OX UX ZX VXSNAN

VSR Data Layout for xvredp

src = VSR[XB]

DP	DP

tgt = VSR[XT]

DP	DP

0 64 127

VSX Vector Reciprocal Estimate Single-Precision XX2-form

xvresp XT,XB

60	T	///	B	154	BX	TX
0	6	11	16	21	30	31

```
XT      ← TX || T
XB      ← BX || B
ex_flag ← 0b0

do i = 0 to 3
   reset_xflags()
   v             ← ReciprocalEstimateSP(VSR[XB].word[i])
   result.word[i] ← RoundToSP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(zx_flag)     then SetFX(ZX)
   ex_flag       ← ex_flag | (VE & vxsnan_flag)
   ex_flag       ← ex_flag | (OE & ox_flag)
   ex_flag       ← ex_flag | (UE & ux_flag)
   ex_flag       ← ex_flag | (ZE & zx_flag)
end

if(ex_flag=0) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

A single-precision floating-point estimate of the reciprocal of src is placed into word element i of VSR[XT] in single-precision format.

Unless the reciprocal of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of src. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\text{src}}}{\frac{1}{\text{src}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
–Infinity	–Zero	None
–Zero	–Infinity[1]	ZX
+Zero	+Infinity[1]	ZX
+Infinity	+Zero	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

1. No result if ZE=1.
2. No result if VE=1.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FX OX UX ZX VXSNAN

VSR Data Layout for xvresp

src = VSR[XB]

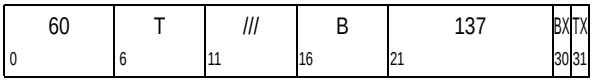
SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

VSX Vector Round to Single-Precision Integer using round to Nearest Away XX2-form

xvrspi XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   result{i:i+31} ← RoundToSPIntegerNearAway(VSR[XB]{i:i+31})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

src is rounded to an integer using the rounding mode Round to Nearest Away.

The result is placed into word element i of VSR[XT] in single-precision format.

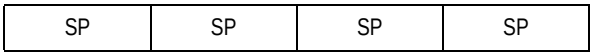
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

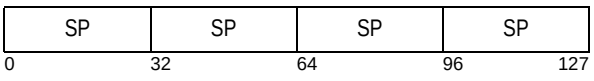
FX VXSNAN

VSR Data Layout for xvrspi

src = VSR[XB]

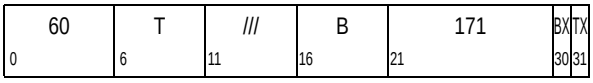


tgt = VSR[XT]



VSX Vector Round to Single-Precision Integer Exact using Current rounding mode XX2-form

xvrspic XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src{0:31} ← VSR[XB]{i:i+31}
   if(RN=0b00) then
      result{i:i+31} ← RoundToSPIntegerNearEven(src)
   if(RN=0b01) then
      result{i:i+31} ← RoundToSPIntegerTrunc(src)
   if(RN=0b10) then
      result{i:i+31} ← RoundToSPIntegerCeil(src)
   if(RN=0b11) then
      result{i:i+31} ← RoundToSPIntegerFloor(src)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

src is rounded to an integer value using the rounding mode specified by RN.

The result is placed into word element i of VSR[XT] in single-precision format.

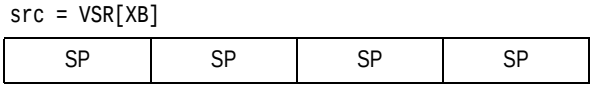
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

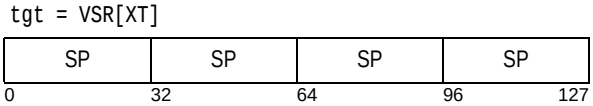
FX XX VXSNAN

VSR Data Layout for xvrspic

src = VSR[XB]

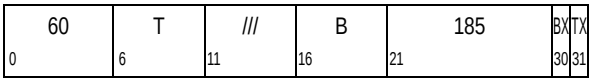


tgt = VSR[XT]



VSX Vector Round to Single-Precision Integer using round toward -Infinity XX2-form

xvrspim XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   result{i:i+31} = RoundToSPIntegerFloor(VSR[XB]{i:i+31})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward -Infinity.

The result is placed into word element i of VSR[XT] in single-precision format.

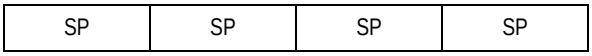
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

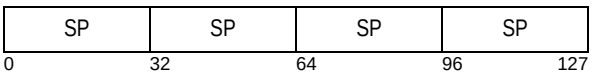
FX VXSNAN

VSR Data Layout for xvrspim

src = VSR[XB]

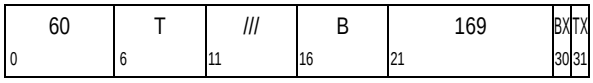


tgt = VSR[XT]



VSX Vector Round to Single-Precision Integer using round toward +Infinity XX2-form

xvrspip XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   result{i:i+31} = RoundToSPIntegerCeil(VSR[XB]{i:i+31})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward +Infinity.

The result is placed into word element i of VSR[XT] in single-precision format.

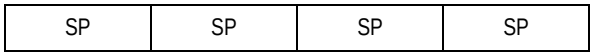
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

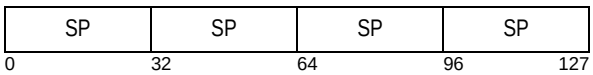
FX VXSNAN

VSR Data Layout for xvrspip

src = VSR[XB]

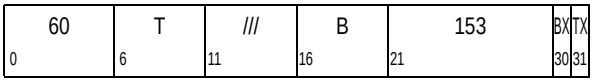


tgt = VSR[XT]



VSX Vector Round to Single-Precision Integer using round toward Zero XX2-form

xvrspiz XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   result{i:i+31} = RoundToSPIntegerTrunc(VSR[XB]{i:i+31})
   if(vxsnan_flag) then SetFX(VXSNAN)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

src is rounded to an integer using the rounding mode Round toward Zero.

The result is placed into word element i of VSR[XT] in single-precision format.

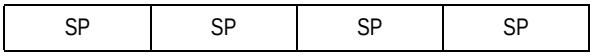
If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

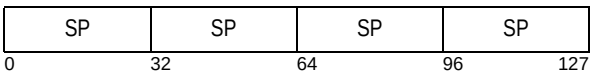
FX VXSNAN

VSR Data Layout for xvrspiz

src = VSR[XB]

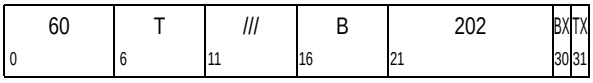


tgt = VSR[XT]



VSX Vector Reciprocal Square Root Estimate Double-Precision XX2-form

xvrsqrtedp XT,XB




```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i←0 to 127 by 64
   reset_xflags()
   v{0:inf}       ← RecipSquareRootEstimateDP(VSR[XB]{i:i+63})
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxsqrt_flag) then SetFX(VXSQRT)
   if(zx_flag)     then SetFX(ZX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxsqrt_flag)
   ex_flag        ← ex_flag | (ZE & zx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

A double-precision floating-point estimate of the reciprocal square root of src is placed into doubleword element i of VSR[XT] in double-precision format.

Unless the reciprocal of the square root of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of src. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\sqrt{src}}}{\frac{1}{\sqrt{src}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
–Infinity	QNaN[1]	VXSQRT
+Infinity	+Zero	None
–Finite	QNaN[1]	VXSQRT
–Zero	–Infinity[2]	ZX
+Zero	+Infinity[2]	ZX
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

1. No result if VE=1.
2. No result if ZE=1.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

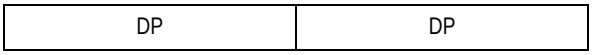
The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

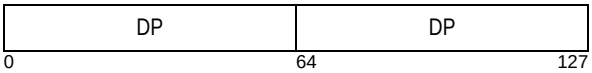
FX ZX VXSNAN VXSQRT

VSR Data Layout for xvrsqrtedp

src = VSR[XB]



tgt = VSR[XT]



VSX Vector Reciprocal Square Root Estimate Single-Precision XX2-form

xvrsqrtesp XT,XB

60	T	///	B	138	BX	TX
0	6	11	16	21	30	31

```
XT        ← TX || T
XB        ← BX || B
ex_flag   ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   v{0:inf}       ← RecipSquareRootEstimateSP(VSR[XB]{i:i+31})
   result{i:i+31} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxsqrt_flag) then SetFX(VXSQRT)
   if(zx_flag)     then SetFX(ZX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxsqrt_flag)
   ex_flag        ← ex_flag | (ZE & zx_flag)
end

if( ex_flag = 0 ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

A single-precision floating-point estimate of the reciprocal square root of src is placed into word element i of VSR[XT] in single-precision format.

Unless the reciprocal of the square root of src would be a zero, an infinity, or a QNaN, the estimate has a relative error in precision no greater than one part in 16384 of the reciprocal of the square root of src. That is,

$$\left| \frac{\text{estimate} - \frac{1}{\sqrt{src}}}{\frac{1}{\sqrt{src}}} \right| \leq \frac{1}{16384}$$

Operation with various special values of the operand is summarized below.

Source Value	Result	Exception
–Infinity	QNaN[1]	VXSQRT
+Infinity	+Zero	None
–Finite	QNaN[1]	VXSQRT
–Zero	–Infinity[2]	ZX
+Zero	+Infinity[2]	ZX
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

1. No result if VE=1.
2. No result if ZE=1.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

The results of executing this instruction is permitted to vary between implementations, and between different executions on the same implementation.

Special Registers Altered

FX ZX VXSNAN VXSQRT

VSR Data Layout for xvrsqrtesp

src = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

VSX Vector Square Root Double-Precision XX2-form

xvsqrtdp XT,XB

60	T	///	B	203	BX	TX
0	6	11	16	21	30	31

```
XT          ← TX || T
XB          ← BX || B
ex_flag     ← 0b0

do i←0 to 127 by 64
   reset_xflags()
   v{0:inf}       ← SquareRootDP(VSR[XB]{i:i+63})
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxsqrt_flag) then SetFX(VXSQRT)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxsqrt_flag)
   ex_flag        ← ex_flag | (XE & xx_flag
end

if( ex_flag ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

The unbounded-precision square root of src is produced.

See Table 127.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, "Scalar Floating-Point Intermediate Result Handling," on page 515.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 98, "Vector Floating-Point Final Result," on page 661.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX XX VXSNAN VXSQRT

VSR Data Layout for xvsqrtdp

src = VSR[XB]

DP	DP

tgt = VSR[XT]

DP	DP

0 64 127

src							
-Infinity	**-NZF**	**-Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
v ← dQNaN vxsqrt_flag ← 1	v ← dQNaN vxsqrt_flag ← 1	v ← +Zero	v ← +Zero	v ← SQRT(src)	v ← +Infinity	v ← src	v ← Q(src) vxsnan_flag ← 1

Explanation:

src	The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (0x7FF8_0000_0000_0000).
NZF	Nonzero finite number.
SQRT(x)	The unbounded-precision square root of the floating-point value x.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 127.Actions for xvsqrtdp

VSX Vector Square Root Single-Precision XX2-form

xvsqrtsp XT,XB

60	T	///	B	139	BX	TX
0	6	11	16	21	30	31

```
XT       ← TX || T
XB       ← BX || B
ex_flag  ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   v{0:inf}       ← SquareRootSP(VSR[XB]{i:i+31})
   result{i:i+31} ← RoundToSP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxsqrt_flag) then SetFX(VXSQRT)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxsqrt_flag)
   ex_flag        ← ex_flag | (XE & xx_flag
end

if( ex_flag ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

The unbounded-precision square root of src is produced.

See Table 128.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 98, “Vector Floating-Point Final Result,” on page 661.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX XX VXSNAN VXSQRT

VSR Data Layout for xvsqrtsp

src = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

src							
-Infinity	**-NZF**	**-Zero**	**+Zero**	**+NZF**	**+Infinity**	**QNaN**	**SNaN**
v ← dQNaN vxsqrt_flag ← 1	v ← dQNaN vxsqrt_flag ← 1	v ← +Zero	v ← +Zero	v ← SQRT(src)	v ← +Infinity	v ← src	v ← Q(src) vxsnan_flag ← 1

Explanation:

src	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (0x7FC0_0000).
NZF	Nonzero finite number.
SQRT(x)	The unbounded-precision square root of the floating-point value x.
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 128.Actions for xvsqrtsp

VSX Vector Subtract Double-Precision XX3-form

xvsubdp XT,XA,XB

60	T	A	B	104	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 64
   reset_xflags()
   src1           ← VSR[XA]{i:i+63}
   src2           ← VSR[XB]{i:i+63}
   v{0:inf}       ← AddDP(src1,NegateDP(src2))
   result{i:i+63} ← RoundToDP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

src2 is negated and added[1] to src1, producing a sum having unbounded range and precision.

The sum is normalized[2].

See Table 129.

The intermediate result is rounded to double-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into doubleword element i of VSR[XT] in double-precision format.

See Table 98, “Vector Floating-Point Final Result,” on page 661.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX XX VXSNAN VXISI

VSR Data Layout for xvsubdp

src1 = VSR[XA]

DP	DP

src2 = VSR[XB]

DP	DP

tgt = VSR[XT]

DP	DP

0 64 127

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← +Infinity	v ← –src2	v ← –Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← +Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The double-precision floating-point value in doubleword element i of VSR[XA] (where i ∈ {0,1}).
src2	The double-precision floating-point value in doubleword element i of VSR[XB] (where i ∈ {0,1}).
dQNaN	Default quiet NaN (`0x7FF8_0000_0000_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 129.Actions for xvsubdp

VSX Vector Subtract Single-Precision XX3-form

xvsubsp XT,XA,XB

60	T	A	B	72	AX	BX	TX
0	6	11	16	21	29	30	31

```
XT          ← TX || T
XA          ← AX || A
XB          ← BX || B
ex_flag     ← 0b0

do i=0 to 127 by 32
   reset_xflags()
   src1           ← VSR[XA]{i:i+31}
   src2           ← VSR[XB]{i:i+31}
   v{0:inf}       ← AddSP(src1,NegateSP(src2))
   result{i:i+31} ← RoundToSP(RN,v)
   if(vxsnan_flag) then SetFX(VXSNAN)
   if(vxisi_flag)  then SetFX(VXISI)
   if(ox_flag)     then SetFX(OX)
   if(ux_flag)     then SetFX(UX)
   if(xx_flag)     then SetFX(XX)
   ex_flag        ← ex_flag | (VE & vxsnan_flag)
   ex_flag        ← ex_flag | (VE & vxisi_flag)
   ex_flag        ← ex_flag | (OE & ox_flag)
   ex_flag        ← ex_flag | (UE & ux_flag)
   ex_flag        ← ex_flag | (XE & xx_flag)
end

if( ex_flag ) then VSR[XT] ← result
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

For each vector element i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

src2 is negated and added[1] to src1, producing a sum having unbounded range and precision.

The sum is normalized[2].

See Table 130.

The intermediate result is rounded to single-precision using the rounding mode specified by RN.

See Table 50, “Scalar Floating-Point Intermediate Result Handling,” on page 515.

The result is placed into word element i of VSR[XT] in single-precision format.

See Table 98, “Vector Floating-Point Final Result,” on page 661.

If a trap-enabled exception occurs in any element of the vector, no results are written to VSR[XT].

Special Registers Altered

FX OX UX XX VXSNAN VXISI

VSR Data Layout for xvsubsp

src1 = VSR[XA]

SP	SP	SP	SP

src2 = VSR[XB]

SP	SP	SP	SP

tgt = VSR[XT]

SP	SP	SP	SP
0	32	64	96	127

1. Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.
2. Floating-point normalization is based on shifting the significand left until the most-significant bit is 1 and decrementing the exponent by the number of bits the significand was shifted.

src1 \ src2	-Infinity	-NZF	-Zero	+Zero	+NZF	+Infinity	QNaN	SNaN
-Infinity	v ← dQNaN vxisi_flag ← 1	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
-Zero	v ← +Infinity	v ← –src2	v ← –Zero	v ← Rezd	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Zero	v ← +Infinity	v ← –src2	v ← Rezd	v ← +Zero	v ← –src2	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+NZF	v ← +Infinity	v ← S(src1,src2)	v ← src1	v ← src1	v ← S(src1,src2)	v ← –Infinity	v ← src2	v ← Q(src2) vxsnan_flag ← 1
+Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← +Infinity	v ← dQNaN vxisi_flag ← 1	v ← src2	v ← Q(src2) vxsnan_flag ← 1
QNaN	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1	v ← src1 vxsnan_flag ← 1
SNaN	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1	v ← Q(src1) vxsnan_flag ← 1

Explanation:

src1	The single-precision floating-point value in word element i of VSR[XA] (where i ∈ {0,1,2,3}).
src2	The single-precision floating-point value in word element i of VSR[XB] (where i ∈ {0,1,2,3}).
dQNaN	Default quiet NaN (`0x7FC0_0000`).
NZF	Nonzero finite number.
Rezd	Exact-zero-difference result (addition of two finite numbers having same magnitude but different signs).
S(x,y)	Return the normalized sum of floating-point value x and negated floating-point value y, having unbounded range and precision. Note: If x = -y, v is considered to be an exact-zero-difference result (Rezd).
Q(x)	Return a QNaN with the payload of x.
v	The intermediate result having unbounded signficand precision and unbounded exponent range.

Table 130.Actions for xvsubsp

VSX Vector Test for software Divide Double-Precision XX3-form

xvtdivdp BF,XA,XB

60	BF	//	A	B	125	AX	BX	/
0	6	9	11	16	21	29	30	31

```
XA      ← AX || A
XB      ← BX || B
eq_flag ← 0b0
gt_flag ← 0b0

do i=0 to 127 by 64
   src1       ← VSR[XA]{i:i+63}
   src2       ← VSR[XB]{i:i+63}
   e_a        ← src1{1:11} - 1023
   e_b        ← src2{1:11} - 1023
   fe_flag    ← fe_flag | IsNaN(src1) | IsInf(src1) |
                IsNaN(src2) | IsInf(src2) | IsZero(src2) |
                ( e_b <= -1022 ) |
                ( e_b >=  1021 ) |
                ( !IsZero(src1) & ( (e_a - e_b) >=  1023 ) ) |
                ( !IsZero(src1) & ( (e_a - e_b) <= -1021 ) ) |
                ( !IsZero(src1) & ( e_a <= -970 ) )
   fg_flag    ← fg_flag | IsInf(src1) | IsInf(src2) |
                IsZero(src2) | IsDen(src2)
end

fl_flag ← xvredp_error() <= 2^-14
CR[BF]  ← 0b1 || fg_flag || fe_flag || 0b0
```

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each vector element i from 0 to 1, do the following.

Let src1 be the double-precision floating-point operand in doubleword element i of VSR[XA].

Let src2 be the double-precision floating-point operand in doubleword element i of VSR[XB].

Let e_a be the unbiased exponent of src1.

Let e_b be the unbiased exponent of src2.

fe_flag is set to 1 for any of the following conditions.

- src1 is a NaN or an infinity.
- src2 is a zero, a NaN, or an infinity.
- e_b is less than or equal to -1022.
- e_b is greater than or equal to 1021.
- src1 is not a zero and the difference, e_a - e_b, is greater than or equal to 1023.
- src1 is not a zero and the difference, e_a - e_b, is less than or equal to -1021.
- src1 is not a zero and e_a is less than or equal to -970

fg_flag is set to 1 for any of the following conditions.

- src1 is an infinity.
- src2 is a zero, an infinity, or a denormalized value.

CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered

CR[BF]

VSR Data Layout for xvtdivdp

src1 = VSR[XA]

.dword[0]	.dword[1]

src2 = VSR[XB]

.dword[0]	.dword[1]

0 64 127

VSX Vector Test for software Divide Single-Precision XX3-form

xvtdivsp BF,XA,XB

60	BF	//	A	B	93	AX	BX	/
0	6	9	11	16	21	29	30	31

```
XA      ← AX || A
XB      ← BX || B
eq_flag ← 0b0
gt_flag ← 0b0

do i=0 to 127 by 32
   src1       ← VSR[XA]{i:i+31}
   src2       ← VSR[XB]{i:i+31}
   e_a        ← src1{1:8} - 127
   e_b        ← src2{1:8} - 127
   fe_flag    ← fe_flag | IsNaN(src1) | IsInf(src1) |
                IsNaN(src2) | IsInf(src2) | IsZero(src2) |
                ( e_b <= -126 ) |
                ( e_b >=  125 ) |
                ( !IsZero(src1) & ( (e_a - e_b) >=  127 ) ) |
                ( !IsZero(src1) & ( (e_a - e_b) <= -125 ) ) |
                ( !IsZero(src1) & ( e_a <= -103 ) )
   fg_flag    ← fg_flag | IsInf(src1) | IsInf(src2) |
                IsZero(src2) | IsDen(src2)
end

fl_flag ← xvredp_error() <= 2^-14
CR[BF]  ← 0b1 || fg_flag || fe_flag || 0b0
```

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each vector element i from 0 to 3, do the following.

Let src1 be the single-precision floating-point operand in word element i of VSR[XA].

Let src2 be the single-precision floating-point operand in word element i of VSR[XB].

Let e_a be the unbiased exponent of src1.

Let e_b be the unbiased exponent of src2.

fe_flag is set to 1 for any of the following conditions.

- src1 is a NaN or an infinity.
- src2 is a zero, a NaN, or an infinity.
- e_b is less than or equal to -126.
- e_b is greater than or equal to 125.
- src1 is not a zero and the difference, e_a - e_b, is greater than or equal to 127.
- src1 is not a zero and the difference, e_a - e_b, is less than or equal to -125.
- src1 is not a zero and e_a is less than or equal to -103.

fg_flag is set to 1 for any of the following conditions.

- src1 is an infinity.
- src2 is a zero, an infinity, or a denormalized value.

CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered

CR[BF]

VSR Data Layout for xvtdivsp

src1 = VSR[XA]

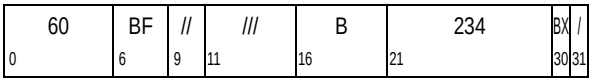
.word[0]	.word[1]	.word[2]	.word[3]

src2 = VSR[XB]

.word[0]	.word[1]	.word[2]	.word[3]
0	32	64	96	127

VSX Vector Test for software Square Root Double-Precision XX2-form

xvtsqrtdp BF,XB




```
XB      ← BX || B
fe_flag ← 0b0
fg_flag ← 0b0

do i=0 to 127 by 64
   src      ← VSR[XB]{i:i+63}
   e_b      ← src2{1:11} - 1023
   fe_flag  ← fe_flag | IsNaN(src) | IsInf(src)  |
              IsZero(src) | IsNeg(src) | ( e_a <= -970 )
   fg_flag  ← fg_flag | IsInf(src) | IsZero(src) |
              IsDen(src)
end

fl_flag ← xvrsqrtedp_error() <= 2^-14
CR[BF]  ← 0b1 || fg_flag || fe_flag || 0b0
```

Let XB be the value 32×BX + B.

fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each vector element i from 0 to 1, do the following.

Let src be the double-precision floating-point operand in doubleword element i of VSR[XB].

Let e_b be the unbiased exponent of src.

fe_flag is set to 1 for any of the following conditions.

- src is a zero, a NaN, an infinity, or a negative value.
- e_b is less than or equal to -970.

fg_flag is set to 1 for the following condition.

- src is a zero, an infinity, or a denormalized value.

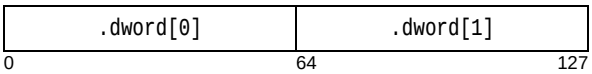
CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered

CR[BF]

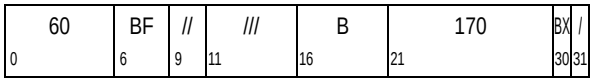
VSR Data Layout for xvtsqrtdp

src = VSR[XB]



VSX Vector Test for software Square Root Single-Precision XX2-form

xvtsqrtsp BF,XB




```
XB      ← BX || B
fe_flag ← 0b0
fg_flag ← 0b0

do i=0 to 127 by 32
   src      ← VSR[XB]{i:i+31}
   e_b      ← src2{1:8} - 127
   fe_flag  ← fe_flag | IsNaN(src) | IsInf(src) |
              IsZero(src) | IsNeg(src) | ( e_a <= -103 )
   fg_flag  ← fg_flag | IsInf(src) | IsZero(src) |
              IsDen(src)
end

fl_flag = xvrsqrtesp_error() <= 2^-14
CR[BF]  = 0b1 || fg_flag || fe_flag || 0b0
```

Let XB be the value 32×BX + B.

fe_flag is initialized to 0.
fg_flag is initialized to 0.

For each vector element i from 0 to 3, do the following.

Let src be the single-precision floating-point operand in word element i of VSR[XB].

Let e_b be the unbiased exponent of src.

fe_flag is set to 1 for any of the following conditions.

- src is a zero, a NaN, an infinity, or a negative value.
- e_b is less than or equal to -103.

fg_flag is set to 1 for the following condition.

- src is a zero, an infinity, or a denormalized value.

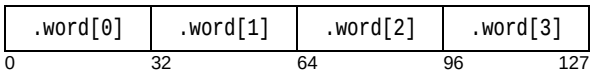
CR field BF is set to the value 0b1 || fg_flag || fe_flag || 0b0.

Special Registers Altered

CR[BF]

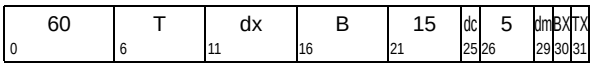
VSR Data Layout for xvtsqrtsp

src = VSR[XB]



VSX Vector Test Data Class Double-Precision XX2-form

xvtstdcdp XT,XB,DCMX




```
if MSR.VSX=0 then VSX_Unavailable()

DCMX ← dc || dm || dx

do i = 0 to 1
   src            ← VSR[32×BX+B].dword[i]
   sign           ← src.bit[0]
   exponent       ← src.bit[1:11]
   fraction       ← src.bit[12:63]

   class.Infinity ← (exponent = 0x7FF) & (fraction  = 0)
   class.NaN      ← (exponent = 0x7FF) & (fraction != 0)
   class.Zero     ← (exponent = 0x000) & (fraction  = 0)
   class.Denormal ← (exponent = 0x000) & (fraction != 0)

   match          ← (DCMX.bit[0] & class.NaN)             |
                    (DCMX.bit[1] & class.Infinity & !sign) |
                    (DCMX.bit[2] & class.Infinity &  sign) |
                    (DCMX.bit[3] & class.Zero     & !sign) |
                    (DCMX.bit[4] & class.Zero     &  sign) |
                    (DCMX.bit[5] & class.Denormal & !sign) |
                    (DCMX.bit[6] & class.Denormal &  sign)

   if match = 1 then
      VSR[32×TX+T].dword[i] ← 0xFFFF_FFFF_FFFF_FFFF
   else
      VSR[32×TX+T].dword[i] ← 0x0000_0000_0000_0000
end
```

Let XB be the sum 32×BX + B.
Let XT be the sum 32×TX + T.
Let DCMX be the value dc concatenated with dm concatenated with dx.

For each integer value i from 0 to 1, do the following.

Let src be the double-precision floating-point value in doubleword element i of VSR[XB].

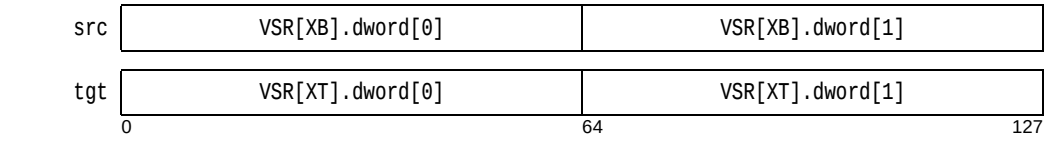
If src matches one of the 7 possible data classes specified by DCMX (Data Class Mask), the contents of doubleword element i of VSR[XT] are set to 0xFFFF_FFFF_FFFF_FFFF. Otherwise, the contents of doubleword element i of VSR[XT] are set to 0x0000_0000_0000_0000.

DCMX bit	Data Class
0	NaN
1	+Infinity
2	-Infinity
3	+Zero
4	-Zero
5	+Denormal
6	-Denormal

Special Registers Altered:

None

VSR Data Layout for xvtstdcdp



VSX Vector Test Data Class Single-Precision XX2-form

xvtstdcsp XT,XB,DCMX

60	T	dx	B	13	dc	5	dm	BX	TX
0	6	11	16	21	25	26	29	30	31

```
if MSR.VSX=0 then VSX_Unavailable()

DCMX ← dc || dm || dx

do i = 0 to 3
   src            ← VSR[32×BX+B].word[i]
   sign           ← src.bit[0]
   exponent       ← src.bit[1:8]
   fraction       ← src.bit[9:31]

   class.Infinity ← (exponent = 0xFF) & (fraction  = 0)
   class.NaN      ← (exponent = 0xFF) & (fraction != 0)
   class.Zero     ← (exponent = 0x00) & (fraction  = 0)
   class.Denormal ← (exponent = 0x00) & (fraction != 0)

   match          ← (DCMX.bit[0] & class.NaN)             |
                    (DCMX.bit[1] & class.Infinity & !sign) |
                    (DCMX.bit[2] & class.Infinity &  sign) |
                    (DCMX.bit[3] & class.Zero     & !sign) |
                    (DCMX.bit[4] & class.Zero     &  sign) |
                    (DCMX.bit[5] & class.Denormal & !sign) |
                    (DCMX.bit[6] & class.Denormal &  sign)

   if match = 1 then
      VSR[32×TX+T].dword[i] ← 0xFFFF_FFFF
   else
      VSR[32×TX+T].dword[i] ← 0x0000_0000
end
```

Let XB be the sum 32×BX + B.
Let XT be the sum 32×TX + T.
Let DCMX be the value dc concatenated with dm concatenated with dx.

For each integer value i from 0 to 3, do the following.

Let src be the single-precision floating-point value in word element i of VSR[XB].

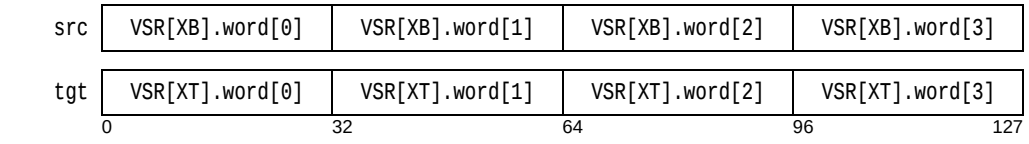
If src matches one of the 7 possible data classes specified by DCMX (Data Class Mask), the contents of word element i of VSR[XT] are set to 0xFFFF_FFFF. Otherwise, the contents of word element i of VSR[XT] are set to 0x0000_0000.

DCMX bit	Data Class
0	NaN
1	+Infinity
2	-Infinity
3	+Zero
4	-Zero
5	+Denormal
6	-Denormal

Special Registers Altered:

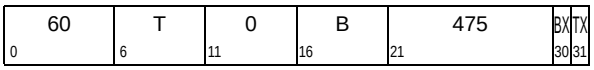
None

VSR Data Layout for xvtstdcsp



VSX Vector Extract Exponent Double-Precision XX2-form

xvxexpdp XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 1
   src ← VSR[32×BX+B].dword[i]
   VSR[32×TX+T].dword[i] ← EXTZ64(src.bit[1:11])
end
```

Let XT be the sum 32×TX + T.
Let XB be the sum 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src be the double-precision floating-point value in doubleword element i of VSR[XB].

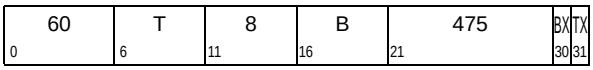
The value of the exponent field in src is placed into doubleword element i of VSR[XT] in unsigned integer format.

Special Registers Altered:

None

VSX Vector Extract Exponent Single-Precision XX2-form

xvxexpsp XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 3
   src ← VSR[32×BX+B].word[i]
   VSR[32×TX+T].word[i] ← EXTZ32(src.bit[1:8])
end
```

Let XT be the sum 32×TX + T.
Let XB be the sum 32×BX + B.

For each integer value i from 0 to 3, do the following.

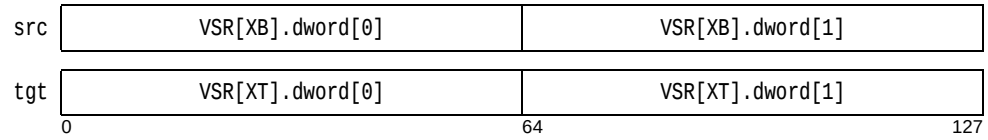
Let src be the single-precision floating-point value in word element i of VSR[XB].

The value of the exponent field in src is placed into word element i of VSR[XT] in unsigned integer format.

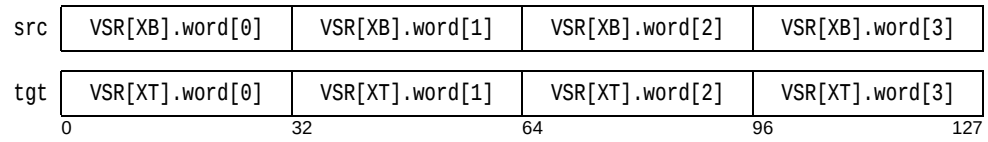
Special Registers Altered:

None

VSR Data Layout for xvxexpdp

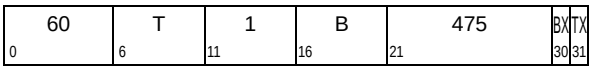


VSR Data Layout for xvxexpsp



VSX Vector Extract Significand Double-Precision XX2-form

xvxsigdp XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 1
   src ← VSR[32×BX+B].dword[i]
   exponent ← EXTZ(src.bit[1:11])
   fraction ← EXTZ64(src.bit[12:63])

   if (exponent != 0) &  (exponent != 2047) then
      fraction ← fraction | 0x0010_0000_0000_0000

   VSR[32×TX+T].dword[i] ← fraction
end
```

Let XT be the sum 32×TX + T.
Let XB be the sum 32×BX + B.

For each integer value i from 0 to 1, do the following.

Let src be the double-precision floating-point value in doubleword element i of VSR[XB].

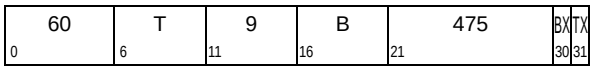
The significand of src is placed into doubleword element i of VSR[XT] in unsigned integer format. If src is a normal value, the implicit leading bit is set to 1.

Special Registers Altered:

None

VSX Vector Extract Significand Single-Precision XX2-form

xvxsigsp XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 3
   src ← VSR[32×BX+B].word[i]
   exponent ← EXTZ(src.bit[1:8])
   fraction ← EXTZ32(src.bit[9:31])

   if (exponent != 0) &  (exponent != 255) then
      fraction ← fraction | 0x0080_0000

   VSR[32×TX+T].word[i] ← fraction
end
```

Let XT be the sum 32×TX + T.
Let XB be the sum 32×BX + B.

For each integer value i from 0 to 3, do the following.

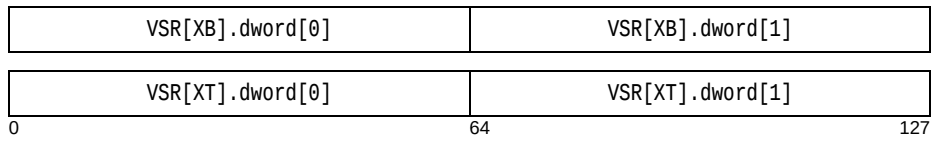
Let src be the single-precision floating-point value in word element i of VSR[XB].

The significand of src is placed into word element i of VSR[XT] in unsigned integer format. If src is a normal value, the implicit leading bit is set to 1.

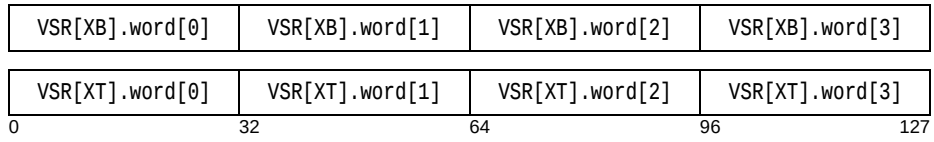
Special Registers Altered:

None

VSR Data Layout for xvxsigdp

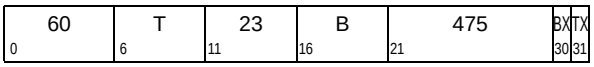


VSR Data Layout for xvxsigsp



VSX Vector Byte-Reverse Doubleword XX2-form

xxbrd XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 1
   do j = 0 to 7
      VSR[32×TX+T].dword[i].byte[j] ← VSR[32×BX+B].dword[i].byte[7-j]
   end
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 1, do the following.

The contents of byte 7 of doubleword element i of VSR[XB] are placed into byte 0 of doubleword element i of VSR[XT].

The contents of byte 6 of doubleword element i of VSR[XB] are placed into byte 1 of doubleword element i of VSR[XT].

The contents of byte 5 of doubleword element i of VSR[XB] are placed into byte 2 of doubleword element i of VSR[XT].

The contents of byte 4 of doubleword element i of VSR[XB] are placed into byte 3 of doubleword element i of VSR[XT].

The contents of byte 3 of doubleword element i of VSR[XB] are placed into byte 4 of doubleword element i of VSR[XT].

The contents of byte 2 of doubleword element i of VSR[XB] are placed into byte 5 of doubleword element i of VSR[XT].

The contents of byte 1 of doubleword element i of VSR[XB] are placed into byte 6 of doubleword element i of VSR[XT].

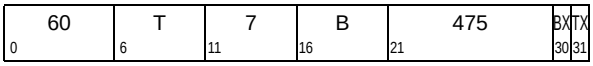
The contents of byte 0 of doubleword element i of VSR[XB] are placed into byte 7 of doubleword element i of VSR[XT].

Special Registers Altered:

None

VSX Vector Byte-Reverse Halfword XX2-form

xxbrh XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 7
   VSR[32×TX+T].hword[i].byte[0] ← VSR[32×BX+B].hword[i].byte[1]
   VSR[32×TX+T].hword[i].byte[1] ← VSR[32×BX+B].hword[i].byte[0]
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 7, do the following.

The contents of byte 1 of halfword element i of VSR[XB] are placed into byte 0 of halfword element i of VSR[XT].

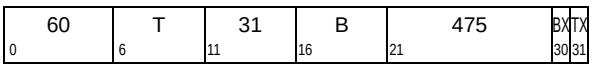
The contents of byte 0 of halfword element i of VSR[XB] are placed into byte 1 of halfword element i of VSR[XT].

Special Registers Altered:

None

VSX Vector Byte-Reverse Quadword XX2-form

xxbrq XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 15
   VSR[32×TX+T].byte[i] ← VSR[32×BX+B].byte[15-i]
end
```

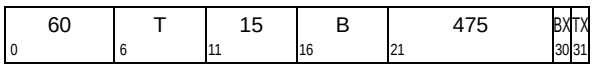
Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 15, do the following.
The contents of byte sub-element 15-i of VSR[XB] are placed into byte sub-element i of VSR[XT].

Special Registers Altered:
None

VSX Vector Byte-Reverse Word XX2-form

xxbrw XT,XB




```
if MSR.VSX=0 then VSX_Unavailable()

do i = 0 to 3
   do j = 0 to 3
      VSR[32×TX+T].word[i].byte[j] ← VSR[32×BX+B].word[i].byte[3-j]
   end
end
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

For each integer value i from 0 to 3, do the following.
The contents of byte 3 of word element i of VSR[XB] are placed into byte 0 of word element i of VSR[XT].

The contents of byte 2 of word element i of VSR[XB] are placed into byte 1 of word element i of VSR[XT].

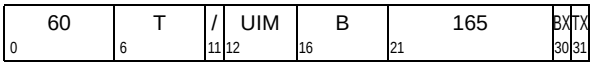
The contents of byte 1 of word element i of VSR[XB] are placed into byte 2 of word element i of VSR[XT].

The contents of byte 0 of word element i of VSR[XB] are placed into byte 3 of word element i of VSR[XT].

Special Registers Altered:
None

VSX Vector Extract Unsigned Word XX2-form

xxextractuw XT,XB,UIM




```
if MSR.VSX=0 then VSX_Unavailable()

src ← VSR[32×BX+B].byte[UIM:UIM+3]

VSR[32×TX+T].dword[0] ← Chop(EXTZ(src),64)
VSR[32×TX+T].dword[1] ← 0x0000_0000_0000_0000
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

The contents of byte elements UIM:UIM+3 of VSR[XB] are placed into word element 1 of VSR[XT]. The contents of the remaining word elements of VSR[XT] are set to 0.

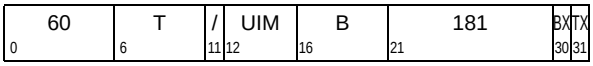
If the value of UIM is greater than 12, the results are undefined.

Special Registers Altered:

None

VSX Vector Insert Word XX2-form

xxinsertw XT,XB,UIM




```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T].byte[UIM:UIM+3] ← VSR[32×BX+B].bit[32:63]
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

The contents of word element 1 of VSR[XB] are placed into byte elements UIM:UIM+3 of VSR[XT]. The contents of the remaining byte elements of VSR[XT] are not modified.

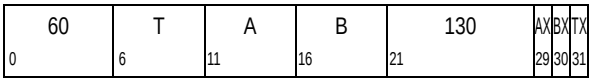
If the value of UIM is greater than 12, the results are undefined.

Special Registers Altered:

None

VSX Logical AND XX3-form

xxland XT,XA,XB



VSR[32×TX+T] ← VSR[32×AX+A] & VSR[32×BX+B]

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

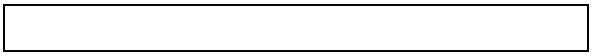
The contents of VSR[XA] are ANDed with the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered

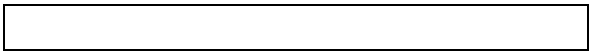
None

VSR Data Layout for xxland

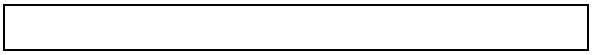
src1 = VSR[XA]



src2 = VSR[XB]



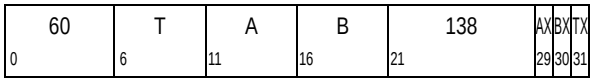
tgt = VSR[XT]



0 127

VSX Logical AND with Complement XX3-form

xxlandc XT,XA,XB



VSR[32×TX+T] ← VSR[32×AX+A] & ~VSR[32×BX+B]

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

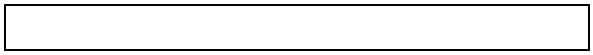
The contents of VSR[XA] are ANDed with the complement of the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered

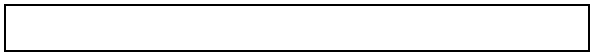
None

VSR Data Layout for xxland

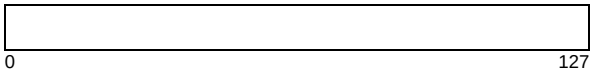
src1 = VSR[XA]



src2 = VSR[XB]



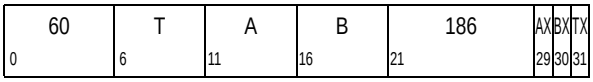
tgt = VSR[XT]



0 127

VSX Logical Equivalence XX3-form

xxleqv XT,XA,XB



VSR[32×TX+T] ← VSR[32×AX+A] ≡ VSR[32×BX+B]

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

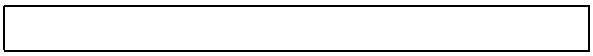
The contents of VSR[XA] are exclusive-ORed with the contents of VSR[XB] and the complemented result is placed into VSR[XT].

Special Registers Altered:

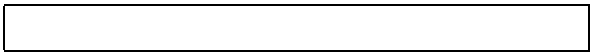
None

VSR Data Layout for xxleqv

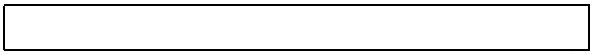
src = VSR[XA]



src = VSR[XB]



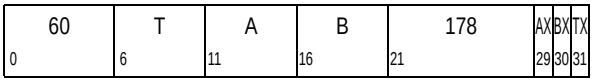
tgt = VSR[XT]



0 127

VSX Logical NAND XX3-form

xxlnand XT,XA,XB



VSR[32×TX+T] ← ¬(VSR[32×AX+A] & VSR[32×BX+B])

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

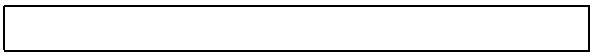
The contents of VSR[XA] are ANDed with the contents of VSR[XB] and the complemented result is placed into VSR[XT].

Special Registers Altered:

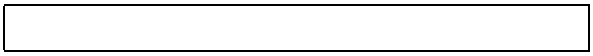
None

VSR Data Layout for xxlnand

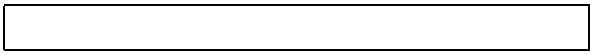
src = VSR[XA]



src = VSR[XB]



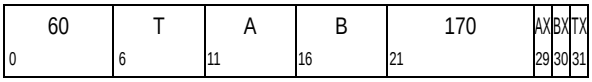
tgt = VSR[XT]



0 127

VSX Logical OR with Complement XX3-form

xxlorc XT,XA,XB



VSR[32×TX+T] ← VSR[32×AX+A] | ¬VSR[32×BX+B]

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

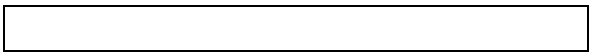
The contents of VSR[XA] are ORed with the complement of the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered:

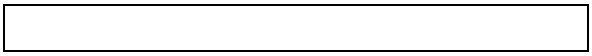
None

VSR Data Layout for xxlorc

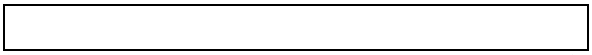
src1 = VSR[XA]



src2 = VSR[XB]



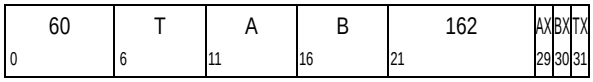
tgt = VSR[XT]



0 127

VSX Logical NOR XX3-form

xxlnor XT,XA,XB



VSR[32×TX+T] ← ~(VSR[32×AX+A] | VSR[32×BX+B])

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

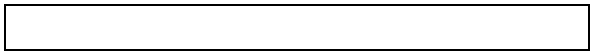
The contents of VSR[XA] are ORed with the contents of VSR[XB] and the complemented result is placed into VSR[XT].

Special Registers Altered

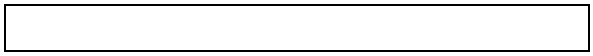
None

VSR Data Layout for xxlnor

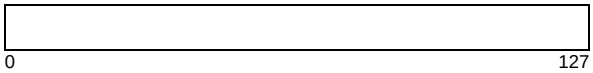
src1 = VSR[XA]



src2 = VSR[XB]



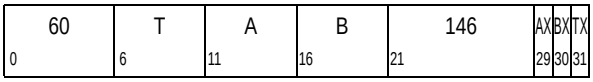
tgt = VSR[XT]



0 127

VSX Logical OR XX3-form

xxlor XT,XA,XB



VSR[32×TX+T] ← VSR[32×AX+A] | VSR[32×BX+B]

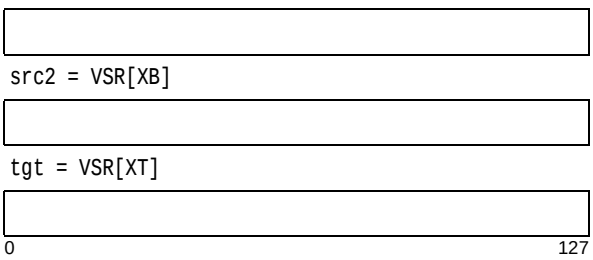
Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

The contents of VSR[XA] are ORed with the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered

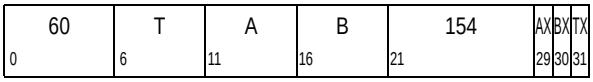
None

VSR Data Layout for xxlor



VSX Logical XOR XX3-form

xxlxor XT,XA,XB



VSR[32×TX+T] ← VSR[32×AX+A] ^ VSR[32×BX+B]

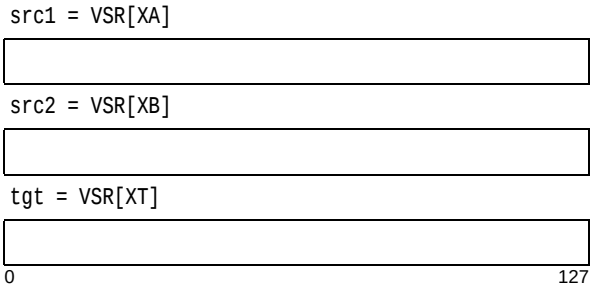
Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

The contents of VSR[XA] are exclusive-ORed with the contents of VSR[XB] and the result is placed into VSR[XT].

Special Registers Altered

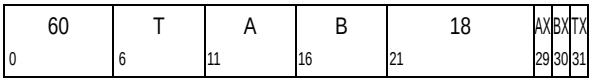
None

VSR Data Layout for xxlxor



VSX Merge High Word XX3-form

xxmrghw XT,XA,XB




```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T].word[0] ← VSR[32×AX+A].word[0]
VSR[32×TX+T].word[1] ← VSR[32×BX+B].word[0]
VSR[32×TX+T].word[2] ← VSR[32×AX+A].word[1]
VSR[32×TX+T].word[3] ← VSR[32×BX+B].word[1]
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

The contents of word element 0 of VSR[XA] are placed into word element 0 of VSR[XT].

The contents of word element 0 of VSR[XB] are placed into word element 1 of VSR[XT].

The contents of word element 1 of VSR[XA] are placed into word element 2 of VSR[XT].

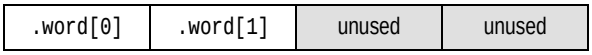
The contents of word element 1 of VSR[XB] are placed into word element 3 of VSR[XT].

Special Registers Altered

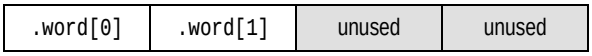
None

VSR Data Layout for xxmrghw

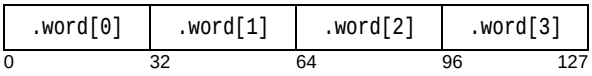
src1 = VSR[XA]



src2 = VSR[XB]

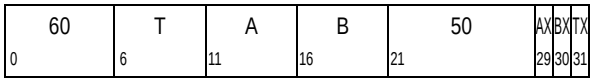


tgt = VSR[XT]



VSX Merge Low Word XX3-form

xxmrglw XT,XA,XB




```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T].word[0] ← VSR[32×AX+A].word[2]
VSR[32×TX+T].word[1] ← VSR[32×BX+B].word[2]
VSR[32×TX+T].word[2] ← VSR[32×AX+A].word[3]
VSR[32×TX+T].word[3] ← VSR[32×BX+B].word[3]
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

The contents of word element 2 of VSR[XA] are placed into word element 0 of VSR[XT].

The contents of word element 2 of VSR[XB] are placed into word element 1 of VSR[XT].

The contents of word element 3 of VSR[XA] are placed into word element 2 of VSR[XT].

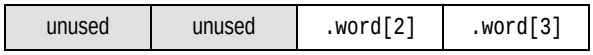
The contents of word element 3 of VSR[XB] are placed into word element 3 of VSR[XT].

Special Registers Altered

None

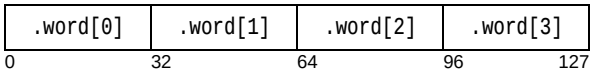
VSR Data Layout for xxmrglw

src1 = VSR[XA]



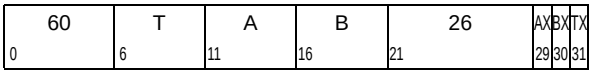
src2 = VSR[XB]

tgt = VSR[XT]



VSX Vector Permute XX3-form

xxperm XT,XA.XB




```
if MSR.VSX=0 then VSX_Unavailable()

src.byte[0:15]  ← VSR[32×AX+A]
src.byte[16:31] ← VSR[32×TX+T]
pcv.byte[0:15]  ← VSR[32×BX+B]

do i = 0 to 15
   idx ← pcv.byte[i].bit[3:7]
   VSR[32×TX+T].byte[i] ← src.byte[idx]
end
```

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XT be the value 32×TX + T.

Let bytes 0:15 of src be the contents of VSR[XA].
Let bytes 16:31 of src be the contents of VSR[XT].

Let the permute control vector pcv be the contents of VSR[XB].

For each integer value i from 0 to 15, do the following.

Let idx be the unsigned integer in bits 3:7 of byte element i of pcv.

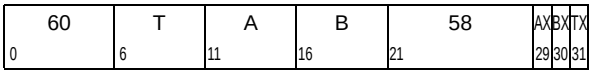
The contents of byte element idx of src is placed into byte element i of VSR[XT].

Special Registers Altered:

None

VSX Vector Permute Right-indexed XX3-form

xxpermr XT,XA.XB




```
if MSR.VSX=0 then VSX_Unavailable()

src.byte[0:15]  ← VSR[32×AX+A]
src.byte[16:31] ← VSR[32×TX+T]
pcv.byte[0:15]  ← VSR[32×BX+B]

do i = 0 to 15
   idx ← pcv.byte[i].bit[3:7]
   VSR[32×TX+T].byte[i] ← src.byte[31-idx]
end
```

Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XT be the value 32×TX + T.

Let bytes 0:15 of src be the contents of VSR[XA].
Let bytes 16:31 of src be the contents of VSR[XT].

Let the permute control vector pcv be the contents of VSR[XB].

For each integer value i from 0 to 15, do the following.

Let idx be the unsigned integer in bits 3:7 of byte element i of pcv.

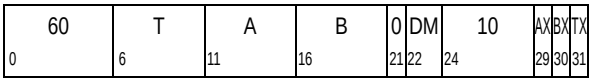
The contents of byte element 31-idx of src is placed into byte element i of VSR[XT].

Special Registers Altered:

None

VSX Permute Doubleword Immediate XX3-form

xxpermdi XT,XA,XB,DM




```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T].dword[0] ← VSR[32×AX+A].dword[DM.bit[0]]
VSR[32×TX+T].dword[1] ← VSR[32×BX+B].dword[DM.bit[1]]
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

If DM.bit[0]=0, the contents of doubleword element 0 of VSR[XA] are placed into doubleword element 0 of VSR[XT]. Otherwise the contents of doubleword element 1 of VSR[XA] are placed into doubleword element 0 of VSR[XT].

If DM.bit[1]=0, the contents of doubleword element 0 of VSR[XB] are placed into doubleword element 1 of VSR[XT]. Otherwise the contents of doubleword element 1 of VSR[XB] are placed into doubleword element 1 of VSR[XT].

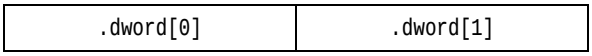
Special Registers Altered

None

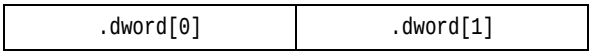
Extended Mnemonic		Equivalent To	
xxspltd	T,A,0	xxpermdi	T,A,A,0b00
xxspltd	T,A,1	xxpermdi	T,A,A,0b11
xxmrghd	T,A,B	xxpermdi	T,A,B,0b00
xxmrgld	T,A,B	xxpermdi	T,A,B,0b11
xxswapd	T,A	xxpermdi	T,A,A,0b10

VSR Data Layout for xxpermdi

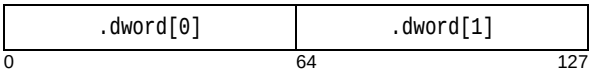
src1 = VSR[XA]



src2 = VSR[XB]



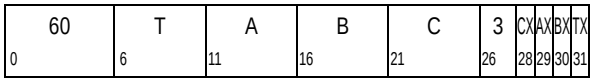
tgt = VSR[XT]



0 64 127

VSX Select XX4-form

xxsel XT,XA,XB,XC




```
if MSR.VSX=0 then VSX_Unavailable()

do i=0 to 127
   if (VSR[32×CX+C].bit[i]=0) then
      VSR[32×TX+T].bit[i] ← VSR[32×AX+A].bit[i]
   else
      VSR[32×TX+T].bit[i] ← VSR[32×BX+B].bit[i]
end
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.
Let XC be the value 32×CX + C.

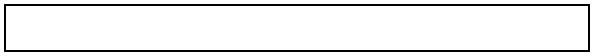
For each bit of VSR[XC] that contains the value 0, the corresponding bit of VSR[XA] is placed into the corresponding bit of VSR[XT]. Otherwise, the corresponding bit of VSR[XB] is placed into the corresponding bit of VSR[XT].

Special Registers Altered

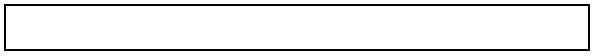
None

VSR Data Layout for xxsel

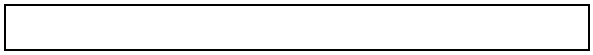
src1 = VSR[XA]



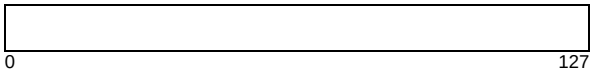
src2 = VSR[XB]



src3 = VSR[XC]



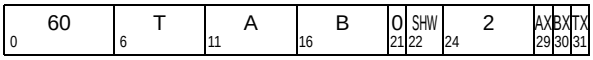
tgt = VSR[XT]



0 127

VSX Shift Left Double by Word Immediate XX3-form

xxsldwi XT,XA,XB,SHW




```
if MSR.VSX=0 then VSX_Unavailable()

source.qword[0] ← VSR[32×AX+A]
source.qword[1] ← VSR[32×BX+B]
VSR[32×TX+T] ← source.word[SHW:SHW+3]
```

Let XT be the value 32×TX + T.
Let XA be the value 32×AX + A.
Let XB be the value 32×BX + B.

Let the source vector be the concatenation of the contents of VSR[XA] followed by the contents of VSR[XB]. Words SHW:SHW+3 of the source vector are placed into VSR[XT].

Special Registers Altered

None

VSR Data Layout for xxsldwi

src1 = VSR[XA]

.word[0]	.word[1]	.word[2]	.word[3]

src2 = VSR[XB]

.word[0]	.word[1]	.word[2]	.word[3]

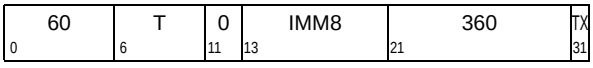
tgt = VSR[XT]

.word[0]	.word[1]	.word[2]	.word[3]

0 32 64 96 127

VSX Vector Splat Immediate Byte X-form

xxspltib XT,IMM8




```
if TX=0 & MSR.VSX=0 then VSX_Unavailable()
if TX=1 & MSR.VEC=0 then Vector_Unavailable()

do i = 0 to 15
   VSR[32×TX+T].byte[i] ← UIM8
end
```

Let XT be the sum 32×TX + T.

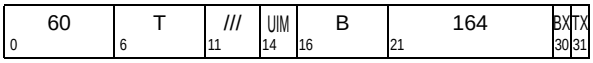
The value IMM8 is copied into each byte element of VSR[XT].

Special Registers Altered:

None

VSX Splat Word XX2-form

xxspltw XT,XB,UIM




```
if MSR.VSX=0 then VSX_Unavailable()

VSR[32×TX+T].word[0] ← VSR[32×BX+B].word[UIM]
VSR[32×TX+T].word[1] ← VSR[32×BX+B].word[UIM]
VSR[32×TX+T].word[2] ← VSR[32×BX+B].word[UIM]
VSR[32×TX+T].word[3] ← VSR[32×BX+B].word[UIM]
```

Let XT be the value 32×TX + T.
Let XB be the value 32×BX + B.

The contents of word element UIM of VSR[XB] are replicated in each word element of VSR[XT].

Special Registers Altered

None

VSR Data Layout for xxspltw

src = VSR[XB]

.word[0]	.word[1]	.word[2]	.word[3]

tgt = VSR[XT]

.word[0]	.word[1]	.word[2]	.word[3]

0 32 64 96 127

Appendix A. Suggested Floating-Point Models

A.1 Floating-Point Round to Single-Precision Model

The following describes algorithmically the operation of the *Floating Round to Single-Precision* instruction.

If $(FRB)_{1:11}$ < 897 and $(FRB)_{1:63}$ > 0 then
    Do
        If $FPSCR_{UE}$ = 0 then goto Disabled Exponent Underflow
        If $FPSCR_{UE}$ = 1 then goto Enabled Exponent Underflow
    End

If $(FRB)_{1:11}$ > 1150 and $(FRB)_{1:11}$ < 2047 then
    Do
        If $FPSCR_{OE}$ = 0 then goto Disabled Exponent Overflow
        If $FPSCR_{OE}$ = 1 then goto Enabled Exponent Overflow
    End

If $(FRB)_{1:11}$ > 896 and $(FRB)_{1:11}$ < 1151 then goto Normal Operand

If $(FRB)_{1:63}$ = 0 then goto Zero Operand

If $(FRB)_{1:11}$ = 2047 then
    Do
        If $(FRB)_{12:63}$ = 0 then goto Infinity Operand
        If $(FRB)_{12}$ = 1 then goto QNaN Operand
        If $(FRB)_{12}$ = 0 and $(FRB)_{13:63}$ > 0 then goto SNaN Operand
    End

Disabled Exponent Underflow:
    sign ← $(FRB)_0$
    If $(FRB)_{1:11}$ = 0 then
        Do
            exp ← -1022
            $frac_{0:52}$ ← 0b0 || $(FRB)_{12:63}$
        End
    If $(FRB)_{1:11}$ > 0 then
        Do
            exp ← $(FRB)_{1:11}$ - 1023
            $frac_{0:52}$ ← 0b1 || $(FRB)_{12:63}$
        End
    Denormalize operand:
        G || R || X ← 0b000
        Do while exp < -126
            exp ← exp + 1
            $frac_{0:52}$ || G || R || X ← 0b0 || $frac_{0:52}$ || G || (R | X)
        End
    $FPSCR_{UX}$ ← ($frac_{24:52}$ || G || R || X) > 0
    Round Single(sign,exp,$frac_{0:52}$,G,R,X)
    $FPSCR_{XX}$ ← $FPSCR_{XX}$ | $FPSCR_{FI}$
    If $frac_{0:52}$ = 0 then
        Do
            FRT_0 ← sign
            $FRT_{1:63}$ ← 0

```
            If sign = 0 then FPSCR_FPRF ← "+ zero"
            If sign = 1 then FPSCR_FPRF ← "- zero"
        End
    If frac_0:52 > 0 then
        Do
            If frac_0 = 1 then
                Do
                    If sign = 0 then FPSCR_FPRF ← "+ normal number"
                    If sign = 1 then FPSCR_FPRF ← "- normal number"
                End
            If frac_0 = 0 then
                Do
                    If sign = 0 then FPSCR_FPRF ← "+ denormalized number"
                    If sign = 1 then FPSCR_FPRF ← "- denormalized number"
                End
            Normalize operand:
                Do while frac_0 = 0
                    exp ← exp-1
                    frac_0:52 ← frac_1:52 || 0b0
                End
            FRT_0 ← sign
            FRT_1:11 ← exp + 1023
            FRT_12:63 ← frac_1:52
        End
    Done
```

Enabled Exponent Underflow:

```
    FPSCR_UX ← 1
    sign ← (FRB)_0
    If (FRB)_1:11 = 0 then
        Do
            exp ← -1022
            frac_0:52 ← 0b0 || (FRB)_12:63
        End
    If (FRB)_1:11 > 0 then
        Do
            exp ← (FRB)_1:11 - 1023
            frac_0:52 ← 0b1 || (FRB)_12:63
        End
    Normalize operand:
        Do while frac_0 = 0
            exp ← exp - 1
            frac_0:52 ← frac_1:52 || 0b0
        End
    Round Single(sign,exp,frac_0:52,0,0,0)
    FPSCR_XX ← FPSCR_XX | FPSCR_FI
    exp ← exp + 192
    FRT_0 ← sign
    FRT_1:11 ← exp + 1023
    FRT_12:63 ← frac_1:52
    If sign = 0 then FPSCR_FPRF ← "+ normal number"
    If sign = 1 then FPSCR_FPRF ← "- normal number"
    Done
```

Disabled Exponent Overflow:

```
    FPSCR_OX ← 1
    If FPSCR_RN = 0b00 then          /* Round to Nearest */
        Do
            If (FRB)_0 = 0 then FRT ← 0x7FF0_0000_0000_0000
            If (FRB)_0 = 1 then FRT ← 0xFFF0_0000_0000_0000
            If (FRB)_0 = 0 then FPSCR_FPRF ← "+ infinity"
            If (FRB)_0 = 1 then FPSCR_FPRF ← "- infinity"
        End
    If FPSCR_RN = 0b01 then          /* Round toward Zero */
        Do
```

```
            If (FRB)0 = 0 then FRT ← 0x47EF_FFFF_E000_0000
            If (FRB)0 = 1 then FRT ← 0xC7EF_FFFF_E000_0000
            If (FRB)0 = 0 then FPSCRFPRF ← "+ normal number"
            If (FRB)0 = 1 then FPSCRFPRF ← "- normal number"
        End
    If FPSCRRN = 0b10 then          /* Round toward +Infinity */
        Do
            If (FRB)0 = 0 then FRT ← 0x7FF0_0000_0000_0000
            If (FRB)0 = 1 then FRT ← 0xC7EF_FFFF_E000_0000
            If (FRB)0 = 0 then FPSCRFPRF ← "+ infinity"
            If (FRB)0 = 1 then FPSCRFPRF ← "- normal number"
        End
    If FPSCRRN = 0b11 then          /* Round toward -Infinity */
        Do
            If (FRB)0 = 0 then FRT ← 0x47EF_FFFF_E000_0000
            If (FRB)0 = 1 then FRT ← 0xFFF0_0000_0000_0000
            If (FRB)0 = 0 then FPSCRFPRF ← "+ normal number"
            If (FRB)0 = 1 then FPSCRFPRF ← "- infinity"
        End
    FPSCRFR ← undefined
    FPSCRFI ← 1
    FPSCRXX ← 1
    Done
```

Enabled Exponent Overflow:

```
    sign ← (FRB)0
    exp ← (FRB)1:11 - 1023
    frac0:52 ← 0b1 || (FRB)12:63
    Round Single(sign,exp,frac0:52,0,0,0)
    FPSCRXX ← FPSCRXX | FPSCRFI
Enabled Overflow:
    FPSCROX ← 1
    exp ← exp - 192
    FRT0 ← sign
    FRT1:11 ← exp + 1023
    FRT12:63 ← frac1:52
    If sign = 0 then FPSCRFPRF ← "+ normal number"
    If sign = 1 then FPSCRFPRF ← "- normal number"
    Done
```

Zero Operand:

```
    FRT ← (FRB)
    If (FRB)0 = 0 then FPSCRFPRF ← "+ zero"
    If (FRB)0 = 1 then FPSCRFPRF ← "- zero"
    FPSCRFRFI ← 0b00
    Done
```

Infinity Operand:

```
    FRT ← (FRB)
    If (FRB)0 = 0 then FPSCRFPRF ← "+ infinity"
    If (FRB)0 = 1 then FPSCRFPRF ← "- infinity"
    FPSCRFRFI ← 0b00
    Done
```

QNaN Operand:

```
    FRT ← (FRB)0:34 || 290
    FPSCRFPRF ← "QNaN"
    FPSCRFR FI ← 0b00
    Done
```

SNaN Operand:

$FPSCR_{VXSNAN} \leftarrow 1$
If $FPSCR_{VE} = 0$ then
  Do
    $FRT_{0:11} \leftarrow (FRB)_{0:11}$
    $FRT_{12} \leftarrow 1$
    $FRT_{13:63} \leftarrow (FRB)_{13:34} \,||\, {}^{29}0$
    $FPSCR_{FPRF} \leftarrow$ "QNaN"
  End
$FPSCR_{FR\ FI} \leftarrow 0b00$
Done

Normal Operand:

sign $\leftarrow (FRB)_0$
exp $\leftarrow (FRB)_{1:11}$ - 1023
$frac_{0:52} \leftarrow 0b1 \,||\, (FRB)_{12:63}$
Round Single(sign,exp,$frac_{0:52}$,0,0,0)
$FPSCR_{XX} \leftarrow FPSCR_{XX} \mid FPSCR_{FI}$
If exp > 127 and $FPSCR_{OE} = 0$ then go to Disabled Exponent Overflow
If exp > 127 and $FPSCR_{OE} = 1$ then go to Enabled Overflow
$FRT_0 \leftarrow$ sign
$FRT_{1:11} \leftarrow$ exp + 1023
$FRT_{12:63} \leftarrow frac_{1:52}$
If sign = 0 then $FPSCR_{FPRF} \leftarrow$ "+ normal number"
If sign = 1 then $FPSCR_{FPRF} \leftarrow$ "- normal number"
Done

Round Single(sign,exp,$frac_{0:52}$,G,R,X):

inc $\leftarrow$ 0
lsb $\leftarrow frac_{23}$
gbit $\leftarrow frac_{24}$
rbit $\leftarrow frac_{25}$
xbit $\leftarrow (frac_{26:52}||G||R||X) \neq 0$
If $FPSCR_{RN} = 0b00$ then /* Round to Nearest */
  Do /* comparisons ignore u bits */
    If sign || lsb || gbit || rbit || xbit = 0bu11uu then inc $\leftarrow$ 1
    If sign || lsb || gbit || rbit || xbit = 0bu011u then inc $\leftarrow$ 1
    If sign || lsb || gbit || rbit || xbit = 0bu01u1 then inc $\leftarrow$ 1
  End
If $FPSCR_{RN} = 0b10$ then /* Round toward + Infinity */
  Do /* comparisons ignore u bits */
    If sign || lsb || gbit || rbit || xbit = 0b0u1uu then inc $\leftarrow$ 1
    If sign || lsb || gbit || rbit || xbit = 0b0uu1u then inc $\leftarrow$ 1
    If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then inc $\leftarrow$ 1
  End
If $FPSCR_{RN} = 0b11$ then /* Round toward - Infinity */
  Do /* comparisons ignore u bits */
    If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc $\leftarrow$ 1
    If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc $\leftarrow$ 1
    If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc $\leftarrow$ 1
  End
$frac_{0:23} \leftarrow frac_{0:23}$ + inc
If carry_out = 1 then
  Do
    $frac_{0:23} \leftarrow 0b1 \,||\, frac_{0:22}$
    exp $\leftarrow$ exp + 1
  End
$frac_{24:52} \leftarrow {}^{29}0$
$FPSCR_{FR} \leftarrow$ inc
$FPSCR_{FI} \leftarrow$ gbit | rbit | xbit
Return

A.2 Floating-Point Convert to Integer Model

The following describes algorithmically the operation of the *Floating Convert To Integer* instructions.

```
if Floating Convert To Integer Word then do
   round_mode    ← FPSCR[RN]
   tgt_precision ← "32-bit signed integer"
end

if Floating Convert To Integer Word Unsigned then do
   round_mode    ← FPSCR[RN]
   tgt_precision ← "32-bit unsigned integer"
end

if Floating Convert To Integer Word with round toward Zero then do
   round_mode    ← 0b01
   tgt_precision ← "32-bit signed integer"
end

if Floating Convert To Integer Word Unsigned with round toward Zero then do
   round_mode    ← 0b01
   tgt_precision ← "32-bit unsigned integer"
end

if Floating Convert To Integer Doubleword then do
   round_mode    ← FPSCR[RN]
   tgt_precision ← "64-bit signed integer"
end

if Floating Convert To Integer Doubleword Unsigned then do
   round_mode    ← FPSCR[RN]
   tgt_precision ← "64-bit unsigned integer"
end

if Floating Convert To Integer Doubleword with round toward Zero then do
   round_mode ← 0b01
   tgt_precision ← "64-bit signed integer"
end

if Floating Convert To Integer Doubleword Unsigned with round toward Zero then do
   round_mode    ← 0b01
   tgt_precision ← "64-bit unsigned integer"
end

sign ← (FRB)[0]
if (FRB)[1:11] = 2047 and (FRB)[12:63] = 0 then goto Infinity Operand
if (FRB)[1:11] = 2047 and (FRB)[12] = 0 then goto SNaN Operand
if (FRB)[1:11] = 2047 and (FRB)[12] = 1 then goto QNaN Operand
if (FRB)[1:11] > 1086 then goto Large Operand

if (FRB)[1:11] > 0 then exp ← (FRB)[1:11] - 1023   /* exp - bias */
if (FRB)[1:11] = 0 then exp ← -1022
if (FRB)[1:11] > 0 then frac[0:64] ← 0b01 || (FRB)[12:63] || [11]0   /* normal */
if (FRB)[1:11] = 0 then frac[0:64] ← 0b00 || (FRB)[12:63] || [11]0   /* denormal */

gbit || rbit || xbit ← 0b000
do i=1,63-exp   /* do the loop 0 times if exp = 63 */
   frac[0:64] || gbit || rbit || xbit ← 0b0 || frac[0:64] || gbit || (rbit | xbit)
end

Round Integer( sign, frac[0:64], gbit, rbit, xbit, round_mode )

if sign = 1 then frac[0:64] ← ¬frac[0:64] + 1  /* needed leading 0 for -2^64<(FRB)<-2^63 */
```

```
if tgt_precision = “32-bit signed integer” and frac0:64 > 2^31-1 then
   goto Large Operand
if tgt_precision = “64-bit signed integer” and frac0:64 > 2^63-1 then
   goto Large Operand
if tgt_precision = “32-bit signed integer” and frac0:64 < -2^31  then
   goto Large Operand
if tgt_precision = “64-bit signed integer” and frac0:64 < -2^63  then
   goto Large Operand

if tgt_precision = “32-bit unsigned integer” & frac0:64 > 2^32-1 then
   goto Large Operand
if tgt_precision = “64-bit unsigned integer” & frac0:64 > 2^64-1 then
   goto Large Operand
if tgt_precision = “32-bit unsigned integer” & frac0:64 < 0 then
   goto Large Operand
if tgt_precision = “64-bit unsigned integer” & frac0:64 < 0 then
   goto Large Operand

FPSCRXX ← FPSCRXX | FPSCRFI

if tgt_precision = “32-bit signed integer”   then FRT ← 0xUUUU_UUUU || frac33:64
if tgt_precision = “32-bit unsigned integer” then FRT ← 0xUUUU_UUUU || frac33:64
if tgt_precision = “64-bit signed integer”   then FRT ← frac1:64
if tgt_precision = “64-bit unsigned integer” then FRT ← frac1:64
FPSCRFPRF ← 0bUUUUU
done
```

Round Integer(sign, $frac_{0:64}$, gbit, rbit, xbit, round_mode):

```
inc ← 0
if round_mode = 0b00 then do   /* Round to Nearest */
   if sign || frac64 || gbit || rbit || xbit = 0bU11UU then inc ← 1
   if sign || frac64 || gbit || rbit || xbit = 0bU011U then inc ← 1
   if sign || frac64 || gbit || rbit || xbit = 0bU01U1 then inc ← 1
end
if round_mode = 0b10 then do   /* Round toward +Infinity */
   if sign || frac64 || gbit || rbit || xbit = 0b0U1UU then inc ← 1
   if sign || frac64 || gbit || rbit || xbit = 0b0UU1U then inc ← 1
   if sign || frac64 || gbit || rbit || xbit = 0b0UUU1 then inc ← 1
end
if round_mode = 0b11 then do   /* Round toward -Infinity */
   if sign || frac64 || gbit || rbit || xbit = 0b1U1UU then inc ← 1
   if sign || frac64 || gbit || rbit || xbit = 0b1UU1U then inc ← 1
   if sign || frac64 || gbit || rbit || xbit = 0b1UUU1 then inc ← 1
end
frac0:64 ← frac0:64 + inc
FPSCRFR ← inc
FPSCRFI ← gbit | rbit | xbit
return
```

Infinity Operand:

```
FPSCRFR ← 0b0
FPSCRFI ← 0b0
FPSCRVXCVI ← 0b1
if FPSCRVE = 0 then do
   if tgt_precision = “32-bit signed integer” then do
      if sign=0 then FRT ← 0xUUUU_UUUU_7FFF_FFFF
      if sign=1 then FRT ← 0xUUUU_UUUU_8000_0000
   end
   else if tgt_precision = “32-bit unsigned integer” then do
      if sign=0 then FRT ← 0xUUUU_UUUU_FFFF_FFFF
      if sign=1 then FRT ← 0xUUUU_UUUU_0000_0000
   end
   else if tgt_precision = “64-bit signed integer” then do
      if sign=0 then FRT ← 0x7FFF_FFFF_FFFF_FFFF
      if sign=1 then FRT ← 0x8000_0000_0000_0000
```

```
      end

      else if tgt_precision = “64-bit unsigned integer” then do
         if sign=0 then FRT ← 0xFFFF_FFFF_FFFF_FFFF
         if sign=1 then FRT ← 0x0000_0000_0000_0000
      end
      FPSCR_FPRF ← 0bUUUUU
   end
   done
```

SNaN Operand:

```
   FPSCR_FR ← 0b0
   FPSCR_FI ← 0b0
   FPSCR_VXSNAN ← 0b1
   FPSCR_VXCVI ← 0b1
   if FPSCR_VE = 0 then do
      if tgt_precision = “32-bit signed integer” then FRT ← 0xUUUU_UUUU_8000_0000
      if tgt_precision = “64-bit signed integer” then FRT ← 0x8000_0000_0000_0000
      if tgt_precision = “32-bit unsigned integer” then FRT ← 0xUUUU_UUUU_0000_0000
      if tgt_precision = “64-bit unsigned integer” then FRT ← 0x0000_0000_0000_0000
      FPSCR_FPRF ← 0bUUUUU
   end
   done
```

QNaN Operand:

```
   FPSCR_FR ← 0b0
   FPSCR_FI ← 0b0
   FPSCR_VXCVI ← 0b1
   if FPSCR_VE = 0 then do
      if tgt_precision = “32-bit signed integer” then FRT ← 0xUUUU_UUUU_8000_0000
      if tgt_precision = “64-bit signed integer” then FRT ← 0x8000_0000_0000_0000
      if tgt_precision = “32-bit unsigned integer” then FRT ← 0xUUUU_UUUU_0000_0000
      if tgt_precision = “64-bit unsigned integer” then FRT ← 0x0000_0000_0000_0000
      FPSCR_FPRF ← 0bUUUUU
   end
   done
```

Large Operand:

```
   FPSCR_FR ← 0b0
   FPSCR_FI ← 0b0
   FPSCR_VXCVI ← 0b1
   if FPSCR_VE = 0 then do
      if tgt_precision = “32-bit signed integer” then do
         if sign = 0 then FRT ← 0xUUUU_UUUU_7FFF_FFFF
         if sign = 1 then FRT ← 0xUUUU_UUUU_8000_0000
      end
      else if tgt_precision = “64-bit signed integer” then do
         if sign = 0 then FRT ← 0x7FFF_FFFF_FFFF_FFFF
         if sign = 1 then FRT ← 0x8000_0000_0000_0000
      end
      else if tgt_precision = “32-bit unsigned integer” then do
         if sign = 0 then FRT ← 0xUUUU_UUUU_FFFF_FFFF
         if sign = 1 then FRT ← 0xUUUU_UUUU_0000_0000
      end
      else if tgt_precision = “64-bit unsigned integer” then do
         if sign = 0 then FRT ← 0xFFFF_FFFF_FFFF_FFFF
         if sign = 1 then FRT ← 0x0000_0000_0000_0000
      end
      FPSCR_FPRF ← 0bUUUUU
   end
   done
```

A.3 Floating-Point Convert from Integer Model

The following describes algorithmically the operation of the *Floating Convert From Integer* instructions.

```
if Floating Convert From Integer Doubleword then do
   tgt_precision ← “double-precision”
   sign    ← (FRB)0
   exp     ← 63
   frac0:63 ← (FRB)
end
if Floating Convert From Integer Doubleword Single then do
   tgt_precision ← “single-precision”
   sign    ← (FRB)0
   exp     ← 63
   frac0:63 ← (FRB)
end
if Floating Convert From Integer Doubleword Unsigned then do
   tgt_precision ← “double-precision”
   sign    ← 0
   exp     ← 63
   frac0:63 ← (FRB)
end
if Floating Convert From Integer Doubleword Unsigned Single then do
   tgt_precision ← “single-precision”
   sign    ← 0
   exp     ← 63
   frac0:63 ← (FRB)
end

if frac0:63 = 0 then go to Zero Operand
if sign = 1 then frac0:63 ← ¬frac0:63 + 1

/* do the loop 0 times if (FRB) = max negative 64-bit integer or */
/*                     if (FRB) = max unsigned 64-bit integer    */
do while frac0 = 0
   frac0:63 ← frac1:63 || 0b0
   exp ← exp - 1
end

Round Float( sign, exp, frac0:63, RN )
if sign = 0 then FPSCRFPRF ← “+normal number”
if sign = 1 then FPSCRFPRF ← “-normal number”
FRT0    ← sign
FRT1:11  ← exp + 1023   /* exp + bias */
FRT12:63 ← frac1:52
done
```

Zero Operand:

```
FPSCRFR ← 0b00
FPSCRFI ← 0b00
FPSCRFPRF ← “+ zero”
FRT ← 0x0000_0000_0000_0000
done
```

Round Float(sign, exp, $frac_{0:63}$, round_mode):

```
inc  ← 0

if tgt_precision = “single-precision” then do
   lsb  ← frac23
   gbit ← frac24
   rbit ← frac25
   xbit ← frac26:63 > 0
end
else do  /* tgt_precision = “double-precision” */
```

```
   lsb  ← frac52
   gbit ← frac53
   rbit ← frac54
   xbit ← frac55:63 > 0
end

if round_mode = 0b00 then do                /* Round to Nearest */
   if sign || lsb || gbit || rbit || xbit = 0bU11UU then inc ← 1
   if sign || lsb || gbit || rbit || xbit = 0bU011U then inc ← 1
   if sign || lsb || gbit || rbit || xbit = 0bU01U1 then inc ← 1
end
if round_mode = 0b10 then do                /* Round toward + Infinity */
   if sign || lsb || gbit || rbit || xbit = 0b0U1UU then inc ← 1
   if sign || lsb || gbit || rbit || xbit = 0b0UU1U then inc ← 1
   if sign || lsb || gbit || rbit || xbit = 0b0UUU1 then inc ← 1
end
if round_mode = 0b11 then do                /* Round toward - Infinity */
   if sign || lsb || gbit || rbit || xbit = 0b1U1UU then inc ← 1
   if sign || lsb || gbit || rbit || xbit = 0b1UU1U then inc ← 1
   if sign || lsb || gbit || rbit || xbit = 0b1UUU1 then inc ← 1
end

if tgt_precision = “single-precision” then
   frac0:23 ← frac0:23 + inc
else  /* tgt_precision = “double-precision” */
   frac0:52 ← frac0:52 + inc

if carry_out = 1 then exp ← exp + 1

FPSCRFR ← inc
FPSCRFI ← gbit | rbit | xbit
FPSCRXX ← FPSCRXX | FPSCRFI
return
```

A.4 Floating-Point Round to Integer Model

The following describes algorithmically the operation of the *Floating Round To Integer* instructions.

If $(FRB)_{1:11}$ = 2047 and $(FRB)_{12:63}$ = 0, then goto Infinity Operand
If $(FRB)_{1:11}$ = 2047 and $(FRB)_{12}$ = 0, then goto SNaN Operand
If $(FRB)_{1:11}$ = 2047 and $(FRB)_{12}$ = 1, then goto QNaN Operand
if $(FRB)_{1:63}$ = 0 then goto Zero Operand
If $(FRB)_{1:11}$ < 1023 then goto Small Operand /* exp < 0; |value| < 1*/
If $(FRB)_{1:11}$ > 1074 then goto Large Operand /* exp > 51; integral value */

sign ← $(FRB)_0$
exp ← $(FRB)_{1:11}$ - 1023 /* exp - bias */
$frac_{0:52}$ ← 0b1 || $(FRB)_{12:63}$
gbit || rbit || xbit ← 0b000

Do i = 1, 52 - exp
    $frac_{0:52}$ || gbit || rbit || xbit ← 0b0 || $frac_{0:52}$ || gbit || (rbit | xbit)
End

Round Integer (sign, $frac_{0:52}$, gbit, rbit, xbit)

Do i = 2, 52 - exp
    $frac_{0:52}$ ← $frac_{1:52}$ || 0b0
End

If $frac_0$ = 1, then exp ← exp + 1
Else $frac_{0:52}$ ← $frac_{1:52}$ || 0b0

FRT_0 ← sign
$FRT_{1:11}$ ← exp + 1023
$FRT_{12:63}$ ← $frac_{1:52}$

If $(FRT)_0$ = 0 then $FPSCR_{FPRF}$ ← “+ normal number”
Else $FPSCR_{FPRF}$ ← “- normal number”
$FPSCR_{FR\ FI}$ ← 0b00
Done

Round Integer(sign, frac0:52, gbit, rbit, xbit):

inc ← 0
If inst = Floating Round to Integer Nearest then /* ties away from zero */
    Do /* comparisons ignore u bits */
        If sign || $frac_{52}$ || gbit || rbit || xbit = 0buu1uu then inc ← 1
    End
If inst = Floating Round to Integer Plus then
    Do /* comparisons ignore u bits */
        If sign || $frac_{52}$ || gbit || rbit || xbit = 0b0u1uu then inc ← 1
        If sign || $frac_{52}$ || gbit || rbit || xbit = 0b0uu1u then inc ← 1
        If sign || $frac_{52}$ || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
    End
If inst = Floating Round to Integer Minus then
    Do /* comparisons ignore u bits */
        If sign || $frac_{52}$ || gbit || rbit || xbit = 0b1u1uu then inc ← 1
        If sign || $frac_{52}$ || gbit || rbit || xbit = 0b1uu1u then inc ← 1
        If sign || $frac_{52}$ || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
    End
$frac_{0:52}$ ← $frac_{0:52}$ + inc
Return

Infinity Operand:

FRT ← (FRB)
If $(FRB)_0$ = 0 then $FPSCR_{FPRF}$ ← "+ infinity"
If $(FRB)_0$ = 1 then $FPSCR_{FPRF}$ ← "- infinity"
$FPSCR_{FR\ FI}$ ← 0b00
Done

SNaN Operand:

$FPSCR_{VXSNAN}$ ← 1
If $FPSCR_{VE}$ = 0 then
    Do
        FRT ← (FRB)
        FRT_{12} ← 1
        $FPSCR_{FPRF}$ ← "QNaN"
    End
$FPSCR_{FR\ FI}$ ← 0b00
Done

QNaN Operand:

FRT ← (FRB)
$FPSCR_{FPRF}$ ← "QNaN"
$FPSCR_{FR\ FI}$ ← 0b00
Done

Zero Operand:

If $(FRB)_0$ = 0 then
    Do
        FRT ← 0x0000_0000_0000_0000
        $FPSCR_{FPRF}$ ← "+ zero"
    End
Else
    Do
        FRT ← 0x8000_0000_0000_0000
        $FPSCR_{FPRF}$ ← "- zero"
    End
$FPSCR_{FR\ FI}$ ← 0b00
Done

Small Operand:

If inst = Floating Round to Integer Nearest and $(FRB)_{1:11}$ < 1022 then goto Zero Operand
If inst = Floating Round to Integer Toward Zero then goto Zero Operand
If inst = Floating Round to Integer Plus and $(FRB)_0$ = 1 then goto Zero Operand
If inst = Floating Round to Integer Minus and $(FRB)_0$ = 0 then goto Zero Operand

If $(FRB)_0$ = 0 then
    Do
        FRT ← 0x3FF0_0000_0000_0000
/* value = 1.0 */
        $FPSCR_{FPRF}$ ← "+ normal number"
    End
Else
    Do
        FRT ← 0xBFF0_0000_0000_0000
/* value = -1.0 */
        $FPSCR_{FPRF}$ ← "- normal number"
    End
$FPSCR_{FR\ FI}$ ← 0b00
Done

Large Operand:

FRT ← (FRB)
If FRT_0 = 0 then $FPSCR_{FPRF}$ ← "+ normal number"
Else $FPSCR_{FPRF}$ ← "- normal number"
$FPSCR_{FR\ FI}$ ← 0b00
Done

Appendix B. Densely Packed Decimal

The trailing significand field of the decimal floating-point data format is encoded using Densely Packed Decimal (DPD). DPD encoding is a compression technique which supports the representation of decimal integers of arbitrary length. Translation operates on three Binary Coded Decimal (BCD) digits at a time compressing the 12 bits into 10 bits with an algorithm that can be applied or reversed using simple Boolean operations. In the following examples, a 3-digit BCD number is represented as (abcd)(efgh)(ijkm), a 10-bit DPD number is represented as (pqr)(stu)(v)(wxy), and the Boolean operations, & (AND), | (OR), and ¬ (NOT) are used.

B.1 BCD-to-DPD Translation

The translation from a 3-digit BCD number to a 10-bit DPD can be performed through the following Boolean operations.

```
p = (f & a & i & ¬e) | (j & a & ¬i) | (b & ¬a)
q = (g & a & i & ¬e) | (k & a & ¬i) | (c & ¬a)
r = d

s = (j & ¬a & e & ¬i) | (f & ¬i & ¬e) |
    (f & ¬a & ¬e) | (e & i)
t = (k & ¬a & e & ¬i) | (g & ¬i & ¬e) |
    (g & ¬a & ¬e) | (a & i)
u = h

v = a | e | i

w = (¬e & j & ¬i) | (e & i) | a
x = (¬a & k & ¬i) | (a & i) | e
y = m
```

Alternatively, the following table can be used to perform the translation. The most significant bit of the three BCD digits (left column) is used to select a specific 10-bit encoding (right column) of the DPD.

aei	pqr stu v wxy
000	bcd fgh 0 jkm
001	bcd fgh 1 00m
010	bcd jkh 1 01m
011	bcd 10h 1 11m
100	jkd fgh 1 10m
101	fgd 01h 1 11m
110	jkd 00h 1 11m
111	00d 11h 1 11m

The full translation of a 3-digit BCD number (000 - 999) to a 10-bit DPD is shown in Table 131 on page 789, with the DPD entries shown in hexadecimal format. The BCD number is produced by replacing '_' in the leftmost column with the corresponding digit along the top row. The table is split into two halves, with the right half being a continuation of the left half.

B.2 DPD-to-BCD Translation

The translation from a 10-bit DPD to a 3-digit BCD number can be performed through the following Boolean operations.

```
a = (¬s & v & w) | (t & v & w & s) | (v & w & ¬x)
b = (p & s & x & ¬t) | (p & ¬w) | (p & ¬v)
c = (q & s & x & ¬t) | (q & ¬w) | (q & ¬v)
d = r

e = (v & ¬w & x) | (s & v & w & x) |
    (¬t & v & x & w)
f = (p & t & v & w & x & ¬s) | (s & ¬x & v) |
    (s & ¬v)
g = (q & t & w & v & x & ¬s) | (t & ¬x & v) |
    (t & ¬v)
h = u

i = (t & v & w & x) | (s & v & w & x) |
    (v & ¬w & ¬x)
j = (p & ¬s & ¬t & w & v) | (s & v & ¬w & x) |
    (p & w & ¬x & v) | (w & ¬v)
k = (q & ¬s & ¬t & v & w) | (t & v & ¬w & x) |
    (q & v & w & ¬x) | (x & ¬v)
m = y
```

Alternatively, the following table can be used to perform the translation. A combination of five bits in the DPD encoding (leftmost column) are used to specify a translation to the 3-digit BCD encoding. Dashes (-) in the table are don't cares, and can be either one or zero.

vwxst	abcd	efgh	ijkm
0----	0pqr	0stu	0wxy
100--	0pqr	0stu	100y
101--	0pqr	100u	0sty
110--	100r	0stu	0pqy
11100	100r	100u	0pqy
11101	100r	0pqu	100y
11110	0pqr	100u	100y
11111	100r	100u	100y

The full translation of the 10-bit DPD to a 3-digit BCD number is shown in Table 132 on page 790. The 10-bit DPD index is produced by concatenating the 6-bit value shown in the left column with the 4-bit index along the top row, both represented in hexadecimal. The values in parentheses are non-preferred translations and are explained further in the following section.

B.3 Preferred DPD encoding

Translating from a 3-digit BCD number (1000 numbers) to a 10-bit DPD encoding (1024 combinations) leaves 24 redundant translations. The 24 redundant combinations are evenly assigned to eight BCD numbers and are shown in the following table, with the non-preferred encoding in parentheses. The preferred encoding is produced by translating a 3-digit BCD number with the translation table or Boolean operations shown in Section B.1. The redundant DPD encodings are all valid and will be correctly translated to their respective BCD value through the mechanisms provided in Section B.2. For decimal floating-point operations all DPD encodings are recognized as source operands.

DPD Code	BCD Value	DPD Code	BCD Value
0x06E	888	0x0EE	988
(0x16E)		(0x1EE)	
(0x26E)		(0x2EE)	
(0x36E)		(0x3EE)	
0x06F	889	0x0EF	989
(0x16F)		(0x1EF)	
(0x26F)		(0x2EF)	
(0x36F)		(0x3EF)	
0x07E	898	0x0FE	998
(0x17E)		(0x1FE)	
(0x27E)		(0x2FE)	
(0x37E)		(0x3FE)	
0x07F	899	0x0FF	999
(0x17F)		(0x1FF)	
(0x27F)		(0x2FF)	
(0x37F)		(0x3FF)	

Table 131:BCD-to-DPD translation

	0	1	2	3	4	5	6	7	8	9
00_	000	001	002	003	004	005	006	007	008	009
01_	010	011	012	013	014	015	016	017	018	019
02_	020	021	022	023	024	025	026	027	028	029
03_	030	031	032	033	034	035	036	037	038	039
04_	040	041	042	043	044	045	046	047	048	049
05_	050	051	052	053	054	055	056	057	058	059
06_	060	061	062	063	064	065	066	067	068	069
07_	070	071	072	073	074	075	076	077	078	079
08_	00A	00B	02A	02B	04A	04B	06A	06B	04E	04F
09_	01A	01B	03A	03B	05A	05B	07A	07B	05E	05F
10_	080	081	082	083	084	085	086	087	088	089
11_	090	091	092	093	094	095	096	097	098	099
12_	0A0	0A1	0A2	0A3	0A4	0A5	0A6	0A7	0A8	0A9
13_	0B0	0B1	0B2	0B3	0B4	0B5	0B6	0B7	0B8	0B9
14_	0C0	0C1	0C2	0C3	0C4	0C5	0C6	0C7	0C8	0C9
15_	0D0	0D1	0D2	0D3	0D4	0D5	0D6	0D7	0D8	0D9
16_	0E0	0E1	0E2	0E3	0E4	0E5	0E6	0E7	0E8	0E9
17_	0F0	0F1	0F2	0F3	0F4	0F5	0F6	0F7	0F8	0F9
18_	08A	08B	0AA	0AB	0CA	0CB	0EA	0EB	0CE	0CF
19_	09A	09B	0BA	0BB	0DA	0DB	0FA	0FB	0DE	0DF
20_	100	101	102	103	104	105	106	107	108	109
21_	110	111	112	113	114	115	116	117	118	119
22_	120	121	122	123	124	125	126	127	128	129
23_	130	131	132	133	134	135	136	137	138	139
24_	140	141	142	143	144	145	146	147	148	149
25_	150	151	152	153	154	155	156	157	158	159
26_	160	161	162	163	164	165	166	167	168	169
27_	170	171	172	173	174	175	176	177	178	179
28_	10A	10B	12A	12B	14A	14B	16A	16B	14E	14F
29_	11A	11B	13A	13B	15A	15B	17A	17B	15E	15F
30_	180	181	182	183	184	185	186	187	188	189
31_	190	191	192	193	194	195	196	197	198	199
32_	1A0	1A1	1A2	1A3	1A4	1A5	1A6	1A7	1A8	1A9
33_	1B0	1B1	1B2	1B3	1B4	1B5	1B6	1B7	1B8	1B9
34_	1C0	1C1	1C2	1C3	1C4	1C5	1C6	1C7	1C8	1C9
35_	1D0	1D1	1D2	1D3	1D4	1D5	1D6	1D7	1D8	1D9
36_	1E0	1E1	1E2	1E3	1E4	1E5	1E6	1E7	1E8	1E9
37_	1F0	1F1	1F2	1F3	1F4	1F5	1F6	1F7	1F8	1F9
38_	18A	18B	1AA	1AB	1CA	1CB	1EA	1EB	1CE	1CF
39_	19A	19B	1BA	1BB	1DA	1DB	1FA	1FB	1DE	1DF
40_	200	201	202	203	204	205	206	207	208	209
41_	210	211	212	213	214	215	216	217	218	219
42_	220	221	222	223	224	225	226	227	228	229
43_	230	231	232	233	234	235	236	237	238	239
44_	240	241	242	243	244	245	246	247	248	249
45_	250	251	252	253	254	255	256	257	258	259
46_	260	261	262	263	264	265	266	267	268	269
47_	270	271	272	273	274	275	276	277	278	279
48_	20A	20B	22A	22B	24A	24B	26A	26B	24E	24F
49_	21A	21B	23A	23B	25A	25B	27A	27B	25E	25F
50_	280	281	282	283	284	285	286	287	288	289
51_	290	291	292	293	294	295	296	297	298	299
52_	2A0	2A1	2A2	2A3	2A4	2A5	2A6	2A7	2A8	2A9
53_	2B0	2B1	2B2	2B3	2B4	2B5	2B6	2B7	2B8	2B9
54_	2C0	2C1	2C2	2C3	2C4	2C5	2C6	2C7	2C8	2C9
55_	2D0	2D1	2D2	2D3	2D4	2D5	2D6	2D7	2D8	2D9
56_	2E0	2E1	2E2	2E3	2E4	2E5	2E6	2E7	2E8	2E9
57_	2F0	2F1	2F2	2F3	2F4	2F5	2F6	2F7	2F8	2F9
58_	28A	28B	2AA	2AB	2CA	2CB	2EA	2EB	2CE	2CF
59_	29A	29B	2BA	2BB	2DA	2DB	2FA	2FB	2DE	2DF
60_	300	301	302	303	304	305	306	307	308	309
61_	310	311	312	313	314	315	316	317	318	319
62_	320	321	322	323	324	325	326	327	328	329
63_	330	331	332	333	334	335	336	337	338	339
64_	340	341	342	343	344	345	346	347	348	349
65_	350	351	352	353	354	355	356	357	358	359
66_	360	361	362	363	364	365	366	367	368	369
67_	370	371	372	373	374	375	376	377	378	379
68_	30A	30B	32A	32B	34A	34B	36A	36B	34E	34F
69_	31A	31B	33A	33B	35A	35B	37A	37B	35E	35F
70_	380	381	382	383	384	385	386	387	388	389
71_	390	391	392	393	394	395	396	397	398	399
72_	3A0	3A1	3A2	3A3	3A4	3A5	3A6	3A7	3A8	3A9
73_	3B0	3B1	3B2	3B3	3B4	3B5	3B6	3B7	3B8	3B9
74_	3C0	3C1	3C2	3C3	3C4	3C5	3C6	3C7	3C8	3C9
75_	3D0	3D1	3D2	3D3	3D4	3D5	3D6	3D7	3D8	3D9
76_	3E0	3E1	3E2	3E3	3E4	3E5	3E6	3E7	3E8	3E9
77_	3F0	3F1	3F2	3F3	3F4	3F5	3F6	3F7	3F8	3F9
78_	38A	38B	3AA	3AB	3CA	3CB	3EA	3EB	3CE	3CF
79_	39A	39B	3BA	3BB	3DA	3DB	3FA	3FB	3DE	3DF
80_	00C	00D	10C	10D	20C	20D	30C	30D	02E	02F
81_	01C	01D	11C	11D	21C	21D	31C	31D	03E	03F
82_	02C	02D	12C	12D	22C	22D	32C	32D	12E	12F
83_	03C	03D	13C	13D	23C	23D	33C	33D	13E	13F
84_	04C	04D	14C	14D	24C	24D	34C	34D	22E	22F
85_	05C	05D	15C	15D	25C	25D	35C	35D	23E	23F
86_	06C	06D	16C	16D	26C	26D	36C	36D	32E	32F
87_	07C	07D	17C	17D	27C	27D	37C	37D	33E	33F
88_	00E	00F	10E	10F	20E	20F	30E	30F	06E	06F
89_	01E	01F	11E	11F	21E	21F	31E	31F	07E	07F
90_	08C	08D	18C	18D	28C	28D	38C	38D	0AE	0AF
91_	09C	09D	19C	19D	29C	29D	39C	39D	0BE	0BF
92_	0AC	0AD	1AC	1AD	2AC	2AD	3AC	3AD	1AE	1AF
93_	0BC	0BD	1BC	1BD	2BC	2BD	3BC	3BD	1BE	1BF
94_	0CC	0CD	1CC	1CD	2CC	2CD	3CC	3CD	2AE	2AF
95_	0DC	0DD	1DC	1DD	2DC	2DD	3DC	3DD	2BE	2BF
96_	0EC	0ED	1EC	1ED	2EC	2ED	3EC	3ED	3AE	3AF
97_	0FC	0FD	1FC	1FD	2FC	2FD	3FC	3FD	3BE	3BF
98_	08E	08F	18E	18F	28E	28F	38E	38F	0EE	0EF
99_	09E	09F	19E	19F	29E	29F	39E	39F	0FE	0FF

	Table 132: DPD-to-BCD translation															
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
00_	000	001	002	003	004	005	006	007	008	009	080	081	800	801	880	881
01_	010	011	012	013	014	015	016	017	018	019	090	091	810	811	890	891
02_	020	021	022	023	024	025	026	027	028	029	082	083	820	821	808	809
03_	030	031	032	033	034	035	036	037	038	039	092	093	830	831	818	819
04_	040	041	042	043	044	045	046	047	048	049	084	085	840	841	088	089
05_	050	051	052	053	054	055	056	057	058	059	094	095	850	851	098	099
06_	060	061	062	063	064	065	066	067	068	069	086	087	860	861	888	889
07_	070	071	072	073	074	075	076	077	078	079	096	097	870	871	898	899
08_	100	101	102	103	104	105	106	107	108	109	180	181	900	901	980	981
09_	110	111	112	113	114	115	116	117	118	119	190	191	910	911	990	991
0A_	120	121	122	123	124	125	126	127	128	129	182	183	920	921	908	909
0B_	130	131	132	133	134	135	136	137	138	139	192	193	930	931	918	919
0C_	140	141	142	143	144	145	146	147	148	149	184	185	940	941	188	189
0D_	150	151	152	153	154	155	156	157	158	159	194	195	950	951	198	199
0E_	160	161	162	163	164	165	166	167	168	169	186	187	960	961	988	989
0F_	170	171	172	173	174	175	176	177	178	179	196	197	970	971	998	999
10_	200	201	202	203	204	205	206	207	208	209	280	281	802	803	882	883
11_	210	211	212	213	214	215	216	217	218	219	290	291	812	813	892	893
12_	220	221	222	223	224	225	226	227	228	229	282	283	822	823	828	829
13_	230	231	232	233	234	235	236	237	238	239	292	293	832	833	838	839
14_	240	241	242	243	244	245	246	247	248	249	284	285	842	843	288	289
15_	250	251	252	253	254	255	256	257	258	259	294	295	852	853	298	299
16_	260	261	262	263	264	265	266	267	268	269	286	287	862	863	(888)	(889)
17_	270	271	272	273	274	275	276	277	278	279	296	297	872	873	(898)	(899)
18_	300	301	302	303	304	305	306	307	308	309	380	381	902	903	982	983
19_	310	311	312	313	314	315	316	317	318	319	390	391	912	913	992	993
1A_	320	321	322	323	324	325	326	327	328	329	382	383	922	923	928	929
1B_	330	331	332	333	334	335	336	337	338	339	392	393	932	933	938	939
1C_	340	341	342	343	344	345	346	347	348	349	384	385	942	943	388	389
1D_	350	351	352	353	354	355	356	357	358	359	394	395	952	953	398	399
1E_	360	361	362	363	364	365	366	367	368	369	386	387	962	963	(988)	(989)
1F_	370	371	372	373	374	375	376	377	378	379	396	397	972	973	(998)	(999)
20_	400	401	402	403	404	405	406	407	408	409	480	481	804	805	884	885
21_	410	411	412	413	414	415	416	417	418	419	490	491	814	815	894	895
22_	420	421	422	423	424	425	426	427	428	429	482	483	824	825	848	849
23_	430	431	432	433	434	435	436	437	438	439	492	493	834	835	858	859
24_	440	441	442	443	444	445	446	447	448	449	484	485	844	845	488	489
25_	450	451	452	453	454	455	456	457	458	459	494	495	854	855	498	499
26_	460	461	462	463	464	465	466	467	468	469	486	487	864	865	(888)	(889)
27_	470	471	472	473	474	475	476	477	478	479	496	497	874	875	(898)	(899)
28_	500	501	502	503	504	505	506	507	508	509	580	581	904	905	984	985
29_	510	511	512	513	514	515	516	517	518	519	590	591	914	915	994	995
2A_	520	521	522	523	524	525	526	527	528	529	582	583	924	925	948	949
2B_	530	531	532	533	534	535	536	537	538	539	592	593	934	935	958	959
2C_	540	541	542	543	544	545	546	547	548	549	584	585	944	945	588	589
2D_	550	551	552	553	554	555	556	557	558	559	594	595	954	955	598	599
2E_	560	561	562	563	564	565	566	567	568	569	586	587	964	965	(988)	(989)
2F_	570	571	572	573	574	575	576	577	578	579	596	597	974	975	(998)	(999)
30_	600	601	602	603	604	605	606	607	608	609	680	681	806	807	886	887
31_	610	611	612	613	614	615	616	617	618	619	690	691	816	817	896	897
32_	620	621	622	623	624	625	626	627	628	629	682	683	826	827	868	869
33_	630	631	632	633	634	635	636	637	638	639	692	693	836	837	878	879
34_	640	641	642	643	644	645	646	647	648	649	684	685	846	847	688	689
35_	650	651	652	653	654	655	656	657	658	659	694	695	856	857	698	699
36_	660	661	662	663	664	665	666	667	668	669	686	687	866	867	(888)	(889)
37_	670	671	672	673	674	675	676	677	678	679	696	697	876	877	(898)	(899)
38_	700	701	702	703	704	705	706	707	708	709	780	781	906	907	986	987
39_	710	711	712	713	714	715	716	717	718	719	790	791	916	917	996	997
3A_	720	721	722	723	724	725	726	727	728	729	782	783	926	927	968	969
3B_	730	731	732	733	734	735	736	737	738	739	792	793	936	937	978	979
3C_	740	741	742	743	744	745	746	747	748	749	784	785	946	947	788	789
3D_	750	751	752	753	754	755	756	757	758	759	794	795	956	957	798	799
3E_	760	761	762	763	764	765	766	767	768	769	786	787	966	967	(988)	(989)
3F_	770	771	772	773	774	775	776	777	778	779	796	797	976	977	(998)	(999)

Appendix C. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided that defines simple shorthand for the most frequently used forms of *Branch Conditional*, *Compare*, *Trap*, *Rotate and Shift*, and certain other instructions.

Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

C.1 Symbols

The following symbols are defined for use in instructions (basic or extended mnemonics) that specify a Condition Register field or a Condition Register bit. The first five (lt, ..., un) identify a bit number within a CR field. The remainder (cr0, ..., cr7) identify a CR field. An expression in which a CR field symbol is multiplied by 4 and then added to a bit-number-within-CR-field symbol and 32 can be used to identify a CR bit.

Symbol	Value	Meaning
lt	0	Less than
gt	1	Greater than
eq	2	Equal
so	3	Summary overflow
un	3	Unordered (after floating-point comparison)
cr0	0	CR Field 0
cr1	1	CR Field 1
cr2	2	CR Field 2
cr3	3	CR Field 3
cr4	4	CR Field 4
cr5	5	CR Field 5
cr6	6	CR Field 6
cr7	7	CR Field 7

The extended mnemonics in Sections C.2.2 and C.3 require identification of a CR bit: if one of the CR field symbols is used, it must be multiplied by 4 and added to a bit-number-within-CR-field (value in the range 0-3, explicit or symbolic) and 32. The extended mnemonics in Sections C.2.3 and C.5 require identification of a CR field: if one of the CR field symbols is used, it must *not* be multiplied by 4 or added to 32. (For the extended mnemonics in Section C.2.3, the bit number within the CR field is part of the extended mnemonic. The programmer identifies the CR field, and the Assembler does the multiplication and addition required to produce a CR bit number for the BI field of the underlying basic mnemonic.)

C.2 Branch Mnemonics

The mnemonics discussed in this section are variations of the *Branch Conditional* instructions.

Note: ***bclr***, ***bclrl***, ***bcctr***, and ***bcctrl*** each serve as both a basic and an extended mnemonic. The Assembler will recognize a ***bclr***, ***bclrl***, ***bcctr***, or ***bcctrl*** mnemonic with three operands as the basic form, and a ***bclr***, ***bclrl***, ***bcctr***, or ***bcctrl*** mnemonic with two operands as the extended form. In the extended form the BH operand is omitted and assumed to be 0b00. Similarly, for all the extended mnemonics described in Sections C.2.2 - C.2.4 that devolve to any of these four basic mnemonics the BH operand can either be coded or omitted. If it is omitted it is assumed to be 0b00.

C.2.1 BO and BI Fields

The 5-bit BO and BI fields control whether the branch is taken. Providing an extended mnemonic for every possible combination of these fields would be neither useful nor practical. The mnemonics described in Sections C.2.2 - C.2.4 include the most useful cases. Other cases can be coded using a basic *Branch Conditional* mnemonic (***bc***[***l***][***a***], ***bclr***[***l***], ***bcctr***[***l***]) with the appropriate operands.

C.2.2 Simple Branch Mnemonics

Instructions using one of the mnemonics in Table 133 that tests a Condition Register bit specify the corresponding bit as the first operand. The symbols defined in Section C.1 can be used in this operand.

Notice that there are no extended mnemonics for relative and absolute unconditional branches. For these the basic mnemonics ***b***, ***ba***, ***bl***, and ***bla*** should be used.

Table 133:Simple branch mnemonics

Branch Semantics	LR not Set				LR Set			
	bc **Relative**	***bca*** **Absolute**	***bclr*** **To LR**	***bcctr*** **To CTR**	***bcl*** **Relative**	***bcla*** **Absolute**	***bclrl*** **To LR**	***bcctrl*** **To CTR**
Branch unconditionally	-	-	blr	bctr	-	-	blrl	bctrl
Branch if CR_{BI}=1	bt	bta	btlr	btctr	btl	btla	btlrl	btctrl
Branch if CR_{BI}=0	bf	bfa	bflr	bfctr	bfl	bfla	bflrl	bfctrl
Decrement CTR, branch if CTR nonzero	bdnz	bdnza	bdnzlr	-	bdnzl	bdnzla	bdnzlrl	-
Decrement CTR, branch if CTR nonzero and CR_{BI}=1	bdnzt	bdnzta	bdnztlr	-	bdnztl	bdnztla	bdnztlrl	-
Decrement CTR, branch if CTR nonzero and CR_{BI}=0	bdnzf	bdnzfa	bdnzflr	-	bdnzfl	bdnzfla	bdnzflrl	-
Decrement CTR, branch if CTR zero	bdz	bdza	bdzlr	-	bdzl	bdzla	bdzlrl	-
Decrement CTR, branch if CTR zero and CR_{BI}=1	bdzt	bdzta	bdztlr	-	bdztl	bdztla	bdztlrl	-
Decrement CTR, branch if CTR zero and CR_{BI}=0	bdzf	bdzfa	bdzflr	-	bdzfl	bdzfla	bdzflrl	-

Examples

1. Decrement CTR and branch if it is still nonzero (closure of a loop controlled by a count loaded into CTR).

   bdnz target (equivalent to: bc 16,0,target)

2. Same as (1) but branch only if CTR is nonzero and condition in CR0 is “equal”.

   bdnzt eq,target (equivalent to: bc 8,2,target)

3. Same as (2), but “equal” condition is in CR5.

   bdnzt 4×cr5+eq,target (equivalent to: bc 8,22,target)

4. Branch if bit 59 of CR is 0.

   bf 27,target (equivalent to: bc 4,27,target)

5. Same as (4), but set the Link Register. This is a form of conditional “call”.

   bfl 27,target (equivalent to: bcl 4,27,target)

C.2.3 Branch Mnemonics Incorporating Conditions

In the mnemonics defined in Table 134, the test of a bit in a Condition Register field is encoded in the mnemonic.

Instructions using the mnemonics in Table 134 specify the CR field as an optional first operand. One of the CR field symbols defined in Section C.1 can be used for this operand. If the CR field being tested is CR Field 0, this operand need not be specified unless the resulting basic mnemonic is ***bclr***[***l***] or ***bcctr***[***l***] and the BH operand is specified.

A standard set of codes has been adopted for the most common combinations of branch conditions.

Code	Meaning
lt	Less than
le	Less than or equal
eq	Equal
ge	Greater than or equal
gt	Greater than
nl	Not less than
ne	Not equal
ng	Not greater than
so	Summary overflow
ns	Not summary overflow
un	Unordered (after floating-point comparison)
nu	Not unordered (after floating-point comparison)

These codes are reflected in the mnemonics shown in Table 134.

Table 134:Branch mnemonics incorporating conditions

Branch Semantics	LR not Set				LR Set			
	bc **Relative**	***bca*** **Absolute**	***bclr*** **To LR**	***bcctr*** **To CTR**	***bcl*** **Relative**	***bcla*** **Absolute**	***bclrl*** **To LR**	***bcctrl*** **To CTR**
Branch if less than	blt	blta	bltlr	bltctr	bltl	bltla	bltlrl	bltctrl
Branch if less than or equal	ble	blea	blelr	blectr	blel	blela	blelrl	blectrl
Branch if equal	beq	beqa	beqlr	beqctr	beql	beqla	beqlrl	beqctrl
Branch if greater than or equal	bge	bgea	bgelr	bgectr	bgel	bgela	bgelrl	bgectrl
Branch if greater than	bgt	bgta	bgtlr	bgtctr	bgtl	bgtla	bgtlrl	bgtctrl
Branch if not less than	bnl	bnla	bnllr	bnlctr	bnll	bnlla	bnllrl	bnlctrl
Branch if not equal	bne	bnea	bnelr	bnectr	bnel	bnela	bnelrl	bnectrl
Branch if not greater than	bng	bnga	bnglr	bngctr	bngl	bngla	bnglrl	bngctrl
Branch if summary overflow	bso	bsoa	bsolr	bsoctr	bsol	bsola	bsolrl	bsoctrl
Branch if not summary overflow	bns	bnsa	bnslr	bnsctr	bnsl	bnsla	bnslrl	bnsctrl
Branch if unordered	bun	buna	bunlr	bunctr	bunl	bunla	bunlrl	bunctrl
Branch if not unordered	bnu	bnua	bnulr	bnuctr	bnul	bnula	bnulrl	bnuctrl

Examples

1. Branch if CR0 reflects condition “not equal”.

   bne target (equivalent to: bc 4,2,target)

2. Same as (1), but condition is in CR3.

```
bne     cr3,target      (equivalent to:   bc      4,14,target)
```

3. Branch to an absolute target if CR4 specifies “greater than”, setting the Link Register. This is a form of conditional “call”.

```
bgtla   cr4,target      (equivalent to:   bcla    12,17,target)
```

4. Same as (3), but target address is in the Count Register.

```
bgtctrl cr4             (equivalent to:   bcctrl  12,17,0)
```

C.2.4 Branch Prediction

Software can use the “at” bits of *Branch Conditional* instructions to provide a hint to the processor about the behavior of the branch. If, for a given such instruction, the branch is almost always taken or almost always not taken, a suffix can be added to the mnemonic indicating the value to be used for the “at” bits.

\+ Predict branch to be taken (at=0b11)

\- Predict branch not to be taken (at=0b10)

Such a suffix can be added to any *Branch Conditional* mnemonic, either basic or extended, that tests either the Count Register or a CR bit (but not both). Assemblers should use 0b00 as the default value for the “at” bits, indicating that software has offered no prediction.

Examples

1. Branch if CR0 reflects condition “less than”, specifying that the branch should be predicted to be taken.

```
blt+    target
```

2. Same as (1), but target address is in the Link Register and the branch should be predicted not to be taken.

```
bltlr-
```

C.3 Condition Register Logical Mnemonics

The *Condition Register Logical* instructions can be used to set (to 1), clear (to 0), copy, or invert a given Condition Register bit. Extended mnemonics are provided that allow these operations to be coded easily.

Table 135:Condition Register logical mnemonics

Operation	Extended Mnemonic	Equivalent to
Condition Register set	crset bx	creqv bx,bx,bx
Condition Register clear	crclr bx	crxor bx,bx,bx
Condition Register move	crmove bx,by	cror bx,by,by
Condition Register not	crnot bx,by	crnor bx,by,by

The symbols defined in Section C.1 can be used to identify the Condition Register bits.

Examples

1. Set CR bit 57.

   crset 25 (equivalent to: creqv 25,25,25)

2. Clear the SO bit of CR0.

   crclr so (equivalent to: crxor 3,3,3)

3. Same as (2), but SO bit to be cleared is in CR3.

   crclr 4×cr3+so (equivalent to: crxor 15,15,15)

4. Invert the EQ bit.

   crnot eq,eq (equivalent to: crnor 2,2,2)

5. Same as (4), but EQ bit to be inverted is in CR4, and the result is to be placed into the EQ bit of CR5.

   crnot 4×cr5+eq,4×cr4+eq (equivalent to: crnor 22,18,18)

C.4 Subtract Mnemonics

C.4.1 Subtract Immediate

Although there is no “Subtract Immediate” instruction, its effect can be achieved by using an Add Immediate instruction with the immediate operand negated. Extended mnemonics are provided that include this negation, making the intent of the computation clearer.

```
subi    Rx,Ry,value   (equivalent to:  addi   Rx,Ry,-value)
subis   Rx,Ry,value   (equivalent to:  addis  Rx,Ry,-value)
subic   Rx,Ry,value   (equivalent to:  addic  Rx,Ry,-value)
subic.  Rx,Ry,value   (equivalent to:  addic. Rx,Ry,-value)
```

C.4.2 Subtract

The *Subtract From* instructions subtract the second operand (RA) from the third (RB). Extended mnemonics are provided that use the more “normal” order, in which the third operand is subtracted from the second. Both these mnemonics can be coded with a final “o” and/or “.” to cause the OE and/or Rc bit to be set in the underlying instruction.

```
sub     Rx,Ry,Rz      (equivalent to:  subf   Rx,Rz,Ry)
subc    Rx,Ry,Rz      (equivalent to:  subfc  Rx,Rz,Ry)
```

C.5 Compare Mnemonics

The L field in the fixed-point *Compare* instructions controls whether the operands are treated as 64-bit quantities or as 32-bit quantities. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

The BF field can be omitted if the result of the comparison is to be placed into CR Field 0. Otherwise the target CR field must be specified as the first operand. One of the CR field symbols defined in Section C.1 can be used for this operand.

Note: The Assembler will recognize a basic *Compare* mnemonic with three operands, and will generate the instruction with L=0. Thus the Assembler must require that the BF field, which normally can be omitted when CR Field 0 is the target, be specified explicitly if L is.

C.5.1 Doubleword Comparisons

Table 136:Doubleword compare mnemonics

Operation	Extended Mnemonic	Equivalent to
Compare doubleword immediate	cmpdi bf,ra,si	cmpi bf,1,ra,si
Compare doubleword	cmpd bf,ra,rb	cmp bf,1,ra,rb
Compare logical doubleword immediate	cmpldi bf,ra,ui	cmpli bf,1,ra,ui
Compare logical doubleword	cmpld bf,ra,rb	cmpl bf,1,ra,rb

Examples

1. Compare register Rx and immediate value 100 as unsigned 64-bit integers and place result into CR0.

   cmpldi Rx,100 (equivalent to: cmpli 0,1,Rx,100)

2. Same as (1), but place result into CR4.

   cmpldi cr4,Rx,100 (equivalent to: cmpli 4,1,Rx,100)

3. Compare registers Rx and Ry as signed 64-bit integers and place result into CR0.

   cmpd Rx,Ry (equivalent to: cmp 0,1,Rx,Ry)

C.5.2 Word Comparisons

Table 137:Word compare mnemonics

Operation	Extended Mnemonic	Equivalent to
Compare word immediate	cmpwi bf,ra,si	cmpi bf,0,ra,si
Compare word	cmpw bf,ra,rb	cmp bf,0,ra,rb
Compare logical word immediate	cmplwi bf,ra,ui	cmpli bf,0,ra,ui
Compare logical word	cmplw bf,ra,rb	cmpl bf,0,ra,rb

Examples

1. Compare bits 32:63 of register Rx and immediate value 100 as signed 32-bit integers and place result into CR0.

   cmpwi Rx,100 (equivalent to: cmpi 0,0,Rx,100)

2. Same as (1), but place result into CR4.

   cmpwi cr4,Rx,100 (equivalent to: cmpi 4,0,Rx,100)

3. Compare bits 32:63 of registers Rx and Ry as unsigned 32-bit integers and place result into CR0.

   cmplw Rx,Ry (equivalent to: cmpl 0,0,Rx,Ry)

C.6 Trap Mnemonics

The mnemonics defined in Table 138 are variations of the *Trap* instructions, with the most useful values of TO represented in the mnemonic rather than specified as a numeric operand.

A standard set of codes has been adopted for the most common combinations of trap conditions.

Code	Meaning	TO encoding	<	>	=	$<^u$	$>^u$
lt	Less than	16	1	0	0	0	0
le	Less than or equal	20	1	0	1	0	0
eq	Equal	4	0	0	1	0	0
ge	Greater than or equal	12	0	1	1	0	0
gt	Greater than	8	0	1	0	0	0
nl	Not less than	12	0	1	1	0	0
ne	Not equal	24	1	1	0	0	0
ng	Not greater than	20	1	0	1	0	0
llt	Logically less than	2	0	0	0	1	0
lle	Logically less than or equal	6	0	0	1	1	0
lge	Logically greater than or equal	5	0	0	1	0	1
lgt	Logically greater than	1	0	0	0	0	1
lnl	Logically not less than	5	0	0	1	0	1
lng	Logically not greater than	6	0	0	1	1	0
u	Unconditionally with parameters	31	1	1	1	1	1
(none)	Unconditional	31	1	1	1	1	1

These codes are reflected in the mnemonics shown in Table 138.

Table 138:Trap mnemonics

Trap Semantics	64-bit Comparison		32-bit Comparison	
	tdi **Immediate**	***td*** **Register**	***twi*** **Immediate**	***tw*** **Register**
Trap unconditionally	-	-	-	trap
Trap unconditionally with parameters	tdui	tdu	twui	twu
Trap if less than	tdlti	tdlt	twlti	twlt
Trap if less than or equal	tdlei	tdle	twlei	twle
Trap if equal	tdeqi	tdeq	tweqi	tweq
Trap if greater than or equal	tdgei	tdge	twgei	twge
Trap if greater than	tdgti	tdgt	twgti	twgt
Trap if not less than	tdnli	tdnl	twnli	twnl
Trap if not equal	tdnei	tdne	twnei	twne
Trap if not greater than	tdngi	tdng	twngi	twng
Trap if logically less than	tdllti	tdllt	twllti	twllt
Trap if logically less than or equal	tdllei	tdlle	twllei	twlle
Trap if logically greater than or equal	tdlgei	tdlge	twlgei	twlge
Trap if logically greater than	tdlgti	tdlgt	twlgti	twlgt
Trap if logically not less than	tdlnli	tdlnl	twlnli	twlnl
Trap if logically not greater than	tdlngi	tdlng	twlngi	twlng

Examples

1. Trap if register Rx is not 0.

   tdnei Rx,0 (equivalent to: tdi 24,Rx,0)

2. Same as (1), but comparison is to register Ry.

   tdne Rx,Ry (equivalent to: td 24,Rx,Ry)

3. Trap if bits 32:63 of register Rx, considered as a 32-bit quantity, are logically greater than 0x7FF.

   twlgti Rx,0x7FF (equivalent to: twi 1,Rx,0x7FF)

4. Trap unconditionally.

   trap (equivalent to: tw 31,0,0)

5. Trap unconditionally with immediate parameters Rx and Ry

   tdu Rx,Ry (equivalent to: td 31,Rx,Ry)

C.7 Integer Select Mnemonics

The mnemonics defined in Table 139, “Integer Select mnemonics,” on page 798 are variations of the *Integer Select* instructions, with the most useful values of BC represented in the mnemonic rather than specified as a numeric operand..

Code	Meaning
lt	Less than
eq	Equal
gt	Greater than

These codes are reflected in the mnemonics shown in Table 139.

Table 139: Integer Select mnemonics

Select semantics	***isel*** **extended mnemonic**
Integer Select if less than	isellt
Integer Select if equal	iseleq
Integer Select if greater than	iselgt

Examples

1. Set register Rx to Ry if the LT bit is set in CR0, and to Rz otherwise.

   isellt Rx,Ry,Rz (equivalent to: isel Rx,Ry,Rz,0)

2. Set register Rx to Ry if the GT bit is set in CR0, and to Rz otherwise.

   iselgt Rx,Ry,Rz (equivalent to: isel Rx,Ry,Rz,1)

3. Set register Rx to Ry if the EQ bit is set in CR0, and to Rz otherwise.

   iseleq Rx,Ry,Rz (equivalent to: isel Rx,Ry,Rz,2)

C.8 Rotate and Shift Mnemonics

The *Rotate and Shift* instructions provide powerful and general ways to manipulate register contents, but can be difficult to understand. Extended mnemonics are provided that allow some of the simpler operations to be coded easily.

Mnemonics are provided for the following types of operation.

Extract Select a field of n bits starting at bit position b in the source register; left or right justify this field in the target register; clear all other bits of the target register to 0.

Insert Select a left-justified or right-justified field of n bits in the source register; insert this field starting at bit position b of the target register; leave other bits of the target register unchanged. (No extended mnemonic is provided for insertion of a left-justified field when operating on doublewords, because such an insertion requires more than one instruction.)

Rotate Rotate the contents of a register right or left n bits without masking.

Shift Shift the contents of a register right or left n bits, clearing vacated bits to 0 (logical shift).

Clear Clear the leftmost or rightmost n bits of a register to 0.

Clear left and shift left
Clear the leftmost b bits of a register, then shift the register left by n bits. This operation can be used to scale a (known nonnegative) array index by the width of an element.

C.8.1 Operations on Doublewords

All these mnemonics can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

Table 140:Doubleword rotate and shift mnemonics

Operation	**Extended Mnemonic**	**Equivalent to**
Extract and left justify immediate	extldi ra,rs,n,b (n > 0)	rldicr ra,rs,b,n-1
Extract and right justify immediate	extrdi ra,rs,n,b (n > 0)	rldicl ra,rs,b+n,64-n
Insert from right immediate	insrdi ra,rs,n,b (n > 0)	rldimi ra,rs,64-(b+n),b
Rotate left immediate	rotldi ra,rs,n	rldicl ra,rs,n,0
Rotate right immediate	rotrdi ra,rs,n	rldicl ra,rs,64-n,0
Rotate left	rotld ra,rs,rb	rldcl ra,rs,rb,0
Shift left immediate	sldi ra,rs,n (n < 64)	rldicr ra,rs,n,63-n
Shift right immediate	srdi ra,rs,n (n < 64)	rldicl ra,rs,64-n,n
Clear left immediate	clrldi ra,rs,n (n < 64)	rldicl ra,rs,0,n
Clear right immediate	clrrdi ra,rs,n (n < 64)	rldicr ra,rs,0,63-n
Clear left and shift left immediate	clrlsldi ra,rs,b,n (n <= b < 64)	rldic ra,rs,n,b-n

Examples

1. Extract the sign bit (bit 0) of register Ry and place the result right-justified into register Rx.

   extrdi Rx,Ry,1,0 (equivalent to: rldicl Rx,Ry,1,63)

2. Insert the bit extracted in (1) into the sign bit (bit 0) of register Rz.

   insrdi Rz,Rx,1,0 (equivalent to: rldimi Rz,Rx,63,0)

3. Shift the contents of register Rx left 8 bits.

   sldi Rx,Rx,8 (equivalent to: rldicr Rx,Rx,8,55)

4. Clear the high-order 32 bits of register Ry and place the result into register Rx.

   clrldi Rx,Ry,32 (equivalent to: rldicl Rx,Ry,0,32)

C.8.2 Operations on Words

All these mnemonics can be coded with a final "." to cause the Rc bit to be set in the underlying instruction. The operations as described above apply to the low-order 32 bits of the registers, as if the registers were 32-bit registers. The Insert operations either preserve the high-order 32 bits of the target register or place rotated data there; the other operations clear these bits.

Table 141:Word rotate and shift mnemonics

Operation	**Extended Mnemonic**	**Equivalent to**
Extract and left justify immediate	extlwi ra,rs,n,b (n > 0)	rlwinm ra,rs,b,0,n-1
Extract and right justify immediate	extrwi ra,rs,n,b (n > 0)	rlwinm ra,rs,b+n,32-n,31
Insert from left immediate	inslwi ra,rs,n,b (n > 0)	rlwimi ra,rs,32-b,b,(b+n)-1
Insert from right immediate	insrwi ra,rs,n,b (n > 0)	rlwimi ra,rs,32-(b+n),b,(b+n)-1
Rotate left immediate	rotlwi ra,rs,n	rlwinm ra,rs,n,0,31
Rotate right immediate	rotrwi ra,rs,n	rlwinm ra,rs,32-n,0,31
Rotate left	rotlw ra,rs,rb	rlwnm ra,rs,rb,0,31
Shift left immediate	slwi ra,rs,n (n < 32)	rlwinm ra,rs,n,0,31-n
Shift right immediate	srwi ra,rs,n (n < 32)	rlwinm ra,rs,32-n,n,31
Clear left immediate	clrlwi ra,rs,n (n < 32)	rlwinm ra,rs,0,n,31
Clear right immediate	clrrwi ra,rs,n (n < 32)	rlwinm ra,rs,0,0,31-n
Clear left and shift left immediate	clrlslwi ra,rs,b,n (n ≤ b < 32)	rlwinm ra,rs,n,b-n,31-n

Examples

1. Extract the sign bit (bit 32) of register Ry and place the result right-justified into register Rx.

   extrwi Rx,Ry,1,0 (equivalent to: rlwinm Rx,Ry,1,31,31)

2. Insert the bit extracted in (1) into the sign bit (bit 32) of register Rz.

   insrwi Rz,Rx,1,0 (equivalent to: rlwimi Rz,Rx,31,0,0)

3. Shift the contents of register Rx left 8 bits, clearing the high-order 32 bits.

   slwi Rx,Rx,8 (equivalent to: rlwinm Rx,Rx,8,0,23)

4. Clear the high-order 16 bits of the low-order 32 bits of register Ry and place the result into register Rx, clearing the high-order 32 bits of register Rx.

   clrlwi Rx,Ry,16 (equivalent to: rlwinm Rx,Ry,0,16,31)

C.9 Move To/From Special Purpose Register Mnemonics

The ***mtspr*** and ***mfspr*** instructions specify a Special Purpose Register (SPR) as a numeric operand. Extended mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as an operand.

Table 142:Extended mnemonics for moving to/from an SPR

Special Purpose Register	Move To SPR		Move From SPR	
	Extended	Equivalent to	Extended	Equivalent to
XER	mtxer Rx	mtspr 1,Rx	mfxer Rx	mfspr Rx,1
DSCR	mtudscr Rx	mtspr 3,Rx	mfudscr Rx	mfspr Rx,3
LR	mtlr Rx	mtspr 8,Rx	mflr Rx	mfspr Rx,8
CTR	mtctr Rx	mtspr 9,Rx	mfctr Rx	mfspr Rx,9
AMR	mtuamr Rx	mtspr 13,Rx	mfuamr Rx	mfspr Rx,13
TFHAR	mttfhar Rx	mtspr 128,Rx	mftfhar Rx	mfspr Rx,128
TFIAR	mttfiar Rx	mtspr 129,Rx	mftfiar Rx	mfspr Rx,129
TEXASR	mttexasr Rx	mtspr 130,Rx	mftexasr Rx	mfspr Rx,130
TEXASRU	mttxasru Rx	mtspr 131,Rx	mftexaru Rx	mfspr Rx,131
CTRL	-	-	mfctrl Rx	mfspr Rx,136
VRSAVE	mtvrsave Rx	mtspr 256,Rx	mfvrsave Rx	mfspr Rx,256
SPRG3	-	-	mfusprg3 Rx	mfspr Rx,259
TB	-	-	mftb Rx	mftb Rx,268 mfspr Rx,268
TBU	-	-	mftbu Rx	mftb Rx,269 mfspr Rx,269
SIER	-	-	mfusier Rx	mfspr Rx,768
MMCR2	mtummcr2 Rx	mtspr 769,Rx	mfummcr2 Rx	mfspr Rx,769
MMCRA	mtummcra Rx	mtspr 770,Rx	mfummcra Rx	mfspr Rx,770
PMC1	mtupmc1 Rx	mtspr 771,Rx	mfupmc1 Rx	mfspr Rx,771
PMC2	mtupmc2 Rx	mtspr 772,Rx	mfupmc2 Rx	mfspr Rx,772
PMC3	mtupmc3 Rx	mtspr 773,Rx	mfupmc3 Rx	mfspr Rx,773
PMC4	mtupmc4 Rx	mtspr 774,Rx	mfupmc4 Rx	mfspr Rx,774
PMC5	mtupmc5 Rx	mtspr 775,Rx	mfupmc5 Rx	mfspr Rx,775
PMC6	mtupmc6 Rx	mtspr 776,Rx	mfupmc6 Rx	mfspr Rx,776
MMCR0	mtummcr0 Rx	mtspr 779,Rx	mfummcr0 Rx	mfspr Rx,779
SIAR	-	-	mfusiar Rx	mfspr Rx,780
SDAR	-	-	mfusdar Rx	mfspr Rx,781
MMCR1	-	-	mfummcr1 Rx	mfspr Rx,782
BESCRS	mtbescrs Rx	mtspr 800,Rx	mfbescrs Rx	mfspr Rx,800
BESCRU	mtbescru Rx	mtspr 801,Rx	mfbescru Rx	mfspr Rx,801
BESCRR	mtbescrr Rx	mtspr 802,Rx	mfbescrr Rx	mfspr Rx,802
BESCRRU	mtbescrru Rx	mtspr 803,Rx	mfbescrru Rx	mfspr Rx,803
EBBHR	mtebbhr Rx	mtspr 804,Rx	mfebbhr Rx	mfspr Rx,804
EBBRR	mtebbrr Rx	mtspr 805,Rx	mfebbrr Rx	mfspr Rx,805
BESCR	mtbescr Rx	mtspr 806,Rx	mfbescr Rx	mfspr Rx,806
TAR	mttar Rx	mtspr 815,Rx	mftar Rx	mfspr Rx,815
PPR	mtppr Rx	mtspr 896,Rx	mfppr Rx	mfspr Rx,896
PPR32	mtppr32 Rx	mtspr 898,Rx	mfppr32 Rx	mfspr Rx,898

Examples

1. Copy the contents of register Rx to the XER.

```
mtxer   Rx          (equivalent to:   mtspr   1,Rx)
```

2. Copy the contents of the LR to register Rx.

```
mflr    Rx          (equivalent to:   mfspr   Rx,8)
```

3. Copy the contents of register Rx to the CTR.

```
mtctr   Rx          (equivalent to:   mtspr   9,Rx)
```

C.10 Miscellaneous Mnemonics

No-op

Many Power ISA instructions can be coded in a way such that, effectively, no operation is performed. An extended mnemonic is provided for the preferred form of no-op. If an implementation performs any type of run-time optimization related to no-ops, the preferred form is the no-op that will trigger this.

```
nop                 (equivalent to:   ori     0,0,0)
```

For some uses of a no-op instruction, optimizations related to no-ops, such as removal from the execution stream, are not desireable. An extended mnemonic is provided for the executed form of no-op. This form of no-op will still consume execution resources.

```
xnop    (equivalent to:   xori    0,0,0)
```

Load Immediate

The ***addi*** and ***addis*** instructions can be used to load an immediate value into a register. Extended mnemonics are provided to convey the idea that no addition is being performed but merely data movement (from the immediate field of the instruction to a register).

Load a 16-bit signed immediate value into register Rx.

```
li      Rx,value    (equivalent to:   addi    Rx,0,value)
```

Load a 16-bit signed immediate value, shifted left by 16 bits, into register Rx.

```
lis     Rx,value    (equivalent to:   addis   Rx,0,value)
```

Load Next Instruction Address

The ***addpcis*** instruction can be used to load the next instruction address into a register. An extended mnemonics is provided to perform this operation.

```
lnia    Rx          (equivalent to:   addpcis Rx,0)
```

Load Address

This mnemonic permits computing the value of a base-displacement operand, using the ***addi*** instruction which normally requires separate register and immediate operands.

```
la      Rx,D(Ry)        (equivalent to:    addi    Rx,Ry,D)
```

The ***la*** mnemonic is useful for obtaining the address of a variable specified by name, allowing the Assembler to supply the base register number and compute the displacement. If the variable v is located at offset Dv bytes from the address in register Rv, and the Assembler has been told to use register Rv as a base for references to the data structure containing v, then the following line causes the address of v to be loaded into register Rx.

```
la      Rx,v            (equivalent to:    addi    Rx,Rv,Dv)
```

Move Register

Several Power ISA instructions can be coded in a way such that they simply copy the contents of one register to another. An extended mnemonic is provided to convey the idea that no computation is being performed but merely data movement (from one register to another).

The following instruction copies the contents of register Ry to register Rx. This mnemonic can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

```
mr      Rx,Ry           (equivalent to:    or      Rx,Ry,Ry)
```

Complement Register

Several Power ISA instructions can be coded in a way such that they complement the contents of one register and place the result into another register. An extended mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of register Ry and places the result into register Rx. This mnemonic can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

```
not     Rx,Ry           (equivalent to:    nor     Rx,Ry,Ry)
```

Move To/From Condition Register

This mnemonic permits copying the contents of the low-order 32 bits of a GPR to the Condition Register, using the same style as the ***mfcr*** instruction.

```
mtcr    Rx              (equivalent to:    mtcrf   0xFF,Rx)
```

The following instructions may generate either the (old) ***mtcrf*** or ***mfcr*** instructions or the (new) ***mtocrf*** or ***mfocrf*** instruction, respectively, depending on the target machine type assembler parameter.

```
mtcrf   FXM,Rx
mfcr    Rx
```

All three extended mnemonics in this subsection are being phased out. In future assemblers the form "mtcr Rx" may not exist, and the ***mtcrf*** and ***mfcr*** mnemonics may generate the old form instructions (with bit 11 = 0) regardless of the target machine type assembler parameter, or may cease to exist.

Book II:

Power ISA Virtual Environment Architecture

Chapter 1. Storage Model

1.1 Definitions

The following definitions, in addition to those specified in Book I, are used in this Book. In these definitions, "*Load* instruction" includes the *Cache Management* and other instructions that are stated in the instruction descriptions to be "treated as a *Load*", and similarly for "*Store* instruction".

- **system**
  A combination of processors, storage, and associated mechanisms that is capable of executing programs. Sometimes the reference to system includes services provided by the privileged software.
- **main storage**
  The level of storage hierarchy in which all storage state is visible to all processors and mechanisms in the system.
- **normal memory**
  Coherently-accessed, well-behaved system memory that holds supervisor software and general purpose applications and data, generally embodied as memory DIMMs attached to a memory controller which is in turn attached to the nest fabric. This is in contrast with memory associated with accelerators or I/O interfaces or attached to other systems
- **primary cache**
  The level of cache closest to the processor.
- **secondary cache**
  After the primary cache, the next closest level of cache to the processor.
- **instruction storage**
  The view of storage as seen by the mechanism that fetches instructions.
- **data storage**
  The view of storage as seen by a *Load* or *Store* instruction.
- **program order**
  The execution of instructions in the order required by the sequential execution model. (See Section 2.2 of Book I.) A ***dcbz*** instruction that modifies storage which contains instructions has the same effect with respect to the sequential execution model as a *Store* instruction as described there.)
  For the instructions and facilities defined in this Book, there are two additional exceptions to the sequential execution model that the processor obeys beyond those described in Section 2.2 of Book I.
  - a transaction failure handler is invoked (see Section 5.3.3)
  - an event-based branch occurs (see Chapter 7)
  - the BHRB is read (see Section 8.2)
- **event-based exception**
  An unusual condition, or external signal, that sets a status bit in the BESCR and may or may not cause an event-based branch, depending upon whether event-based branches are enabled.
- **storage location**
  A contiguous sequence of one or more bytes in storage. When used in association with a specific instruction or the instruction fetching mechanism, the length of the sequence of one or more bytes is typically implied by the operation. In other uses, it may refer more abstractly to a group of bytes which share common storage attributes.
- **storage access**
  An access to a storage location. There are three (mutually exclusive) kinds of storage access.
  - **data access**

    An access to the storage location specified by a *Load* or *Store* instruction, or, if the access is performed "out-of-order" (see Section 5.5 of Book III), an access to a storage location as if it were the storage location specified by a *Load* or *Store* instruction.
  - **instruction fetch**

    An access for the purpose of fetching an instruction.

- **implicit access**

  An access by the processor for the purpose of finding the address translation tables, translating an address, or recording reference and change information (see Book III).

- **caused by, associated with**

  - **caused by**

    A storage access is said to be caused by an instruction if the instruction is a *Load* or *Store* and the access (data access) is to the storage location specified by the instruction.

  - **associated with**

    A storage access is said to be associated with an instruction if the access is for the purpose of fetching the instruction (instruction fetch), or is a data access caused by the instruction, or is an implicit access that occurs as a side effect of fetching or executing the instruction.

- **prefetched instructions**
  Instructions for which a copy of the instruction has been fetched from instruction storage, but the instruction has not yet been executed.

- **uniprocessor**
  A system that contains one processor.

- **multiprocessor**
  A system that contains two or more processors.

- **shared storage multiprocessor**
  A multiprocessor that contains some common storage, which all the processors in the system can access.

- **performed**
  A load or instruction fetch by a processor or mechanism (P1) is performed with respect to any processor or mechanism (P2) when the value to be returned by the load or instruction fetch can no longer be changed by a store by P2. A store by P1 is performed with respect to P2 when a load by P2 from the location accessed by the store will return the value stored (or a value stored subsequently). An instruction cache block invalidation by P1 is performed with respect to P2 when the instruction that requested the invalidation has caused the specified block, if present, to be made invalid in P2's instruction cache, and similarly for a data cache block invalidation.

  The preceding definitions apply regardless of whether P1 and P2 are the same entity.

- **page (virtual page)**
  2^n contiguous bytes of storage aligned such that the effective address of the first byte in the page is an integral multiple of the page size for which protection and control attributes are independently specifiable and for which reference and change status are independently recorded.

- **block**
  The aligned unit of storage operated on by the *Cache Management* instructions. The size of an instruction cache block may differ from the size of a data cache block, and both sizes may vary between implementations. The maximum block size is equal to the minimum page size.

- **aggregate store**
  The set of stores caused by a successful transaction, which are performed as an atomic unit.

1.2 Introduction

The Power ISA User Instruction Set Architecture, discussed in Book I, defines storage as a linear array of bytes indexed from 0 to a maximum of 2^{64}-1. Each byte is identified by its index, called its address, and each byte contains a value. This information is sufficient to allow the programming of applications that require no special features of any particular system environment. The Power ISA Virtual Environment Architecture, described herein, expands this simple storage model to include caches, virtual storage, and shared storage multiprocessors. The Power ISA Virtual Environment Architecture, in conjunction with services based on the Power ISA Operating Environment Architecture (see Book III) and provided by the operating system, permits explicit control of this expanded storage model. A simple model for sequential execution allows at most one storage access to be performed at a time and requires that all storage accesses appear to be performed in program order. In contrast to this simple model, the Power ISA specifies a relaxed model of storage consistency. In a multiprocessor system that allows multiple copies of a storage location, aggressive implementations of the architecture can permit intervals of time during which different copies of a storage location have different values. This chapter describes features of the Power ISA that enable programmers to write correct programs for this storage model.

1.3 Virtual Storage

The Power ISA system implements a virtual storage model for applications. This means that a combination of hardware and software can present a storage model that allows applications to exist within a "virtual" address space larger than either the effective address space or the real address space.

Each program can access 2^{64} bytes of "effective address" (EA) space, subject to limitations imposed by the operating system. In a typical Power ISA system, each program's EA space is a subset of a larger "virtual

address" (VA) space managed by the operating system.

Each effective address is translated to a real address (i.e., to an address of a byte in real storage or on an I/O device) before being used to access storage. The hardware accomplishes this, using the address translation mechanism described in Book III. The operating system manages the real (physical) storage resources of the system, by setting up the tables and other information used by the hardware address translation mechanism.

In general, real storage may not be large enough to map all the virtual pages used by the currently active applications. With support provided by hardware, the operating system can attempt to use the available real pages to map a sufficient set of virtual pages of the applications. If a sufficient set is maintained, "paging" activity is minimized. If not, performance degradation is likely.

The operating system can support restricted access to virtual pages (including read/write, read only, and no access; see Book III), based on system standards (e.g., program code might be read only) and application requests.

1.4 Single-Copy Atomicity

An access is *single-copy atomic*, or simply *atomic*, if it is always performed in its entirety with no visible fragmentation. Atomic accesses are thus serialized: each happens in its entirety in some order, even when that order is not specified in the program or enforced between processors.

The access caused by an instruction other than a *Load/Store Multiple* or *Move Assist* instruction is guaranteed to be atomic if the storage operand is not larger than a doubleword and is aligned (see Section 1.11.1 of Book I).

Quadword accesses with aligned storage operands are guaranteed to be atomic when caused by the following instructions.

- ***lq***
- ***stq***
- ***lqarx***
- ***stqcx.***

Quadword atomicity applies only to storage that is neither Write Through Required nor Caching Inhibited. The cases described above are the only cases in which the access to the storage operand is guaranteed to be atomic. For example, the access caused by the following instructions is not guaranteed to be atomic.

- any *Load* or *Store* instruction for which the storage operand is unaligned
- ***lmw***, ***stmw***, ***lswi***, ***lswx***, ***stswi***, ***stswx***
- ***lfdp***, ***lfdpx***, ***stfdp***, ***stfdpx***
- any *Cache Management* instruction

An access that is not atomic is performed as a set of smaller disjoint atomic accesses. If the non-atomic access is caused by an instruction other than a *Load/Store Multiple* or *Move Assist* instruction and one of the following conditions is satisfied, the non-atomic access is performed as described in the corresponding list item. The first list item matching a given situation applies.

- The storage operand is one quadword and is doubleword-aligned:
  the access is performed as two disjoint aligned doubleword atomic accesses.
- The storage operand is at least eight bytes long and is word-aligned:
  the access is performed as a set of disjoint atomic accesses each of which consists of one or more aligned words.
- The storage operand is at least four bytes long and is halfword-aligned:
  the access is performed as a set of disjoint atomic accesses each of which consists of one or more aligned halfwords.

In all other cases the number, length, and alignment of the component disjoint atomic accesses are implementation-dependent. In all cases the relative order in which the component disjoint atomic accesses are performed is implementation-dependent.

The results for several combinations of loads and stores to the same or overlapping locations are described below.

1. When two processors perform atomic stores to locations that do not overlap, and no other stores are performed to those locations, the contents of those locations are the same as if the two stores were performed by a single processor.
2. When two processors perform atomic stores to the same storage location, and no other store is performed to that location, the contents of that location are the result stored by one of the processors.
3. When two processors perform stores that have the same target location and are not guaranteed to be atomic, and no other store is performed to that location, the result is some combination of the bytes stored by both processors.
4. When two processors perform stores to overlapping locations, and no other store is performed to those locations, the result is some combination of the bytes stored by the processors to the overlapping bytes. The portions of the locations that do not overlap contain the bytes stored by the processor storing to the location.
5. When a processor performs an atomic store to a location, a second processor performs an atomic load from that location, and no other store is performed to that location, the value returned by the

load is the contents of the location before the store or the contents of the location after the store.

6. When a load and a store with the same target location can be performed simultaneously, and the accesses are not guaranteed to be atomic, and no other store is performed to that location, the value returned by the load is some combination of the contents of the location before the store and the contents of the location after the store.

1.5 Cache Model

A cache model in which there is one cache for instructions and another cache for data is called a "Harvard-style" cache. This is the model assumed by the Power ISA, e.g., in the descriptions of the *Cache Management* instructions in Section 4.3. Alternative cache models may be implemented (e.g., a "combined cache" model, in which a single cache is used for both instructions and data, or a model in which there are several levels of caches), but they support the programming model implied by a Harvard-style cache.

The processor is not required to maintain copies of storage locations in the instruction cache consistent with modifications to those storage locations (e.g., modifications caused by *Store* instructions).

A location in the data cache is considered to be modified in that cache if the location has been modified (e.g., by a *Store* instruction) and the modified data have not been written to main storage.

Cache Management instructions are provided so that programs can manage the caches when needed. For example, program management of the caches is needed when a program generates or modifies code that will be executed (i.e., when the program modifies data in storage and then attempts to execute the modified data as instructions). The *Cache Management* instructions are also useful in optimizing the use of memory bandwidth in such applications as graphics and numerically intensive computing. The functions performed by these instructions depend on the storage control attributes associated with the specified storage location (see Section 1.6, "Storage Control Attributes").

The *Cache Management* instructions allow the program to do the following.

- invalidate the copy of storage in an instruction cache block (***icbi***)
- provide a hint that an instruction will probably soon be accessed from a specified instruction cache block (***icbt***)
- provide a hint that the program will probably soon access a specified data cache block (***dcbt***, ***dcbtst***)
- set the contents of a data cache block to zeros (***dcbz***)
- copy the contents of a modified data cache block to main storage (***dcbst***)
- copy the contents of a modified data cache block to main storage and make the copy of the block in the data cache invalid (***dcbf or dcbfl***)

1.6 Storage Control Attributes

Some operating systems may provide a means to allow programs to specify the storage control attributes described in this section. Because the support provided for these attributes by the operating system may vary between systems, the details of the specific system being used must be known before these attributes can be used.

Storage control attributes are associated with units of storage that are multiples of the page size. Each storage access is performed according to the storage control attributes of the specified storage location, as described below. The storage control attributes are the following.

- Write Through Required
- Caching Inhibited
- Memory Coherence Required
- Guarded
- Strong Access Order

These attributes have meaning only when an effective address is translated by the processor performing the storage access.

Programming Note

The Write Through Required and Caching Inhibited attributes are mutually exclusive because, as described below, the Write Through Required attribute permits the storage location to be in the data cache while the Caching Inhibited attribute does not.

Storage that is Write Through Required or Caching Inhibited is not intended to be used for general-purpose programming. For example, the ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, ***lqarx***, ***stbcx.***, ***sthcx.***, ***stwcx***., ***stdcx.***, and ***stqcx***. instructions may cause the system data storage error handler to be invoked if they specify a location in storage having either of these attributes. To obtain the best performance across the widest range of implementations, storage that is Write Through Required or Caching Inhibited should be used only when the use of such storage meets specific functional or semantic needs or enables a performance optimization.

In the remainder of this section, "*Load* instruction" includes the *Cache Management* and other instructions that are stated in the instruction descriptions to be "treated as a *Load*" unless they are explicitly excluded, and similarly for "*Store* instruction".

1.6.1 Write Through Required

A store to a Write Through Required storage location is performed in main storage. A Store instruction that specifies a location in Write Through Required storage may cause additional locations in main storage to be accessed. If a copy of the block containing the specified location is retained in the data cache, the store is also performed in the data cache. The store does not cause the block to be considered to be modified in the data cache.

In general, accesses caused by separate *Store* instructions that specify locations in Write Through Required storage may be combined into one access. Such combining does not occur if the *Store* instructions are separated by a ***sync***, ***eieio*** instruction.

1.6.2 Caching Inhibited

An access to a Caching Inhibited storage location is performed in main storage. A *Load* instruction that specifies a location in Caching Inhibited storage may cause additional locations in main storage to be accessed unless the specified location is also Guarded. An instruction fetch from Caching Inhibited storage may cause additional words in main storage to be accessed. No copy of the accessed locations is placed into the caches.

In general, non-overlapping accesses caused by separate *Load* instructions that specify locations in Caching Inhibited storage may be combined into one access, as may non-overlapping accesses caused by separate *Store* instructions that specify locations in Caching Inhibited storage. Such combining does not occur if the *Load* or *Store* instructions are separated by a ***sync*** instruction. Combining may also occur among such accesses from multiple processors that share a common memory interface. No combining occurs if the storage is also Guarded.

Programming Note

None of the memory barrier instructions prevent the combining of accesses from different processors. The Guarded storage attribute must be used in combination with Caching Inhibited to prevent such combining.

1.6.3 Memory Coherence Required

An access to a Memory Coherence Required storage location is performed coherently, as follows.

Memory coherence refers to the ordering of stores to a single location. Atomic stores to a given location are *coherent* if they are serialized in some order, and no processor or mechanism is able to observe any subset of those stores as occurring in a conflicting order. This serialization order is an abstract sequence of values; the physical storage location need not assume each of the values written to it. For example, a processor may update a location several times before the value is written to physical storage. The result of a store operation is not available to every processor or mechanism at the same instant, and it may be that a processor or mechanism observes only some of the values that are written to a location. However, when a location is accessed atomically and coherently by all processors and mechanisms, the sequence of values loaded from the location by any processor or mechanism during any interval of time forms a subsequence of the sequence of values that the location logically held during that interval. That is, a processor or mechanism can never load a “newer” value first and then, later, load an “older” value.

Memory coherence is managed in blocks called coherence *blocks*. Their size is implementation-dependent, but is larger than a word and is usually the size of a cache block.

For storage that is not Memory Coherence Required, software must explicitly manage memory coherence to the extent required by program correctness. The operations required to do this may be system-dependent.

Because the Memory Coherence Required attribute for a given storage location is of little use unless all processors that access the location do so coherently, in statements about Memory Coherence Required storage elsewhere in this document it is generally assumed that the storage has the Memory Coherence Required attribute for all processors that access it.

Programming Note

Operating systems that allow programs to request that storage not be Memory Coherence Required should provide services to assist in managing memory coherence for such storage, including all system-dependent aspects thereof.

In most systems the default is that all storage is Memory Coherence Required. For some applications in some systems, software management of coherence may yield better performance. In such cases, a program can request that a given unit of storage not be Memory Coherence Required, and can manage the coherence of that storage by using the ***sync*** instruction, the *Cache Management* instructions, and services provided by the operating system.

1.6.4 Guarded

A data access to a Guarded storage location is performed only if either (a) the access is caused by an instruction that is known to be required by the sequential execution model, or (b) the access is a load and the storage location is already in a cache. If the storage is

also Caching Inhibited, only the storage location specified by the instruction is accessed; otherwise any storage location in the cache block containing the specified storage location may be accessed.

Instructions are not fetched from virtual storage that is Guarded. If the instruction addressed by the current instruction address is in such storage, the system instruction storage error handler may be invoked (see Section 6.5.5 of Book III).

Programming Note

In some implementations, instructions may be executed before they are known to be required by the sequential execution model. Because the results of instructions executed in this manner are discarded if it is later determined that those instructions would not have been executed in the sequential execution model, this behavior does not affect most programs.

This behavior does affect programs that access storage locations that are not "well-behaved" (e.g., a storage location that represents a control register on an I/O device that, when accessed, causes the device to perform an operation). To avoid unintended results, programs that access such storage locations should request that the storage be Guarded, and should prevent such storage locations from being in a cache (e.g., by requesting that the storage also be Caching Inhibited).

1.6.5 Strong Access Order

All accesses to storage with the Strong Access Order (SAO) attribute (referred to as *SAO storage*) will be performed using a set of ordering rules different from that of the weakly consistent model that is described in Section 1.7.1, "Storage Access Ordering". These rules apply only to accesses that are caused by a *Load* or a *Store*, and not to accesses associated with those instructions. Furthermore, these rules do not apply to accesses that are caused by or associated with instructions that are stated in their descriptions to be "treated as a *Load*" or "treated as a *Store*." The details are described below, from the programmer's point of view. (The processor may deviate from these rules if the programmer cannot detect the deviation.) The SAO attribute is not intended to be used for general purpose programming. It is provided in a manner that is not fully independent of the other storage attributes. Specifically, it is only provided for storage that is Memory Coherence Required, but not Write Through Required, not Caching Inhibited, and not Guarded. See Section 5.8.2.1, "Storage Control Bit Restrictions", in Book III for more details. Accesses to SAO storage are likely to be performed more slowly than similar accesses to non-SAO storage.

The order in which a processor performs storage accesses to SAO storage, the order in which those accesses are performed with respect to other processors and mechanisms, and the order in which those accesses are performed in main storage are the same except in the circumstances described in the following paragraph. The ordering rules for accesses performed by a single processor to SAO storage are as follows. Stores are performed in program order. When a store accesses data adjacent to that which is accessed by the next store in program order, the two storage accesses may be combined into a single larger access. Loads are performed in program order. When a load accesses data adjacent to that which is accessed by the next load in program order, the two storage accesses may be combined into a single larger access. Stores may not be performed before loads which precede them in program order. Loads may be performed before stores which precede them in program order, with the provision that a load which follows a store of the same datum (to the same address) must obtain a value which is no older (in consideration of the possibility of programs on other processors sharing the same storage) than the value stored by the preceding store.

When any given processor loads the datum it just stored, as described above, the load may be performed by the processor before the preceding store has been performed with respect to other processors and mechanisms, and in main storage. This may cause the processor to see its store earlier relative to stores performed by other processors than it is observed by other processors and mechanisms, and than it is performed in memory. A direct consequence of this consideration is that although programs running on each processor will see the same sequence of accesses from any individual processor to SAO storage, each may in general see a different interleaving of the individual sequences. The memory barrier instructions may be used to establish stronger ordering, as described in Section 1.7.1, "Storage Access Ordering", beginning with the third major bullet.

1.7 Shared Storage

This architecture supports the sharing of storage between programs, between different instances of the same program, and between processors and other mechanisms. It also supports access to a storage location by one or more programs using different effective addresses. All these cases are considered storage sharing. Storage is shared in blocks that are an integral number of pages.

When the same storage location has different effective addresses, the addresses are said to be *aliases*. Each application can be granted separate access privileges to aliased pages.

1.7.1 Storage Access Ordering

The Power ISA defines two models for the ordering of storage accesses: weakly consistent and strong access ordering. The predominant model is weakly consistent. This model provides an opportunity for improved performance over a model that has stronger consistency rules, but places the responsibility on the program to ensure that ordering or synchronization instructions are properly placed when storage is shared by two or more programs. Implementations which support SAO apply a stronger consistency model among accesses to SAO storage. The order between accesses to SAO storage and those performed using the weakly consistent model is characteristic of the weakly consistent model. The following description, through the second major bullet, applies only to the weakly consistent model. The corresponding description for SAO storage is found in Section 1.6.5, "Strong Access Order". The rest of the description following the second bulletted item applies to both models.

The order in which the processor performs storage accesses, the order in which those accesses are performed with respect to another processor or mechanism, and the order in which those accesses are performed in main storage may all be different. Several means of enforcing an ordering of storage accesses are provided to allow programs to share storage with other programs, or with mechanisms such as I/O devices. These means are listed below. The phrase "to the extent required by the associated Memory Coherence Required attributes" refers to the Memory Coherence Required attribute, if any, associated with each access.

- If two *Store* instructions or two *Load* instructions specify storage locations that are both Caching Inhibited and Guarded, the corresponding storage accesses are performed in program order with respect to any processor or mechanism.
- If a *Load* instruction depends on the value returned by a preceding *Load* instruction (because the value is used to compute the effective address specified by the second *Load*), the corresponding storage accesses are performed in program order with respect to any processor or mechanism to the extent required by the associated Memory Coherence Required attributes. This applies even if the dependency has no effect on program logic (e.g., the value returned by the first *Load* is ANDed with zero and then added to the effective address specified by the second *Load*).
- When a processor (P1) executes a *Synchronize or* ***eieio*** instruction a *memory barrier* is created, which orders applicable storage accesses pairwise, as follows. Let A be a set of storage accesses that includes all storage accesses associated with instructions preceding the barrier-creating instruction, and let B be a set of storage accesses that includes all storage accesses associated with instructions following the barrier-creating instruction. For each applicable pair a_i,b_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor or mechanism.

  The ordering done by a memory barrier is said to be "cumulative" if it also orders storage accesses that are performed by processors and mechanisms other than P1, as follows.

  - A includes all applicable storage accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.
  - B includes all applicable storage accesses by any such processor or mechanism that are performed after a Load instruction executed by that processor or mechanism has returned the value stored by a store that is in B.

No ordering should be assumed among the storage accesses caused by a single instruction (i.e, by an instruction for which the access is not atomic), even if the accesses are to SAO storage, and no means are provided for controlling that order.

Programming Note

Because stores cannot be performed "out-of-order" (see Book III), if a *Store* instruction depends on the value returned by a preceding *Load* instruction (because the value returned by the *Load* is used to compute either the effective address specified by the *Store* or the value to be stored), the corresponding storage accesses are performed in program order. The same applies if *whether* the *Store* instruction is executed depends on a conditional *Branch* instruction that in turn depends on the value returned by a preceding *Load* instruction.

Because an ***isync*** instruction prevents the execution of instructions following the ***isync*** until instructions preceding the ***isync*** have completed, if an ***isync*** follows a conditional *Branch* instruction that depends on the value returned by a preceding *Load* instruction, the load on which the *Branch* depends is performed before any loads caused by instructions following the ***isync***. This applies even if the effects of the "dependency" are independent of the value loaded (e.g., the value is compared to itself and the *Branch* tests the EQ bit in the selected CR field), and even if the branch target is the sequentially next instruction.

With the exception of the cases described above and earlier in this section, data dependencies and control dependencies do not order storage accesses. Examples include the following.

- If a *Load* instruction specifies the same storage location as a preceding *Store* instruction and the location is in storage that is not Caching Inhibited, the load may be satisfied from a "store queue" (a buffer into which the processor places stored values before presenting them to the storage subsystem), and not be visible to other processors and mechanisms. A consequence is that if a subsequent *Store* depends on the value returned by the *Load*, the two stores need not be performed in program order with respect to other processors and mechanisms.
- Because a *Store Conditional* instruction may complete before its store has been performed, a conditional *Branch* instruction that depends on the CR0 value set by a *Store Conditional* instruction does not order the *Store Conditional*'s store with respect to storage accesses caused by instructions that follow the *Branch*.
- Because processors may predict branch target addresses and branch condition resolution, control dependencies (e.g., branches) do not order storage accesses except as described above. For example, when a subroutine returns to its caller the return address may be predicted, with the result that loads caused by instructions at or after the return address may be performed before the load that obtains the return address is performed.

Because processors may implement nonarchitected duplicates of architected resources (e.g., GPRs, CR fields, and the Link Register), resource dependencies (e.g., specification of the same target register for two Load instructions) do not order storage accesses.

Examples of correct uses of dependencies, ***sync*** and ***lwsync*** to order storage accesses can be found in Appendix B. "Programming Examples for Sharing Storage" on page 913.

Because the storage model is weakly consistent, the sequential execution model as applied to instructions that cause storage accesses guarantees only that those accesses appear to be performed in program order with respect to the processor executing the instructions. For example, an instruction may complete, and subsequent instructions may be executed, before storage accesses caused by the first instruction have been performed. However, for a sequence of atomic accesses to the same storage location, if the location is in storage that is Memory Coherence Required the definition of coherence guarantees that the accesses are performed in program order with respect to any processor or mechanism that accesses the location coherently, and similarly if the location is in storage that is Caching Inhibited.

Because accesses to storage that is Caching Inhibited are performed in main storage, memory barriers and dependencies on *Load* instructions order such accesses with respect to any processor or mechanism even if the storage is not Memory Coherence Required.

Programming Note

The first example below illustrates cumulative ordering of storage accesses preceding a memory barrier, and the second illustrates cumulative ordering of storage accesses following a memory barrier. Assume that locations X, Y, and Z initially contain the value 0.

Example 1:

Processor A:
stores the value 1 to location X

Processor B:
loads from location X obtaining the value 1, executes a ***sync*** instruction, then stores the value 2 to location Y

Processor C:
loads from location Y obtaining the value 2, executes a ***sync*** instruction, then loads from location X

Example 2:

Processor A:
stores the value 1 to location X, executes a ***sync*** instruction, then stores the value 2 to location Y

Processor B:
loops loading from location Y until the value 2 is obtained, then stores the value 3 to location Z

Processor C:
loads from location Z obtaining the value 3, executes a ***sync*** instruction, then loads from location X

In both cases, cumulative ordering dictates that the value loaded from location X by processor C is 1.

1.7.2 Storage Ordering of Copy/ Paste-Initiated Data Transfers

The Copy-Paste Facility (see Section 4.4) uses pairs of instructions to initiate 128-byte data transfers. They are referred to as "data transfers" to differentiate them from the "normal" storage accesses caused by or associated with loads, stores, and instructions that are treated as loads and stores. In the absence of barriers, the relative ordering among adjacent data transfers or data transfers and storage accesses is not defined, and the sequential execution model and coherence-required ordering relationships do not apply. To establish order between adjacent data transfers or between data transfers and storage accesses, ***hwsync*** must be used. See the description of the *Synchronize* instruction in Section 4.6.3 for more information.

Programming Note

It may be helpful to think of a ***copy***/***paste.*** pair sending the real storage addresses of the 128-byte source and destination to an asynchronous data transfer engine completely separate from the processor that is executing the ***copy*** and ***paste.*** instructions. The data transfers collect in the engine's queue. The engine may perform the data transfers in any order, and with the only relative timing relationship to adjacent transfers and accesses being determined by ***hwsync***.

1.7.3 Storage Ordering of I/O Accesses

A "coherence domain" consists of all processors and all interfaces to main storage. Memory reads and writes initiated by mechanisms outside the coherence domain are performed within the coherence domain in the order in which they enter the coherence domain and are performed as coherent accesses.

1.7.4 Atomic Update

The *Load And Reserve* and *Store Conditional* instructions together permit atomic update of a shared storage location. There are byte, halfword, word, doubleword, and quadword forms of each of these instructions. Described here is the operation of the word forms ***lwarx*** and ***stwcx.***; operation of the byte, halfword, doubleword, and quadword forms ***lbarx***, ***stbcx.***, ***lharx***, ***sthcx.***, ***ldarx, stdcx.***, ***lqarx,*** and ***stqcx.*** is the same except for obvious substitutions.

The ***lwarx*** instruction is a load from a word-aligned location that has two side effects. Both of these side effects occur at the same time that the load is performed.

1. A reservation for a subsequent ***stwcx.*** instruction is created.
2. The memory coherence mechanism is notified that a reservation exists for the storage location specified by the ***lwarx***.

The ***stwcx.*** instruction is a store to a word-aligned location that is conditioned on the existence of the reservation created by the ***lwarx*** and on whether the same storage location is specified by both instructions. To emulate an atomic operation with these instructions, it is necessary that both the ***lwarx*** and the ***stwcx***. specify the same storage location.

A ***stwcx.*** performs a store to the target storage location only if the reservation created by the ***lwarx*** still exists at the time the ***stwcx***. is executed, and only if the storage locations specified by the two instructions are in the same aligned block of real storage whose size is the smallest real page size supported by the implementa-

tion. The remainder of this paragraph assumes that these two conditions are satisfied. If the storage locations specified by the two instructions differ, or if a *Store Conditional* instruction is used with a preceding *Load And Reserve* instruction that has a different storage operand length (e.g., ***stwcx***. with ***ldarx***), whether the store is performed is undefined. Otherwise the store is performed.

A ***stwcx.*** that performs its store is said to "succeed".

Examples of the use of ***lwarx*** and ***stwcx.*** are given in Appendix B. "Programming Examples for Sharing Storage" on page 913.

A successful ***stwcx.*** to a given location may complete before its store has been performed with respect to other processors and mechanisms. As a result, a subsequent load or ***lwarx*** from the given location by another processor may return a "stale" value. However, a subsequent ***lwarx*** from the given location by the other processor followed by a successful ***stwcx.*** by that processor is guaranteed to have returned the value stored by the first processor's ***stwcx***. (in the absence of other stores to the given location).

Programming Note

The store caused by a successful ***stwcx.*** is ordered, by a dependence on the reservation, with respect to the load caused by the ***lwarx*** that established the reservation, such that the two storage accesses are performed in program order with respect to any processor or mechanism.

Programming Note

If a virtual address is reassigned to a different real page, a reservation established at the virtual address before the reassignment will not be cleared by a store to the new real page by some other processor or mechanism. (As described in Section 1.7.4.1, reservations are held on real addresses.) If *Store Conditional* instructions did not suppress the store when the storage location specified by the *Store Conditional* instruction is in a different real page from the storage location specified by the corresponding *Load And Reserve* instruction, such virtual address reassignment could permit a *Store Conditional* instruction that specifies the same virtual address as the corresponding *Load And Reserve* instruction, and logically should fail because the other processor or mechanism stored to the virtual address, to succeed.

This real address checking cannot detect that the virtual page in which the reservation was established has been moved to a new real page and back again to the original real page that was accessed by the *Load And Reserve* instruction. It also cannot detect that the real address of the storage location specified by a *Store Conditional* instruction is the same as the real address of the reservation, or is in the same real page as the reservation, *only* because the virtual page containing the storage location specified by the *Store Conditional* instruction has been moved to the real page that was accessed by the corresponding *Load And Reserve* instruction. Privileged software that moves a virtual page should clear the reservation on the processor it is running on in order to ensure that a *Store Conditional* instruction executed by that processor does not succeed in these cases. (If the software that moves the virtual page uses *Load And Reserve* and *Store Conditional* for its own purposes, the clearing of the original reservation will happen naturally. The stores that occur naturally as part of moving the virtual page will cause any reservations, held by other processors, in the target real page to be cleared.)

1.7.4.1 Reservations

The ability to emulate an atomic operation using ***lwarx*** and ***stwcx.*** is based on the conditional behavior of ***stwcx.***, the reservation created by ***lwarx***, and the clearing of that reservation if the target storage location is modified by another processor or mechanism before the ***stwcx.*** performs its store.

A reservation is held on an aligned unit of real storage called a reservation granule. The size of the reservation granule is 2^n bytes, where n is implementation-dependent but is always at least 4 (thus the minimum reservation granule size is a quadword), and where 2^n is not larger than the smallest real page size

supported by the implementation. The reservation granule associated with effective address EA contains the real address to which EA maps. ("real_addr(EA)" in the RTL for the *Load And Reserve* and *Store Conditional* instructions stands for "real address to which EA maps".) The reservation also has an associated length, which is equal to the storage operand length, in bytes, of the *Load and Reserve* instruction that established the reservation.

A processor has at most one reservation at any time. A reservation is established by executing a ***lbarx***, ***lharx***, ***lwarx, ldarx***, or ***lqarx*** instruction, as described in item 1 below, and is lost or may be lost, depending on the item, if any of the following occur. Items 1-9 apply only if the relevant access is performed. (For example, an access that would ordinarily be caused by an instruction might not be performed if the instruction causes the system error handler to be invoked.)

1. The processor holding the reservation executes another ***lbarx***, ***lharx***, ***lwarx,*** or ***ldarx***: this clears the first reservation and establishes a new one.
2. The processor holding the reservation executes any ***stbcx.***, ***sthcx.***, ***stwcx., stdcx.***, or ***stqcx.***, regardless of whether the specified address matches the address specified by the ***lbarx***, ***lharx***, ***lwarx, ldarx***, or ***lqarx*** that established the reservation, and regardless of whether the storage operand lengths of the two instructions are the same.
3. The processor holding the reservation executes an AMO that updates the same reservation granule: whether the reservation is lost is undefined.
4. Any of the following occurs on the processor holding the reservation.
    a. The transaction state changes (from Non-transactional, Transactional, or Suspended state to one of the other two states; see Section 5.2, "Transactional Memory Facility States"), except in the following cases
        - If the change is from Transactional state to Suspended state, the reservation is not lost.
        - If the change is from Suspended state to Transactional state, the reservation is not lost if it was established in Transactional state.
        - If the change is caused by a ***treclaim.*** or ***trechkpt.*** instruction, whether the reservation is lost is undefined.

    b. The transaction nesting depth (see Section 5.4, "Transactional Memory Facility Registers") changes; whether the reservation is lost is undefined. (This item applies only if the processor is in Transactional state both before and after the change.)

    c. The processor is in Suspended state and executes a *Store Conditional* instruction (***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.***) or a ***waitrsv*** instruction; the reservation is lost if it was established in Transactional state. In this case the *Store Conditional* instruction's store is not performed, and the ***waitrsv*** does not wait. (For *Store Conditional*, the reservation is also lost if it was established in Suspended state; see item 2.)
5. Some other processor executes a *Store or* ***dcbz*** that specifies a location in the same reservation granule.
6. Some other processor executes a ***dcbtst***, or ***dcbt*** that specifies a location in the same reservation granule: whether the reservation is lost is undefined. (For a ***dcbtst*** instruction that specifies a data stream, "location" in the preceding sentence includes all locations in the data stream.)
7. Any processor modifies a Reference or Change bit (see Book III in the same reservation granule: whether the reservation is lost is undefined.
8. Some mechanism other than a processor modifies a storage location in the same reservation granule.
9. An interrupt (see Book III) occurs on the processor holding the reservation: the interrupt itself does not clear the reservation, but system software invoked by the interrupt may clear the reservation.
10. Implementation-specific characteristics of the coherence mechanism cause the reservation to be lost.

Virtualized Implementation Note

A reservation may be lost if:

- Software executes a privileged instruction or utilizes a privileged facility
- Software accesses storage not intended for general-purpose programming
- Software accesses a Device Control Register

Programming Note

One use of ***lwarx*** and ***stwcx.*** is to emulate a "Compare and Swap" primitive like that provided by the IBM System/370 Compare and Swap instruction; see Section B.1, "Atomic Update Primitives" on page 913. A System/370-style Compare and Swap checks only that the old and current values of the word being tested are equal, with the result that programs that use such a Compare and Swap to control a shared resource can err if the word has been modified and the old value subsequently restored. The combination of ***lwarx*** and ***stwcx.*** improves on such a Compare and Swap, because the reservation reliably binds the ***lwarx*** and ***stwcx.*** together. The reservation is always lost if the word is modified by another processor or mechanism between the ***lwarx*** and ***stwcx.***, so the ***stwcx.*** never succeeds unless the word has not been stored into (by another processor or mechanism) since the ***lwarx***.

Programming Note

In general, programming conventions must ensure that ***lwarx*** and ***stwcx.*** specify addresses that match; a ***stwcx.*** should be paired with a specific ***lwarx*** to the same storage location. Situations in which a ***stwcx.*** may erroneously be issued after some ***lwarx*** other than that with which it is intended to be paired must be scrupulously avoided. For example, there must not be a context switch in which the processor holds a reservation in behalf of the old context, and the new context resumes after a ***lwarx*** and before the paired ***stwcx.***. The ***stwcx.*** in the new context might succeed, which is not what was intended by the programmer. Such a situation must be prevented by executing a ***stbcx.***, ***sthcx***., ***stwcx.***, ***stdcx.***, or ***stqcx.*** that specifies a dummy writable aligned location as part of the context switch; see Section 6.4.3 of Book III.

Programming Note

Because the reservation is lost if another processor stores anywhere in the reservation granule, lock words (or bytes, halfwords, or doublewords) should be allocated such that few such stores occur, other than perhaps to the lock word itself. (Stores by other processors to the lock word result from contention for the lock, and are an expected consequence of using locks to control access to shared storage; stores to other locations in the reservation granule can cause needless reservation loss.) Such allocation can most easily be accomplished by allocating an entire reservation granule for the lock and wasting all but one word. Because reservation granule size is implementation-dependent, portable code must do such allocation dynamically.

Similar considerations apply to other data that are shared directly using ***lwarx*** and ***stwcx.*** (e.g., pointers in certain linked lists; see Section B.3, "List Insertion" on page 917).

1.7.4.2 Forward Progress

Forward progress in loops that use ***lwarx*** and ***stwcx.*** is achieved by a cooperative effort among hardware, system software, and application software.

The architecture guarantees that when a processor executes a ***lwarx*** to obtain a reservation for location X and then a ***stwcx.*** to store a value to location X, either

1. the ***stwcx.*** succeeds and the value is written to location X, or
2. the ***stwcx.*** fails because some other processor or mechanism modified location X, or
3. the ***stwcx.*** fails because the processor's reservation was lost for some other reason.

In Cases 1 and 2, the system as a whole makes progress in the sense that some processor successfully modifies location X. Case 3 covers reservation loss required for correct operation of the rest of the system. This includes cancellation caused by some other processor or mechanism writing elsewhere in the reservation granule, cancellation caused by the operating system in managing certain limited resources such as real storage, and cancellation caused by any of the other effects listed in see Section 1.7.4.1.

An implementation may make a forward progress guarantee, defining the conditions under which the system as a whole makes progress. Such a guarantee must specify the possible causes of reservation loss in Case 3. While the architecture alone cannot provide such a guarantee, the characteristics listed in Cases 1 and 2 are necessary conditions for any forward progress guarantee. An implementation and operating system can build on them to provide such a guarantee.

Virtualized Implementation Note

On a virtualized implementation, Case 3 includes reservation loss caused by the virtualization software. Thus, on a virtualized implementation, a reservation may be lost at any time without apparent cause. The virtualization software participates in any forward progress assurances, as described above.

Programming Note

The architecture does not include a “fairness guarantee”. In competing for a reservation, two processors can indefinitely lock out a third.

1.8 Transactions

A transaction is a group of instructions that collectively have unique storage access behavior intended to facilitate parallel programming. (It is possible to nest transactions within one another. The description in this chapter will ignore nesting because it does not have a significant impact on the properties of the memory model. Nesting and its consequences will be described elsewhere.) Sequences of instructions that are part of the transaction may be interleaved with sequences of Suspended state instructions that are not part of the transaction. A transaction is said to “succeed” or to “fail,” and failure may happen before all of the instructions in the transaction have completed. If the transaction fails, it is as if the instructions that are part of the transaction were never executed. If the transaction succeeds, it appears to execute as an atomic unit as viewed by other processors and mechanisms. (Although the transaction appears to execute atomically, some knowledge of the inner workings will be necessary to avoid apparent paradoxes in the rest of the model. These details are described below.) The execution of Suspended state sequences have the same effect that the sequence would have in the absence of a transaction, independent of the success or failure of the transaction, including accessing storage according to the weakly consistent storage model or SAO, based on storage attributes. Upon failure, normal execution continues at the failure handler. Except for the rollback of the effects of transactional instructions upon transaction failure, as viewed by the executing thread, the interleaved sequences of Transactional and Suspended state instructions appear to execute according to the sequential execution model. See Chapter 5. “Transactional Memory Facility” on page 877 for more details. The unique attributes of the storage model for transactions are described below.

Transaction processing does not support the rollback of operations on the reservation mechanism. To prevent this possibility, a reservation is lost as a result of a state change from Transactional to Non-transactional or Non-transactional to Transactional. It is possible to successfully complete an atomic update in Transactional state, though such a sequence would have no benefit. It is also possible to complete an atomic update in Suspended state, or straddling an interval in Suspended state if Suspended state is entered via an interrupt or ***tsuspend.*** and exited via ***tresume.***, ***rfebb***, ***rfid***, ***rfscv***, ***hrfid***, or ***mtmsrd***. However, an atomic update will not succeed if only one of the *Load and Reserve* / *Store Conditional* instruction pair is executed in Suspended state.

Programming Note

Note that if a *Store Conditional* instruction within a transaction does not store, it may still be possible for the transaction to succeed. Software must not depend on the two operations having the same outcome. For example, software must not use success of an enclosing transaction as a replacement for checking the condition code from a transactional *Store Conditional* instruction.

Programming Note

Accessing storage locations in Suspended state that have been accessed transactionally has the potential to create apparent storage paradoxes. Consider, for example, a case where variable X has intial value zero, is updated transactionally to one, is read in Suspended state, subsequently the transaction fails, and variable X is read again. In the absence of external conflicts, the observed sequence of values will be zero, one, zero: old, new, old.

Performing an atomic update on X in Suspended state may be even more confusing. Suppose the atomic sequence increments X, but that the only way to have X=1 is via the transactional store that occurs before entering Suspended state. The store conditional, if it succeeds, will store X=2 and in so doing, kill the transaction. But with the transaction having failed, X was never equal to one.

The flexibility of the Suspended state programming model can create unintuitive results. It must be used with care.

Successful transactions are serialized in some order, and no processor or mechanism is able to observe the accesses caused by any subset of these transactions as occurring in an order that conflicts with this order. Specifically, let processor i execute transactions 0, 1,…, j, j+1, …, where only successful transactions are numbered, and the numbering reflects program order. Let T_{ij} be transaction j on processor i. Then there is an ordering of the T_{ij} such that no processor or mechanism is able to observe the accesses caused by the transactions T_{ij} in an order that conflicts with this ordering. Note that Suspended state storage accesses are not included in the serialization property.

Programming Note

The ordering of the T_{ij} for a given i is consistent with program order for processor i.

Because of the difference between a transaction's instantaneous appearance and the finite time required to execute it in an implementation, it is exposed to changes in memory management state in a way that is not true for individual accesses. A change to the translation or protection state that would prevent any access from taking place at any time during its processing for the transaction compromises the integrity of the transaction. Any such change must either be prevented or must cause the transaction to fail. The architecture will automatically fail a transaction if the memory management state change is accomplished using ***tlbie*** or ***slbieg***. An implementation may overdetect such conflicts between the ***tlbie*** or ***slbieg*** and the transaction footprint. (Overdetection may result from the technique used to detect the conflict. A bloom filter may be used, as an example. Subsequent references to translation invalidation conflicts implicitly include any cases of spurious overdetection.) Changes made in some other manner must be managed by software, for example by explicitly terminating any affected transactions. Examples of instructions that require software management are ***tlbiel***, ***slbie***, ***slbia***, and ***slbiag***.

The atomic nature of a transaction, together with the cumulative memory barrier created by the transaction and the memory barriers created by ***tbegin.*** and ***tend.*** described below, has the potential to eliminate the need for explicit memory barriers within the transaction, and before and after the transaction as well. However, since there may be a desire to preserve existing algorithms while exploiting transactions, the interaction of memory barriers and transactions is defined. In the presence of transactions, storage access ordering is the same as if no transactions are present, with the following exceptions. Memory barriers that are created while the transaction is running (other than the integrated cumulative memory barrier of the transaction described below), data dependencies, and SAO do not order transactional stores. Instead, transactional stores are grouped together into an "aggregate store," which is performed as an atomic unit with respect to other processors and mechanisms when the transaction succeeds, after all the transactional loads have been performed. With this store behavior, the appearance of transactional atomicity is created in a manner similarly to that for a *Load and Reserve* / *Store Conditional* pair. Success of the transaction is conditional on the storage locations specified by the loads not having been stored into by a more recent Suspended state store or by any store by another processor or mechanism since the load was performed. (There are additional conditions for the success of transactions.)

A ***tbegin.*** instruction that begins a successful transaction creates a memory barrier that immediately precedes the transaction and orders storage accesses pairwise, as follows. Let A and B be sets of storage accesses as defined below. For each pair a_ib_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor or mechanism. Set A contains all data accesses caused by instructions preceding the ***tbegin.*** that are neither Write Through Required nor Caching Inhibited. Set B contains all data accesses caused by instructions following the ***tbegin.***, including Suspended state accesses, that are neither Write Through Required nor Caching Inhibited. The ordering done by this memory barrier is cumulative.

Programming Note

The reason the creation of the memory barrier by ***tbegin.*** is specified to be contingent on the transaction succeeding is that delaying the creation may improve performance, and does not seriously inconvenience software.

A successful transaction has an integrated cumulative memory barrier behavior. When a processor (P1) executes a ***tend.*** instruction and ***tend.*** processing determines that the transaction will succeed, a memory barrier is created, which orders storage accesses pairwise, as follows. Let A and B be sets of storage accesses as defined below. For each pair a_ib_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor or mechanism. Set A contains all non-transactional data accesses by other processors and mechanisms that have been performed with respect to P1 before the memory barrier is created and are neither Write Through Required nor Caching Inhibited. Set B contains the aggregate store and all non-transactional data accesses by other processors and mechanisms that are performed after a *Load* instruction executed by that processor or mechanism has returned the value stored by a store that is in set B. Note that the integrated cumulative memory barrier does not order Suspended state storage accesses interleaved with the transaction.

A ***tend.*** instruction that ends a successful transaction creates a memory barrier that immediately follows the transaction and orders storage accesses pairwise, as follows. Let A and B be sets of storage accesses as defined below. For each pair a_ib_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor or

mechanism. Set A contains all data accesses caused by instructions preceding the ***tend.,*** including Suspended state accesses, that are neither Write Through Required nor Caching Inhibited. Set B contains all data accesses caused by instructions following the ***tend.*** that are neither Write Through Required nor Caching Inhibited. The ordering done by this memory barrier is cumulative.

Programming Note

The memory barriers that are created by the execution of a successful transaction (those associated with ***tbegin.***, ***tend.***, and the integrated cumulative memomry barrier) render most explicit memory barriers in and around transactions redundant. An exception is when there is a need to establish order among Suspended state accesses.

1.8.1 Rollback-Only Transactions

A Rollback-Only Transaction (ROT) is a sequence of instructions that is executed, or not, as a unit. The purpose of the ROT is to enable bulk speculation of instructions with minimum overhead. It leverages the rollback mechanism that is invoked as part of transaction failure handling, but has reduced overhead in that it does not have the full atomic nature of the transaction and its synchronization and serialization properties. The absence of a (normal) transaction's atomic quality means that a ROT must not be used to manipulate shared data.

More specifically, a ROT differs from a normal transaction as follows.

- ROTs are not serialized.
- There are no memory barriers created by ***tbegin.*** and ***tend.***
- A ROT has no integrated cumulative memory barrier.
- There is no monitoring of storage locations specified by loads for modification by other processors and mechanisms between the performing of the loads and the completion of the ROT.
- The stores that are included in the ROT need not appear to be performed as an aggregate store. (Implementations are likely to provide an aggregate store appearance, but the correctness of the program must not depend on the aggregate store appearance.)

1.9 Instruction Storage

The instruction execution properties and requirements described in this section, including its subsections, apply only to instruction execution that is required by the sequential execution model.

In this section, including its subsections, it is assumed that all instructions for which execution is attempted are in storage that is not Caching Inhibited and (unless instruction address translation is disabled; see Book III) is not Guarded, and from which instruction fetching does not cause the system error handler to be invoked (e.g., from which instruction fetching is not prohibited by the "address translation mechanism" or the "storage protection mechanism"; see Book III).

Programming Note

The results of attempting to execute instructions from storage that does not satisfy this assumption are described in Section 1.6.2 and Section 1.6.4 of this Book and in Book III.

For each instance of executing an instruction from location X, the instruction may be fetched multiple times.

The instruction cache is not necessarily kept consistent with the data cache or with main storage. It is the responsibility of software to ensure that instruction storage is consistent with data storage when such consistency is required for program correctness.

After one or more bytes of a storage location have been modified and before an instruction located in that storage location is executed, software must execute the appropriate sequence of instructions to make instruction storage consistent with data storage. Otherwise the result of attempting to execute the instruction is boundedly undefined except as described in Section 1.9.1, "Concurrent Modification and Execution of Instructions" on page 825.

Programming Note

Following are examples of how to make instruction storage consistent with data storage. Because the optimal instruction sequence to make instruction storage consistent with data storage may vary between systems, many operating systems will provide a system service to perform this function.

Case 1: The given program does not modify instructions executed by another program nor does another program modify the instructions executed by the given program.

Assume that location X previously contained the instruction A0; the program modified one of more bytes of that location such that, in data storage, the location contains the instruction A1; and location X is wholly contained in a single cache block. The following instruction sequence will make instruction storage consistent with data storage such that if the ***isync*** was in location X-4, the instruction A1 in location X would be executed immediately after the ***isync***.

```
dcbst   X       #copy the block to main storage
sync            #order copy before invalidation
icbi    X       #invalidate copy in instr cache
isync           #discard prefetched instructions
```

Case 2: One or more programs execute the instructions that are concurrently being modified by another program.

Assume program A has modified the instruction at location X and other programs are waiting for program A to signal that the new instruction is ready to execute. The following instruction sequence will make instruction storage consistent with data storage and then set a flag to indicate to the waiting programs that the new instruction can be executed.

```
li      r0,1    #put a 1 value in r0
dcbst   X       #copy the block in main storage
sync            #order copy before invalidation
icbi    X       #invalidate copy in instr cache
sync            #order invalidation before store
                #  to flag
stw r0,flag     #set flag indicating instruction
                #  storage is now consistent
```

The following instruction sequence, executed by the waiting program, will prevent the waiting programs from executing the instruction at location X until location X in instruction storage is consistent with data storage, and then will cause any prefetched instructions to be discarded.

```
lwz     r0,flag #loop until flag = 1 (when 1 is
cmpwi   r0,1    #   loaded, location X in inst'n
bne     $-8     #   storage is consistent with
                #   location X in data storage)
isync           #discard any prefetched inst'ns
```

In the preceding instruction sequence any context synchronizing instruction (e.g., ***rfid***) can be used instead of ***isync***. (For Case 1 only ***isync*** can be used.)

For both cases, if two or more instructions in separate data cache blocks have been modified, the ***dcbst*** instruction in the examples must be replaced by a sequence of ***dcbst*** instructions such that each block containing the modified instructions is copied back to main storage. Similarly, for ***icbi*** the sequence must invalidate each instruction cache block containing a location of an instruction that was modified. The ***sync*** instruction that appears above between "***dcbst*** X" and "***icbi*** X" would be placed between the sequence of ***dcbst*** instructions and the sequence of ***icbi*** instructions.

1.9.1 Concurrent Modification and Execution of Instructions

The phrase “concurrent modification and execution of instructions” (CMODX) refers to the case in which a processor fetches and executes an instruction from instruction storage which is not consistent with data storage or which becomes inconsistent with data storage prior to the completion of its processing. This section describes the only case in which executing this instruction under these conditions produces defined results.

In the remainder of this section the following terminology is used.

- Location X is an arbitrary word-aligned storage location.
- X_0 is the value of the contents of location X for which software has made the location X in instruction storage consistent with data storage.
- X_1, X_2, ..., X_n are the sequence of the first n values occupying location X after X_0.
- X_n is the first value of X subsequent to X_0 for which software has again made instruction storage consistent with data storage.
- The “patch class” of instructions consists of the I-form *Branch* instruction (***b***[***l***][***a***]) and the preferred no-op instruction (***ori*** 0,0,0).

If the instruction from location X is executed after the copy of location X in instruction storage is made consistent for the value X_0 and before it is made consistent for the value X_n, the results of executing the instruction are defined if and only if the following conditions are satisfied.

1. The stores that place the values X_1, ..., X_n into location X are atomic stores that modify all four bytes of location X.
2. Each X_i, $0 \le i \le n$, is a patch class instruction.
3. Location X is in storage that is Memory Coherence Required.

If these conditions are satisfied, the result of each execution of an instruction from location X will be the execution of some X_i, $0 \le i \le n$. The value of the ordinate i associated with each value executed may be different and the sequence of ordinates i associated with a sequence of values executed is not constrained, (e.g., a valid sequence of executions of the instruction at location X could be the sequence X_i, X_{i+2}, then X_{i-1}). If these conditions are not satisfied, the results of each such execution of an instruction from location X are boundedly undefined, and may include causing inconsistent information to be presented to the system error handler.

Programming Note

An example of how failure to satisfy the requirements given above can cause inconsistent information to be presented to the system error handler is as follows. If the value X_0 (an illegal instruction) is executed, causing the system illegal instruction handler to be invoked, and before the error handler can load X_0 into a register, X_0 is replaced with X_1, an *Add Immediate* instruction, it will appear that a legal instruction caused an illegal instruction exception.

Programming Note

It is possible to apply a patch or to instrument a given program without the need to suspend or halt the program. This can be accomplished by modifying the example shown in the Programming Note at the end of Section 1.9 where one program is creating instructions to be executed by one or more other programs.

In place of the *Store* to a flag to indicate to the other programs that the code is ready to be executed, the program that is applying the patch would replace a patch class instruction in the original program with a *Branch* instruction that would cause any program executing the *Branch* to branch to the newly created code. The first instruction in the newly created code must be an ***isync***, which will cause any prefetched instructions to be discarded, ensuring that the execution is consistent with the newly created code. The instruction storage location containing the ***isync*** instruction in the patch area must be consistent with data storage with respect to the processor that will execute the patched code before the *Store* which stores the new *Branch* instruction is performed.

Programming Note

It is believed that all processors that comply with versions of the architecture that precede Version 2.01 support concurrent modification and execution of instructions as described in this section if the requirements given above are satisfied, and that most such processors yield boundedly undefined results if the requirements given above are not satisfied. However, in general such support has not been verified by processor testing. Also, one such processor is known to yield undefined results in certain cases if the requirements given above are not satisfied.

Chapter 2. Performance Considerations and Instruction Restart

2.1 Performance-Optimized Instruction Sequences

Performance-optimized instruction sequences are instruction sequences that provide better performance than other ways of achieving the same results. The supported performance-optimized sequences are shown in the following sections. In order to achieve the improved performance, the sequences must be coded exactly as shown, including instruction order, register re-use, and lack of intervening instructions. The processor achieves the improved performance by executing the sequence as a single operation, or in some other highly efficient, sequence-specific, manner. (The improved performance may not be obtained if the sequence causes the system error handler to be invoked, or for implementation-dependent reasons.)

2.1.1 Load and Store Operations

The following instruction sequences will optimize performance for storage accesses to effective addresses that are offset from (RA) by magnitudes of up to 2^{32}.

Operation	Load Instruction Sequence		Store Instruction Sequence	
Fixed-point byte accesses	addis lbz	Rx,RA,SI Rt,D(Rx)	addis stb	Rx,RA,SI_h RS,D(Rx)
Fixed-point halfword accesses	addis lhz	Rx,RA,SI_h Rt,D(Rx)	addis sth	Rx,RA,SI_h RS,D(Rx)
Fixed-point word accesses	addis lwz	Rx,RA,SI_h Rt,D(Rx)	addis stw	Rx,RA,SI_h RS,D(Rx)
Fixed-point doubleword accesses	addis ld	Rx,RA,SI_h Rt,D(Rx)	addis std	Rx,RA,SI_h RS,D(Rx)
Floating-point single-precision accesses	addis lfs	Rx,RA,SI_h FRT,D(Rx)	addis stfs	Rx,RA,SI_h FRS,D(Rx)
Floating-point double-precision accesses	addis lfd	Rx,RA,SI_h FRT,D(Rx)	addis stfd	Rx,RA,SI_h FRS,D(Rx)
VSX Scalar doubleword accesses	addis lxsd	Rx,RA,SI_h XT,DS(Rx)	addis stxsd	Rx,RA,SI_h XS,DS(Rx)
VSX Scalar single-precision accesses	addis lxssp	Rx,RA,SI_h XT,DS(Rx)	addis stxssp	Rx,RA,SI_h XS,DS(Rx)
VSX Vector accesses	addis lxv	Rx,RA,SI_h XT,DQ(Rx)	addis stxv	Rx,RA,SI_h XS,DQ(Rx)

Table 1: Loads and Stores with offsets of up to 2^{32} offsets from base register

The following instruction sequences will optimize performance for storage accesses to effective addresses that are offset from (RB) by magnitudes of up to 2^{16}.

Operation	Load Istruction Sequence	Store Instruction Sequence
Fixed-point doubleword accesses	addi Rx,0,SI ldx Rt,RA,Rx	addi Rx,0,SI stdx RS,RA,Rx
Floating-point as integer word accesses	addi Rx,0,SI lfiwzx FRT,RA,Rx	addi Rx,0,SI stfiwx FRS,RA,Rx
Vector byte accesses	addi Rx,0,SI lvebx VRT,RA,Rx	addi Rx,0,SI stvebx VRS,RA,Rx
Vector halfword accesses	addi Rx,0,SI lvehx VRT,RA,Rx	addi Rx,0,SI stvehx VRS,RA,Rx
Vector word accesses	addi Rx,0,SI lvewx VRT,RA,Rx	addi Rx,0,SI stvewx VRS,RA,Rx
Vector accesses	addi Rx,0,SI lvx VRT,RA,Rx	addi Rx,0,SI stvx VRS,RA,Rx
VSX Vector accesses	addi Rx,0,SI lxvx XT,RA,Rx	addi Rx,0,SI stxvx XS,RA,Rx
VSX Vector doubleword accesses	addi Rx,0,SI lxvd2x XT,RA,Rx	addi Rx,0,SI stxvd2x XS,RA,Rx
VSX Vector word accesses	addi Rx,0,SI lxvw4x XT,RA,Rx	addi Rx,0,SI stxvw4x XS,RA,Rx
VSX Vector halfword accesses	addi Rx,0,SI lxvh8x XT,RA,Rx	addi Rx,0,SI stxvh8x XS,RA,Rx
VSX Vector byte accesses	addi Rx,0,SI lxvb16x XT,RA,Rx	addi Rx,0,SI stxvb16x XS,RA,Rx
VSX Vector word splat accesses	addi Rx,0,SI lxvwsx XT,RA,Rx	n/a
VSX Vector doubleword splat accesses	addi Rx,0,SI lxvdsx XT,RA,Rx	n/a
VSX Scalar doubleword accesses	addi Rx,0,SI lxsdx XT,RA,Rx	addi Rx,0,SI stxsdx XS,RA,Rx
VSX Scalar single-precision accesses	addi Rx,0,SI lxsspx XT,RA,Rx	addi Rx,0,SI stxsspx XS,RA,Rx
VSX Scalar byte accesses	addi Rx,0,SI lxsibzx XT,RA,Rx	addi Rx,0,SI stxsibx XS,RA,Rx
VSX Scalar halfword accesses	addi Rx,0,SI lxsihzx XT,RA,Rx	addi Rx,0,SI stxsihx XS,RA,Rx
VSX Scalar word accesses	addi Rx,0,SI lxsiwzx XT,RA,Rx	addi Rx,0,SI stxsiwx XS,RA,Rx

Table 2: Loads and Stores with Offsets from (RA) by Magnitudes of Up to 2^{16}.

Programming Note

Even independent of the performance optimization described above, the techniques illustrated in Table 1 and Table 2 generally perform better than other ways of achieving the effect of having a large displacement field for D-form and DS-form fixed-point *Load/Store* instructions (Table 1), and of having a displacement field for X-form Vector and VSX *Load/Store* instructions (Table 2).

The technique for the fixed-point *Load/Store* instructions is complicated by the fact that D-form and DS-form *Loads* and *Stores* treat the D/DS value as signed.

For simplicity, most of this Note assumes that the fixed-point *Load/Store* instruction is D-form; the modifications for DS-form fixed-point *Load/Store* instructions are straightforward.

Let the desired effective address to load from or store to be (RA) + DISP, where DISP is a signed 32-bit value.

$$(RA) + DISP = (RA) + DISP_{0:15} \,||\, DISP_{16:31}$$
$$= (RA) + (DISP_{0:15} \,||\, 0x0000) + DISP_{16:31}$$

where $DISP_{0:15}$ is a signed 16-bit value.

If $DISP_{0:15}$ is used as the SI value for the ***addis***, the ***addis*** forms the sum

$$(RA) + (DISP_{0:15} \,||\, 0x0000)$$

and places the result into Rx.

If $DISP_{16:31}$ is used as the D value for the *Load* or *Store* and Rx is used as the base register for the *Load* or *Store*, and $DISP_{16} = 0$, the *Load* or *Store* computes the EA to load from as

$$(Rx) + DISP_{16:31} = (RA) + (DISP_{0:15} \,||\, 0x0000) + DISP_{16:31}$$
$$= (RA) + DISP$$

However, because D-form *Loads* and *Stores* treat the D value as signed, if $DISP_{16} = 1$ the *Load* or *Store* computes the EA as

$$(Rx) + DISP_{16:31} = (RA) + (DISP_{0:15} \,||\, 0x0000) + DISP_{16:31} + 0xFFFF_FFFF_FFFF_0000$$
$$= (RA) + (DISP_{0:15} \,||\, 0x0000) + DISP_{16:31} - 2^{16}$$
$$= (RA) + DISP - 2^{16}$$

To compensate for this effective subtraction of 2^{16}, if $DISP_{16} = 1$ the SI value used for the ***addis*** must be $DISP_{0:15} + 1$. Then the ***addis*** sets Rx to

$$(RA) + ((DISP_{0:15} + 1) \,||\, 0x0000) = (RA) + (DISP_{0:15} \,||\, 0x0000) + 2^{16}$$

and the *Load* or *Store* computes the EA as

$$(Rx) + DISP_{16:31} = (RA) + (DISP_{0:15} \,||\, 0x0000) + 2^{16} + DISP_{16:31} - 2^{16}$$
$$= (RA) + DISP$$

as desired.

Thus the rules for using the technique illustrated in Table 1 are as follows.

- For the RA field of the ***addis***, use the desired base register for the *Load* or *Store.*
- For the D field of the *Load* or *Store*, use $DISP_{16:31}$.
  (For DS-form *Loads* and *Stores*, for the DS field use $DISP_{16:29}$; $DISP_{30:31}$ are 0b00.)
- For the SI field of the ***addis***:
  - if $DISP_{16} = 0$ use $DISP_{0:15}$;
  - if $DISP_{16} = 1$ use $DISP_{0:15} + 1$.

2.1.2 32-Bit Constant Generation

The following instruction sequences will optimize performance when generating zero-extended 32-bit unsigned constants (when $RA_{0:63}$ equal 0) and when performing 32-bit logical operations on $RA_{32:63}$).

Operation	Instruction Sequence
Unsigned constant (UI_h,UI_l zero extended)	oris Rx,RA,UI_h ori Rt,Rx,UI_l
Unsigned constant (UI_h,UI_l zero extended)	xoris Rx,RA,UI_h xori Rt,Rx,UI_l

Table 3: 32-bit Unsigned Constant Generation

The following instruction sequences will optimize performance when generating 32-bit signed constants.

Operation	Instruction Sequence
Signed consant (SI_h,SI_l sign extended)	addis Rx,RA,SI_h addi Rt,Rx,SI_l
Signed consant (SI_h sign extended; UI zero extended)	addis Rx,0,SI_h ori Rt,Rx,UI_l

Table 4: 32-bit Signed Constant Generation

2.1.3 Sign and Zero Extension

The following instruction sequences will optimize performance when converting 32-bit signed constants into 64-bit signed constants or performing other operations that require the result of an arithmetic operation to be sign extended.

Instruction Sequence
add Rx,RA,RB extsw[.] Rt,Rx
addi Rx,RA,SI extsw[.] Rt,Rx
addis Rx,RA,SI extsw[.] Rt,Rx
subf Rx,RA,RB extsw[.] Rt,Rx
neg Rx,RA extsw[.] Rt,Rx

Table 5: 32-bit Sign Extended Addition

The following instruction sequence will optimize performance when zero-extending the result of a 32-bit addition.

Operation	Instruction Sequence
Unsigned constant (RA + RB zero extended)	add Rx,RA,RB rldicl Rt,Rx,0,32

Table 6: 32-bit Zero-Extended addition

2.1.4 Load/Store Addressing Relative to Program Counter

The following instruction sequences will optimize performance for storage accesses to effective addresses that are offset from the CIA by magnitudes of up to 2^{32}.

Operation	Load Instruction Sequence	Store Instruction Sequence
Fixed-point byte accesses	addpcis Rx,SI_h lbz Rt,D(Rx)	addpcis Rx,SI_h stb RS,D(Rx)
Fixed-point halfword accesses	addpcis Rx,SI_h lhz Rt,D(Rx)	addpcis Rx,SI_h sth RS,D(Rx)
Fixed-point word accesses	addpcis Rx,SI_h lwz Rt,D(Rx)	addpcis Rx,SI_h stw RS,D(Rx)
Fixed-point doubleword accesses	addpcis Rx,SI_h ld Rt,D(Rx)	addpcis Rx,SI_h std RS,D(Rx)
Fixed-point doubleword accesses	addpcis Rx,SI_h ldx Rt,D(Rx)	addpcis Rx,SI_h stdx RS,D(Rx)
Floating-point single-precision accesses	addpcis Rx,SI_h lfs FRT,D(Rx)	addpcis Rx,SI_h stfs FRS,D(Rx)
Floating-point double-precision accesses	addpcis Rx,SI_h lfd FRT,D(Rx)	addpcis Rx,SI_h stfd FRS,D(Rx)
VSX Scalar doubleword accesses	addpcis Rx,SI_h lxsd VRT,DS(Rx)	addpcis Rx,SI_h stxsd VRS,DS(Rx)
VSX Scalar single-precision accesses	addpcis Rx,SI_h lxssp VRT,DS(Rx)	addpcis Rx,SI_h stxssp VRS,DS(Rx)
VSX Vector accesses	addpcis Rx,SI_h lxv XT,DQ(Rx)	addpcis Rx,SI_h stxv XS,DQ(Rx)

Table 7: Fixed-Point, Floating-Point and VSX ***Load/Store*** Fusion with offset up to 2^{32} from Program Counter

Programming Note

See the Programming Notes for Table 1.

2.1.5 Destructive Operation Operand Preservation

A destructive operation is an operation that modifies one of its inputs. The *VSX Vector Permute* and *VSX Vector Multiply-Add* instructions are destructive operations because they use their destination register as a source register.

When there is a need to preserve the contents of the overwritten source register for the various *VSX Vector Permute* and *VSX Vector Multiply-Add* instructions, performance will be optimized if the ***xxlor*** instruction is used to copy the contents of the source operand into another register, and then that register is used as the destination (and source) register for the *VSX Vector Permute* or *VSX Vector Multiply-Add* instruction.

As an example, to preserve the XT source register in the ***xxperm*** instruction, the following sequence will optimize performance.

```
xxlor  XT,XC,XC     /* Copy (XC) to XT
xxperm XT,XA,XB     /* Permute, overwriting XT
```

The set of instructions listed below, when immediately preceded by the ***xxlor*** XT,XC,XC instruction in a sequence similar to the above example, will provide optimal performance.

Mnemonic		Instruction Name
xxperm	XT,XA,XB	VSX Vector Permute
xxpermr	XT,XA,XB	VSX Vector Permute Right Indexed
xsmaddasp	XT,XA,XB	VSX Scalar Multiply-Add Type-A Single-Precision
xsmsubasp	XT,XA,XB	VSX Scalar Multiply-Subtract Type-A Single-Precision
xsnmaddasp	XT,XA,XB	VSX Scalar Negative Multiply-Add Type-A Single-Precision
xsnmsubasp	XT,XA,XB	VSX Scalar Negative Multiply-Subtract Type-A Single-Precision
xsmaddadp	XT,XA,XB	VSX Scalar Multiply-Add Type-A Double-Precision
xsmsubadp	XT,XA,XB	VSX Scalar Multiply-Subtract Type-A Double-Precision
xsnmaddadp	XT,XA,XB	VSX Scalar Negative Multiply-Add Type-A Double-Precision
xsnmsubadp	XT,XA,XB	VSX Scalar Negative Multiply-Subtract Type-A Double-Precision
xsmaddqp[o]	XT,XA,XB	VSX Scalar Multiply-Add Quad-Precision [using round to Odd]
xsmsubqp[o]	XT,XA,XB	VSX Scalar Multiply-Subtract Quad-Precision [using round to Odd]
xsnmaddqp[o]	XT,XA,XB	VSX Scalar Negative Multiply-Add Quad-Precision [using round to Odd]
xsnmsubqp[o]	XT,XA,XB	VSX Scalar Negative Multiply-Subtract Quad-Precision [using round to Odd]
xvmaddasp	XT,XA,XB	VSX Vector Multiply-Add Type-A Single-Precision
xvmsubasp	XT,XA,XB	VSX Vector Multiply-Subtract Type-A Single-Precision
xvnmaddasp	XT,XA,XB	VSX Vector Negative Multiply-Add Type-A Single-Precision
xvnmsubasp	XT,XA,XB	VSX Vector Negative Multiply-Subtract Type-A Single-Precision
xvmaddadp	XT,XA,XB	VSX Vector Multiply-Add Type-A Double-Precision
xvmsubadp	XT,XA,XB	VSX Vector Multiply-Subtract Type-A Double-Precision
xvnmaddadp	XT,XA,XB	VSX Vector Negative Multiply-Add Type-A Double-Precision
xvnmsubadp	XT,XA,XB	VSX Vector Negative Multiply-Subtract Type-A Double-Precision

Table 8. VSX Multiply-Add Arithmetic Instructions Providing Optimal Performance When Preceded by *xxlor*

Programming Note

Table 8 includes only the Type-A *Multiply-Add* instructions because supporting only one of the two types (i.e. either Type-A or Type-M) is sufficient to preserve the contents of the destination operand of the permute or *Multiply-Add* instruction. The ***xxlor*** instruction "preserves" the contents of the destination operand by copying it into another register, and the copy is then used as the destination operand of the *Multiply-Add* instruction, which is overwritten upon execution.

2.2 Instruction Restart

In this section, "*Load* instruction" includes the *Cache Management* and other instructions that are stated in the instruction descriptions to be "treated as a *Load*", and similarly for "*Store* instruction".

The following instructions are never restarted after having accessed any portion of the storage operand (unless the instruction causes a "Data Address Watchpoint match", for which the corresponding rules are given in Book III).

1. A *Store* instruction that causes an atomic access
2. A *Load* instructionthat causes an atomic access to storage that is both Caching Inhibited and Guarded

Any other *Load* or *Store* instruction may be partially executed and then aborted after having accessed a portion of the storage operand, and then re-executed (i.e., restarted, by the processor or the operating system). If an instruction is partially executed, the contents of registers are preserved to the extent that the correct result will be produced when the instruction is re-executed. Additional restrictions on the partial execution of instructions are described in Section 6.6 of Book III.

Programming Note

In order to ensure that the contents of registers are preserved to the extent that a partially executed instruction can be re-executed correctly, the registers that are preserved must satisfy the following conditions. For any given instruction, zero or more of the conditions applies.

- For a fixed-point *Load* instruction that is not a multiple or string form, if RT=RA or RT=RB then the contents of register RT are not altered.
- For an update form *Load* or *Store* instruction, the contents of register RA are not altered.

Programming Note

There are many events that might cause a *Load* or *Store* instruction to be restarted. For example, a hardware error may cause execution of the instruction to be aborted after part of the access has been performed, and the recovery operation could then cause the aborted instruction to be re-executed.

When an instruction is aborted after being partially executed, the contents of the instruction pointer indicate that the instruction has not been executed, however, the contents of some registers may have been altered and some bytes within the storage operand may have been accessed. The following are examples of an instruction being partially executed and altering the program state even though it appears that the instruction has not been executed.

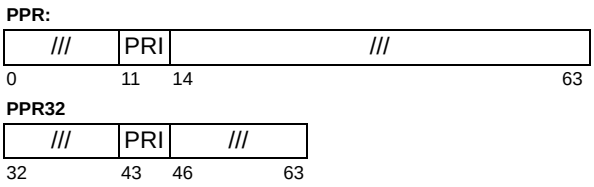
1. *Load Multiple*, *Load String*: Some registers in the range of registers to be loaded may have been altered.
2. Any *Store* instruction, ***dcbz***: Some bytes of the storage operand may have been altered.

Chapter 3. Management of Shared Resources

The facilities described in this section provide the means to control the use of resources that are shared with other processors.

3.1 Program Priority Registers

The Program Priority Register (PPR) is a 64-bit register that controls the program's priority. The PPR provides access to the full 64-bit PPR, and the Program Priority Register 32-bit (PPR32) provides access to the upper 32 bits of the PPR. The layouts of the PPR and PPR32 are shown in Figure 1.



Bit(s) Description

11:13 ***Program Priority (PRI) ($PPR32_{43:45}$)***

001 very low
010 low
011 medium low
100 medium
101 medium high

Programs can always set the PRI field to very low, low, medium low, and medium priorities; programs may be allowed to set the PRI field to medium high priority during certain time intervals. (See Section 4.3.8.) If the program priority is medium high when the time interval expires or if an attempt is made to set the priority to medium high when it is not allowed, the PRI field is set to medium.

If other values are written to this field, the PRI field is not changed. (See Section 4.3.7 of Book III for additional information.)

All other fields are reserved.

Figure 1. Program Priority Register

Programming Note

The ability to access the low-order half of the PPR (and thus the use of ***mfppr*** and ***mtppr***) might be phased out in a future version of the architecture.

Programming Note

By setting the PRI field, a programmer may be able to improve system throughput by causing system resources to be used more efficiently.

E.g., if a program is waiting on a lock (see Section B.2), it could set low priority, with the result that more processor resources would be diverted to the program that holds the lock. This diversion of resources may enable the lock-holding program to complete the operation under the lock more quickly, and then relinquish the lock to the waiting program.

Programming Note

or Rx,Rx,Rx can be used to modify the PRI field; see Section 3.2.

Programming Note

When the system error handler is invoked, the PRI field may be set to an undefined value.

3.2 “or” Instruction

Setting the PPR

The *or Rx,Rx,Rx* (see Book I) instruction can be used to set PPR_{PRI} as shown in Table 9. *or. Rx,Rx,Rx* does not set PPR_{PRI}.

Rx	PPR_{PRI}	Priority
31	001	very low
1	010	low
6	011	medium low
2	100	medium
5	101	medium high

Table 9: Priority levels for *or Rx,Rx,Rx*

Programs can always set the PRI field to very low, low, medium low, and medium priorities; programs may be allowed to set the PRI field to medium high priority during certain time intervals. (See Section 4.3.8 of Book III.) If the program priority is medium high when the time interval expires or if an attempt is made to set the priority to medium high when it is not allowed, the PRI field is set to medium.

I

Programming Note

Warning: Other forms of *or Rx,Rx,Rx* that are not described in this section and in Section 4.3.3 may also cause program priority to change. Use of these forms should be avoided except when software explicitly intends to alter program priority. If a no-op is needed, the preferred no-op (*ori 0,0,0*) should be used.

Chapter 4. Storage Control Instructions

4.1 Parameters Useful to Application Programs

It is suggested that the operating system provide a service that allows an application program to obtain the following information.

1. The virtual page sizes
2. Coherence block size
3. Reservation granule size
4. An indication of the cache model implemented (e.g., Harvard-style cache, combined cache)
5. Instruction cache size
6. Data cache size
7. Instruction cache block size
8. Data cache block size
9. Instruction cache associativity
10. Data cache associativity
11. Number of stream IDs supported for the stream variant of ***dcbt***
12. Factors for converting the Time Base to seconds
13. Maximum transaction level

If the caches are combined, the same value should be given for an instruction cache attribute and the corresponding data cache attribute.

4.2 Data Stream Control Register (DSCR)

The layout of the Data Stream Control Register (DSCR) is shown in Figure 2 below.

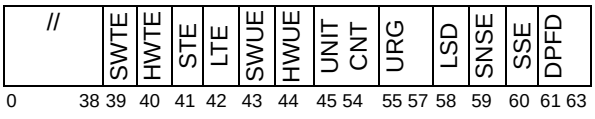


Figure 2. Data Stream Control Register

Bit(s) Description

39 ***Software Transient Enable*** (SWTE)

0 SWTE is disabled.

1 Applies the transient attribute to software-defined streams.

40 ***Hardware Transient Enable*** (HWTE)

0 HWTE is disabled.

1 Applies the transient attribute to hardware-detected streams.

41 ***Store Transient Enable*** (STE)

0 STE is disabled.

1 Applies the transient attribute to store streams.

42 ***Load Transient Enable*** (LTE)

0 LTE is disabled.

1 Applies the transient attribute to load streams.

43 ***Software Unit count Enable*** (SWUE)

0 SWUE is disabled.

1 Applies the unit count to software-defined streams.

44 ***Hardware Unit count Enable*** (HWUE)

0 HWUE is disabled.

1 Applies the unit count to hardware-detected streams.

45:54 ***Unit Count*** (UNITCNT)

Number of units in data stream.

55:57 ***Depth Attainment Urgency*** (URG)

This field indicates how quickly the prefetch depth should be reached for hardware-detected streams. Values and their meanings are as follows.

0 default
1 not urgent
2 least urgent
3 less urgent
4 medium
5 urgent
6 more urgent
7 most urgent

58 ***Load Stream Disable*** (LSD)

0 No effect.

1 Disables hardware detection and initiation of load streams.

59 ***Stride-N Stream Enable*** (SNSE)

0 No effect.
1 Enables the hardware detection and initiation of load and store streams that have a stride greater than a single cache block. Such load streams are detected only when LSD is also zero. Such store streams are detected only when SSE is also one.

60 ***Store Stream Enable*** (SSE)

0 No effect.
1 Enables hardware detection and initiation of store streams.

61:63 ***Default Prefetch Depth*** (DPFD)

This field supplies a prefetch depth for hardware-detected streams and for software-defined streams for which a depth of zero is specified or for which ***dcbt/dcbtst*** with TH=1010 is *not* used in their description. Values and their meanings are as follows.

0 default ($LPCR_{DPFD}$)
1 none
2 shallowest
3 shallow
4 medium
5 deep
6 deeper
7 deepest

The contents of the DSCR affect how a processor handles hardware-detected and software-defined data streams. The DSCR provides the only means by which software can control or supply information for hardware-detected data streams. The DPFD, UNITCNT, and transient fields may also be used instead of the TH=01010 variant of ***dcbt*** for software-defined data streams, especially when multiple streams have these attributes in common. See Section 4.3.2, “Data Cache Instructions” on page 841, for information on streams and how software may specify them.

Programming Note

The URG, LSD, SNSE and SSE fields do not affect the initiation of streams specified using the ***dcbt*** and ***dcbtst*** instructions.

Note that even when SNSE is not set, hardware may detect Stride-N streams in intervals when they access elements that map to sequential cache blocks.

Programming Note

In order for the DSCR to apply the transient attribute to streams, at least two of the four enable bits must be set: one to choose a type of access (load or store), and one to choose a kind of prefetching (software-defined or hardware-detected).

Programming Note

The purpose of Depth Attainment Urgency is to regulate the rate of prefetch generation from the cycle at which the hardware first detects an incipient stream until the cycle when the prefetch Depth is reached. A more urgent setting will benefit applications that are dominated by short to medium length streams, because otherwise prefetching does not occur rapidly enough to benefit them. In contrast, applications that frequently cause unproductive prefetches due to stream mispredicts will benefit from a less urgent setting.

Unlike the Depth, the Depth Attainment Urgency applies only to hardware-detected streams. Furthermore, the DSCR provides the only point of control for this parameter. Software-defined streams are assumed not to have the correctness risk associated with hardware streams, and therefore are set to reach their depth relatively quickly.

Programming Note

In versions of the architecture that precede Version 2.07, ***mtspr*** specifying the DSCR caused all active and nascent data streams to cease to exist. In those versions of the architecture, the DSCR was used as an overall control mechanism to specify a single global profile for all streams. Beginning with Version 2.07, the DSCR is intended to control and accelerate the creation of new streams without disturbing existing streams.

4.3 Cache Management Instructions

The *Cache Management* instructions obey the sequential execution model except as described in Section 4.3.1.

In the instruction descriptions the statements “this instruction is treated as a *Load*” and “this instruction is treated as a *Store*” mean that the instruction is treated as a *Load* (*Store*) from (to) the addressed byte with respect to address translation, the definition of program order on page 809, storage protection, reference and change recording, the storage access ordering described in Section 1.7.1, and Performance Monitor events (see Section 9.4.5 of Book III).

Programming Note

Accesses that are caused by or associated with *Cache Management* instructions that are “treated as a *Load*” or “treated as a *Store*” are not subject to the special ordering rules described for SAO storage. These accesses are always performed in accordance with the weakly consistent storage model.

Some *Cache Management* instructions contain a CT field that is used to specify a cache level within a cache hierarchy or a portion of a cache structure to which the instruction is to be applied. The correspondence between the CT value specified and the cache level is shown below.

CT Field Value	Cache Level
0	Primary Cache
2	Secondary Cache

CT values not shown above may be used to specify implementation-dependent cache levels or implementation-dependent portions of a cache structure.

4.3.1 Instruction Cache Instructions

Instruction Cache Block Invalidate X-form

icbi RA,RB

31	///	RA	RB	982	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the instruction cache of any processors, the block is invalidated in those instruction caches.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the instruction cache of this processor, the block is invalidated in that instruction cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a *Load* (see Section 4.3), except that reference and change recording need not be done.

Special Registers Altered:

None

Programming Note

Because the instruction is treated as a *Load*, the effective address is translated using translation resources that are used for data accesses, even though the block being invalidated was copied into the instruction cache based on translation resources used for instruction fetches (see Book III).

Programming Note

The invalidation of the specified block need not have been performed with respect to the processor executing the ***icbi*** instruction until a subsequent ***isync*** instruction has been executed by that processor. No other instruction or event has the corresponding effect.

Instruction Cache Block Touch X-form

icbt CT, RA, RB

31	/	CT	RA	RB	22	/
0	6	7	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

The ***icbt*** instruction provides a hint that the program will probably soon execute code from the block containing the byte addressed by EA, and that the block containing the byte addressed by EA is to be loaded into the cache specified by the CT field. (See Section 4.3 of Book II.) If the CT field is set to a value not supported by the implementation, no operation is performed.

The hint is ignored if the block is Caching Inhibited.

This instruction treated as a *Load* (see Section 4.3), except that the system data storage error handler is not invoked, and reference and change recording need not be done.

Special Registers Altered:

None

4.3.2 Data Cache Instructions

The Data Cache instructions control various aspects of the data cache.

TH field in the *dcbt* and *dcbtst* instructions

Described below are the TH field values for the ***dcbt*** and ***dcbtst*** instructions. For all TH field values which are not listed, the hint provided by the instruction is undefined.

TH=0b00000

If TH=0b00000, the ***dcbt/dcbtst*** instruction provides a hint that the program will probably soon access the block containing the byte addressed by EA.

TH=0b01000 - 0b01111

The ***dcbt/dcbtst*** instructions provide hints regarding a sequence of accesses to data elements, or indicate the expected use thereof. Such a sequence is called a “data stream”, and a ***dcbt/dcbtst*** instruction in which TH is set to one of these values is said to be a “data stream variant” of ***dcbt/dcbtst***. In the remainder of this section, “data stream” may be abbreviated to “stream”.

A data stream to which a program may perform *Load* accesses is said to be a “load data stream”, and is described using the data stream variants of the ***dcbt*** instruction. A data stream to which a program may perform *Store* accesses is said to be a “store data stream”, and is described using the data stream variants of the ***dcbtst*** instruction.

When, and how often, effective addresses for a data stream are translated is implementation-dependent.

Each data element is associated with a *unit* of storage, which is the aligned 128-byte location in storage that contains the first byte of the element. The data stream variants may be used to specify the address of the beginning of the data stream, the displacement (stride) between the first byte of successive elements, and the number of unique units of storage that are associated with all of the data elements. If the stride is specified, both the stride and the address of the first element are specified at 4 byte granularity. If the stride is not specified, the address of the first element is the address of the first unit.

Programming Note

The architecture does not provide a way to specify the size of the data elements that compose a stream. An implementation may assume some fixed size for all data elements. As a result, depending on the offset, stride, and size (and in particular whether the elements are aligned), the implementation may reduce the latency for accessing only a portion of some of the elements. A future version of the architecture may enable the specification of element size to avoid this limitation.

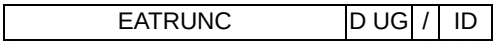
Each such data stream is associated, by software, with a stream ID, which is a resource that the processor uses to distinguish the data stream from other such data streams. The number of stream IDs is an implementation-dependent value in the range 1:16. Stream IDs are numbered sequentially starting from 0.

The encodings of the TH field and of the corresponding EA values are as follows. In the EA layout diagrams, fields shown as "/"s are reserved. These reserved fields are treated in the same manner as the corresponding case for instruction fields (see Section 1.3.3 of Book I). If a reserved value is specified for a defined EA field, or if a TH value is specified that is not explicitly defined below, the hint provided by the instruction is undefined.

TH **Description**

01000 The ***dcbt/dcbtst*** instruction provides a hint that describes certain attributes of a data stream, and may indicate that the program will probably soon access the stream.

The EA is interpreted as follows.



Bit(s) Description

0:56 ***EATRUNC***

High-order 57 bits of the effective address of the first element of the data stream. (i.e., the effective address of the first unit of the stream is EATRUNC || $^{7}0$)

57 ***Direction*** (D)

- 0 Subsequent elements have increasing addresses.
- 1 Subsequent elements have decreasing addresses.

58 ***Unlimited/GO*** (UG)

0 No information is provided by the UG field.
1 The number of elements in the data stream is unlimited, the elements are adjacent to each other, the program's need for each element of the stream is not likely to be transient, and the program will probably soon access the stream.

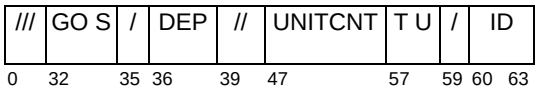
59 Reserved

60:63 ***Stream ID*** (ID)

Stream ID to use for this data stream.

01010 The ***dcbt/dcbtst*** instruction provides a hint that describes certain attributes of a data stream, or indicates that the program will probably soon access data streams that have been described using data stream variants of the ***dcbt/dcbtst*** instruction, or will probably no longer access such data streams.

The EA is interpreted as follows. If GO=1 and S≠0b00 the hint provided by the instruction is undefined; the remainder of this instruction description assumes that this combination is not used.



Bit(s) Description

0:31 Reserved

32 ***GO***

0 No information is provided by the GO field.
1 For ***dcbt***, the program will probably soon access all nascent load and store data streams that have been completely described, and will probably no longer access all other nascent load and store data streams. All other fields of the EA are ignored. ("Nascent" and "completely described" are defined below.) For ***dcbtst***, this field value holds no meaning and is treated as though it were zero.

33:34 ***Stop*** (S)

00 No information is provided by the S field.
01 Reserved
10 The program will probably no longer access the data stream (if any) associated with the specified stream ID. (All other fields of the EA except the ID field are ignored.)
11 For ***dcbt***, the program will probably no longer access the load and store data streams associated with all stream IDs. (All other fields of the EA are ignored.) For ***dcbtst***, this field value holds no meaning, and is treated as though it were 0b00.

35 Reserved

36:38 ***Depth*** (DEP)

The DEP field provides a relative estimate of how many elements ahead of the point of stream use the latency-reducing actions should go. This value reflects a comparison of the rate of consumption of the elements of the data stream and the latency to bring an arbitrary element of the stream into cache. The values are as follows.

0 default = $DSCR_{DPFD}$
1 none
2 shallowest
3 shallow
4 medium
5 deep
6 deeper
7 deepest

39:46 Reserved

47:56 ***UNITCNT***

Number of units in data stream.

57 ***Transient*** (T)

If T=1, the program's need for each element of the data stream is likely to be transient (i.e., the time interval during which the program accesses the element is likely to be short).

58 ***Unlimited*** (U)

If U=1, the number of units in the data stream is unlimited (and the UNITCNT field is ignored).

59 Reserved

60:63 ***Stream ID*** (ID)

Stream ID to use for this data stream (GO=0 and S=0b00), or stream ID associated with the data stream which the program will probably no longer access(S=0b10).

Programming Note

To maximize the utility of the Depth control mechanism, the architecture provides a hierarchy of three ways to program it. The DPFD field in the LPCR is used by the provisory/firmware to set a safe or appropriate default depth for unaware operating systems and applications. The DPFD field in the DSCR may be initialized by the aware OS and overwritten by an application via the OS-provided service when per stream control is unnecessary or unaffordable. The DEP field in the EA specification when TH=0b01010 may be used by the application to specify the depth on a per-stream basis.

The number of elements ahead of the point of stream use indicated by a given depth value may differ across implementations, as may the latency to bring a given element into the cache. To achieve optimum performance, some experimentation with different depth values may be necessary.

01011 The ***dcbt/dcbtst*** instruction provides a hint that describes certain attributes of a data stream.

The EA is interpreted as follows.

Bit(s) Description

0:31 Reserved

32:49 ***Stride***

The displacement, in words, between the first byte of successive elements in the stream. The effective address of the N^{th} element in the stream is

$$(N-1)\times STRIDE\times 4$$

greater than or less than the effective address of the first element of the stream, depending on the direction specified for the stream.

50 Reserved

51:55 ***Offset***

The word-offset of the first element of the stream in its unit (i.e., the effective address of the first element of the stream is (EATRUNC || OFFSET || 0b00)).

56:59Reserved

60:63 ***Stream ID*** (ID)

Stream ID to use for this data stream.

Programming Note

A program should use a ***dcbt/dcbtst*** instruction with TH=0b01011 only when the stride is larger than 128 bytes. Otherwise, consecutive units will be accessed, so the additional stream information has no benefit.

If the specified stream ID value is greater than m-1, where m is the number of stream IDs provided by the implementation, and either (a) TH=0b01000 or TH=0b01011, or (b) TH=0b01010 with GO=0 and S≠0b11, no hint is provided by the instruction.

The following terminology is used to describe the state of a data stream. Except as described in the paragraph after the next paragraph, the state of a data stream at a given time is determined by the most recently provided hint(s) for the stream.

- A data stream for which only descriptive hints have been provided (by ***dcbt/dcbtst*** instructions with TH=0b01000 and UG=0, TH=0b01010 and GO=0 and S=0b00, and/or with TH=0b01011) is said to be “nascent”. A nascent data stream for which all relevant descriptive hints have been provided (by the ***dcbt/dcbtst*** usages listed in the preceding sentence) is considered to be “completely described”. The order of descriptive hints with respect to one another is unimportant.
- A data stream for which a hint has been provided (by a ***dcbt/dcbtst*** instruction with TH=0b01000 and UG=1 or ***dcbt*** with TH=0b01010 and GO=1) that the program will probably soon access it is said to be “active”.
- A data stream that is either nascent or active is considered to “exist”.
- A data stream for which a hint has been provided (e.g., by a ***dcbt*** instruction with TH=0b01010 and S≠0b00) that the program will probably no longer access it is considered no longer to exist.

The hint provided by a ***dcbt/dcbtst*** instruction with TH=0b01000 and UG=1 implicitly includes a hint that the program will probably no longer access the data stream (if any) previously associated with the specified stream ID. The hint provided by a ***dcbt/dcbtst*** instruction with TH=0b01000 and UG=0, or with TH=0b01010 and GO=0 and S=0b00, or with TH=0b01011 implicitly includes a hint that the program will probably no longer access the *active* data stream (if any) previously associated with the specified stream ID.

If a data stream is specified without using a ***dcbt/dcbtst*** instruction with TH=0b01010 and GO=0 and S=0b00, then the number of elements in the stream is unlimited, and the program’s need for each element of the stream is not likely to be transient. If a data stream is specified without using a ***dcbt/dcbtst*** instruction with

TH=0b01011, then the stream will access consecutive units of storage.

Interrupts (see Book III) cause all existing data streams to cease to exist. In addition, depending on the implementation, certain conditions and events may cause an existing data stream to cease to exist; for example, in some implementations an existing data stream ceases to exist when it comes to the end of a page.

Programming Note

To obtain the best performance across the widest range of implementations that support the data stream variants of ***dcbt/dcbtst***, the programmer should assume the following model when using those variants.

- The processor's response to a hint that the program will probably soon access a given data stream is to take actions that reduce the latency of accesses to the first few elements of the stream. (Such actions may include prefetching cache blocks into levels of the storage hierarchy that are "near" the processor.) Thereafter, as the program accesses each successive element of the stream, the processor takes latency-reducing actions for additional elements of the stream, pacing these actions with the program's accesses (i.e., taking the actions for only a limited number of elements ahead of the element that the program is currently accessing).

  The processor's response to a hint that the program will probably no longer access a given data stream, or to the cessation of existence of a data stream, is to stop taking latency-reducing actions for the stream.

- A data stream having finite length ceases to exist when the latency-reducing actions have been taken for all elements of the stream.

- If the program ceases to need a given data stream before having accessed all elements of the stream (always the case for streams having unlimited length), performance may be improved if the program then provides a hint that it will no longer access the stream (e.g., by executing the appropriate ***dcbt*** instruction with TH=0b01010 and S≠0b00).

- At each level of the storage hierarchy that is "near" the processor, elements of a data stream that is specified as transient are most likely to be replaced. As a result, it may be desirable to stagger addresses of streams (choose addresses that map to different cache congruence classes) to reduce the likelihood that an element of a transient stream will be replaced prior to being accessed by the program.

- Processors that comply with versions of the architecture that do not support the TH field at all treat TH = 0b01000, 0b01010, and 0b01011 as if TH = 0b00000.

- A single set of stream IDs is shared between the ***dcbt*** and ***dcbtst*** instructions.

- On some implementations, data streams that are not specified by software may be detected by the processor. Such data streams are called "hardware-detected data streams". On some such implementations, data stream resources (resources that are used primarily to support data streams) are shared between software-specified data streams and hardware-detected data streams. On these latter implementations, the programming model includes the following.

  - Software-specified data streams take precedence over hardware-detected data streams in use of data stream resources.

  - The processor's response to a hint that the program will probably no longer access a given data stream, or to the cessation of existence of a data stream, includes releasing the associated data stream resources, so that they can be used by hardware-detected data streams.

Programming Note

The latency-reducing actions taken in response to a program's hints about access to a data stream, including the depth and urgency parameters, may vary based on its behavior and on the behavior of other programs sharing platform resources, as well as on the design of the platform resources they use. Without actually changing the stream specification or DSCR parameters, the processor may adjust its actions (e.g. slow down prefetches or be more selective choosing them) based on their effectiveness and on the availability of storage bandwidth. In general, the goal of this variation is to improve overall system performance and fairness across the set of programs that share resources. There often will be a performance benefit, however, from adjusting stream specifications to the platform and co-resident programs to adjust for these actions by the processor.

Programming Note

This Programming Note describes several aspects of using the data stream variants of the ***dcbt*** and ***dcbtst*** instructions.

- A non-transient data stream having unlimited length and which will access consecutive units in storage can be completely specified, including providing the hint that the program will probably soon access it, using one ***dcbt*** instruction. The corresponding specification for a data stream having other attributes requires two or three ***dcbt/dcbtst*** instructions to describe the stream and one additional ***dcbt*** instruction to start the stream. However, one ***dcbt*** instruction with TH=0b01010 and GO=1 can apply to a set of the data streams described in the preceding sentence, so the corresponding specification for n such data streams requires 2×n to 3×n ***dcbt/dcbtst*** instructions plus one ***dcbt*** instruction. (There is no need to execute a ***dcbt/dcbtst*** instruction with TH=0b01010 and S=0b10 for a given stream ID before using the stream ID for a new data stream; the implicit portion of the hint provided by ***dcbt/dcbtst*** instructions that describe data streams suffices.)
- If it is desired that the hint provided by a given ***dcbt/dcbtst*** instruction be provided in program order with respect to the hint provided by another ***dcbt/dcbtst*** instruction, the two instructions must be separated by an ***eieio*** instruction. For example, if a ***dcbt*** instruction with TH=0b01010 and GO=1 is intended to indicate that the program will probably soon access nascent data streams described (completely) by preceding ***dcbt/dcbtst*** instructions, and is intended *not* to indicate that the program will probably soon access nascent data streams described (completely) by following ***dcbt/dcbtst*** instructions, an ***eieio*** instruction must separate the ***dcbt*** instruction with GO=1 from the preceding ***dcbt/dcbtst*** instructions, and another ***eieio*** instruction must separate that ***dcbt*** instruction from the following ***dcbt/dcbtst*** instructions.
- In practice, the second ***eieio*** described above can sometimes be omitted. For example, if the program consists of an outer loop that contains the ***dcbt/dcbtst*** instructions and an inner loop that contains the *Load* or *Store* instructions that access the data streams, the characteristics of the inner loop and of the implementation's branch prediction mechanisms may make it highly unlikely that hints corresponding to a given iteration of the outer loop will be provided out of program order with respect to hints corresponding to the previous iteration of the outer loop. (Also, any providing of hints out of program order affects only performance, not program correctness.)
- To mitigate the effects of interrupts on data streams, it may be desirable to specify a given "logical" data stream as a sequence of shorter, component data streams. Similar considerations apply to conditions and events that, depending on the implementation, may cause an existing data stream to cease to exist; for example, in some implementations an existing data stream ceases to exist when it comes to the end of a virtual page.
- If it is desired to specify data streams without regard to the number of stream IDs provided by the implementation, stream IDs should be assigned to data streams in order of decreasing stream importance (stream ID 0 to the most important stream, stream ID 1 to the next most important stream, etc.). This order ensures that the hints for the most important data streams will be provided.

TH=0b10000

If TH=0b10000, the ***dcbt*** instruction provides a hint that the program will probably soon load from the block containing the byte addressed by EA, and that the program's need for the block will be transient (i.e., the time interval during which the program accesses the block is likely to be short).

Programming Note

The processor's response to the hint that access to the block will be transient is to prefetch data into the cache hierarchy in a way that minimizes the displacement of data that has not been identified as transient.

TH=0b10001

If TH=0b10001, the ***dcbt*** instruction provides a hint that the program will probably not access the block containing the byte addressed by EA for a relatively long period of time.

Data Cache Block Touch *X-form*

dcbt RA,RB,TH

31	TH	RA	RB	278	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

The ***dcbt*** instruction provides a hint that describes a block or data stream to which the program may perform a *Load* access. The instruction is also used to indicate imminent access or end of access to described load and store data streams. A hint that the program will probably soon load from a given storage location is ignored if the location is Caching Inhibited or Guarded.

The only operation that is "caused" by the ***dcbt*** instruction is the providing of the hint. The actions (if any) taken by the processor in response to the hint are not considered to be "caused by" or "associated with" the ***dcbt*** instruction (e.g., ***dcbt*** is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by the memory barrier created by a ***sync*** instruction.

The ***dcbt*** instruction may complete before the operation it causes has been performed.

The nature of the hint depends, in part, on the value of the TH field, as specified at the beginning of this section. If TH≠0b01010 and TH≠0b01011, this instruction is treated as a *Load* (see Section 4.3), except that the system data storage error handler is not invoked, and reference and change recording need not be done.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics are provided for the *Data Cache Block Touch* instruction so that it can be coded with the TH value as the last operand for all categories, and so that the transient hint can be specified without coding the TH field explicitly.

Extended:	**Equivalent to:**
dcbtct RA,RB,TH	dcbt for TH values of 0b00000 - 0b00111; other TH values are invalid.
dcbtds RA,RB,TH	dcbt for TH values of 0b00000 or 0b01000 - 0b01111; other TH values are invalid.
dcbtt RA,RB	dcbt for TH value of 0b10000
dcbna RA,RB	dcbt for TH value of 0b10001

Programming Notes

New programs should avoid using the ***dcbt*** and ***dcbtst*** mnemonics; one of the extended mnemonics should be used exclusively.

If the ***dcbt*** mnemonic is used with only two operands, the TH operand is assumed to be 0b00000.

Processors that comply with versions of the architecture that precede Version 2.01 do not necessarily ignore the hint provided by ***dcbt*** and ***dcbtst*** if the specified block is in storage that is Guarded and not Caching Inhibited.

Programming Note

See the Programming Notes at the beginning of this section.

Data Cache Block Touch for Store *X-form*

dcbtst RA,RB,TH

31	TH	RA	RB	246	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

The ***dcbtst*** instruction provides a hint that describes a block or data stream to which the program may perform a *Store* access, or indicates the expected use thereof. A hint that the program will soon store to a given storage location is ignored if the location is Caching Inhibited or Guarded.

The only operation that is "caused by" the ***dcbtst*** instruction is the providing of the hint. The actions (if any) taken by the processor in response to the hint are not considered to be "caused by" or "associated with" the ***dcbtst*** instruction (e.g., ***dcbtst*** is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by memory barriers.

The ***dcbtst*** instruction may complete before the operation it causes has been performed.

The nature of the hint depends, in part, on the value of the TH field, as specified at the beginning of this section. If TH≠0b01010 and TH≠0b01011, this instruction is treated as a *Store* (see Section 4.3), except that the system data storage error handler is not invoked, reference recording need not be done, and change recording is not done.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics are provided for the *Data Cache Block Touch for Store* instruction so that it can be coded with the TH value as the last operand for all categories, and so that the transient hint can be specified without coding the TH field explicitly.

Extended:	Equivalent to:
dcbtstct RA,RB,TH	dcbtst for TH values of 0b00000 or 0b00000 - 0b00111; other TH values are invalid.
dcbtstds RA,RB,TH	dcbtst for TH values of 0b00000 or 0b01000 - 0b01111; other TH values are invalid.
dcbtstt RA,RB	dcbtst for TH value of 0b10000.

Programming Note

See the Programming Notes at the beginning of this section.

Data Cache Block set to Zero *X-form*

dcbz RA,RB

31	///	RA	RB	1014	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
n ← block size (bytes)
m ← log2(n)
ea ← EA0:63-m || m0
MEM(ea, n) ← n0x00
```

Let the effective address (EA) be the sum (RA|0)+(RB).

All bytes in the block containing the byte addressed by EA are set to zero.

This instruction is treated as a Store (see Section 4.3).

Special Registers Altered:

None

Programming Note

dcbz does not cause the block to exist in the data cache if the block is in storage that is Caching Inhibited.

For storage that is neither Write Through Required nor Caching Inhibited, ***dcbz*** provides an efficient means of setting blocks of storage to zero. It can be used to initialize large areas of such storage, in a manner that is likely to consume less memory bandwidth than an equivalent sequence of *Store* instructions.

For storage that is either Write Through Required or Caching Inhibited, ***dcbz*** is likely to take significantly longer to execute than an equivalent sequence of *Store* instructions. For example, on some implementations dcbz for such storage may cause the system alignment error handler to be invoked; on such implementations the system alignment error handler sets the specified block to zero using Store instructions.

See Section 5.9.1 of Book III for additional information about ***dcbz***.

Data Cache Block Store *X-form*

dcbst RA,RB

31	///	RA	RB	54	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, those locations are written to main storage, additional locations in the block may be written to main storage, and the block ceases to be considered to be modified in that data cache.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the data cache of this processor and any locations in the block are considered to be modified there, those locations are written to main storage, additional locations in the block may be written to main storage, and the block ceases to be considered to be modified in that data cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a *Load* (see Section 4.3), except that reference and change recording need not be done.

Special Registers Altered:

None

Data Cache Block Flush *X-form*

dcbf RA,RB,L

31	///	L	RA	RB	86	/
0	6	9	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

L=0

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, those locations are written to main storage and additional locations in the block may be written to main storage. The block is invalidated in the data caches of all processors.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and the block is in the data cache of this processor and any locations in the block are considered to be modified there, those locations are written to main storage and additional locations in the block may be written to main storage. The block is invalidated in the data cache of this processor.

L=1 (“dcbf local”)

The L=1 form of the ***dcbf*** instruction permits a program to limit the scope of the “flush” operation to the data cache of this processor. If the block containing the byte addressed by EA is in the data cache of this processor, it is removed from this cache. The coherence of the block is maintained to the extent required by the Memory Coherence Required storage attribute.

L = 3 (“dcbf local primary”)

The L=3 form of the ***dcbf*** instruction permits a program to limit the scope of the “flush” operation to the primary data cache of this processor. If the block containing the byte addressed by EA is in the primary data cache of this processor, it is removed from this cache. The coherence of the block is maintained to the extent required by the Memory Coherence Required storage attribute.

For the L operand, the value 2 is reserved. The results of executing a ***dcbf*** instruction with L=2 are boundedly undefined.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a *Load* (see Section 4.3), except that reference and change recording need not be done.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics are provided for the *Data Cache Block Flush* instruction so that it can be coded with the L value as part of the mnemonic rather than as a numeric operand. These are shown as examples with the instruction. See Appendix A. “Assembler Extended Mnemonics” on page 911. The extended mnemonics are shown below.

Extended:	Equivalent to:
dcbf RA,RB	dcbf RA,RB,0
dcbfl RA,RB	dcbf RA,RB,1
dcbflp RA,RB	dcbf RA,RB,3

Except in the ***dcbf*** instruction description in this section, references to “***dcbf***” in Books I-III imply L=0 unless otherwise stated or obvious from context; “***dcbfl***” is used for L=1 and “***dcbflp***” is used for L=3.

Programming Note

dcbf serves as both a basic and an extended mnemonic. The Assembler will recognize a ***dcbf*** mnemonic with three operands as the basic form, and a ***dcbf*** mnemonic with two operands as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Programming Note

dcbf with L=1 can be used to provide a hint that a block in this processor’s data cache will not be reused soon.

dcbf with L=3 can be used to flush a block from the processor’s primary data cache but reduce the latency of a subsequent access. For example, the block may be evicted from the primary data cache but a copy retained in a lower level of the cache hierarchy.

Programs which manage coherence in software must use ***dcbf*** with L=0.

4.3.2.1 Obsolete Data Cache Instructions

The *Data Stream Touch* (***dst***), *Data Stream Touch for Store* (***dststst***), and *Data Stream Stop* (***dss***) instructions (primary opcode 31, extended opcodes 342, 374, and 822 respectively), which were proposed for addition to the Power ISA and were implemented by some processors, must be treated as no-ops (rather than as illegal instructions).

The treatment of these instructions is independent of whether other Vector instructions are available (i.e., is independent of the contents of MSR_{VEC} (see Book III).

Programming Note

These instructions merely provided hints, and thus were permitted to be treated as no-ops even on processors that implemented them.

The treatment of these instructions is independent of whether other Vector instructions are available because, on processors that implemented the instructions, the instructions were available even when other Vector instructions were not.

The extended mnemonics for these instructions were ***dstt, dststt,*** and ***dssall***.

4.3.3 “or” Instruction

“or” Cache Control Hint

or 26,26,26

This form of ***or*** provides a hint that stores caused by preceding *Store* and ***dcbz*** instructions should be performed with respect to other processors and mechanisms as soon as is feasible.

Extended Mnemonics:

Additional extended mnemonic for the ***or*** hint*:*

Extended:	Equivalent to:	
miso	or	26,26,26

“miso” is short for “make it so.”

Programming Note

This form of the ***or*** instruction can be used to reduce latency in producer-consumer applications by requesting that modified data be made visible to other processors quickly. In this example it is assumed that the base register is GPR3.

Producer:

```
addi r1,r0,0x1234
sth r1,0x1000(r3) # store data value 0x1234
lwsync            # order data store before
                    flag store
addi r2,r0,0x0001
stb r2,0x1002(r3) # store nonzero flag byte
or r26,r26,r26    # miso

p_loop:
lbz r2,0x1002(r3) # load flag byte
andi. r2,r2,0x00FF
bne p_loop        # wait for consumer to clear
                  # flag
```

Consumer:

```
c_loop:
lbz r2,0x1002(r3) # load flag byte
andi. r2,r2,0x00FF
beq c_loop        # wait for producer to set
                  # flag to nonzero
lwsync            # order flag load before
                  # data load
lhz r1,0x1000(r3) # load data value
lwsync            # order data load before
                  # flag store
addi r2,r0,0x0000
stb r2,0x1002(r3) # clear  flag byte
or r26,r26,r26    # miso
```

Programming Note

Warning: Other forms of *or Rx,Rx,Rx* that are not described in this section and in Section 3.2 may also cause program priority to change. Use of these forms should be avoided except when software explicitly intends to alter program priority. If a no-op is needed, the preferred no-op (*ori 0,0,0*) should be used.

4.4 Copy-Paste Facility

The Copy-Paste Facility provides a means to copy a block of data to an accelerator. It uses pairs of instructions, ***copy*** followed by ***paste.***, to define the data transfers. (See Section 1.7.2, "Storage Ordering of Copy/Paste-Initiated Data Transfers" for the memory model characteristics of these data transfers.) Authority to use an accelerator is established through a call to the hypervisor, the details of which are beyond the scope of the architecture. The format of the data block is accelerator-specific. The transfer preserves the order of bytes in storage and is not affected by the endian mode of the processor.

Since the buffer that holds the block until a data transfer is performed is hidden state (cannot be saved and restored) and there is no way to save the state of the ***copy***, any disruption of program execution (e.g. interrupts, event-based branch) has the potential to prevent the data transfer from completing correctly. The software that handles the disruption is responsible for executing ***cpabort*** to clear the state associated with an outstanding data transfer if it will use the Copy-Paste Facility itself or transfer control to another program that might use the facility prior to returning control to the original program.

Programming Note

A ***paste.*** instruction is ordered with respect to its preceding ***copy*** by a dependency on the copy buffer. No explicit synchronization or barrier is required.

Correct use of the Copy-Paste Facility consists of a series of ***copy***/***paste.*** pairs. The two instructions in a pair need not be adjacent in the instruction stream. Two or more ***copy*** instructions with no intervening ***paste.*** produces a "copy-paste sequence error." Similarly, a bare ***paste.*** with no preceding ***copy*** produces a copy-paste sequence error. Copy-paste sequence errors are reported by the ***paste.*** for the malformed sequence of instructions.

Programming Note

WARNING: In rare circumstances, ***paste.*** may falsely report successful completion when the copy-paste sequence is coded incorrectly. This may occur if the instruction sequence includes a redundant ***copy*** and the sequence is interrupted just prior to the redundant ***copy***. Since interrupts should be rare, any sequence that returns a false positive CR0 value should fail for most executions.

Programming Note

It is always best to avoid unnecessary instructions between the ***copy*** and the ***paste.***

Successful transfers are indicated when ***paste.*** returns 0b001x in CR0. Transient errors (a copy-paste sequence error, a memory management state change (***tlbie***[***l***]) during the transfer, or an implementation-specific transient problem) are indicated by a CR0 value of 0b000x, indicating the sequence should be retried. (A sequence error is considered transient because it could have been caused by an interruption between the ***copy*** and ***paste.***.) Fatal errors unique to the Copy-Paste Facility (attempting to copy from an accelerator, attempting to paste to normal memory, and attempting to use an accelerator that has not been properly configured) cause the system data storage error handler to be invoked when the (associated) ***paste.*** instruction is executed. ***paste.*** instructions that cause or report transient errors, fatal errors unique to the Copy-Paste Facility, or successful transfer completion reset the state of the facility so that a subsequent copy-paste sequence can begin with a clean slate.

Programming Note

A failure of a data transfer may be the result of a shortage of the resources required to complete the operation. When the resources are known to be shared by multiple programs, a credit-based system is frequently used to improve quality of service. If such a credit system is in use, or if the resources are not shared, the program should continually repeat the ***copy***/***paste.*** pair until it succeeds. However, if no credit system is in use for shared resources, it may be appropriate to apply some sort of backoff algorithm after having retried the ***copy***/***paste.*** pair a few times.

The Copy-Paste Facility is the only means to address an accelerator. If any other storage access (implicit or explicit, instruction or data) addresses an accelerator, a Machine Check exception will result. Unlike other Machine Check exceptions, this one will generally be presented with ordering and priority similar to that for a storage protection exception.

Programming Note

Accelerator address space is to be marked No-execute by the hypervisor, so that an instruction fetch will violate storage protection rather than causing a Machine Check.

Copy *X-form*

copy RA,RB

31	///	1	RA	RB	774	/
0	6 9	10	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +(RB)
copy_buffer ←  memory(EA,128)
```

Let the effective address (EA) be the sum (RA|0)+(RB).

The 128 bytes in storage addressed by EA is loaded into the copy buffer.

If the EA is not a multiple of 128, the system alignment error handler is invoked.

If the specified block is in storage that is Caching Inhibited, the system data storage error handler is invoked

When successful, this instruction is treated as a *Load* (see Section 4.3, “Cache Management Instructions”), except that the data transfer ordering is described in Section 1.7.2, “Storage Ordering of Copy/Paste-Initiated Data Transfers”.

Special Registers Altered:

None

Paste *X-form*

paste. RA,RB

31	///	1	RA	RB	902	1
0	6 9	10	11	16	21	31

```
if there was a copy-paste sequence error or a
translation conflict
  CR0←0b000||XER_SO
else
  if RA = 0 then b ← 0
  else           b ← (RA)
  EA ← b +(RB)
  post(memory(EA,128)) ←  copy_buffer
  wait for completion status
  if there was a data transfer problem
    CR0←0b000||XER_SO
  else
    CR0←0b001||XER_SO
clear the state of the Copy-Paste Facility
```

If there was a copy-paste sequence error or a translation conflict, set CR0 to indicate failure. Otherwise, continue as follows.

Let the effective address (EA) be the sum (RA|0)+(RB).

Post the contents of the copy buffer to be sent to the accelerator addressed by EA and wait for completion status on the data transfer. Set CR0 as follows based on the completion status.

CR0	*Description*
0b000||XER_{SO}	Data transfer failed due to a sequence error or a conflict with ***tlbie*** or some implementation-specific problem.
0b001||XER_{SO}	Data transfer successful.

Clear the state of the Copy-Paste Facility.

If the EA is not a multiple of 128, the system alignment error handler is invoked.

If the specified block is in storage that is Caching Inhibited, the system data storage error handler is invoked.

If the associated ***copy*** specified an accelerator, if the ***paste.*** specifies an accelerator that was not properly configured, or if the paste. specifies normal storage, the data storage error handler will be invoked.

When successful, this instruction is treated as a *Store* (see Section 4.3, “Cache Management Instructions”), except that the data transfer ordering is described in Section 1.7.2, “Storage Ordering of Copy/Paste-Initiated Data Transfers”.

Special Registers Altered:

CR0

Copy-Paste Abort *X-form*

cpabort

31	///	///	///	838	/
0	6	11	16	21	31

```
clear the state of the Copy-Paste Facility
```

The ***cpabort*** instruction causes a data transfer to fail if one is in progress.

Any pending errors in the Copy-Paste Facility are cleared and the state is reset to prepare for a new ***copy***.

Special Registers Altered:

None

4.5 Atomic Memory Operations

The Atomic Memory Operation (AMO) facility may be used to optimize performance when many software threads are manipulating shared control structures concurrently. In such situations, accessing the shared data frequently involves transfering the data from one processor's cache to another. The latency of such transfers can become the limiting factor in the performance of some environments. Rather than moving the data to the work, AMOs move the work to the data. The mental model is of an agent consisting of an execution unit and a work queue near memory that receives atomic update requests from all the processors in the system.

Despite that AMOs are performed at memory, their function is only defined for storage that is not Caching Inhibited. This is done so that software can transparently access the same data using normal loads and stores. But furthermore, AMOs generally behave as typical explicit storage accesses performed by the thread, with respect to both the weakly consistent and SAO storage models. The few complications are described below. Since the performance advantage of AMOs derives from avoiding time of flight through cache hierarchies, software should avoid frequent mixing of normal loads and stores and AMOs to the same storage locations. AMOs are also restricted to storage that is not Guarded and storage that is not Write Through Required to limit implementation complexity.

The facility specifies a set of atomic update operations that a processor may send, accompanied by operands from GPRs, to the memory to be performed. The operations are expressed using the *Load Atomic* (LAT) and *Store Atomic* (STAT) instructions. Each of these instructions performs an atomic update operation (load followed by some manipulation and a store) on some location in storage. As a result, these instructions are considered to be both fixed-point loads and fixed-point stores, and any reference elsewhere in the architecture to fixed-point loads or fixed-point stores apply to these instructions as well, except where explicitly stated otherwise or obvious from context. For example, in order to perform an AMO, it is necessary to have both read and write access to the storage location. Another example is that the DAWR will detect a match if either Data Read or Data Write is selected. Yet another example is that a Trace interrupt will indicate both a load and a store have been executed. Barrier action will be based on whether the barrier would give a load or a store the stronger ordering. The difference between the loads and stores is simply that the loads return a result to a GPR, while the stores do not. In the RTL in the following subsections, the "lat" and "stat" functions represent the manipulations performed by the memory agent. The parameters shown are the maximum storage footprint, the maximum list of registers, and the function code that are provided to the agent. If the specified registers wrap (e.g. RT=R31 and RT+1=R0), the wrapping is permitted. Such an instruction is not an invalid form. Destructive encodings are also permitted (i.e. a LAT specified with RT=RA).

Except in this section, references to "atomic update" in Books I-III imply use of the *Load And Reserve* and *Store Conditional* instructions unless otherwise stated or obvious from context.

Programming Note

The best performance for the Atomic Memory Operations will be realized when the targeted storage locations are accessed only using AMOs. If it is necessary to perform other I=0 loads and stores to those addresses, the result will still be correct, but performance will suffer. In such circumstances, it is not helpful to performance to flush the data to memory using ***dcbf***.

Programming Note

Note that the descriptions of AMO operations are Endian independent. The only effect of Endian on these operations is the obvious one that byte significance within an individual datum reflects the Endian mode.

Engineering Note

4.5.1 Load Atomic

The Atomic Loads perform an atomic update to an aligned memory location and return a value to a GPR. The manipulation performed on the memory value and the value that is returned in the GPR are determined by the function code (FC) specified by the instruction. The name of each function and its associated RTL are shown in Figure 3.

Function Code	GPR operands	Storage operands	Function name and RTL
00000	RT, RT+1	mem(EA,s)	Fetch and Add t ← mem(EA, s) t2 ← t + (RT+1) mem(EA,s) ← t2 RT ← t
00001	RT, RT+1	mem(EA,s)	Fetch and XOR t ← mem(EA, s) t2 ← t ⊕ (RT+1) mem(EA,s) ← t2 RT ← t
00010	RT, RT+1	mem(EA,s)	Fetch and OR t ← mem(EA, s) t2 ← t | (RT+1) mem(EA,s) ← t2 RT ← t
00011	RT, RT+1	mem(EA,s)	Fetch and AND t ← mem(EA, s) t2 ← t & (RT+1) mem(EA,s) ← t2 RT ← t
00100	RT, RT+1	mem(EA,s)	Fetch and Maximum Unsigned t ← mem(EA, s) if (RT+1) $>^{u}$ t then mem(EA,s) ← (RT+1) RT ← t
00101	RT, RT+1	mem(EA,s)	Fetch and Maximum Signed t ← mem(EA, s) if (RT+1) > t then mem(EA,s) ← (RT+1) RT ← t
00110	RT, RT+1	mem(EA,s)	Fetch and Minimum Unsigned t ← mem(EA, s) if (RT+1) $<^{u}$ t then mem(EA,s) ← (RT+1) RT ← t
00111	RT, RT+1	mem(EA,s)	Fetch and Minimum Signed t ← mem(EA, s) if (RT+1) < t then mem(EA,s) ← (RT+1) RT ← t
01000	RT, RT+1	mem(EA,s)	Swap t ← mem(EA, s) mem(EA,s) ← (RT+1) RT ← t
10000	RT, RT1, RT+2	mem(EA,s)	Compare and Swap Not Equal t ← mem(EA, s) if t != (RT+1) then mem(EA,s) ← (RT+2) RT ← t
11000	RT	mem(EA,s) mem(EA+s, s)	Fetch and Increment Bounded t ← mem(EA, s) t2 ← mem(EA+s, s) if t != t2 then mem(EA,s) ← t+1 RT ← t else RT ← 1 << (s*8-1)

11001	RT	mem(EA,s)
		mem(EA+s, s)
			Fetch and Increment Equal
			`t ← mem(EA, s)`
			`t2 ← mem(EA+s, s)`
			`if t = t2 then`
			`  mem(EA,s) ← t+1`
			`  RT ← t`
			`else RT ← 1 << (s*8-1)`

11100	RT	mem(EA-s,s)
		mem(EA, s)
			Fetch and Decrement Bounded
			`t ← mem(EA, s)`
			`t2 ← mem(EA-s, s)`
			`if t != t2 then`
			`  mem(EA,s) ← t-1`
			`  RT ← t`
			`else RT ← 1 << (s*8-1)`

Notes: s = operand size in number of bytes Function codes not listed in this table are considered invalid. For word atomics, only the least significant word of each source register is used, and the least significant word of the target register is updated with the result, while the upper word is set to zero.			

Figure 3. Load Atomic function codes

Load Word Atomic *X-form*

lwat RT,RA,FC

31	RT	RA	FC	582	/
0	6	11	16	21	31

```
if RA=0 then EA ← 0
else         EA ← (RA)
(RT32:63,mem(EA,4))← lat(mem(EA-4,12), RT+132:63,
RT+232:63, FC)
RT0:31←0
```

Let the effective address (EA) be (RA). The least significant word of RT and the word of storage at EA are updated as specified by load atomic function code FC. The most significant word of RT is set to zero. Input operands are function code specific, and may include the least significant words of RT+1 and RT+2, and mem(EA-4,12)

Figure 3 contains the valid function codes. An attempt to execute ***lwat*** specifying an invalid function code will cause the system data storage error handler to be invoked.

The portion of mem(EA-4,12) accessed by the instruction must be contained within an aligned 32-byte block of storage. If it is not, the system alignment error handler will be invoked.

Special Registers Altered:

None

Load Doubleword Atomic *X-form*

ldat RT,RA,FC

31	RT	RA	FC	614	/
0	6	11	16	21	31

```
if RA=0 then EA ← 0
else         EA ← (RA)
(RT,mem(EA,8))← lat(mem(EA-8,24), RT+1, RT+2, FC)
```

Let the effective address (EA) be (RA). RT and the doubleword of storage at EA are updated as specified by load atomic function code FC. Input operands are function code specific, and may include RT+1, RT+2, and mem(EA-8,24)

Figure 3 contains the valid function codes. An attempt to execute ***ldat*** specifying an invalid function code will cause the system data storage error handler to be invoked.

The portion of mem(EA-8,24) accessed by the instruction must be contained within an aligned 32-byte block of storage. If it is not, the system alignment error handler will be invoked.

Special Registers Altered:

None

4.5.2 Store Atomic

The Atomic Stores perform an atomic update to an aligned memory location. The manipulation performed on the memory value is determined by the function code (FC) specified by the instruction. The name of each function and its associated RTL are shown in Figure 4.

Function Code	GPR operands	Storage operands	Function name and RTL
00000	RS	mem(EA,s)	Store Add t ← mem(EA, s) t2 ← t + (RS) mem(EA,s) ← t2
00001	RS	mem(EA,s)	Store XOR t ← mem(EA, s) t2 ← t ⊕ (RS) mem(EA,s) ← t2
00010	RS	mem(EA,s)	Store OR t ← mem(EA, s) t2 ← t | (RS) mem(EA,s) ← t2
00011	RS	mem(EA,s)	Store AND t ← mem(EA, s) t2 ← t & (RS) mem(EA,s) ← t2
00100	RS	mem(EA,s)	Store Maximum Unsigned t ← mem(EA, s) if (RS) $>^u$ t then mem(EA,s) ← (RS)
00101	RS	mem(EA,s)	Store Maximum Signed t ← mem(EA, s) if (RS) > t then mem(EA,s) ← (RS)
00110	RS	mem(EA,s)	Store Minimum Unsigned t ← mem(EA, s) if (RS) $<^u$ t then mem(EA,s) ← (RS)
00111	RS	mem(EA,s)	Store Minimum Signed t ← mem(EA, s) if (RS) < t then mem(EA,s) ← (RS)
11000	RS	mem(EA,s) mem(EA+s, s)	Store Twin t ← mem(EA, s) t2 ← mem(EA+s, s) if t = t2 then mem(EA,s) ← (RS) mem(EA+s,s) ← (RS)

Notes:
s = operand size in number of bytes
Function codes not listed in this table are considered invalid.
For word atomics, only the least significant word of each source register is used.

Figure 4. Store Atomic function codes

Store Word Atomic *X-form*

stwat RS,RA,FC

31	RS	RA	FC	710	/
0	6	11	16	21	31

```
if RA=0 then EA ← 0
else         EA ← (RA)
mem(EA,8)← stat(mem(EA,8), RS32:63, FC)
```

Let the effective address (EA) be (RA). Four or eight bytes of storage at EA are updated as specified by store atomic function code FC. Input operands are function code specific, and may include $RS_{32:63}$ and mem(EA,8).

Figure 4 contains the valid function codes. An attempt to execute ***stwat*** specifying an invalid function code will cause the system data storage error handler to be invoked.

The portion of mem(EA,8) accessed by the instruction must be contained within an aligned 32-byte block of storage. If it is not, the system alignment error handler will be invoked.

Special Registers Altered:

None

Store Doubleword Atomic *X-form*

stdat RS,RA,FC

31	RS	RA	FC	742	/
0	6	11	16	21	31

```
if RA=0 then EA ← 0
else         EA ← (RA)
mem(EA,16)← stat(mem(EA,16), RS, FC)
```

Let the effective address (EA) be (RA). Eight or sixteen bytes of storage at EA are updated as specified by store atomic function code FC. Input operands are function code specific, and may include RS and mem(EA,16).

Figure 4 contains the valid function codes. An attempt to execute ***stdat*** specifying an invalid function code will cause the system data storage error handler to be invoked.

The portion of mem(EA,16) accessed by the instruction must be contained within an aligned 32-byte block of storage. If it is not, the system alignment error handler will be invoked.

Special Registers Altered:

None

4.6 Synchronization Instructions

The synchronization instructions are used to ensure that certain instructions have completed before other instructions are initiated, or to control storage access ordering, or to support debug operations.

4.6.1 Instruction Synchronize Instruction

Instruction Synchronize ***XL-form***

isync

19	///	///	///	150	/
0	6	11	16	21	31

Executing an ***isync*** instruction ensures that all instructions preceding the ***isync*** instruction have completed before the ***isync*** instruction completes, and that no subsequent instructions are initiated until after the ***isync*** instruction completes. It also ensures that all instruction cache block invalidations caused by ***icbi*** instructions preceding the ***isync*** instruction have been performed with respect to the processor executing the ***isync*** instruction, and then causes any prefetched instructions to be discarded.

Except as described in the preceding sentence, the ***isync*** instruction may complete before storage accesses associated with instructions preceding the ***isync*** instruction have been performed.

This instruction is context synchronizing (see Book III).

Special Registers Altered:

None

4.6.2 Load and Reserve and Store Conditional Instructions

The *Load And Reserve* and *Store Conditional* instructions can be used to construct a sequence of instructions that appears to perform an atomic update operation on an aligned storage location. See Section 1.7.4, “Atomic Update” for additional information about these instructions.

The *Load And Reserve* and *Store Conditional* instructions are fixed-point *Storage Access* instructions; see Section 3.3.1, “Fixed-Point Storage Access Instructions”, in Book I.

The storage location specified by the *Load And Reserve* and *Store Conditional* instructions must be in storage that is Memory Coherence Required if the location may be modified by another processor or mechanism. If the specified location is in storage that is Write Through Required or Caching Inhibited, the system data storage error handler is invoked.

The *Load and Reserve* instructions include an Exclusive Access hint (EH), which can be used to indicate that the instruction sequence being executed is implementing one of two types of algorithms:

Atomic Update (EH=0)
This hint indicates that the program is using a fetch and operate (e.g., fetch and add) or some similar algorithm and that all programs accessing the shared variable are likely to use a similar operation to access the shared variable for some time.

Exclusive Access (EH=1)
This hint indicates that the program is attempting to acquire a lock and if it succeeds, will perform another store to the lock variable (releasing the lock) before another program attempts to modify the lock variable.

Programming Note

The Memory Coherence Required attribute on other processors and mechanisms ensures that their stores to the reservation granule will cause the reservation created by the *Load And Reserve* instruction to be lost.

Programming Note

Because the *Load And Reserve* and *Store Conditional* instructions have implementation dependencies (e.g., the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (Test and Set, Compare and Swap, locking, etc.; see Appendix B) that are needed by application programs. Application programs should use these library programs, rather than use the *Load And Reserve* and *Store Conditional* instructions directly.

Programming Note

EH = 1 should be used when the program is obtaining a lock variable which it will subsequently release before another program attempts to perform a store to it. When contention for a lock is significant, using this hint may reduce the number of times a cache block is transferred between processor caches.

EH = 0 should be used when all accesses to a mutex variable are performed using an instruction sequence with *Load and Reserve* followed by *Store Conditional* (e.g., emulating atomic update primitives such as “Fetch and Add;” see Appendix B). The processor may use this hint to optimize the cache to cache transfer of the block containing the mutex variable, thus reducing the latency of performing an operation such as ‘Fetch and Add’.

Programming Note

Either value of the EH field is appropriate for a *Load and Reserve* instruction that is intended to establish a reservation for a subsequent ***waitrsv*** and not a subsequent *Store Conditional* instruction.

Programming Note

Warning: On some processors that comply with versions of the architecture that precede Version 2.00, executing a *Load And Reserve* instruction in which EH = 1 will cause the illegal instruction error handler to be invoked.

Load Byte And Reserve Indexed X-form

lbarx RT,RA,RB,EH

31	RT	RA	RB	52	EH
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 1
RESERVE_ADDR ← real_addr(EA)
RT ← 56 0 || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

This instruction creates a reservation for use by a ***stbcx.*** instruction. A real address computed from the EA as described in Section 1.7.4.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 1 byte is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the byte in storage addressed by EA before some other processor attempts to modify it.

- 0 Other programs might attempt to modify the byte in storage addressed by EA regardless of the result of the corresponding ***stbcx.*** instruction.
- 1 Other programs will not attempt to modify the byte in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

Special Registers Altered:

None

Programming Note

lbarx serves as both a basic and an extended mnemonic. The Assembler will recognize a ***lbarx*** mnemonic with four operands as the basic form, and a ***lbarx*** mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

Load Halfword And Reserve Indexed X-form

lharx RT,RA,RB,EH

31	RT	RA	RB	116	EH
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 2
RESERVE_ADDR ← real_addr(EA)
RT ← 48^0 || MEM(EA, 2)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

This instruction creates a reservation for use by a ***sthcx.*** instruction. A real address computed from the EA as described in Section 1.7.4.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 2 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the halfword in storage addressed by EA before some other processor attempts to modify it.

- 0 Other programs might attempt to modify the halfword in storage addressed by EA regardless of the result of the corresponding ***sthcx.*** instruction.
- 1 Other programs will not attempt to modify the halfword in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 2. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

None

Programming Note

lharx serves as both a basic and an extended mnemonic. The Assembler will recognize a ***lharx*** mnemonic with four operands as the basic form, and a ***lharx*** mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

Load Word And Reserve Indexed X-form

lwarx RT,RA,RB,EH

31	RT	RA	RB	20	EH
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 4
RESERVE_ADDR ← real_addr(EA)
RT ← 32^0 || MEM(EA, 4)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

This instruction creates a reservation for use by a ***stwcx.*** instruction. A real address computed from the EA as described in Section 1.7.4.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 4 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the word in storage addressed by EA before some other processor attempts to modify it.

- 0 Other programs might attempt to modify the word in storage addressed by EA regardless of the result of the corresponding ***stwcx.*** instruction.
- 1 Other programs will not attempt to modify the word in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

None

Programming Note

lwarx serves as both a basic and an extended mnemonic. The Assembler will recognize a ***lwarx*** mnemonic with four operands as the basic form, and a ***lwarx*** mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

Store Byte Conditional Indexed X-form

stbcx. RS,RA,RB

31	RS	RA	RB	694	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 1 &
     RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 1) ← (RS)56:63
      undefined_case  ← 0
      store_performed ← 1
   else
      z ← smallest real page size supported by
          implementation
      if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
         undefined_case  ← 1
      else
        undefined_case  ← 0
        store_performed ← 0
else
  undefined_case  ← 0
  store_performed ← 0
if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    MEM(EA, 1) ← (RS)56:63
  u2 ← undefined 1-bit value
  CR0 ← 0b00 || u2 || XERSO
else
  CR0 ← 0b00 || store_performed || XERSO
RESERVE ← 0
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 1 byte, and the real storage location specified by the ***stbcx.*** is the same as the real storage location specified by the ***lbarx*** instruction that established the reservation, $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

If a reservation exists, and either the length associated with the reservation is not 1 byte or the real storage location specified by the ***stbcx.*** is not the same as the real storage location specified by the ***lbarx*** instruction that established the reservation, the following applies. Let z denote the smallest real page size supported by the implementation. If the real storage location specified by the ***stbcx.*** is in the same aligned z-byte block of real storage as the real storage location specified by the ***lbarx*** instruction that established the reservation, it is undefined whether $(RS)_{56:63}$ are stored into the byte in storage addressed by EA. Otherwise, no store is performed.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

$CR0_{LT\ GT\ EQ\ SO} = 0b00 \mid\mid n \mid\mid XER_{SO}$

The reservation is cleared.

Special Registers Altered:

CR0

Store Halfword Conditional Indexed X-form

sthcx. RS,RA,RB

31	RS	RA	RB	726	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 2 &
     RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 2) ← (RS)48:63
      undefined_case  ← 0
      store_performed ← 1
   else
      z ← smallest real page size supported by
         implementation
      if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
         undefined_case  ← 1
      else
        undefined_case  ← 0
        store_performed ← 0
else
  undefined_case  ← 0
  store_performed ← 0
if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    MEM(EA, 2) ← (RS)48:63
  u2 ← undefined 1-bit value
  CR0 ← 0b00 || u2 || XERSO
else
  CR0 ← 0b00 || store_performed || XERSO
RESERVE ← 0
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 2 bytes, and the real storage location specified by the ***sthcx.*** is the same as the real storage location specified by the ***lharx*** instruction that established the reservation, $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

If a reservation exists, and either the length associated with the reservation is not 2 bytes or the real storage location specified by the ***sthcx.*** is not the same as the real storage location specified by the ***lharx*** instruction that established the reservation, the following applies. Let z denote the smallest real page size supported by the implementation. If the real storage location specified by the ***sthcx.*** is in the same aligned z-byte block of real storage as the real storage location specified by the ***lharx*** instruction that established the reservation, it is undefined whether $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA. Otherwise, no store is performed.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

$CR0_{LT\ GT\ EQ\ SO} = 0b00 \,||\, n \,||\, XER_{SO}$

The reservation is cleared.

EA must be a multiple of 2. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

CR0

Store Word Conditional Indexed X-form

stwcx. RS,RA,RB

31	RS	RA	RB	150	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 4 &
     RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 4) ← (RS)32:63
      undefined_case  ← 0
      store_performed ← 1
   else
      z ← smallest real page size supported by
          implementation
      if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
        undefined_case  ← 1
      else
        undefined_case  ← 0
        store_performed ← 0
else
  undefined_case  ← 0
  store_performed ← 0
if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    MEM(EA, 4) ← (RS)32:63
  u2 ← undefined 1-bit value
  CR0 ← 0b00 || u2 || XERSO
else
  CR0 ← 0b00 || store_performed || XERSO
RESERVE ← 0
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 4 bytes, and the real storage location specified by the ***stwcx.*** is the same as the real storage location specified by the ***lwarx*** instruction that established the reservation, $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

If a reservation exists, and either the length associated with the reservation is not 4 bytes or the real storage location specified by the ***stwcx.*** is not the same as the real storage location specified by the ***lwarx*** instruction that established the reservation, the following applies. Let z denote the smallest real page size supported by the implementation. If the real storage location specified by the ***stwcx.*** is in the same aligned z-byte block of real storage as the real storage location specified by the ***lwarx*** instruction that established the reservation, it is undefined whether $(RS)_{32:63}$ are stored into the word in storage addressed by EA. Otherwise, no store is performed.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

$CR0_{LT\ GT\ EQ\ SO} = 0b00 \mid\mid n \mid\mid XER_{SO}$

The reservation is cleared.

EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

CR0

4.6.2.1 64-Bit Load and Reserve and Store Conditional Instructions

Load Doubleword And Reserve Indexed X-form

ldarx RT,RA,RB,EH

31	RT	RA	RB	84	EH
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 8
RESERVE_ADDR ← real_addr(EA)
RT ← MEM(EA, 8)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The doubleword in storage addressed by EA is loaded into RT.

This instruction creates a reservation for use by a ***stdcx.*** instruction. A real address computed from the EA as described in Section 1.7.4.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 8 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the doubleword in storage addressed by EA before some other processor attempts to modify it.

- 0 Other programs might attempt to modify the doubleword in storage addressed by EA regardless of the result of the corresponding ***stdcx.*** instruction.
- 1 Other programs will not attempt to modify the doubleword in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 8. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

None

Programming Note

ldarx serves as both a basic and an extended mnemonic. The Assembler will recognize a ***ldarx*** mnemonic with four operands as the basic form, and a ***ldarx*** mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

Store Doubleword Conditional Indexed X-form

stdcx. RS,RA,RB

31	RS	RA	RB	214	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 8 &
     RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 8) ← (RS)
      undefined_case  ← 0
      store_performed ← 1
   else
      z ← smallest real page size supported by
          implementation
      if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
        undefined_case  ← 1
      else
        undefined_case  ← 0
        store_performed ← 0
else
  undefined_case  ← 0
  store_performed ← 0
if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    MEM(EA, 8) ← (RS)
  u2 ← undefined 1-bit value
  CR0 ← 0b00 || u2 || XER_SO
else
  CR0 ← 0b00 || store_performed || XER_SO
RESERVE ← 0
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, and either the length associated with the reservation is not 8 bytes or the real storage location specified by the ***stdcx.*** is the same as the real storage location specified by the ***ldarx*** instruction that established the reservation, (RS) is stored into the doubleword in storage addressed by EA.

If a reservation exists, and either the length associated with the reservation is not 8 bytes or the real storage location specified by the ***stdcx.*** is not the same as the real storage location specified by the ***ldarx*** instruction that established the reservation, the following applies. Let z denote the smallest real page size supported by the implementation. If the real storage location specified by the ***stdcx.*** is in the same aligned z-byte block of real storage as the real storage location specified by the ***ldarx*** instruction that established the reservation, it is undefined whether (RS) is stored into the doubleword in storage addressed by EA. Otherwise, no store is performed.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

$CR0_{LT\ GT\ EQ\ SO} = 0b00 \mid\mid n \mid\mid XER_{SO}$

The reservation is cleared.

EA must be a multiple of 8. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:

CR0

4.6.2.2 128-bit Load and Reserve Store Conditional Instructions

For ***lqarx***, the quadword in storage addressed by EA is loaded into an even-odd pair of GPRs as follows. In Big-Endian mode, the even-numbered GPR is loaded with the doubleword from storage addressed by EA and the odd-numbered GPR is loaded with the doubleword addressed by EA+8. In Little-Endian mode, the even-numbered GPR is loaded with the byte-reversed doubleword from storage addressed by EA+8 and the odd-numbered GPR is loaded with the byte-reversed doubleword addressed by EA.

In the preferred form of the *Load Quadword* instruction RA ≠ RTp+1 and RB ≠ RTp+1.

For ***stqcx.***, the contents of an even-odd pair of GPRs is stored into the quadword in storage addressed by EA as follows. In Big-Endian mode, the even-numbered GPR is stored into the doubleword in storage addressed by EA and the odd-numbered GPR is stored into the doubleword addressed by EA+8. In Little-Endian mode, the even-numbered GPR is stored byte-reversed into the doubleword in storage addressed by EA+8 and the odd-numbered GPR is stored byte-reversed into the doubleword addressed by EA.

Load Quadword And Reserve Indexed X-form

lqarx RTp,RA,RB,EH

31	RTp	RA	RB	276	EH
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +(RB)
RESERVE ← 1
RESERVE_LENGTH ← 16
RESERVE_ADDR ← real_addr(EA)
RTp ← MEM(EA, 16)
```

Let the effective address (EA) be the sum (RA|0)+(RB). The quadword in storage addressed by EA is loaded into RTp.

This instruction creates a reservation for use by a ***stqcx.*** instruction. A real address computed from the EA as described in Section 1.7.4.1 is associated with the reservation, and replaces any address previously associated with the reservation. A length of 16 bytes is associated with the reservation, and replaces any length previously associated with the reservation.

The value of EH provides a hint as to whether the program will perform a subsequent store to the doubleword in storage addressed by EA before some other processor attempts to modify it.

- 0 Other programs might attempt to modify the doubleword in storage addressed by EA regardless of the result of the corresponding ***stqcx.*** instruction.
- 1 Other programs will not attempt to modify the doubleword in storage addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

EA must be a multiple of 16. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

If RTp is odd, RTp=RA, or RTp=RB the instruction form is invalid. If RTp=RA or RTp=RB, an attempt to execute this instruction will invoke the system illegal instruction error handler. (The RTp=RA case includes the case of RTp=RA=0.)

Special Registers Altered:

None

Programming Note

lqarx serves as both a basic and an extended mnemonic. The Assembler will recognize a ***lqarx*** mnemonic with four operands as the basic form, and a ***lqarx*** mnemonic with three operands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

Store Quadword Conditional Indexed X-form

stqcx. RSp,RA,RB

31	RSp	RA	RB	182	1
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
if RESERVE then
  if RESERVE_LENGTH = 16 &
     RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 16) ← (RSp)
      undefined_case  ← 0
      store_performed ← 1
   else
      z ← smallest real page size supported by
          implementation
      if RESERVE_ADDR ÷ z = real_addr(EA) ÷ z then
         undefined_case  ← 1
      else
         undefined_case  ← 0
         store_performed ← 0
else
  undefined_case  ← 0
  store_performed ← 0
if undefined_case then
  u1 ← undefined 1-bit value
  if u1 then
    MEM(EA, 16) ← (RSp)
  u2 ← undefined 1-bit value
  CR0 ← 0b00 || u2 || XER_SO
else
  CR0 ← 0b00 || store_performed || XER_SO
RESERVE ← 0
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists, the length associated with the reservation is 16 bytes, and the real storage location specified by the ***stqcx.*** is the same as the real storage location specified by the ***lqarx*** instruction that established the reservation, (RSp) is stored into the quadword in storage addressed by EA.

If a reservation exists, and either the length associated with the reservation is not 16 bytes or the real storage location specified by the ***stqcx.*** is not the same as the real storage location specified by the ***lqarx*** instruction that established the reservation, the following applies. Let z denote the smallest real page size supported by the implementation. If the real storage location specified by the ***stqcx.*** is in the same aligned z-byte block of real storage as the real storage location specified by the ***lqarx*** instruction that established the reservation, it is undefined whether (RSp) is stored into the quadword in storage addressed by EA. Otherwise, no store is performed.

If a reservation does not exist, no store is performed.

CR Field 0 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if, per the preceding description, it is undefined whether the store is performed, the value of n is undefined (and need not reflect whether the store was performed).

$CR0_{LT\ GT\ EQ\ SO} = 0b00 \ || \ n \ || \ XER_{SO}$

The reservation is cleared.

EA must be a multiple of 16. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

If RSp is odd, the instruction form is invalid.

Special Registers Altered:

CR0

4.6.3 Memory Barrier Instructions

The *Memory Barrier* instructions can be used to control the order in which storage accesses and data transfers are performed. See Section 1.8, "Transactions" for a description of how the *Memory Barrier* instructions interact with transactions. Additional information about these instructions and about related aspects of storage management can be found in Book III.

Synchronize *X-form*

sync L

31	///	L	/	///	598	/
0	6	9	11	16	21	31

```
switch(L)
   case(0): hwsync
   case(1): lwsync
   case(2): ptesync
```

The ***sync*** instruction creates a memory barrier (see Section 1.7.1). The set of storage accesses and/or data transfers that is ordered by the memory barrier depends on the contents of the L field as follows.

- **L=0 ("heavyweight sync")**
  The memory barrier provides an ordering function for the storage accesses and data transfers associated with all instructions that are executed by the processor executing the ***sync*** instruction. The applicable pairs are all pairs a_i,b_j of storage accesses and data transfers in which b_j is a data access or data transfer, except that if a_i is the storage access caused by an ***icbi*** instruction then b_j may be performed with respect to the processor executing the ***sync*** instruction before a_i is performed with respect to that processor.

- **L=1 ("lightweight sync")**
  The memory barrier provides an ordering function for the storage accesses caused by *Load*, *Store*, and *dcbz* instructions that are executed by the processor executing the *sync* instruction and for which the specified storage location is in storage that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited. The applicable pairs are all pairs a_i,b_j of storage accesses except those in which a_i is an access caused by a *Store* or *dcbz* instruction and b_j is an access caused by a *Load* instruction.

- **L=2 ("ptesync")**
  The set of storage accesses that is ordered by the memory barrier is described in Section 5.9.2 of Book III, as are additional properties of the ***sync*** instruction with L=2.

The ordering done by the memory barrier is cumulative (regardless of L value).

If L=0 (or L=2), the ***sync*** instruction has the following additional properties.

- Executing the ***sync*** instruction ensures that all instructions preceding the ***sync*** instruction have completed before the ***sync*** instruction completes, and that no subsequent instructions are initiated until after the ***sync*** instruction completes.
- The ***sync*** instruction is execution synchronizing (see Book III). However, address translation and reference and change recording (see Book III) associated with subsequent instructions may be performed before the ***sync*** instruction completes.
- The memory barrier provides the additional ordering function such that if a given instruction that is the result of a store in set B is executed, all applicable storage accesses in set A have been performed with respect to the processor executing the instruction to the extent required by the associated memory coherence properties. The single exception is that any storage access in set A that is caused by an ***icbi*** instruction executed by the processor executing the ***sync*** instruction (P1) may not have been performed with respect to P1 (see the description of the ***icbi*** instruction on page 840).

  The cumulative properties of the memory barrier apply to the execution of the given instruction as they would to a load that returned a value that was the result of a store in set B.

Programming Note

Section 1.9 contains a detailed description of how to modify instructions such that a well-defined result is obtained.

The value L=3 is reserved.

The ***sync*** instruction may complete before storage accesses associated with instructions preceding the ***sync*** instruction have been performed.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics for *Synchronize*:

Extended:	Equivalent to:
sync	sync 0
lwsync	sync 1
ptesync	sync 2

Except in the ***sync*** instruction description in this section, references to "***sync***" in Books I-III imply L=0 unless otherwise stated or obvious from context; the appropriate extended mnemonics are used when other L values are intended.

Programming Note

sync serves as both a basic and an extended mnemonic. Assemblers will recognize a ***sync*** mnemonic with one operand as the basic form, and a ***sync*** mnemonic with no operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Programming Note

The ***sync*** instruction can be used to ensure that all stores into a data structure, caused by *Store* instructions executed in a "critical section" of a program, will be performed with respect to another processor before the store that releases the lock is performed with respect to that processor; see Section B.2, "Lock Acquisition and Release, and Related Techniques" on page 915.

The memory barrier created by a ***sync*** instruction with L=1 does not order implicit storage accesses or instruction fetches. The memory barrier created by a ***sync*** instruction with L=0 (or L=2) orders implicit storage accesses and instruction fetches associated with instructions preceding the ***sync*** instruction but not those associated with instructions following the ***sync*** instruction.

In order to obtain the best performance across the widest range of implementations, the programmer should use the ***sync*** instruction with L=1, or the ***eieio*** instruction, if any of these is sufficient for his needs; otherwise he should use ***sync*** with L=0. ***sync*** with L=2 should not be used by application programs.

Programming Note

The functions provided by ***sync*** with L=1 are a strict subset of those provided by ***sync*** with L=0. (The functions provided by ***sync*** with L=2 are a strict superset of those provided by ***sync*** with L=0; see Book III.)

Enforce In-order Execution of I/O X-form

eieio

31	///	///	///	854	/
0	6	11	16	21	31

The ***eieio*** instruction creates a memory barrier (see Section 1.7.1, “Storage Access Ordering”), which provides an ordering function for the storage accesses caused by *Load*, *Store*, and ***dcbz*** instructions executed by the processor executing the ***eieio*** instruction. These storage accesses are divided into the two sets listed below. The storage access caused by a ***dcbz*** instruction is ordered as a store.

1. Loads and stores to storage that is both Caching Inhibited and Guarded, and stores to main storage caused by stores to storage that is Write Through Required.

   The applicable pairs are all pairs a_i,b_j of such accesses.

2. Stores to storage that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited.

   The applicable pairs are all pairs a_i,b_j of such accesses.

The operations caused by the stream variants of the ***dcbt*** and ***dcbtst*** instructions (i.e., the providing of hints) are ordered by ***eieio*** as a third set of operations, the operations caused by ***tlbie*** and ***tlbsync*** instructions (see Book III) are ordered by ***eieio*** as a fourth set of operations, and the operations caused by ***slbieg*** or ***slbiag*** and ***slbsync*** instructions (see Book III) are ordered by ***eieio*** as a fifth set of operations.

Each of the five sets of storage accesses or operations is ordered independently of the other four sets. The ordering done by ***eieio***'s memory barrier for the second set is cumulative; the ordering done by ***eieio***'s memory barrier for the other four sets is not cumulative.

The ***eieio*** instruction may complete before storage accesses associated with instructions preceding the ***eieio*** instruction have been performed. The ***eieio*** instruction may complete before operations caused by ***dcbt*** and ***dcbtst*** instructions preceding the ***eieio*** instruction have been performed

Special Registers Altered:

None

Programming Note

The ***eieio*** instruction is intended for use in doing memory-mapped I/O). Because loads, and separately stores, to storage that is both Caching Inhibited and Guarded are performed in program order (see Section 1.7.1, “Storage Access Ordering” on page 815), ***eieio*** is needed for such storage only when loads must be ordered with respect to stores.

For the ***eieio*** instruction, accesses in set 1, a_i and b_j need not be the same kind of access or be to storage having the same storage control attributes. For example, a_i can be a load to Caching Inhibited, Guarded storage, and b_j a store to Write Through Required storage.

If stronger ordering is desired than that provided by ***eieio***, the ***sync*** instruction must be used, with the appropriate value in the L field.

Programming Note

The functions provided by ***eieio*** for its second set are a strict subset of those provided by ***sync*** with L=1.

4.6.4 Wait Instruction

The wait instruction is used to stop instruction fetching and execution until certain events occur. These events include exceptions (see Section 1.2.1 of Book III), event-based branch exceptions (see Section 1.1), and task completion by accelerators in the platform (referred to as “platform notify”).

Wait *X-form*

wait WC

31	///	WC	///	///	30	/
0	6	9	11	16	21	31

The ***wait*** instruction causes instruction fetching and execution to be suspended. Instruction fetching and execution are resumed when the events specified by the WC field occur.

The values of the WC field are as follows.

0b00 Resume instruction fetching and execution when an exception, an event-based branch exception, or a platform notify occurs.

0b01:11 Reserved.

The exception, EBB exception, or platform notify causes the ***wait*** instruction to complete and instruction fetching to resume.

When the ***wait*** instruction completes, processing is resumed either at the instruction following the ***wait*** (if interrupts and/or event-based branches are disabled) or in the corresponding interrupt or event-based branch handler (if interrupts and/or event-based branches are enabled). If an interrupt or event-based branch causes resumption of instruction execution, the interrupt or event-based branch handler will return to the instruction after the ***wait***.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for ***wait***:

Extended:	**Equivalent to:**
wait	wait 0

Programming Note

The ***wait*** instruction frees computational resources which might be allocated to another program or converted into power savings.

Programming Note

Since exceptions corresponding to system-caused interrupts (see Section 6.4 of Book III) may occur at any time, including immediately prior to the ***wait*** instruction, applications should not depend on them to cause ***wait*** to resume. In order to ensure timely resumption, therefore, applications should execute ***wait*** only in order to suspend processing until an event-based branch exception or a platform notify occurs.

Also, since exceptions corresponding to interrupts can cause ***wait*** to resume at any time without any EBB exception or platform notify having occurred, programs that execute ***wait*** should check that the expected condition has actually occurred after the ***wait*** instruction completes. If the expected condition has not occurred, ***wait*** should be re-executed. An example code usage is shown below.

```
while (¬expected condition), wait
```

Programming Note

Applications that execute ***wait*** in order to suspend processing until an external event-based branch exception occurs (see Section 7.2) should enable external event-based branch exceptions (by setting $BESCR_{EE}$=1) and disable event-based branches (by setting $BESCR_{GE}$=0) before executing ***wait***. If $BESCR_{GE}$=1, then the expected event-based branch exception may cause the corresponding event-based branch to occur immediately prior to execution of the ***wait*** instruction. This will result in a hang condition since the EBB exception that was expected to cause ***wait*** to resume will have already occurred.

Programming Note

The values in LPIDR, PIDR, and TIDR uniquely identify a thread that has initiated processing on an accelerator. Platforms may use these resources to track the locations of threads in the system. This service enables an accelerator to cause its initiating thread to resume processing when its results are available.

Chapter 5. Transactional Memory Facility

5.1 Transactional Memory Facility Overview

This chapter describes the registers and instructions that make up the transactional memory (TM) facility. Transactional memory is a shared-memory synchronization construct allowing an application to perform a sequence of storage accesses that appear to occur atomically with respect to other threads.

A set of instructions, special-purpose registers, and state bits in the MSR (see Book III) are used to control a transactional facility that is associated with each hardware thread. A ***tbegin.*** instruction is used to initiate transactional execution, and a ***tend.*** instruction is used to terminate transactional execution. Loads and stores that occur between the ***tbegin.*** and ***tend.*** instructions appear to occur atomically. An implementation may prematurely terminate transactional execution for a variety of reasons, rolling back all transactional storage updates that have been made by the thread since the ***tbegin.*** was executed, and rolling back the contents of a subset of the thread's Book I registers to their contents before the ***tbegin.*** was executed. In the event of such premature termination, control is transferred to a software failure handler associated with the transaction, which may then retry the transaction or choose an alternate path depending on the cause of transaction failure. A transaction can be explicitly aborted via a set of *conditional abort* instructions and an *unconditional abort* instruction, ***tabort.***. A ***tsr.*** instruction is used to suspend or resume transactional execution, while allowing the transaction to remain active.

Programming Note

A ***tbegin.*** should always be followed immediately by a ***beq*** as the first instruction of the failure handler, that branches to the main body of the failure handler. The failure handler should always either retry the transaction or use non-transactional code to perform the same operation. (The number of retries should be limited to avoid the possibility of an infinite loop. The limit could be based on the perceived permanence / transience of the failure.) A failure handler policy which includes trying a different transaction before returning to the one that failed may fail to make forward progress.

Programming Note

In code that may be executed transactionally, conditional branches should hint in favor of successful transactional execution where such a distinction exists. For example, the branch immediately following ***tbegin.*** should hint that the branch is very likely not to be taken. As another example, consider a method of coding a failure handler that executes the body of a transaction non-transactionally by branching past the TM control instructions (e.g. ***tsuspend.***). Branches that bypass the TM control instructions should also hint that the branch is very likely not to be taken. These predictions will improve the efficiency of transactional execution, and may also help prevent the addition of spurious accesses to the transactional footprint.

Programming Note

The architecture does not include a “fairness guarantee” or a “forward progress” guarantee for transactions. If two processors repeatedly conflict with one another in an attempt to complete a transaction, one of the two may always succeed while the other may always fail. If two processors repeatedly conflict with one another in an attempt to complete a transaction, both may always fail, depending on the details of the transaction. This is different from the behavior of a typical locking routine, in which one or the other of the competitors will generally get the lock.

Transactions performed using this facility are “strongly atomic”, meaning that they appear atomic with respect to both transactional and non-transactional accesses performed by other threads. Transactions are isolated from reads and writes performed by other threads; i.e., transactional reads and writes will not appear to be interleaved with the reads and writes of other threads.

Nesting of transactions is supported using a form of nesting called “flattened nesting,” in which transactions that are initiated during transactional execution are subsumed by the pre-existing transaction. Consequently, the effects of a nested transaction do not become visible until the outer transaction commits, and if a nested transaction fails, the entire set of transactions (outer as well as nested) is rolled back, and control is transferred to the outer transaction’s failure handler. The memory barriers created by ***tbegin.*** and ***tend.*** and the integrated cumulative memory barrier that are described in Section 1.8, “Transactions” are only created for outer transactions and not for any transactions nested within them.

References to *Store* instructions, and stores, include ***dcbz*** and the storage accesses that it causes.

Rollback-Only Transactions

Rollback-Only Transactions (ROTs) differ from normal transactions in that they are speculative but not atomic. They are initated by a unique variant of ***tbegin.*** They may be nested with other ROTs or with normal transactions. When a normal transaction is nested within a ROT, the behavior from the normal ***tbegin.*** until the end of the outer transaction is characteristic of a normal transaction. Although subject to failure from storage conflicts, the typical cause of ROT failure is via a ***Tabort*** variant that is executed after the program detects an error in its (software) speculation. Except where specifically differentiated or where differences follow from specific differentiation, the following description applies to ROTs as well as normal transactions.

5.1.1 Definitions

Commit: A transaction is said to commit when it successfully completes execution. When a transaction is committed, its transactional accesses become irrevocable, and are made visible to other threads. A transaction completes by either commiting or failing.

Checkpointed registers: The set of registers that are saved to the “checkpoint area” when a transaction is initiated, and restored upon transaction failure, is a subset of the architected register state, consisting of the General Purpose Registers, Floating-Point Registers, Vector Registers, Vector-Scalar Registers, and the following Special Registers and fields: CR fields other than CR0, XER, LR, CTR, FPSCR, AMR, PPR, VRSAVE, VSCR, DSCR, and TAR. The checkpointed registers include all problem state writable registers with the exception of CR0, EBBHR, EBBRR, BESCR, the Performance Monitor registers, and the Transactional Memory registers. With the exception of updates of CR0, the GSR, and the Transactional Memory registers, explicit updates of registers that are not included in the set of checkpointed registers are disallowed in Transactional state (i.e., will cause the transaction to fail), but are permitted in Suspended state. Suspended state modifications of these registers will not be rolled back in the event of transaction failure. (Modifications of Transactional Memory registers are permitted in Non-transactional state, and modifications of the TFHAR are also permitted in Suspended state. Other attempts to modify Transactional Memory registers will cause a TM Bad Thing type Program interrupt.)

Programming Note

The GSR is not checkpointed because it has no contents; see Chapter 11 of Book III.

Programming Note

CR0, and the Transactional Memory registers (TFHAR, TEXASR, TFIAR) are not saved and are not restored when the transaction fails because restoring them would lose information needed by the failure handler. The Performance Monitor registers and the event-based branching registers (BESCR, EBBHR, EBBRR) are not saved or restored because saving and restoring them would add significant implementation complexity and is not needed by software. Also, these registers, except EBBHR, can be modified asynchronously by the processor, so restoring them when the transaction fails could cause loss of information.

The BHRB is not saved and is not restored when the transaction fails for reasons similar to those for Performance Monitor registers, and because, even if the BHRB were saved and restored, including by the ***treclaim.*** and ***trechkpt.*** instructions (see Section 4.4.3 of Book III), there is no way for privileged software to write the desired contents into the BHRB before executing ***trechkpt.***.

Transactional accesses: Data accesses that are caused by an instruction that is executed when the thread is in the Transactional state (see Section 5.2) are said to be "transactional," or to have been "performed transactionally." The set of accesses caused by a committed normal transaction is performed as if it were a single atomic access. That is, it is always performed in its entirety with no visible fragmentation. The sets performed by normal transactions are thus serialized: each happens in its entirety in some order, even when that order is not specified in the program or enforced between processors. Until a transaction commits, its set of transactional accesses is provisional, and will be discarded should the transaction fail. The set of transactional accesses is also referred to as the "transactional footprint."

Non-transactional accesses: Storage accesses performed in the existing Power storage model are said to be "non-transactional." In contrast to transactional storage accesses, there is no provision of atomicity across multiple non-transactional accesses. Non-transactional storage updates are not discarded in the event of a transaction failure.

Outer transaction: A transaction that is initiated from the Non-transactional state is said to be an outer transaction. A ***tbegin.*** instruction that initiates an outer transaction is sometimes referred to as an "outer ***tbegin.***." Similarly, a ***tend.*** instruction with A=0 that ends an outer transaction is sometimes referred to as an "outer ***tend.***."

Nested Transaction: A transaction that is initiated while already executing a transaction is said to be "nested" within the pre-existing transaction. The set of active nested transactions forms a stack growing from the outer transaction. A ***tend.*** with A=0 will remove the most recently nested transaction from the stack.

Failure: A transaction failure is an exceptional condition causing the transactional footprint to be discarded, the checkpointed registers to be reverted to their pre-transactional values, and the failure handler to get control.

Failure handler: A failure handler is a software component responsible for handling transaction failure. On transaction failure, hardware redirects control to the failure handler associated with the outer transaction.

Conflict: A transactional storage access is said to conflict with another transactional or non-transactional storage access if the two accesses overlap--i.e. if there is at least one byte that is referenced by both accesses-- and at least one of the accesses is a store. If two transactions make conflicting accesses, at least one of them will fail. If a transaction fails as a result of a conflict with a store, the store may have been executed by another processor or may have been executed in Suspended state by the processor with the failing transaction. For a ROT, no conflict is caused if the ROT performs a load and another program performs a non-transactional store to the same storage location. The granularity at which conflict between storage accesses is detected is implementation-dependent, and may vary between accesses, but is never larger than a cache block.

A transactional storage access is said to conflict with a ***tlbie*** or ***slbieg*** if the storage location being accessed is in the page or segment the translation for which is being invalidated by the ***tlbie*** or ***slbieg***. For a ROT, no conflict is caused if the access is a load.

A Suspended state cache control instruction is said to cause a conflict if it would cause the destruction of a transactional update or if it would make a transactional update visible to another thread.

Programming Note

Warning: In descriptions of the transactional memory facility that precede V. 2.07B, the granularity at which conflict between storage accesses is detected was specified to be the cache block. Programs that were based on these early descriptions and depend on this granularity may need to be revised so as not to depend on it.

A future version of the architecture may define "transaction conflict granule", as the aligned unit of storage having the property that the granularity at which conflict between storage accesses is detected is never larger than the transaction conflict granule. The size of the transaction conflict granule would be implementation-dependent and would be added to the list of parameters useful to application programs in Section 4.1 and the last sentence of the first paragraph of the definition of "conflict" would use "transaction conflict granule" instead of "cache block".

5.2 Transactional Memory Facility States

The transactional memory facility supports several modes of operation, referred to in this document as the “transaction state.” These states control the behavior of storage accesses made during the transaction and the handling of transaction failure. Changes to transaction state affect all transactions currently using the transactional facility on the affected thread: the outer transaction as well as any nested transactions, should they exist.

Non-transactional: The default, initial state of execution; no transaction is executing. The transactional facility is available for the initiation of a new transaction.

Transactional: This state is initiated by the execution of a ***tbegin.*** instruction in the Non-transactional state. Storage accesses (data accesses) caused by instructions executed in the Transactional state are performed transactionally. Other storage accesses associated with instructions executed in the Transactional state (instruction fetches, implicit accesses) are performed non-transactionally. In the event of transaction failure, failure is recorded as defined in Section 5.3.2, and control is transferred to the failure handler as described in Section 5.3.3.

Suspended: The Suspended execution state is explicitly entered with the execution of a ***tsuspend.*** form of ***tsr.*** instruction during a transaction, the execution of a ***trechkpt.*** instruction from Non-transactional state, or as a side-effect of an interrupt while in the Transactional state. Storage accesses and accesses to SPRs that are not part of the checkpointed registers are performed non-transactionally; they will be performed independently of the outcome of the transaction. The initiation of a new transaction is prevented in this state. In the event of transaction failure, failure recording is performed as defined in Section 5.3.2, but failure handling is usually deferred until transactional execution is resumed (see Section 5.3.3 for details).

Until failure occurs, *Load* instructions that access storage locations that were transactionally written by the same thread will return the transactionally written data. After failure is detected, but before failure handling is performed, such loads may return either the transactionally written data, or the current non-transactional contents of the accessed location. The ***tcheck*** instruction can be used to determine whether any previous such loads may have returned non-transactional contents.

Suspended state *Store* instructions that access storage locations that have been accessed transactionally (due to load or store) by the same thread cause the transaction to fail.

Programming Note

The intent of the Suspended execution state is to temporarily escape from transactional handling when transactional semantics are undesirable. Examples of such cases include storage updates that should be retained in the event of transactional failure, which is useful for debugging, interthread communication, the access of Caching Inhibited storage, and the handling of interrupts. In the event of transaction failure during the Suspended execution state, failure handling is deferred until transactional execution is resumed, allowing the block of Suspended state code to complete its activities.

Programming Note

During Suspended state execution, accessing storage locations that have been transactionally accessed by the same thread prior to entering Suspended state requires special care, because failure may occur due to uncontrollable events such as interactions with other threads or the operating system. Up until a transaction fails, loads from transactionally modified storage locations will return the transactionally modified data. However once the transaction fails, the loads may return either the transactionally updated version of storage, or a non-transactional version. Suspended state stores to transactionally modified blocks cause the thread’s transaction to fail.

Table 9 enumerates the set of Transactional Memory instructions and events that can cause changes to the transaction state. Transaction states are abbreviated N (Non-transactional), T (Transactional), and S (Suspended). (Interrupts, and the ***rfebb***, ***rfid***, ***rfscv***, ***hrfid***,

and ***mtmsrd*** instructions, can also cause changes to the transaction state; see Book III.)

Programming Note

tbegin. in Suspended state merely updates CR0. When ***tbegin.*** is followed by ***beq***, this will result in a transfer to the failure handler. Nothing more severe (e.g. an interrupt) is required. The failure handler for a transaction for which initiation may be attempted in Suspended state should test CR0 to determine whether tbegin. was executed in Suspended state. If so, it should attempt to emulate the transaction non-transactionally. (This case can arise, for example, if a transaction enters Suspended state and then calls a library routine that independently attempts to use transactions.)

Notice that, although a failure handler runs in Non-transactional state when reached because the transaction has failed, it runs in Suspended state for the case discussed in this Programmng Note.)

Instr/ Event / State	*tbegin.*	*tend.*	Abort caused by *tabort.* and conditional *tabort.* variants	*tsuspend.*	*tresume.*	Failure	treclaim.	trechkpt.
N	T	N^2	N^2	N^2	N^2	Not applicable	N^6	S^7
T	T	N, if outer transaction or A=1 form; otherwise T	$N^{3,4}$	S	T	$N^{3,4}$	N^3	S^6
S	S^1	S^6	S^3	S^2	T^5	S^3	N^3	S^6

Notes
1. **CR0 updated indicating transactional initiation was unsuccessful, due to a pre-existing transaction occupying the transactional facility.**
2. **Execution of these operations does not affect transaction state, allowing for the instructions to be used in software modules called from Non-transactional, Transactional, and Suspended paths.**
3. **If failure recording has not previously occurred, failure recording is performed as defined in Section 5.3.2.**
4. **Failure handling is performed as defined in Section 5.3.3.**
5. **If failure has occurred during Suspended execution, failure handling will be performed sometime after the execution of *tresume*, and no later than the set of events listed in Section 5.3.3.**
6. **Generate TM Bad Thing type Program interrupt.**
7. **If $TEXASR_{FS}$=0, generate a TM Bad Thing type Program interrupt.**

Table 9: Transaction state transitions caused by TM instructions and transaction failure

5.2.1 The TDOOMED Bit

The status of an active transaction is summarized by a transaction doomed bit (TDOOMED) that resides in an implementation-dependent location. When 0, it indicates that the active transaction is valid, meaning that it remains possible for the transaction to commit successfully, if failure does not occur before committing. When 1 it indicates that transaction failure has already occurred for the transaction.

The TDOOMED bit is set to 0 upon the successful initiation of an outer transaction by ***tbegin.***. It is set to 1 when failure occurs or as a result of executing ***trechkpt.***. When failure occurs, TDOOMED is set to 1 before any other effects of the transaction failure (recording the failure in TEXASR, rollback of transactional stores, over-writing of the transactionally accessed locations by a conflicting store, etc.) are visible to software executing on the processor that executed the transaction. In Non-transactional state, the value of TDOOMED is undefined.

5.3 Transaction Failure

5.3.1 Causes of Transaction Failure

A transaction failure is said to be “externally-induced” if the failure is caused by a thread other than the transactional thread. Likewise, a transaction failure is said to be “self-induced” if the failure is caused by the transactional thread itself.

For self-induced failure as a result of attempting to execute an instruction that is disallowed in Transactional state, or an ***mtspr*** specifying an SPR that is not part of the checkpointed registers and is not the GSR or a Transactional Memory SPR, Privileged Instruction type Program interrupt, Hypervisor Emulation Assistance interrupt, and [Hypervisor] Facility Unavailable interrupt take precedence over transaction failure. (For example, an attempt to execute ***stdcix*** in Transactional state and problem state will result in a Privileged Instruction type Program interrupt.) For these instructions, transaction failure takes precedence over all other interrupt types. The relevant instructions are listed in the fourth and fifth bullets of the second set of bullets below and the first and second bullets in the third set of bullets below.

In general, a ROT will not fail in the following scenarios when the failure is specified as a conflict on a transactional access and the access is a load.

Transactions will fail for the following externally-induced causes

- Conflict with transactional access by another thread
- Conflict with non-transactional access by another thread
- In either of the previous two cases, if a successful *Store Conditional* would have conflicted, but the *Store Conditional* is not successful, it is implementation-dependent whether a conflict is detected
- Conflict with a translation invalidation caused by a ***tlbie*** or ***slbieg*** performed by another thread
- ***copy*** from a block that was previously written transactionally is executed by another thread.
- ***paste.*** to a block that was previously accessed transactionally is executed by another thread.
- Footprint overflow that occurs when the thread is sharing the transactional footprint tracking resources with other threads. Footprint overflow is defined as an attempt to perform a storage access in Transactional state which exceeds the capacity for tracking transactional accesses.

Transactions will fail for the following self-induced causes

- Termination caused by the execution of ***tabort.***, ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, ***tabortwci.*** or ***treclaim.*** instruction.
- Transaction level overflow, defined as an attempt to execute ***tbegin.*** when the transaction level is already at its maximum value
- Footprint overflow that occurs when the thread is the only thread using the transactional footprint tracking resources.
- Execution of the following instructions while in the Transactional state: ***icbi, copy, paste., cpabort, lwat, ldat, stwat, stdat, dcbf, dcbi, dcbst, rfscv, [h]rfid, rfebb, mtmsr[d], msgsnd, msgsndp, msgclr, msgclrp, slbie[g], slbia, slbmte, slbfee, stop,*** and ***tlbie[l].*** (These instructions are considered to be *disallowed* in Transactional state.) The disallowed instruction is not executed; failure handling occurs before it has been executed.

Programming Note

Note that execution of a ***stop*** instruction in Suspended state causes a TM Bad Thing type Program interrupt.

- Execution, while in Transactional state, of ***mtspr*** specifying an SPR that is not part of the checkpointed registers and is not the GSR or a Transactional Memory SPR. The ***mtspr*** is not executed; failure handling occurs before it has been executed. (Modification of XER_{FXCC} and CR_{CR0} are allowed, but the changes will not be rolled back in the event of transaction failure.)
- Conflict caused by a Suspended state store to a storage location that was previously accessed transactionally. If the store would have been performed by a successful *Store Conditional* instruction, but the *Store Conditional* instruction does not succeed, it is implementation-dependent whether a conflict is detected.
- Conflict caused by a Suspended state *Load Atomic* or *Store Atomic* instruction updating a block that was previously accessed transactionally.
- Conflict caused by a Suspended state ***tlbie*** or ***slbieg*** that specifies a translation that was previously used transactionally. (This case will be recorded as a translation invalidation conflict because it may be hard to differentiate from a conflict caused by a ***tlbie*** or ***slbieg*** performed by another thread and because it is highly likely to be a transient failure.)

For each of the following potential causes, the transaction will fail if the absence of failure would compromise transaction semantics; otherwise, whether the transaction fails is undefined.

- Execution of the following instructions while in the Transactional state: ***lbzcix, ldcix, lhzcix, lwzcix, stbcix, stdcix, sthcix, stwci***. The disallowed instruction is not executed; failure handling occurs before it has been executed. (These instructions are considered to be disallowed in Transactional state if they cause transaction failure in Transactional state.) Execution of these instructions in the Suspended state is allowed and does not cause transaction failure.
- Execution of the following instruction in the Transactional state: ***wait***. The disallowed instruction is not executed; failure handling occurs before it has been executed. (This instruction is considered to be disallowed in a transaction if it causes transaction failure.)
- Execution of the following instruction in the Suspended state: ***wait***. The disallowed instruction is treated as a no-op; failure recording occurs. (This instruction is considered to be disallowed in a transaction if it causes transaction failure.)
- Access of a disallowed type while in the Transactional state: Caching Inhibited, Write Through Required, and Memory Coherence not Required for data access; Caching Inhibited for instruction fetch. The disallowed access is not performed; failure handling occurs such that the instruction that would cause (or be associated with, for instruction fetch) the disallowed access type appears not to have been executed. Accesses of this type in the Suspended state are allowed and do not cause transaction failure.
- Instruction fetch from a storage location that was previously written transactionally (reported as a unique cause that includes both self-induced and externally-induced instances)

- ***dcbf***, ***dcbi***, or ***icbi*** specifying a block that was previously accessed transactionally, in either of the following cases.

Programming Note

Note that ***dcbf*** with L=3 should never compromise transactional semantics, but it is still permitted to cause transaction failure in Suspended state and it is disallowed in Transactional state.

- the instruction (***dcbf***, ***dcbi***, or ***icbi***) is executed in Suspended state on the processor executing the transaction (self-induced conflict)
- the instruction is executed by another processor (externally-induced conflict)

- ***dcbst*** specifying a block that was previously written transactionally, in either of the following cases.
  - ***dcbst*** is executed in Suspended state on the processor executing the transaction (self-induced conflict)
  - ***dcbst*** is executed by another processor (externally-induced conflict)
- ***copy***, in any of the following cases.
  - ***copy*** from a block that was previously accessed transactionally is executed in Suspended state on the processor executing the transaction (self-induced conflict)
  - ***copy*** from a block that was previously accessed transactionally is executed by another processor (externally-induced conflict)
- Cache eviction of a block that was previously accessed transactionally. (This case will be recorded as an externally-induced footprint overflow for several reasons. First, it is also a case in which over-use of hardware resources precludes complete tracking of the transaction. Second, eviction that is self-induced (i.e., due solely to cache use by the executing thread) may be difficult for hardware to differentiate from eviction that is due partly to cache use by other threads. Finally, this case is expected to occur only rarely, and therefore not to be worth the one or two additional TEXASR bits that would be needed to record it separately.)

Transactions may also fail due to implementation-specific characteristics of the transactional memory mechanism.

Programming Note

Warning: Software should not depend for its correct execution on the behavior (whether or not the relevant transaction fails) of the cases described in the preceding set of bullets. The behavior is likely to vary from design to design. Such a dependence would impact the software's portability without any tangible advantage.

Programming Note

Because the atomic nature of a transaction implies an apparent delay of its component accesses until they can be performed in unison, the use of cache control instructions to manage cache residency and/or the performing of storage accesses may have unexpected consequences. Although they may not cause transaction failure directly, their use in a transaction is strongly discouraged.

If an instruction or event does not cause transaction failure, it behaves as defined in the architecture.

The set of failure causes and events are further classified as "precise" and "imprecise" failure causes. All externally induced events are imprecise, and all self-induced events are precise with the exception of the following cases:

- Self-induced conflicts caused by instruction fetch
- Self-induced conflicts caused by footprint overflow
- Self-induced conflicts in Suspended state (because failure handling is deferred in Suspended state).

When failure recording and handling occur (as defined in Section 5.3.2 and 5.3.3) for a precise failure, they will occur precisely according to the sequential execution model, adhering to the following rules:

1. Effects of the failure occur such that all instructions preceding the instruction causing the failure appear to have completed with respect to the executing thread.
2. The instruction causing the failure may appear not to have begun execution (except for causing the failure), or may have completed, depending on the failure cause.
3. Architecturally, no subsequent instruction has begun execution.

Failure handling for imprecise failure types is guaranteed to occur no later than the execution of ***tend.*** with A=1 or $TEXASR_{TL}$ =1. Failure recording for imprecise failure types is guaranteed to occur no later than failure handling. Any operation that can cause imprecise failure if performed in-order can also cause imprecise failure if performed out-of-order.

Programming Note

Because instruction fetch from a transactionally modified storage location may result in transaction failure, and because conflict between storage accesses may be detected at granularity as large as a cache block, it is recommended that instructions and transactionally accessed data not be co-located within a single cache block.

Programming Note

The architecture does not detect and cause transaction failure for translation invalidations to transactionally accessed pages or segments, when the translation invalidation is caused by instructions other than ***tlbie*** or ***slbieg*** (i.e., ***slbie, slbia, tlbiel***). Consequently, software is responsible for terminating transactions in circumstances where such local translation invalidations may affect a local transaction.

5.3.2 Recording of Transaction Failure

When transaction failure occurs, information about the cause and circumstances of failure are recorded in SPRs associated with the transactional facility. Failure recording is performed a single time per transaction that fails, controlled by the state of the TEXASR failure summary (FS) bit; when 0, FS indicates that failure recording has not already been performed, and is therefore permissible.

The following RTL function specifies the actions taken during the recording of transaction failure:

```
TMRecordFailure(FailureCause)
                  #FailureCause is 32-bit cause
code
if TEXASR_FS = 0
   if failure IA known then
      TFIAR ← CIA
      TEXASR_37 ← 1
   else
      TFIAR ← approximate instruction address
      TEXASR_37 ← 0
   TEXASR_0:31 ← FailureCause
   if MSR_TS=0b01 then TEXASR_Suspended ← 1
   TEXASR_PRIVILEGE ← MSR_HV PR
   TFIAR_PRIVILEGE ← MSR_HV PR
  TEXASR_FS ← 1
  TDOOMED ← 1
```

When failure recording occurs, the TEXASR and TFIAR SPRs are set indicating the source of failure. When possible, TFIAR is set to the effective address of the instruction that caused the failure, and $TEXASR_{37}$ is set to 1 indicating that the contents of TFIAR are exact. When the instruction address is not known exactly, an approximate value is placed in TFIAR and $TEXASR_{37}$ is set to 0. TEXASR bits 0:31 are set indicating the cause of the failure, and the $TEXASR_{Suspended}$, $TEXASR_{Privilege}$, and $TFIAR_{Privilege}$ fields are set indicating the machine state in which the failure was recorded. $TEXASR_{TL}$ is unchanged. The TDOOMED bit is set to 1.

Programming Note

TFIAR is intended for use in the debugging of transactional programs by identifying the source of transaction failure. Because TFIAR may not always be set exactly, software should test $TEXASR_{37}$ before use; if zero, the contents of TFIAR are an approximation.

5.3.3 Handling of Transaction Failure

Discarding of the transactional footprint may begin immediately after detection of failure and, except in the case of an abort in Suspended state, may continue until the rest of failure handling is complete. However, the timing of the rest of failure handling is dependent on the state of the transactional facility. In the case of an abort in Suspended state, the transactional footprint is discarded immediately, despite that the rest of failure handling is deferred.

In Transactional state, failure handling may occur immediately, but an implementation is free to delay handling until one of the following failure synchronizing events occurs in Transactional state.

- An abort caused by the execution of a ***tabort.***, ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, or ***tabortwci.*** instruction.
- The execution of a ***treclaim.*** instruction.
- An attempt, in Transactional state, to execute a disallowed instruction, perform an access of a disallowed type, or execute an ***mtspr*** instruction that specifies an SPR that is not part of the checkpointed registers and is not the GSR or a Transactional Memory SPR.
- Nesting level overflow.
- An attempt to transition from Transactional to Suspended state caused by ***tsuspend.*** or by an interrupt or event.
- An attempt to commit a transaction, caused by the execution of ***tend.*** with A = 1 or when $TEXASR_{TL}$ = 1.

When a failure synchronizing event occurs in Transactional state, the processor waits until all preceding Transactional and Suspended state loads have been performed with respect to all processors and mechanisms and all failures that have occurred up to that point have been recorded. Then failure handling occurs if a failure has been recorded; otherwise, processing of the failure synchronizing event continues. If failure is caused by the failure synchronizing event, failure handling occurs immediately.

When failure handling occurs, checkpointed registers are reverted to their pre-transactional values, the transactional footprint is discarded if it has not previously been discarded, and any resources occupied by the transaction are discarded. If the failure is not caused by

treclaim., the following things occur. CR0 is set to 0b101 || 0. The transaction state is set to Non-transactional, and control flow is redirected to the instruction address stored in TFHAR. If the failure is caused by ***treclaim.***, CR0 is not set to indicate failure and the transaction's failure handler is not invoked.

The following RTL function specifies the actions taken during the handling of transaction failure:

```
TMHandleFailure()
   If the transactional footprint has not previously been discarded
       Discard transactional footprint
   Revert checkpointed registers to pre-transactional values
   Discard all resources related to current transaction
   MSR_TS ← 0b00                    #Non-transactional
   If failure was not caused by treclaim.,
     NIA ← TFHAR
     CR0 ← 0b101 || 0
```

Upon failure detected in Suspended state from causes other than the execution of a ***treclaim.*** instruction, failure handling is deferred until the transaction is resumed. Once resumed, failure handling will occur no later than the set of failure synchronizing events listed above. Upon failure in Suspended state caused by ***treclaim.***, failure handling is immediate (but CR0 is not set to indicate failure and the transaction's failure handler is not invoked).

Programming Note

A *Load* instruction executed immediately after ***treclaim.*** or a conditional or unconditional *Abort* instruction is guaranteed not to load a transactional storage update.

5.4 Transactional Memory Facility Registers

The architecture is augmented with three Special Purpose Registers in support of transactional memory. TFHAR stores the effective address of the software failure handler used in the event of transaction failure. TFIAR is used to inform software of the exact location of the transaction failure, when possible. TEXASR contains a transaction level indicating the nesting depth of an active transaction, as well as an indicator of the cause of transaction failure and some machine state when the transaction failed. These registers can be written only when in Non-transactional state, and for TFHAR, also when in Suspended state.

5.4.1 Transaction Failure Handler Address Register (TFHAR)

The Transaction Failure Handler Address Register is a 64-bit SPR that records the effective address of a software failure handler used in the event of transaction failure. Bits 62:63 are reserved.

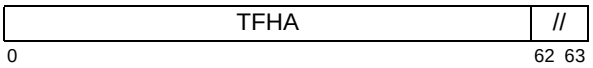


Figure 5. Transaction Failure Handler Address Register (TFHAR)

This register is written with the NIA for the ***tbegin.*** as a side-effect of the execution of an outer ***tbegin.*** instruction (***tbegin.*** executed in the Non-transactional state).

5.4.2 Transaction EXception And Status Register (TEXASR)

The Transaction EXception And Status Register is a 64-bit register, containing a transaction level (TEXAS-R_{TL}) and status information for use by transaction failure handlers. The identification of the cause and persistence of transaction failure reported in bits 7:30 may rarely be inaccurate, except that if bit 31 is set to 1 then bits 7:30 are always accurate. Bits 0:31 are called the *failure cause* in the instruction descriptions.

Programming Note

Warning: In addition to the contents of bits 7:30 of the TEXASR being rarely inaccurate, new failure causes may be added over time, and/or an existing cause may be divided into two or more subsets of causes (which may differ in their persistence). Further, speculative execution can cause unexpected contents of these bits. As a result, except when failure is caused by a ***treclaim.*** or ***tabort.*** instruction (including conditional ***tabort.***), software must not depend on the contents of bits 7:30 for its correct execution. Guidelines follow.

- The bits should only be used to determine the approach to the computation, e.g. whether the computation is suitable to use the Transactional Memory facility, how to adapt it best for TM, or how many retries to attempt before performing the operation non-transactionally.
- Software should use the persistence indication when none of the causes that were defined at the time the software was written is indicated.
- Under no circumstances should software depend on the transience of a failure. There must always be a limit to the number of retries before performing the operation non-transactionally.

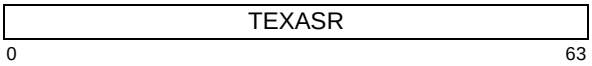


Figure 6. Transaction EXception And Status Register (TEXASR)

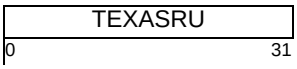


Figure 7. Transaction EXception And Status Register Upper (TEXASRU)

Bit(s Description

0:6 ***Failure Code***
The Failure Code is copied from the ***tabort.*** or ***treclaim.*** source operand. When set, TFIAR is exact.

7 ***Failure Persistent***
The failure is likely to recur on each execution of the transaction. This bit is set to 1 for causes in bits 8:11, copied from the ***tabort.*** or ***treclaim.*** source operand when RA is non-zero, and set to 0 for all other failure causes.

Programming Note

For ***tabort.*** and ***treclaim.***, the Failure Persistent bit may be viewed as an eighth bit in the failure code in that both fields are supplied by the least significant byte of RA and software may use all eight to differentiate among the cases for which it performs an abort or reclaim. However, software is expected to organize its cases so that bit 7 predicts the persistence of the case.

Programming Note

The inaccuracy of the Failure Persistent bit may arise from either of two causes. First, a kind of failure that is usually transient, such as conflict with another thread, may in certain unusual circumstances be persistent. Second, if the cause of transaction failure is identified incorrectly, the Failure Persistent bit will inherit this inaccuracy -- i.e., will be set to 0 or 1 based on the identified failure cause. (Neither of these causes applies if $TEXASR_{31}$=1.)

8 ***Disallowed***
The instruction, SPR, or access type is not permitted. When set, TFIAR is exact. See Section 5.3.1, “Causes of Transaction Failure”.

Programming Note

An instruction fetch to storage that is Caching Inhibited, while nominally disallowed, will be reported as Implementation-specific (bit 15). This choice was made because it seems like a relatively unlikely programming error, and there is a significant chance that data from an external conflict (store by another thread) could indirectly cause a wild branch to storage that is Caching Inhibited.

9 ***Nesting Overflow***
The maximum transaction level was exceeded. When set, TFIAR is exact.

10 ***Footprint Overflow, Self-Induced***
The tracking limit for transactional storage accesses was exceeded when this thread was the only thread using the transactional footprint tracking resources. When set, TFIAR is an approximation.

11 ***Self-Induced Conflict***
A self-induced conflict occurred in Suspended state, due to one of the following: a store to a storage location that was previously accessed transactionally; a ***dcbf***, ***dcbi,*** or ***icbi*** specifying a block that was previously accessed transactionally; a ***dcbst*** specifying a block that was previously written transactionally; a *Load Atomic* or *Store Atomic* instruction specifying a block that was previously accessed transactionally, or a ***copy*** from a block that was previously accessed transactionally. When set, TFIAR may be exact.

12 ***Non-Transactional Conflict***
A conflict occurred with a non-transactional access by another processor. When set, TFIAR is an approximation.

13 ***Transaction Conflict***
A conflict occurred with another transaction. When set, TFIAR may be exact.

14 ***Translation Invalidation Conflict***
A conflict occurred with a TLB or SLB invalidation. When set, TFIAR is an approximation.

15 ***Implementation-specific***
An implementation-specific condition caused the transaction to fail. Such conditions are transient and the value in the TFIAR may be exact.

16 ***Instruction Fetch Conflict***
An instruction fetch (by this or another thread) was performed from a storage location that

was previously written transactionally. Such conditions are transient and the value in the TFIAR may be exact.

17 ***Footprint Overflow, Externally-Induced***
The tracking limit for transactional storage accesses was exceeded when other threads, in addition to this thread, were using the transactional footprint tracking resources. This bit is also set when a cache block eviction causes the transaction to fail. When set, TFIAR is an approximation.

Programming Note

Appropriate behavior of the failure handler when the tracking limit is exceeded due partly to transactions running on other threads may include re-executing the transaction after a significant and randomized amount of time has elapsed. (This policy will tend to spread out the contending transactions in time, and thereby reduce their simultaneous use of the transactional footprint tracking resources.) Some designs may provide hardware assistance in reducing contention for the tracking resources. Writers of failure handlers should see the Users' Manual for the implementation to understand how to benefit from the hardware behavior.

Transaction failure due to cache block eviction is expected to be sufficiently rare that handling it as if the failure were caused by exceeding the tracking limit is acceptable.

18:30 Reserved for future failure causes

31 ***Abort***
Termination was caused by the execution of a ***tabort.***, ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, ***tabortwci.*** or ***treclaim.*** instruction. When due to ***tabort.*** or ***treclaim.***, bits in $TEXASR_{0:7}$ are user-supplied. When set, TFIAR is exact.

32 ***Suspended***
When set to 1, the failure was recorded in Suspended state. When set to 0, the failure was recorded in Transactional state.

33 Reserved

34:35 ***Privilege***
The thread was in this privilege state when the failure was recorded. This was the value $MSR_{HV\ PR}$ when the failure was recorded.

36 ***Failure Summary (FS)***
Set to 1 when a failure has been detected and failure recording has been performed.

37 ***TFIAR Exact***
Set to 1 when the value in the TFIAR is exact. Otherwise the value in the TFIAR is approximate.

38 ***ROT***
Set to 1 when a ROT is initiated. Set to zero when a non-ROT ***tbegin.*** is executed.

39 Reserved

40:51 Reserved

52:63 ***Transaction Level (TL)***
Transaction level (nesting depth + 1) for the active transaction, if any; otherwise 0 if the most recently executed transaction completed successfully, or the transaction level at which the most recently executed transaction failed if the most recently executed transaction did not complete successfully.

The transaction level in $TEXASR_{TL}$ contains an unsigned integer indicating whether the current transaction is an outer transaction ($TEXASR_{TL}$ = 1), or is nested ($TEXASR_{TL}$ > 1), and if nested, its depth. The maximum transaction level supported by a given implementation is of the form 2^t - 1. The value of *t* corresponding to the smallest maximum is 4; the value of *t* corresponding to the largest maximum is 12. This value is tied to the "Maximum transaction level" parameter useful for application programmers, as specified in Section 4.1. The high-order 12-*t* bits of $TEXASR_{TL}$ are treated as reserved.

Transaction failure information is contained in $TEXASR_{0:37}$. The fields of TEXASR are initialized upon the successful initiation of a transaction from the Non-transactional state, by setting $TEXASR_{TL}$ to 1, indicating an outer transaction, and all other fields to 0.

When transaction failure is recorded, the failure summary bit $TEXASR_{FS}$ is set to 1, indicating that failure has been detected for the active transaction and that failure recording has been performed. $TEXASR_{0:31}$ are set indicating the source of the failure. Exactly one of bits 8 through 31 will be set indicating the instruction or event that caused failure. In the event of failure due to the execution of a ***tabort.***, ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, ***tabortwci.*** or ***treclaim.*** instruction, $TEXASR_{31}$ is set to 1, and, for ***tabort.*** and ***treclaim.***, a software defined failure code is copied from a register operand to $TEXASR_{0:7}$. $TEXASR_{Suspended}$ indicates whether the transaction was in the Suspended state at the time that failure occurred. The values of MSR_{HV} and MSR_{PR} at the time that failure occurs are copied to $TEXASR_{34}$ and $TEXASR_{35}$, respectively. In some circumstances, the failure causing instruction address in TFIAR may not be exact. In such circumstances, $TEXASR_{37}$ is set to 0 indicating that the contents of TFIAR are not exact; otherwise $TEXASR_{37}$ is set to 1.

Programming Note

The transaction level contained in $TEXASR_{TL}$ should be interpreted by software as follows:

When in the Transactional or Suspended state, this field contains an unsigned integer representing the transaction level of the active transaction, with 1 indicating an outer transaction, and a number greater than 1 indicating a nested transaction. The nesting depth of the active transaction is $TEXASR_{TL} - 1$.

When in the Non-transactional state, $TEXASR_{TL}$ contains 0 if the last transaction committed successfully, otherwise it contains the transaction level at which the most recent transaction failed.

Programming Note

The Privilege bits in TEXASR represent the state of the machine at the point when failure occurs. This information may be used by problem state software to determine whether an unexpected hypervisor or operating system interaction was responsible for transaction failure. This information may be useful to operating systems or hypervisors when restoring register state for failure handling after the transactional facility was reclaimed, to determine which of the operating system or the hypervisor has retained the pre-transactional version of the checkpointed registers.

5.4.3 Transaction Failure Instruction Address Register (TFIAR)

The Transaction Failure Instruction Address Register is a 64-bit SPR that is set to the exact effective address of the instruction causing the failure, when possible. Bits 62:63 contain the privilege state when the failure was recorded. This was the value $MSR_{HV\ PR}$ when the failure was recorded.

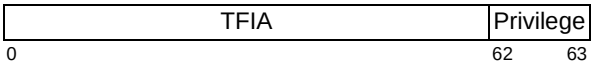


Figure 8. Transaction Failure Instruction Address Register (TFIAR)

In certain cases, the exact address may not be available, and therefore TFIAR will be an approximation. An approximate value will point to an instruction near the instruction that was executing at the time of the failure. TFIAR accuracy is recorded in an Exact bit residing in $TEXASR_{37}$.

5.5 Transactional Facility Instructions

Similar to the *Floating-Point Status and Control Register* instructions, modifications of transaction state caused by the execution of *Transactional Memory* instructions or by failure handling synchronize the effects of exception-causing floating-point instructions executed by a given processor. Executing a Transactional Memory instruction, or invocation of the failure handler, ensures that all floating-point instructions previously initiated by the given processor have completed before the transaction state is modified, and that no subsequent floating-point instructions are initiated by the given processor until the transaction state has been modified. In particular:

- All exceptions that will be caused by the previously initiated instructions are recorded in the FPSCR before the transaction state is modified.
- All invocations of the system floating-point enabled exception error handler that will be caused by the previously initiated instructions have occurred before the transaction state is modified.
- No subsequent floating-point instruction that alters the settings of any FPSCR bits is initiated until the transaction state has been modified.

(*Floating-point Storage Access* instructions are not affected.)

Transaction Begin *X-form*

tbegin. R

31	A	//	R	///	///	654	1
0	6	7	10	11	16	21	31

```
ROT  ← R
CR0  ← 0 || MSR_TS || 0

if MSR_TS = 0b00 then              #Non-transactional
   TEXASR ← 0x000000000 || 0b00 || ROT || 0b0 ||
0x000001
   TFHAR ← CIA + 4
   TDOOMED ← 0
   MSR_TS ← 0b10
   checkpoint area ← (checkpointed registers)
   if not ROT and the transaction succeeds then
     insert tbegin memory barrier
else if MSR_TS = 0b10 then          #Transactional
   if TEXASR_TL=TL_max then
      cause ← 0x01400000
      TMRecordFailure(cause)
      TMHandleFailure()
   else
      TEXASR_TL ← TEXASR_TL + 1
      if (TEXASR_ROT=1) & (not ROT)
        TEXASR_ROT ← 0
        if the transaction succeeds
          insert tbegin memory barrier
```

The ***tbegin.*** instruction initiates execution of a transaction, either an outer transaction or a nested transaction, as described below.

An outer transaction is initiated when ***tbegin.*** is executed in the Non-transactional state. If R=0 and the transaction is successful, the ***tbegin*** memory barrier described in Section 1.8 is inserted. TEXASR and TFHAR are initialized, and the TDOOMED bit is set to 0. A nested transaction is initiated when ***tbegin.*** is executed in the Transactional state unless the transaction level is already at its maximum value, in which case failure recording is performed with a failure cause of 0x01400000 and failure handling is performed. When initiating a nested transaction, the transaction level held in $TEXASR_{TL}$ is incremented by 1, and if $TEXASR_{ROT}$ =1 but R=0, $TEXASR_{ROT}$ is set to 0, and if additionally the transaction succeeds, the ***tbegin*** memory barrier described in Section 1.8 is inserted. The effects of a nested transaction will not be visible until the outer transaction commits, and in the event of failure, the checkpointed registers are reverted to the pre-transactional values of the outer transaction. Initiation of a transaction is unsuccessful when in the Suspended state.

When successfully initiated, transactional execution continues until the transaction is terminated using a ***tend.***, ***tabort.***, ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, ***tabortwci.***, or ***treclaim.*** instruction, suspended using a ***tsr*** instruction, or failure occurs. Upon transaction failure while in the Transactional state, transaction failure recording and failure handling are performed as defined in Section 5.3. Upon transaction failure while in the Suspended state, failure recording is performed as defined in Section 5.3.2, but failure handling is usually deferred.

CR0 is set as follows.

CR0	*Description*
000 || 0	Transaction initiation successful, unnested (Transaction state of Non-transactional prior to ***tbegin.***)
010 || 0	Transaction initiation successful, nested (Transaction state of Transactional prior to ***tbegin.***)
001 || 0	Transaction initiation unsuccessful, (Transaction state of Suspended prior to ***tbegin.***)

Other than the setting of CR0, ***tbegin.*** in the Suspended state is treated as a no-op.

The use of the A field is implementation specific.

Special Registers Altered:

CR0 TEXASR TFHAR TS

Programming Note

When a transaction is successfully initiated, and failure subsequently occurs, control flow will be redirected to the instruction following the *tbegin.* instruction. When failure handling occurs, as described in Section 5.3.3, CR0 is set to 0b101 || 0. Consequently, instructions following ***tbegin.*** should also expect this value as an indication of transaction failure. Most applications will follow ***tbegin.*** with a conditional branch predicated on $CR0_2$; code at this target is responsible for handling the transaction failure.

Transaction End *X-form*

tend. A

31	A	//	/	///	///	686	1
0	6	7	10	11	16	21	31

```
CR0  ← 0b0 || MSR_TS || 0

if MSR_TS = 0b10 then           #Transactional
   if A = 1 | TEXASR_TL = 1 then
      if (TDOOMED) then
         TMHandleFailure()
      else
         if not TEXASR_ROT
           insert integrated cumulative
           memory barrier
         Commit transaction
         TEXASR_TL ← 0
          Discard all resources related to current
transaction
          MSR_TS ← 0b00          #Non-transactional
          if not TEXASR_ROT
            insert tend memory barrier
   else TEXASR_TL ← TEXASR_TL - 1  # nested
```

The A=0 variant of ***tend.*** supports nested transactions, in which the transaction is committed only if the execution of ***tend.*** completes an outer transaction. Execution of this variant by a nested transaction ($TEXASR_{TL} > 1$) causes $TEXASR_{TL}$ to be decremented by 1. The A=1 variant of ***tend.*** unconditionally completes the current outer transaction and all nested transactions.

When the ***tend.*** instruction completes an outer transaction, transaction commit is predicated on the TDOOMED bit. If TDOOMED is 1, failure handling occurs as defined in Section 5.3.3. If TDOOMED is 0, the transaction is committed, and $TEXASR_{TL}$ is set to 0. In both cases, the transaction state is set to Non-transactional.

When the ***tend.*** instruction commits a transaction, it atomically commits its writes to storage. If $TEXASR_{ROT}$=0, the integrated cumulative memory barrier is inserted prior to the creation of the aggregate store, and the ***tend*** memory barrier described in Section 1.8 is inserted after the aggregate store. If the transaction has failed prior to the execution of ***tend.,*** no storage updates are performed and no memory barrier is inserted. In either case (success or failure), all resources associated with the transaction are discarded.

If the transaction succeeds, Condition Register field 0 is set to 0 || MSR_{TS} || 0. If the transaction fails, CR0 is set to 0b101 || 0.

Other than the setting of CR0, ***tend.*** in Non-transactional state is treated as a no-op. If an attempt is made to execute ***tend.*** in Suspended state, a TM Bad Thing type Program interrupt occurs.

Special Registers Altered:

CR0 TEXASR TS

Extended Mnemonics

Examples of extended mnemonics for *Transaction End.*

Extended:	***Equivalent To:***
tend.	tend. 0
tendall.	tend. 1

Programming Note

When an outer ***tend.*** or a ***tend.*** with A=1 is executed in the Transactional state, the CR0 value 0b101 || 0 will never be visible to the instruction that immediately follows ***tend.***, because in the event of failure the failure handler will have been invoked not later than the completion of the ***tend.*** instruction.

Transaction Abort *X-form*

tabort. RA

31	///	RA	///	910	1
0	6	11	16	21	31

```
CR0 ← 0 || MSR_TS || 0

if MSR_TS = 0b10 | MSR_TS = 0b01 then
#Transactional, or Suspended
   if RA = 0 then cause ←0x00000001
   else           cause ← GPR(RA)_56:63 || 0x000001
   if MSR_TS= 0b01 & TEXASR_FS = 0 then  #Suspended
      Discard the transactional footprint
   TMRecordFailure(cause)
   if MSR_TS = 0b10 then                 #Transactional
      TMHandleFailure()
```

The ***tabort.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. When in the Transactional state or the Suspended state the ***tabort.*** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2. If RA is 0, the failure cause is set to 0x00000001, otherwise it is set to $GPR(RA)_{56:63}$ || 0x000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of ***tabort.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered:

CR0 TEXASR TFIAR TS

Transaction Abort Word Conditional *X-form*

tabortwc. TO,RA,RB

31	TO	RA	RB	782	1
0	6	11	16	21	31

```
a ← EXTS((RA)32:63)
b ← EXTS((RB)32:63)
abort ← 0

CR0 ←  0 || MSRTS || 0

if (a < b)  & TO0 then abort ← 1
if (a > b)  & TO1 then abort ← 1
if (a = b)  & TO2 then abort ← 1
if (a u< b) & TO3 then abort ← 1
if (a >u b) & TO4 then abort ← 1

if abort & (MSRTS = 0b10 | MSRTS = 0b01) then
           #Transactional or Suspended
   cause ← 0x00000001
   if MSRTS= 0b01 & TEXASRFS = 0 then  #Suspended
      Discard transactional footprint
   TMRecordFailure(cause)
   if MSRTS = 0b10 then               #Transactional
      TMHandleFailure()
```

The ***tabortwc.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. The contents of register $RA_{32:63}$ are compared with the contents of register $RB_{32:63}$. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended, then the ***tabortwc.*** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of ***tabortwc.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered:

CR0 TEXASR TFIAR TS

Transaction Abort Word Conditional Immediate *X-form*

tabortwci. TO,RA,SI

31	TO	RA	SI	846	1
0	6	11	16	21	31

```
a ← EXTS((RA)32:63)
abort ← 0

CR0 ←  0 || MSRTS || 0


if a < EXTS(SI)  & TO0 then abort ← 1
if a > EXTS(SI)  & TO1 then abort ← 1
if a = EXTS(SI)  & TO2 then abort ← 1
if a u< EXTS(SI) & TO3 then abort ← 1
if a >u EXTS(SI) & TO4 then abort ← 1

if abort & (MSRTS = 0b10 | MSRTS = 0b01) then
           #Transactional or Suspended
   cause ← 0x00000001
   if MSRTS= 0b01 & TEXASRFS = 0 then  #Suspended
      Discard transactional footprint
   TMRecordFailure(cause)
   if MSRTS = 0b10 then               #Transactional
      TMHandleFailure()
```

The ***tabortwci.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. The contents of register $RA_{32:63}$ are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended then the ***tabortwci.*** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of ***tabortwci.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered:

CR0 TEXASR TFIAR TS

Transaction Abort Doubleword Conditional *X-form*

tabortdc. TO,RA,RB

31	TO	RA	RB	814	1
0	6	11	16	21	31

```
a ← ( RA )
b ← ( RB )
abort ← 0

CR0 ← 0 || MSR_TS || 0

if (a < b)  & TO_0 then abort ← 1
if (a > b)  & TO_1 then abort ← 1
if (a = b)  & TO_2 then abort ← 1
if (a u< b) & TO_3 then abort ← 1
if (a >u b) & TO_4 then abort ← 1

if abort & (MSR_TS = 0b10 | MSR_TS = 0b01) then
           #Transactional or Suspended
   cause ← 0x00000001
   if MSR_TS= 0b01 & TEXASR_FS = 0 then  #Suspended
      Discard transactional footprint
   TMRecordFailure(cause)
   if MSR_TS = 0b10 then                 #Transactional
      TMHandleFailure()
```

The ***tabortdc.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. The contents of register RA are compared with the contents of register RB. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended, then the ***tabortdc.*** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of ***tabortdc.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered:

CR0 TEXASR TFIAR TS

Transaction Abort Doubleword Conditional Immediate *X-form*

tabortdci. TO,RA, SI

31	TO	RA	SI	878	1
0	6	11	16	21	31

```
a ← (RA)
abort ← 0

CR0 ← 0 || MSR_TS || 0

if a < EXTS(SI)  & TO_0 then abort ← 1
if a > EXTS(SI)  & TO_1 then abort ← 1
if a = EXTS(SI)  & TO_2 then abort ← 1
if a u< EXTS(SI) & TO_3 then abort ← 1
if a >u EXTS(SI) & TO_4 then abort ← 1

if abort & (MSR_TS = 0b10 | MSR_TS = 0b01) then
           #Transactional or Suspended
   cause ← 0x00000001
   if MSR_TS= 0b01 & TEXASR_FS = 0 then  #Suspended
      Discard transactional footprint
   TMRecordFailure(cause)
   if MSR_TS = 0b10 then                 #Transactional
      TMHandleFailure()
```

The ***tabortdci.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. The contents of register RA are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended then the ***tabortdci.*** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 5.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 5.3.3 (this includes discarding the transactional footprint).

If the transaction state is Suspended, the transactional footprint is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of ***tabortdci.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered:

CR0 TEXASR TFIAR TS

Transaction Suspend or Resume *X-form*

tsr. L

31	///	L	///	///	750	1
0	6	10	11	16	21	31

```
CR0 ← 0 || MSR_TS || 0
if L = 0 then
   if MSR_TS = 0b10 then              #Transactional
      MSR_TS ← 0b01                    #Suspended
else
   if MSR_TS = 0b01                    #Suspended
      MSR_TS ← 0b10                   #Transactional
```

The ***tsr.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. Based on the value of the L field, two variants of ***tsr.*** are used to change the transaction state.

If L = 0, and the transaction state is Transactional, the transaction state is set to Suspended.

If L = 1, and the transaction state is Suspended, the transaction state is set to Transactional.

Other than the setting of CR0, the execution of ***tsr.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered:

CR0 TS

Programming Note

When resuming a transaction that has encountered failure while in the Suspended state, failure handling is performed after the execution of ***tresume.*** and no later than the next failure synchronizing event.

Extended Mnemonics

Examples of extended mnemonics for *Transaction Suspend* or *Resume.*

Extended:	***Equivalent To:***
tsuspend.	tsr. 0
tresume.	tsr. 1

Transaction Check *X-form*

tcheck BF

31	BF	//	///	///	718	/
0	6	9	11	16	21	31

```
if MSR_TS = 0b10 | MSR_TS = 0b01 then   #Transactional
                                        #or Suspended
  for each load caused by an instruction following
  the outer tbegin and preceding this tcheck
    if (Load instruction was executed in T state
       with TEXASR_ROT=0 or accessing a location
       previously stored transactionally) |
       (Load instruction was executed in S state
        with TEXASR_ROT=0 and accessed a location
        previously accessed transactionally)|
       (Load instruction was executed in S state
        with TEXASR_ROT=1 and accessed a location
        previously stored transactionally)
    then wait until load has been performed with
    respect to all processors and mechanisms
CR field BF ← TDOOMED || MSR_TS || 0
```

If the transaction state is Transactional or Suspended, the ***tcheck*** instruction ensures that all loads that are caused by instructions that follow the outer *tbegin.* instruction and precede the ***tcheck*** instruction and satisfy one of the following properties, have been performed with respect to all processors and mechanisms.

- The load is caused by an instruction that was executed in Transactional state, either while $TEXASR_{ROT}$=0 or accessing a location previously stored transactionally.
- The load is caused by an instruction that was executed in Suspended state while $TEXASR_{ROT}$=0 and accesses a location that was accessed transactionally.
- The load is caused by an instruction that was executed in Suspended state while $TEXASR_{ROT}$=1 and accesses a location that was stored transactionally.

The ***tcheck*** instruction then copies the TDOOMED bit into bit 0 of CR field BF, copies MSR_{TS} to bits 1:2 of CR field BF, and sets bit 3 of CR field BF to 0.

Other than the setting of CR field BF, execution of ***tcheck*** in the Non-transactional state is treated as a no-op.

Special Registers Altered:

CR field BF

Programming Note

One use of the ***tcheck*** instruction in Suspended state is to determine whether preceding loads from transactionally modified locations have returned the data the transaction stored. (If the transaction has failed, some of the loads may have returned a more recent value that was stored by a conflicting store, or may have returned the pre-transaction contents of the location.). It is important to use tcheck. between any Suspended state loads that might access transactionally modified locations and subsequent computation using the Suspended-state-loaded data. Otherwise, corrupt data could cause problems such as wild branches or infinite loops.

Another use of ***tcheck*** in Suspended state is to determine whether the contents of storage, as seen in Suspended state, are consistent with the transaction succeeding -- e.g., whether no location that has been accessed transactionally (stored transactionally, for ROTs), and has been seen in Suspended state, has been subject to a conflict thus far. (A location is seen in Suspended state either by being loaded in Suspended state or by being loaded in Transactional state and the value (or a value derived therefrom) passed, in a register, into Suspended state.)

A use of ***tcheck*** in Transactional state is to determine whether the transaction still has the potential to succeed.

Note that ***tcheck*** provides an instantaneous check on the integrity of a subset of the accesses performed within a transaction. ***tcheck*** is not a failure synchronizing mechanism. Even if no accesses follow the ***tcheck***, there may still be latent failures that haven't been recorded, for example caused by accesses that ***tcheck*** does not wait for, by external conflicts that will happen in the future, or simply by time of flight to the failure detection mechanism for operations that have already been performed.

Programming Note

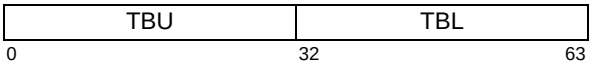
The ***tcheck*** instruction can return 1 in bit 0 of CR field BF before the failure has been recorded in TEXASR and TFIAR.

Programming Note

The ***tcheck*** instruction may cause pipeline synchronization. As a result, programs that use ***tcheck*** excessively may perform poorly.

Chapter 6. Time Base

The Time Base (TB) is a 64-bit register (see Figure 9) containing a 64-bit unsigned integer that is incremented periodically as described below.



Field	Description
TBU	Upper 32 bits of Time Base
TBL	Lower 32 bits of Time Base

Figure 9. Time Base

The Time Base monotonically increments until its value becomes 0xFFFF_FFFF_FFFF_FFFF (2^{64} - 1); at the next increment its value becomes 0x0000_0000_0000_0000. There is no interrupt or other indication when this occurs.

The suggested frequency at which the time base increments is 512 MHz, however, variation from this rate is allowed provided the following requirements are met.

- The contents of the Time Base differ by no more than +/- four counts from what they would be if they incremented at the required frequency.
- Bit 63 of the Time Base is set to 1 between 30% and 70% of the time over any time interval of at least 16 counts.

The Power ISA does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock. The Time Base update frequency is not required to be constant. What *is* required, so that system software can keep time of day and operate interval timers, is one of the following.

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the Time Base changes, and a means to determine what the current update frequency is.
- The update frequency of the Time Base is under the control of the system software.

Programming Note

If the operating system initializes the Time Base on power-on to some reasonable value and the update frequency of the Time Base is constant, the Time Base can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the Time Base are monotonically increasing (except when the Time Base wraps from 2^{64}-1 to 0). If a trace entry is recorded each time the update frequency changes, the sequence of Time Base values can be post-processed to become actual time values.

Successive readings of the Time Base may return identical values.

6.1 Time Base Instructions

Move From Time Base *XFX-form*

mftb RT,TBR
[Phased-Out]

31	RT	tbr	371	/
0	6	11	21	31

This instruction behaves as if it were an ***mfspr*** instruction; see the ***mfspr*** instruction description in Section 3.3.17 of Book I.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics for *Move From Time Base*:

Extended:	Equivalent to:
mftb Rx	mftb Rx,268 mfspr Rx,268
mftbu Rx	mftb Rx,269 mfspr Rx,269

Programming Note

New programs should use ***mfspr*** instead of ***mftb*** to access the Time Base.

Programming Note

mftb serves as both a basic and an extended mnemonic. The Assembler will recognize an ***mftb*** mnemonic with two operands as the basic form, and an ***mftb*** mnemonic with one operand as the extended form. In the extended form the TBR operand is omitted and assumed to be 268 (the value that corresponds to TB).

Programming Note

The ***mfspr*** instruction can be used to read the Time Base on all processors that comply with Version 2.01 of the architecture or with any subsequent version.

It is believed that the ***mfspr*** instruction can be used to read the Time Base on most processors that comply with versions of the architecture that precede Version 2.01. Processors for which ***mfspr*** cannot be used to read the Time Base include the following.

- 601
- POWER3

(601 implements neither the Time Base nor ***mftb***, but depends on software using ***mftb*** to read the Time Base, so that the attempt causes the Illegal Instruction error handler to be invoked and thereby permits the operating system to emulate the Time Base.)

Programming Note

Since the update frequency of the Time Base is implementation-dependent, the algorithm for converting the current value in the Time Base to time of day is also implementation-dependent.

As an example, assume that the Time Base increments at the constant rate of 512 MHz. (Note, however, that programs should allow for the possibility that some implementations may not increment the least-significant 4 bits of the Time Base at a constant rate.) What is wanted is the pair of 32-bit values comprising a POSIX standard clock:[1] the number of whole seconds that have passed since 00:00:00 January 1, 1970, UTC, and the remaining fraction of a second expressed as a number of nanoseconds.

Assume that:

- The value 0 in the Time Base represents the start time of the POSIX clock (if this is not true, a simple 64-bit subtraction will make it so).
- The integer constant *ticks_per_sec* contains the value 512,000,000, which is the number of times the Time Base is updated each second.
- The integer constant *ns_adj* contains the value

  $$\frac{1{,}000{,}000{,}000}{512{,}000{,}000} \times 2^{32} / 2 = 4194304000$$

  which is the number of nanoseconds per tick of the Time Base, multiplied by 2^{32} for use in ***mulhwu*** (see below), and then divided by 2 in order to fit, as an unsigned integer, into 32 bits.

When the processor is in 64-bit mode, The POSIX clock can be computed with an instruction sequence such as this:

```
mfspr  Ry,268  # Ry = Time Base
lwz    Rx,ticks_per_sec
divdu  Rz,Ry,Rx   # Rz = whole seconds
stw    Rz,posix_sec
mulld  Rz,Rz,Rx   # Rz = quotient * divisor
sub    Rz,Ry,Rz   # Rz = excess ticks
lwz    Rx,ns_adj
slwi   Rz,Rz,1 # Rz = 2 * excess ticks
mulhwu Rz,Rz,Rx   # mul by (ns/tick)/2 * 2^32
stw    Rz,posix_ns# product[0:31] = excess ns
```

Non-constant update frequency

In a system in which the update frequency of the Time Base may change over time, it is not possible to convert an isolated Time Base value into time of day. Instead, a Time Base value has meaning only with respect to the current update frequency and the time of day that the update frequency was last changed. Each time the update frequency changes, either the system software is notified of the change via an interrupt (see Book III), or the change was instigated by the system software itself. At each such change, the system software must compute the current time of day using the old update frequency, compute a new value of *ticks_per_sec* for the new frequency, and save the time of day, Time Base value, and tick rate. Subsequent calls to compute Time of Day use the current Time Base Value and the saved value.

1. Described in POSIX Draft Standard P1003.4/D12, *Draft Standard for Information Technology -- Portable Operating System Interface (POSIX) -- Part 1: System Application Program Interface (API) - Amendment 1: Real-time Extension* [*C Language*]. Institute of Electrical and Electronics Engineers, Inc., Feb. 1992.

Chapter 7. Event-Based Branch Facility

7.1 Event-Based Branch Overview

The Event-Based Branch facility allows application programs to enable hardware to change the effective address of the next instruction to be executed when certain events occur to an effective address specified by the program.

The operation of the Event-Based Branch facility is summarized as follows:

- The Event-Based Branch facility is available only when the system software has made it available. See Section 9.5 of Book III for additional information.
- When the Event-Based Branch facility is available, event-based branches are caused by event-based exceptions. Event-based exceptions can be enabled to occur by setting bits in the BESCR.
- When an event-based exception occurs, the bit in the BESCR control field corresponding to the event-based exception is set to 0 and the bit in the Event Status field in the BESCR corresponding to the event-based exception is set to 1.
- If the global enable bit in the BESCR is set to 1 when any of the bits in the status field are set to 1 (i.e., when an event-based exception exists), an event-based branch occurs.
- The event-based branch causes the following to occur.
  - The global enable bit is set to 0.
  - The TS field of the BESCR is set to indicate the transaction state of the processor when the event-based branch occurred; if the processor was in Transactional state when the event-based branch occurred, it is put into Suspended state.
  - Bits 0:61 of the EBBRR are set to the effective address of the instruction that would have attempted to execute next if the event-based branch did not occur.
  - Instruction fetch and execution continues at the effective address contained in the EBBHR.
- The event-based branch handler performs the necessary processing in response to the event, and then executes an ***rfebb*** instruction in order to resume execution at the instruction at the address indicated in the EBBRR. The ***rfebb*** instruction also restores the processor to the transaction state indicated by BESCR$_{TS}$. See the Programming Notes in Section 7.3 for an example sequence of operations of the event-based branch handler.

Additional information about the Event-Based Branch facility is given in Section 3.4 of Book III.

Programming Note

Since system software controls the availability of the Event-Based Branch facility (see Section 9.5 of Book III), an interface must be provided that enables applications to request access to the facility and determine when it is available.

Programming Note

In order to initialize the Event-Based Branch facility for Performance Monitor event-based exceptions, software performs the following operations.

- Software requests control of the Event-Based Branch facility from the system software.
- Software requests the system software to initialize the Performance Monitor as desired.
- Software sets the EBBHR to the effective address of the event-based branch handler.
- Software enables Performance Monitor event-based exceptions by setting $BESCR_{PME\ PMEO}$ = 1 0, and also sets $MMCR0_{PMAE\ PMAO}$ = 1 0. See Section 9.4.4 of Book III for the description of MMCR0.
- Software sets the GE bit in the BESCR to enable event-based branches.

Initializing the Event-Based Branch facility for External EBB exceptions follows a similar process except that EBB exceptons for these facilities are controlled by different bits in the BESCR.

7.2 Event-Based Branch Registers

7.2.1 Branch Event Status and Control Register

The Branch Event Status and Control Register (BESCR) is a 64-bit register that contains control and status information about the Event-Based Branch facility.

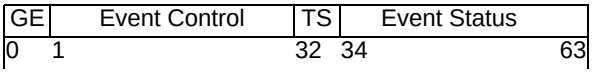


Figure 10. Branch Event Status and Control Register (BESCR)

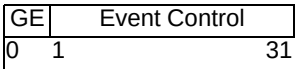


Figure 11. Branch Event Status and Control Register Upper (BESCRU)

System software controls whether or not event-based branches occur regardless of the contents of the BESCR. See Section 9.4.4 of Book III and Section 6.2.12 of Book III.

The entire BESCR can be read or written using SPR 806. Individual bits of the BESCR can be set or reset using two sets of additional SPR numbers.

- When ***mtspr*** indicates SPR 800 (Branch Event Status and Control Set, or BESCRS), the bits in BESCR which correspond to "1" bits in the source register are set to 1; all other bits in the BESCR are unaffected. SPR 801 (BESCRSU) provides the same capability to each of the upper 32 bits of the BESCR.
- When ***mtspr*** indicates SPR 802 (Branch Event Status and Control Reset, or BESCRR), the bits in BESCR which correspond to "1" bits in the source register are set to 0; all other bits in the BESCR are unaffected. SPR 803 (BESCRRU) provides the same capability to each of the upper 32 bits of the BESCR.

When ***mfspr*** indicates any of the above SPR numbers, the current value of the register is returned.

Programming Note

Event-based branch handlers typically reset event status bits upon entry, and enable event enable bits after processing an event. Execution of ***rfebb*** then re-enables the GE bit so that additional event-based branches can occur.

0 **Global Enable (GE)**

0 Event-based branches are disabled
1 Event-based branches are enabled.

When an event-based branch occurs, GE is set to 0 and is not altered by hardware until ***rfebb*** 1 is executed or software sets GE=1 and another event-based branch occurs.

1:31 **Event Control**

1:29 **Reserved**

30 **External Event-Based Exception Enable (EE)**

0 External event-based (EBB) exceptions are disabled.
1 External EBB exceptions are enabled until an external event-based exception occurs, at which time:
- EE is set to 0
- EEO is set to 1

External event-based exceptions exist in any privilege state when an external EBB input from the platform is active. See the system documentation for information about the external EBB input.

31 **Performance Monitor Event-Based Exception Enable (PME)**
- 0 Performance Monitor event-based exceptions are disabled.
- 1 Performance Monitor event-based exceptions are enabled until a Performance Monitor event-based exception occurs, at which time:
  - PME is set to 0
  - PMEO is set to 1

See Chapter 9 of Book III for information about Performance Monitor event-based exceptions and about the effects of this bit on the Performance Monitor.

Programming Note

Performance Monitor event-based exceptions can only occur in problem state. See Section 9.2 of Book III.

32:33 **Transaction State (TS)**

When an event-based branch occurs, hardware sets this field to indicate the transaction state of the processor when the event-based branch occurred.

The values and their associated meanings are as follows.

00 Non-transactional
01 Suspended
10 Transactional
11 Reserved

$BESCR_{TS}$ is part of the Transactional Memory facility. (The entire BESCR is part of the Event-Based Branch facility.)

Programming Note

Event-based branch handlers should not modify this field since its value is used by the processor to determine the transaction state of the processor after the ***rfebb*** instruction is executed.

34:63 **Event Status**

34:61 **Reserved**

62 **External Event-Based Exception Occurred (EEO)**
- 0 An external EBB exception has not occurred since the last time software set this bit to 0.
- 1 An external EBB exception has occurred since the last time software set this bit to 0.

Programming Note

As part of processing an External EBB exception, it may also be necessary to perform additional operations to manage the external EBB input from the system. See the system documentation for details.

63 **Performance Monitor Event-Based Exception Occurred (PMEO)**
- 0 A Performance Monitor event-based exception has not occurred since the last time software set this bit to 0.
- 1 A Performance Monitor event-based exception has occurred since the last time software set this bit to 0.

This bit is set to 1 by the hardware when a Performance Monitor event-based exception occurs. This bit can be set to 0 only by the ***mtspr*** instruction.

See Chapter 9 of Book III for information about Performance Monitor event-based exceptions and about the effects of this bit on the Performance Monitor.

Programming Note

After handling an event-based branch, software should set the “exception occurred” bit(s) corresponding to the event-based exception(s) that have occurred to 0. See the Programming Notes in Section 7.3 for additional information.

7.2.2 Event-Based Branch Handler Register

The Event-Based Branch Handler Register (EBBHR) is a 64-bit register register that contains the 62 most significant bits of the effective address of the instruction that is executed next after an event-based branch occurs. Bits 62:63 must be available to be read and written by software.

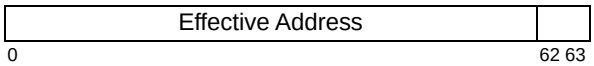


Figure 12. Event-Based Branch Handler Register (EBBHR)

Programming Note

The EBBHR can be used by software as a scratch-pad register after entry into an event-based branch handler, provided that its contents are restored prior to executing ***rfebb*** 1. An example of such usage is as follows. In the example, SPRG3 is used to contain a pointer to a storage area where private application data may be saved, however, refer to the applicable operating system documentation to determine if an alternate register or storage area should be used.

```
E:mtspr EBBHR, r1      // Save r1 in EBBHR
  mfspr r1, SPRG3      // Move SPRG3 to r1
  std r2, r1,offset1   // Store r2
  mfspr EBBHR,r2       // Copy original contents
                       // of r1 to r2
  std r2,offset2(r1)   // save original r1
  ..                   // Store rest of state
 ...                    // Process event(s)
 ...                  // Restore all state except
                       // r1,r2
  r2 = &E             // Generate original value
                       // of EBBHR in r2
  mtspr EBBHR,r2       // Restore EBBHR
  ld r2 offset1(r1)    // restore r2
  ld r1 offset2(r1)    // restore r1
  rfebb 1              // Return from handler
```

7.2.3 Event-Based Branch Return Register

The Event-Based Branch Return Register (EBBRR) is a 64-bit register that contains the 62 most significant bits of an instruction effective address as specified below.

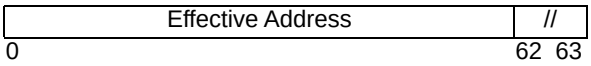


Figure 13. Event-Based Branch Return Register (EBBRR)

When an event-based branch occurs, bits 0:61 of the EBBRR are set to the effective address of the instruction that would have attempted to execute next if the event-based branch did not occur.

Bits 62:63 are reserved.

7.3 Event-Based Branch Instructions

Return from Event-Based Branch *XL-form*

rfebb S

19	///	///	///	S	146	/
0	6	11	16	20	21	31

$BESCR_{GE}$ ← S
MSR_{TS} ← $BESCR_{TS}$
NIA ←$_{iea}$ $EBBRR_{0:61}$ || 0b00

$BESCR_{GE}$ is set to S. The processor is placed in the transaction state indicated by $BESCR_{TS}$.

If there are no pending event-based exceptions, then the next instruction is fetched from the address $EBBRR_{0:61}$ || 0b00 (when MSR_{SF}=1) or $^{32}0$ || $EBBRR_{32:61}$ || 0b00 (when MSR_{SF}=0). If one or more pending event-based exceptions exist, an event-based branch is generated; in this case the value placed into EBBRR by the Event-Based Branch facility is the address of the instruction that would have been executed next had the event-based branch not occurred.

See Section 3.4 of Book III for additional information about this instruction.

Special Registers Altered:

BESCR
MSR (See Book III)

Extended Mnemonics:

Extended:	Equivalent to:
rfebb	rfebb 1

Programming Note

rfebb serves as both a basic and an extended mnemonic. The Assembler will recognize an ***rfebb*** mnemonic with one operand as the basic form, and an ***rfebb*** mnemonic with no operand as the extended form. In the extended form, the S operand is omitted and assumed to be 1.

Programming Note

If the $BESCR_{TS}$ has been modified by software after an event-based branch occurs, an illegal transaction state transition may occur. See Chapter 3.2.2 of Book III.

Programming Note

When an event-based branch occurs, the event-based branch handler can execute the following sequence of operations. This sequence of operations assumes that the handler routine has access to a stack or other area in memory in which state information from the main program can be stored. Note also that in this example, the handler entry point is labeled “E,” r1 and r2 are used as scratch registers, and both external EBB and Performance Monitor EBB exceptions are enabled.

```
E:Save  state          // This is the entry pt
  mfspr r1, BESCR      // Check event status
  if r1_63=1, then
  Process PM exception
   r2 ← 0x0000 0000 0000 0001
   mtspr BESCRR, r2   //Reset PMEO status bit
   r2 ← 0x0000 0001 0000 0000
   mtspr BESCRS, r1   //Re-enable PM exceptions
         //Note: The PMAE bit of MMCR0 must also
         //      be enabled. See Book III.
  if r1_62=1, then
  Process external exception
   r2 ← 0x0000 0000 0000 0002
   mtspr BESCRR, r2   //Reset EEO status bit
   r2 ← 0x0000 0002 0000 0000
                     // De-activate external EBB
                        input from platform
   mtspr BESCRS, r1  //Re-enable external EBB
                       exceptions
                     // .  .  .
                     //Other exceptions
                     //are processed similarly.
                     // .  .  .
  Restore state
  rfebb 1             // return & global enable
```

Note that before resetting the $BESCR_{EEO}$, the external EBB input from the platform should be deactivated, and additional operations to manage the external EBB input may be required. See the system documentation for details.

In the above sequence, if other exceptions occur after they are enabled, another event-based branch will occur immediately after ***rfebb*** is executed.

Chapter 8. Branch History Rolling Buffer

The Branch History Rolling Buffer (BHRB) is a buffer containing an implementation-dependent number of entries, referred to as BHRB Entries (BHRBEs), that contain information related to branches that have been taken. Entries are numbered from 0 through n, where n is implementation-dependent but no more than 1023. Entry 0 is the most-recently written entry. The BHRB is read by means of the ***mfbhrbe*** instruction.

System software typically controls the availability of the BHRB as well as the number of entries that it contains. If the BHRB is accessed when it is unavailable, the system facility unavailable error handler is invoked.

Various events or actions by the system software may result in the BHRB occasionally being cleared. If BHRB entries are read after this has occurred, 0s will be returned. See the description of the ***mfbhrbe*** instruction for additional information.

The BHRB is typically used in conjunction with Performance Monitor event-based branches. (See Chapter 7 of Book II.) When used in conjunction with this facility, $BESCR_{PME}$ is set to 1 to enable Performance Monitor event-based exceptions, and Performance Monitor alerts are enabled to enable the writing of BHRB entries. When a Performance Monitor alert occurs, Performance Monitor alerts are disabled, BHRB entries are no longer written, and an event-based branch occurs. (See Chapter 9 of Book III for additional information on the Performance Monitor.) The event-based branch handler can then access the contents of the BHRB for analysis.

When the BHRB is written by hardware, only those *Branch* instructions that meet the filtering criteria are written. See Section 9.4.7 of Book III.

The following paragraphs describe the entries written into the BHRB for various types of *Branch* instructions for which the branch was taken. In some circumstances, however, the hardware may be unable to make the entry even though the following paragraphs require it. In such cases, the hardware sets the EA field to 0, and indicates any missed entries using the T and P fields. (See Section 8.1.)

When an I-form or B-form *Branch* instruction is entered into the BHRB, bits 0:61 of the effective address of the *Branch* instruction are written into the next available entry, except that the entry may or may not be written in the following cases.

- The effective address of the branch target exceeds the effective address of the *Branch* instruction by 4.
- The instruction is a B-form *Branch*, the effective address of the branch target exceeds the effective address of the *Branch* instruction by 8, and the instruction immediately following the *Branch* instruction is not another *Branch* instruction.

The determination of whether the effective address of the branch target exceeds the effective address of the Branch instruction by 4 or 8 is made modulo 2^{64}.

Programming Note

The cases described above, for which the BHRBE need not be written, are cases for which some implementations may optimize the execution of the Branch instruction (first case) or of the Branch instruction and the following instruction (second case) in a manner that makes writing the BHRBE difficult. Such implementations may provide a means by which system software can disable these optimizations, thereby ensuring that the corresponding BHRBEs are written normally.

When an XL-form *Branch* instruction is entered into the BHRB, bits 0:61 of the effective address of the *Branch* instruction are written into the next available entry if allowed by the filtering mode; subsequently, bits 0:61 of the effective address of the branch target are written into the following entry.

BHRB entries are written as described above without regard to transaction state and are not removed due to transaction failures.

8.1 Branch History Rolling Buffer Entry Format

Branch History Rolling Buffer Entries (BHRBEs) have the following format.

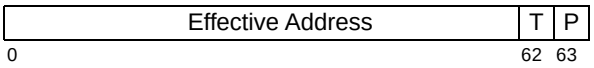


Figure 14. Branch History Rolling Buffer Entry

0:61 **Effective Address (EA)**
When this field is set to a non-zero value, it contains bits 0:61 of the effective address of the instruction indicated by the T field; otherwise this field indicates that the entry is a marker with the meaning specified by the T and P fields.

When the EA field contains a non-zero value, bits 62:63 have the following meanings.

62 **Target Address (T)**

0 The EA field contains bits 0:61 of the effective address of a *Branch* instruction for which the branch was taken.
1 The EA field contains bits 0:61 of the branch effective address of the branch target of an XL-form *Branch* instruction for which the branch was taken.

63 **Prediction (P)**
When T=0, this field has the following meaning.

0 The outcome of the *Branch* instruction was correctly predicted.
1 The outcome of the *Branch* instruction was mispredicted.

When T=1, this field has the following meaning.
0 The Branch instruction was predicted to be taken and the target address was predicted correctly, or the target address was not predicted because the branch was predicted to be not taken.
1 The target address was mispredicted.

When the EA field contains a zero value, bits 62:63 specify the type of marker as described below.

Programming Note

It is expected that programs will not contain *Branch* instructions with instruction or target effective address equal to 0. If such instructions exist, programs cannot distinguish between entries that are markers and entries that correspond to instructions with instruction or target effective address 0.

Value	Meaning
00	This entry either is not implemented or has been cleared. There are no valid entries beyond the current entry.
01-11	Reserved.

8.2 Branch History Rolling Buffer Instructions

The Branch History Rolling Buffer instructions enable application programs to clear and read the BHRB. The availability of these instructions is controlled by the system software. (See Chapter 9 of Book III.) When an attempt is made to execute these instructions when they are unavailable, the system facility unavailable error handler is invoked.

Clear BHRB *X-form*

clrbhrb

31	///	///	///	430	/
0	6	11	16	21	31

```
for n = 0 to (number_of_BHRBEs implemented - 1)
  BHRB(n) ← 0
```

All BHRB entries are set to 0s.

Special Registers Altered:

None.

Move From Branch History Rolling Buffer Entry *XFX-form*

mfbhrbe RT,BHRBE

31	RT	BHRBE	302	/
0	6	11	21	31

```
n ← BHRBE0:9
If n < number of BHRBEs implemented then
 RT ← BHRBE(n)
else
 RT ← 640
```

The BHRBE field denotes an entry in the BHRB. If the designated entry is within the range of BHRB entries implemented and Performance Monitor alterts are disable (see Section 9.5 of Book III), the contents of the designated BHRB entry are placed into register RT; otherwise, $^{64}0$s are placed into register RT.

In order to ensure that the current BHRB contents are read by this instruction, one of the following must have occurred prior to this instruction and after all previous *Branch* and ***clrbhrb*** instructions have completed.

- an event-based branch has occurred
- an ***rfebb*** (see Chapter 7 of Book II) has been executed
- a context synchronizing event (see Section 1.5 of Book III) other than ***isync*** (see Section 4.6.1 of Book II) has occurred.

Special Registers Altered:

None

Programming Note

In order to read all the BHRB entries containing information about taken branches, software should read the entries starting from entry number 0 and continuing until an entry containing all 0s is read or until all implemented BHRB entries have been read.

Since the number of BHRB entries may decrease or the BHRB may be cleared at any time, if a given entry, m, is read as not containing all 0s and is read again subsequently, the subsequent read may return all 0s even though the program has not executed ***clrbhrb***.

Appendix A. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided for certain instructions. This appendix defines extended mnemonics and symbols related to instructions defined in Book II. Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

A.1 Data Cache Block Touch [for Store] Mnemonics

The TH field in the *Data Cache Block Touch* and *Data Cache Block Touch for Store* instructions control the actions performed by the instructions. Extended mnemonics are provided that represent the TH value in the mnemonic rather than requiring it to be coded as a numeric operand.

dcbtct RA,RB,TH	(equivalent to: dcbt for TH values of 0b00000 - 0b00111); other TH values are invalid.
dcbtds RA,RB,TH	(equivalent to: dcbt for TH values of 0b00000 or 0b01000 - 0b01111); other TH values are invalid.
dcbtt RA,RB	(equivalent to: dcbt for TH value of 0b10000)
dcbna RA,RB	(equivalent to: dcbt for TH value of 0b10001)
dcbtstct RA,RB,TH	(equivalent to: dcbtst for TH values of 0b00000 or 0b00000 - 0b00111); other TH values are invalid.
dcbtstds RA,RB,TH	(equivalent to: dcbtst for TH values of 0b00000 or 0b01000 - 0b01111); other TH values are invalid.
dcbtstt RA,RB	(equivalent to: dcbtst for TH value of 0b10000)

A.2 Data Cache Block Flush Mnemonics

The L field in the *Data Cache Block Flush* instruction controls the scope of the flush function performed by the instruction. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

Note: ***dcbf*** serves as both a basic and an extended mnemonic. The Assembler will recognize a ***dcbf*** mnemonic with three operands as the basic form, and a ***dcbf*** mnemonic with two operands as the extended form. In the extended form the L operand is omitted and assumed to be 0.

dcbf RA,RB	(equivalent to: dcbf RA,RB,0)
dcbfl RA,RB	(equivalent to: dcbf RA,RB,1)
dcbflp RA,RB	(equivalent to: dcbf RA,RB,3)

A.3 Or Mnemonics

The three register fields in the ***or*** instruction can be used to specify a hint indicating how the processor should handle stores caused by previous *Store* or ***dcbz*** instructions. An extended mnemonic is supported that represents the operand values in the mnemonic rather than requiring them to be coded as numeric operands.

miso	(equivalent to: or 26,26,26)

A.4 Load and Reserve Mnemonics

The EH field in the *Load and Reserve* instructions provides a hint regarding the type of algorithm implemented by the instruction sequence being executed. Extended mnemonics are provided that allow the EH value to be omitted and assumed to be 0b0.

Note: ***lbarx***, ***lharx***, ***lwarx, ldarx,*** and ***lqarx*** serve as both basic and extended mnemonics. The Assembler will recognize these mnemonics with four operands as the basic form, and these mnemonics with three oper-

ands as the extended form. In the extended form the EH operand is omitted and assumed to be 0.

lbarx	RT,RA,RB	(equivalent to: lbarx RT,RA,RB,0)
lharx	RT,RA,RB	(equivalent to: lharx RT,RA,RB,0)
lwarx	RT,RA,RB	(equivalent to: lwarx RT,RA,RB,0)
ldarx	RT,RA,RB	(equivalent to: ldarx RT,RA,RB,0)
lqarx	RT,RA,RB	(equivalent to: lqarx RT,RA,RB,0)

A.5 Synchronize Mnemonics

The L field in the *Synchronize* instruction controls the scope of the synchronization function performed by the instruction. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand. Two extended mnemonics are provided for the L=0 value in order to support Assemblers that do not recognize the ***sync*** mnemonic.

Note: *sync* serves as both a basic and an extended mnemonic. Assemblers will recognize a ***sync*** mnemonic with one operand as the basic form, and a ***sync*** mnemonic with no operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

sync	(equivalent to:	sync	0)
lwsync	(equivalent to:	sync	1)
ptesync	(equivalent to:	sync	2)

A.6 Wait Mnemonics

The WC field in the ***wait*** instruction is reserved for future use. It may be be used in the future to indicate the condition that causes instruction execution to resume. An extended mnemonic is provided that represent the WC value in the mnemonic rather than requiring it to be coded as a numeric operand.

Note: *wait* serves as both a basic and an extended mnemonic. The Assembler will recognize a ***wait*** mnemonic with one operand as the basic form, and a ***wait*** mnemonic with no operands as the extended form. In the extended form the WC operand is omitted and assumed to be 0.

wait	(equivalent to: wait 0)

A.7 Transactional Memory Instruction Mnemics

The A field in the *Transaction End* instruction controls whether the instruction ends only the current (possibly nested) transaction or the entire set of nested transactions. Extended mnemonics are provided that represent the A value in the mnemonic rather than requiring it to be coded as a numeric operand..

tend.	(equivalent to: tend. 0)
tendall.	(equivalent to: tend. 1)

The L field in the *Transaction Suspend or Resume* instruction determines how to change the transaction state. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

tsuspend.	(equivalent to: tsr. 0)
tresume.	(equivalent to: tsr. 1)

A.8 Move To/From Time Base Mnemonics

The tbr field in the *Move From Time Base* instruction specifies whether the instruction reads the entire Time Base or only the high-order half of the Time Base.

mftb Rx	(equivalent to: mftb Rx,268) or: mfspr Rx,268
mftbu Rx	(equivalent to: mftb Rx,269) or: mfspr Rx,269

A.9 Return From Event-Based Branch Mnemonic

The S field in the *Return from Event-Based Branch* instruction specifies the value to which the instruction sets the GE field in the BESCR. Extended mnemonics are provided that represent the S value in the mnemonic rather than requiring it to be coded as a numeric operand.

rfebb	(equivalent to: rfebb 1)

Note: *rfebb* serves as both a basic and an extended mnemonic. The Assembler will recognize this mnemonic with one operand as the basic form, and this mnemonic with no operands as the extended form. In the extended form the S operand is omitted and assumed to be 1.

Appendix B. Programming Examples for Sharing Storage

This appendix gives examples of how dependencies and the *Synchronization* instructions can be used to control storage access ordering when storage is shared between programs.

Many of the examples use extended mnemonics (e.g., ***bne***, ***bne-***, ***cmpw***) that are defined in Appendix C of Book I.

Many of the examples use the *Load And Reserve* and *Store Conditional* instructions, in a sequence that begins with a *Load And Reserve* instruction and ends with a *Store Conditional* instruction (specifying the same storage location as the *Load Conditional*) followed by a *Branch Conditional* instruction that tests whether the *Store Conditional* instruction succeeded.

In these examples it is assumed that contention for the shared resource is low; the conditional branches are optimized for this case by using "+" and "-" suffixes appropriately.

The examples deal with words; they can be used for doublewords by changing all word-specific mnemonics to the corresponding doubleword-specific mnemonics (e.g., ***lwarx*** to ***ldarx***, ***cmpw*** to ***cmpd***).

In this appendix it is assumed that all shared storage locations are in storage that is Memory Coherence Required, and that the storage locations specified by *Load And Reserve* and *Store Conditional* instructions are in storage that is neither Write Through Required nor Caching Inhibited.

B.1 Atomic Update Primitives

This section gives examples of how the *Load And Reserve* and *Store Conditional* instructions can be used to emulate atomic read/modify/write operations.

An atomic read/modify/write operation reads a storage location and writes its next value, which may be a function of its current value, all as a single atomic operation. The examples shown provide the effect of an atomic read/modify/write operation, but use several instructions rather than a single atomic instruction.

Fetch and No-op

The "Fetch and No-op" primitive atomically loads the current value in a word in storage.

In this example it is assumed that the address of the word to be loaded is in GPR 3 and the data loaded are returned in GPR 4.

```
loop:
   lwarx  r4,0,r3 #load and reserve
   stwcx. r4,0,r3 #store old value if
                  #  still reserved
   bne-   loop    #loop if lost reservation
```

Note:

1. The ***stwcx***., if it succeeds, stores to the target location the same value that was loaded by the preceding ***lwarx***. While the store is redundant with respect to the value in the location, its success ensures that the value loaded by the ***lwarx*** is still the current value at the time the ***stwcx.*** is executed.

Fetch and Store

The "Fetch and Store" primitive atomically loads and replaces a word in storage.

In this example it is assumed that the address of the word to be loaded and replaced is in GPR 3, the new value is in GPR 4, and the old value is returned in GPR 5.

```
loop:
   lwarx  r5,0,r3 #load and reserve
   stwcx. r4,0,r3 #store new value if
                  #  still reserved
   bne-   loop    loop if lost reservation
```

Fetch and Add

The “Fetch and Add” primitive atomically increments a word in storage.

In this example it is assumed that the address of the word to be incremented is in GPR 3, the increment is in GPR 4, and the old value is returned in GPR 5.

```
loop:
   lwarx  r5,0,r3 #load and reserve
   add    r0,r4,r5#increment word
   stwcx. r0,0,r3 #store new value if still res’ved
   bne-   loop    #loop if lost reservation
```

Fetch and AND

The “Fetch and AND” primitive atomically ANDs a value into a word in storage.

In this example it is assumed that the address of the word to be ANDed is in GPR 3, the value to AND into it is in GPR 4, and the old value is returned in GPR 5.

```
loop:
   lwarx  r5,0,r3 #load and reserve
   and    r0,r4,r5#AND word
   stwcx. r0,0,r3 #store new value if still res’ved
   bne-   loop    #loop if lost reservation
```

Note:

1. The sequence given above can be changed to perform another Boolean operation atomically on a word in storage, simply by changing the ***and*** instruction to the desired Boolean instruction (***or***, ***xor***, etc.).

Test and Set

This version of the “Test and Set” primitive atomically loads a word from storage, sets the word in storage to a nonzero value if the value loaded is zero, and sets the EQ bit of CR Field 0 to indicate whether the value loaded is zero.

In this example it is assumed that the address of the word to be tested is in GPR 3, the new value (nonzero) is in GPR 4, and the old value is returned in GPR 5.

```
loop:
   lwarx  r5,0,r3 #load and reserve
   cmpwi  r5,0    #done if word not equal to 0
   bne-   exit
   stwcx. r4,0,r3 #try to store non-0
   bne-   loop    #loop if lost reservation
exit: ...
```

Compare and Swap

The “Compare and Swap” primitive atomically compares a value in a register with a word in storage, if they are equal stores the value from a second register into the word in storage, if they are unequal loads the word from storage into the first register, and sets the EQ bit of CR Field 0 to indicate the result of the comparison.

In this example it is assumed that the address of the word to be tested is in GPR 3, the comparand is in GPR 4 and the old value is returned there, and the new value is in GPR 5.

```
loop:
   lwarx  r6,0,r3 #load and reserve
   cmpw   r4,r6   #1st 2 operands equal?
   bne-   exit    #skip if not
   stwcx. r5,0,r3 #store new value if still res’ved
   bne-   loop    #loop if lost reservation
exit:
   mr     r4,r6   #return value from storage
```

Notes:

1. The semantics given for “Compare and Swap” above are based on those of the IBM System/370 Compare and Swap instruction. Other architectures may define a Compare and Swap instruction differently.
2. “Compare and Swap” is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by ***lwarx*** and ***stwcx.***. A major weakness of a System/370-style Compare and Swap instruction is that, although the instruction itself is atomic, it checks only that the old and current values of the word being tested are equal, with the result that programs that use such a Compare and Swap to control a shared resource can err if the word has been modified and the old value subsequently restored. The sequence shown above has the same weakness.
3. In some applications the second ***bne-*** instruction and/or the ***mr*** instruction can be omitted. The ***bne-*** is needed only if the application requires that if the EQ bit of CR Field 0 on exit indicates “not equal” then (r4) and (r6) are in fact not equal. The ***mr*** is needed only if the application requires that if the comparands are not equal then the word from storage is loaded into the register with which it was compared (rather than into a third register). If either or both of these instructions is omitted, the resulting Compare and Swap does not obey System/370 semantics.

B.2 Lock Acquisition and Release, and Related Techniques

This section gives examples of how dependencies and the *Synchronization* instructions can be used to implement locks, import and export barriers, and similar constructs.

B.2.1 Lock Acquisition and Import Barriers

An "import barrier" is an instruction or sequence of instructions that prevents storage accesses caused by instructions following the barrier from being performed before storage accesses that acquire a lock have been performed. An import barrier can be used to ensure that a shared data structure protected by a lock is not accessed until the lock has been acquired. A ***sync*** instruction can be used as an import barrier, but the approaches shown below will generally yield better performance because they order only the relevant storage accesses.

B.2.1.1 Acquire Lock and Import Shared Storage

If ***lwarx*** and ***stwcx.*** instructions are used to obtain the lock, an import barrier can be constructed by placing an ***isync*** instruction immediately following the loop containing the ***lwarx*** and ***stwcx.***. The following example uses the "Compare and Swap" primitive to acquire the lock.

In this example it is assumed that the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, the value to which the lock should be set is in GPR 5, the old value of the lock is returned in GPR 6, and the address of the shared data structure is in GPR 9.

```
loop:
   lwarx  r6,0,r3,1 #load lock and reserve
   cmpw   r4,r6     #skip ahead if
   bne-   wait      #  lock not free
   stwcx. r5,0,r3   #try to set lock
   bne-   loop      #loop if lost reservation
   isync            #import barrier
   lwz    r7,data1(r9)#load shared data
   .
   .
wait...           #wait for lock to free
```

The hint provided with ***lwarx*** indicates that after the program acquires the lock variable (i.e., ***stwcx***. is successful), it will release it (i.e., store to it) prior to another program attempting to modify it.

The second ***bne-*** does not complete until CR0 has been set by the ***stwcx.***. The ***stwcx.*** does not set CR0 until it has completed (successfully or unsuccessfully). The lock is acquired when the ***stwcx.*** completes successfully. Together, the second ***bne-*** and the subsequent ***isync*** create an import barrier that prevents the load from "data1" from being performed until the branch has been resolved not to be taken.

If the shared data structure is in storage that is neither Write Through Required nor Caching Inhibited, an ***lwsync*** instruction can be used instead of the ***isync*** instruction. If ***lwsync*** is used, the load from "data1" may be performed before the ***stwcx.***. But if the ***stwcx.*** fails, the second branch is taken and the ***lwarx*** is re-executed. If the ***stwcx.*** succeeds, the value returned by the load from "data1" is valid even if the load is performed before the ***stwcx.***, because the ***lwsync*** ensures that the load is performed after the instance of the ***lwarx*** that created the reservation used by the successful ***stwcx.***.

B.2.1.2 Obtain Pointer and Import Shared Storage

If ***lwarx*** and ***stwcx.*** instructions are used to obtain a pointer into a shared data structure, an import barrier is not needed if all the accesses to the shared data structure depend on the value obtained for the pointer. The following example uses the "Fetch and Add" primitive to obtain and increment the pointer.

In this example it is assumed that the address of the pointer is in GPR 3, the value to be added to the pointer is in GPR 4, and the old value of the pointer is returned in GPR 5.

```
loop:
   lwarx  r5,0,r3 #load pointer and reserve
   add    r0,r4,r5#increment the pointer
   stwcx. r0,0,r3 #try to store new value
   bne-   loop    #loop if lost reservation
   lwz    r7,data1(r5) #load shared data
```

The load from "data1" cannot be performed until the pointer value has been loaded into GPR 5 by the ***lwarx***. The load from "data1" may be performed before the ***stwcx.***. But if the ***stwcx.*** fails, the branch is taken and the value returned by the load from "data1" is discarded. If the ***stwcx.*** succeeds, the value returned by the load from "data1" is valid even if the load is performed before the ***stwcx.***, because the load uses the pointer value returned by the instance of the ***lwarx*** that created the reservation used by the successful ***stwcx.***.

An ***isync*** instruction could be placed between the ***bne-*** and the subsequent ***lwz***, but no ***isync*** is needed if all accesses to the shared data structure depend on the value returned by the ***lwarx***.

B.2.2 Lock Release and Export Barriers

An “export barrier” is an instruction or sequence of instructions that prevents the store that releases a lock from being performed before stores caused by instructions preceding the barrier have been performed. An export barrier can be used to ensure that all stores to a shared data structure protected by a lock will be performed with respect to any other processor before the store that releases the lock is performed with respect to that processor.

B.2.2.1 Export Shared Storage and Release Lock

A ***sync*** instruction can be used as an export barrier independent of the storage control attributes (e.g., presence or absence of the Caching Inhibited attribute) of the storage containing the shared data structure. Because the lock must be in storage that is neither Write Through Required nor Caching Inhibited, if the shared data structure is in storage that is Write Through Required or Caching Inhibited a ***sync*** instruction *must* be used as the export barrier.

In this example it is assumed that the shared data structure is in storage that is Caching Inhibited, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

```
stw    r7,data1(r9)#store shared data (last)
sync           #export barrier
stw    r4,lock(r3)#release lock
```

The ***sync*** ensures that the store that releases the lock will not be performed with respect to any other processor until all stores caused by instructions preceding the ***sync*** have been performed with respect to that processor.

B.2.2.2 Export Shared Storage and Release Lock using lwsync

If the shared data structure is in storage that is neither Write Through Required nor Caching Inhibited, an ***lwsync*** instruction can be used as the export barrier. Using ***lwsync*** rather than ***sync*** will yield better performance in most systems.

In this example it is assumed that the shared data structure is in storage that is neither Write Through Required nor Caching Inhibited, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

```
stw    r7,data1(r9)#store shared data (last)
lwsync         #export barrier
stw    r4,lock(r3)#release lock
```

The ***lwsync*** ensures that the store that releases the lock will not be performed with respect to any other processor until all stores caused by instructions preceding the ***lwsync*** have been performed with respect to that processor.

B.2.3 Safe Fetch

If a load must be performed before a subsequent store (e.g., the store that releases a lock protecting a shared data structure), a technique similar to the following can be used.

In this example it is assumed that the address of the storage operand to be loaded is in GPR 3, the contents of the storage operand are returned in GPR 4, and the address of the storage operand to be stored is in GPR 5.

```
lwz    r4,0(r3)#load shared data
cmpw   r4,r4   #set CR0 to "equal"
bne-   $-8     #branch never taken
stw    r7,0(r5)#store other shared data
```

An alternative is to use a technique similar to that described in Section B.2.1.2, by causing the ***stw*** to depend on the value returned by the ***lwz*** and omitting the ***cmpw*** and ***bne-***. The dependency could be created by ANDing the value returned by the ***lwz*** with zero and then adding the result to the value to be stored by the ***stw***. If both storage operands are in storage that is neither Write Through Required nor Caching Inhibited, another alternative is to replace the ***cmpw*** and ***bne-*** with an ***lwsync*** instruction.

B.3 List Insertion

This section shows how the ***lwarx*** and ***stwcx.*** instructions can be used to implement simple insertion into a singly linked list. (Complicated list insertion, in which multiple values must be changed atomically, or in which the correct order of insertion depends on the contents of the elements, cannot be implemented in the manner shown below and requires a more complicated strategy such as using locks.)

The "next element pointer" from the list element after which the new element is to be inserted, here called the "parent element", is stored into the new element, so that the new element points to the next element in the list; this store is performed unconditionally. Then the address of the new element is conditionally stored into the parent element, thereby adding the new element to the list.

In this example it is assumed that the address of the parent element is in GPR 3, the address of the new element is in GPR 4, and the next element pointer is at offset 0 from the start of the element. It is also assumed that the next element pointer of each list element is in a reservation granule separate from that of the next element pointer of all other list elements.

```
loop:
   lwarx  r2,0,r3 #get next pointer
   stw    r2,0(r4)#store in new element
   lwsync or sync #order stw before stwcx
   stwcx. r4,0,r3 #add new element to list
   bne-   loop    #loop if stwcx. failed
```

In the preceding example, if two list elements have next element pointers in the same reservation granule then, in a multiprocessor, "livelock" can occur. (Livelock is a state in which processors interact in a way such that no processor makes forward progress.)

If it is not possible to allocate list elements such that each element's next element pointer is in a different reservation granule, then livelock can be avoided by using the following, more complicated, sequence.

```
   lwz    r2,0(r3)#get next pointer
loop1:
   mr     r5,r2   #keep a copy
   stw    r2,0(r4)#store in new element
   sync           #order stw before stwcx.
                  and before lwarx
loop2:
   lwarx  r2,0,r3 #get it again
   cmpw   r2,r5   #loop if changed (someone
   bne-   loop1   #  else progressed)
   stwcx. r4,0,r3 #add new element to list
   bne-   loop2   #loop if failed
```

In the preceding example, livelock is avoided by the fact that each processor re-executes the ***stw*** only if some other processor has made forward progress.

B.4 Notes

The following notes apply to Section B.1 through Section B.3.

1. To increase the likelihood that forward progress is made, it is important that looping on ***lwarx***/***stwcx.*** pairs be minimized. For example, in the "Test and Set" sequence shown in Section B.1, this is achieved by testing the old value before attempting the store; were the order reversed, more ***stwcx.*** instructions might be executed, and reservations might more often be lost between the ***lwarx*** and the ***stwcx.***
2. The manner in which ***lwarx*** and ***stwcx.*** are communicated to other processors and mechanisms, and between levels of the storage hierarchy within a given processor, is implementation-dependent. In some implementations performance may be improved by minimizing looping on a ***lwarx*** instruction that fails to return a desired value. For example, in the "Test and Set" sequence shown in Section B.1, if the programmer wishes to stay in the loop until the word loaded is zero, he could change the "bne- exit" to "bne- loop". However, in some implementations better performance may be obtained by using an ordinary Load instruction to do the initial checking of the value, as follows.

   ```
   loop:
    lwz   r5,0(r3)#load the word
    cmpwi r5,0    #loop back if word
    bne-  loop    #  not equal to 0
    lwarx r5,0,r3 #try again, reserving
    cmpwi r5,0    #  (likely to succeed)
    bne-  loop
    stwcx.r4,0,r3 #try to store non-0
    bne-  loop    #loop if lost reserv'n
   ```

3. In a multiprocessor, livelock is possible if there is a *Store* instruction (or any other instruction that can clear another processor's reservation; see Section 1.7.4.1) between the ***lwarx*** and the ***stwcx.*** of a ***lwarx***/***stwcx.*** loop and any byte of the storage location specified by the Store is in the reservation granule. For example, the first code sequence shown in Section B.3 can cause livelock if two list elements have next element pointers in the same reservation granule.

B.5 Transactional Lock Elision

This section illustrates the use of the Transactional Memory facility to implement transactional lock elision (TLE), in which lock-based critical sections are speculatively executed as a transaction without first acquiring a lock. This locking protocol is an alternative to the routines described above, yielding increased concurrency when the lock that guards a critical section is frequently unnecessary.

B.5.1 Enter Critical Section

The following example shows the entry point to a critical section using transactional lock elision. The entry code starts a transaction using the *tbegin.* instruction and checks whether the transaction was aborted or not. If not, it checks whether the lock is free or not. If the lock is found to be free, the thread proceeds to execute the critical section.

In this example it is assumed that the address of the lock is in GPR 3, and the value indicating that the lock is free is in GPR 4. The handling of cases of transaction abort and busy lock are described in subsequent examples.

```
tle_entry:
   tbegin.            #Start TLE transaction
   beq- tle_abort     #Handle TLE transaction abort
   lwz r6,0(r3)       #Read lock
   cmpw r6,r4         #Check if lock is free
   bne- busy_lock    #If not, handle lock busy case

critical_section1:
```

B.5.2 Handling Busy Lock

In the event that the lock is already held, by either another thread or the current thread, the transaction is aborted using the *tabort* instruction, using a software-defined code *TLE_BUSY_LOCK* indicating the cause of the abort. The abort returns control to the ***beq*** following ***tbegin.*** in the critical section entrance sequence, allowing for an abort handler to react appropriately.

```
busy_lock:
   li r3, TLE_BUSY_LOCK
   tabort r3             #Abort TLE transaction
```

B.5.3 Handling TLE Abort

A TLE transaction may fail for one of a variety of causes, persistent and transient. Persistent causes are certain—or at least highly likely—to cause future attempts to execute the same transaction to fail. However, for transient causes, it is possible that the failure cause may not be re-encountered in a subsequent attempt. Thus, persistent aborts are handled by taking a non-transactional path that involves the actual acquisition of the lock, while transient aborts retry the critical section using TLE.

The following example illustrates the handling of aborts in TLE. It is assumed that the address of the lock is in GPR 3. The immediate value of the *andis.* instruction selects the Failure Persistent bit in the upper half of TEXASR to be tested.

```
tle_abort:
   mfspr r4, TEXASRU    # Read high-order half
                        # of TEXASR
   andis. r5,r4,0x0100  # determine whether failure
                       # is likely to be persistent
   bne tle_acquire_lock   #Persistent, acquire lock
                          #enter critical sec
   b tle_entry            #Transient, try TLE again
```

This example can be extended to keep track of the number of transient aborts and fall back on the acquisition of the lock after the number of transient failures reaches some threshold. It can also be extended to handle reentrant locks. Acquisition of TLE locks is described in a subsequent example.

B.5.4 TLE Exit Section Critical Path

The following example illustrates the instruction sequence used to exit a TLE critical section. The CR0 value set by ***tend.*** indicates whether the current thread was in a transaction. If so, the exited critical section was entered speculatively, and the transaction is ended. If not, the execution takes a path to release the lock.

Release of an acquired TLE lock is described in a subsequent example.

```
tle_exit:
   tend.                  #End the current trans-
                          #action, if any
   bng- tle_release_lock  #Release lock, if was
                          #not in a transaction
```

B.5.5 Acquisition and Release of TLE Locks

The steps for acquiring and releasing a lock associated with a TLE critical section are identical to those for acquiring and releasing conventional locks that are not elided, as described in Section B.2.1.1 and Section B.2.2 respectively.

Programming Note

A future version of the architecture will revise the ***isync*** and ***lwsync*** instruction descriptions to make them consistent with the use of these instructions, as shown in Section B.2.1.1, to acquire a lock associated with a TLE critical section.

Book III:

Power ISA Operating Environment Architecture

Chapter 1. Introduction

1.1 Overview

Chapter 1 of Book I describes computation modes, document conventions, a general systems overview, instruction formats, and storage addressing. This chapter augments that description as necessary for the Power ISA Operating Environment Architecture.

1.2 Document Conventions

The notation and terminology used in Book I apply to this Book also, with the following substitutions.

- For “system alignment error handler” substitute “Alignment interrupt”.
- For “system data storage error handler” substitute “Data Storage interrupt”, “Hypervisor Data Storage interrupt”, or “Data Segment interrupt”, as appropriate.
- For “system error handler” substitute “interrupt”.
- For “system floating-point enabled exception error handler” substitute “Floating-Point Enabled Exception type Program interrupt”.
- For “system illegal instruction error handler” substitute “Hypervisor Emulation Assistance interrupt”.
- For “system instruction storage error handler” substitute “Instruction Storage interrupt”, “Hypervisor Instruction Storage interrupt”, or “Instruction Segment interrupt”, as appropriate.
- For “system privileged instruction error handler” substitute “Privileged Instruction type Program interrupt”.
- For “system service program” substitute “System Call interrupt” or “System Call Vectored interrupt”, as appropriate.
- For “system trap handler” substitute “Trap type Program interrupt”.
- For “system facility unavailable error handler” substitute “Facility Unavailable interrupt” or “Hypervisor Facility Unavailable interrupt.”

1.2.1 Definitions and Notation

The definitions and notation given in Book I and Book II are augmented by the following.

- **Threaded processor, single-threaded processor, thread**

  A threaded processor implements one or more “threads”, where a thread corresponds to the Book I/II concept of “processor”. That is, the definition of “thread” is the same as the Book I definition of “processor”, and “processor” as used in Books I and II can be thought of as either a single-threaded processor or as one thread of a multi-threaded processor. Except where the meaning is clear in context or the number of threads does not matter, the only unqualified uses of “processor” in Book III are in resource names (e.g. Processor Identification Register); such uses should be regarded as meaning “threaded processor”. The threads of a multi-threaded processor typically share certain resources, such as the hardware components that execute certain kinds of instructions (e.g., Fixed-Point instructions), certain caches, the address translation mechanism, and certain hypervisor resources.

- **real page**
  A unit of real storage that is aligned at a boundary that is a multiple of its size. The real page size is 4KB.

- **context of a program**
  The state (e.g., privilege and relocation) in which the program executes. The context is controlled by the contents of certain System Registers, such as the MSR and PTCR, of certain lookaside buffers, such as the SLB and TLB, and of the Page Table.

- **performed**
  The definition of “performed” given in Section 1.1 of Book II is extended to apply to implicit storage accesses and to invalidations of entries in caches of information derived from address translation tables, as follows.

  – The definition of “load is performed” applies to accesses for performing address translation.

- The definition of “store is performed” applies to accesses for recording reference and change information.
- A TLB entry invalidation by thread T1 is performed with respect to thread T2 when the instruction that requested the invalidation has caused the specified entry, if present, to be made invalid in T2’s TLB, and similarly for invalidations of entries in other caches of information derived from tables used in address translation.

- **exception**
  An error, unusual condition, or external signal, that may set a status bit and may or may not cause an interrupt, depending upon whether the corresponding interrupt is enabled.
- **interrupt**
  The act of changing the machine state in response to an exception, as described in Chapter 6. “Interrupts” on page 1049.
- **trap interrupt**
  An interrupt that results from execution of a *Trap* instruction.
- Additional exceptions to the rule that the thread obeys the sequential execution model, beyond those described in Section 2.2 of Book I and in the bullet defining “program order” in Section 1.1 of Book II, are the following.
  - A System Reset or Machine Check interrupt may occur. The determination of whether an instruction is required by the sequential execution model is not affected by the potential occurrence of a System Reset or Machine Check interrupt. (The determination is affected by the potential occurrence of any other kind of interrupt.)
  - A context-altering instruction is executed (Chapter 11. “Synchronization Requirements for Context Alterations” on page 1133). The context alteration need not take effect until the required subsequent synchronizing operation has occurred.
  - A Reference and Change bit is updated by the thread. The update need not be performed with respect to that thread until the required subsequent synchronizing operation has occurred.
  - A *Branch* instruction is executed and the branch is taken. The update of the Come-From Address Register (see Section 8.2 of Book III) need not occur until a subsequent context synchronizing operation has occurred.
  - An ***mtgsr*** is executed and an interrupt occurs before the ***mtspr*** sequence following ***mtgsr*** has finished executing. The contents of SPRs that are the targets of ***mtspr*** instructions between the point of interruption and the end of the ***mtspr*** sequence may be altered.

- **“must”**
  If hypervisor software violates a rule that is stated using the word “must” (e.g., “this field must be set to 0”), and the rule pertains to the contents of a hypervisor resource, to executing an instruction that can be executed only in hypervisor state, or to accessing storage in real addressing mode, the results are undefined, and may include altering resources belonging to other partitions, causing the system to “hang”, etc.
- **hardware**
  Any combination of hard-wired implementation, emulation assist, or interrupt for software assistance. In the last case, the interrupt may be to an architected location or to an implementation-dependent location. Any use of emulation assists or interrupts to implement the architecture is implementation-dependent.
- **hypervisor privileged**
  A term used to describe an instruction or facility that is available only when the thread is in hypervisor state.
- **privileged state** and **supervisor mode**
  Used interchangeably to refer to a state in which privileged facilities are available.
- **problem state** and **user mode**
  Used interchangeably to refer to a state in which privileged facilities are not available.
- /, //, ///, ... denotes a field that is reserved in an instruction, in a register, or in an architected storage table.
- ?, ??, ???, ... denotes a field that is implementation-dependent in an instruction, in a register, or in an architected storage table.

1.2.2 Reserved Fields

Book I's description of the handling of reserved bits in System Registers, and of reserved values of defined fields of System Registers, applies also to the SLB. Book I's description of the handling of reserved values of defined fields of System Registers applies also to architected storage tables (e.g., the Page Table).

Software should set reserved fields in the SLB and in architected storage tables to zero, because these fields may be assigned a meaning in some future version of the architecture.

Some fields of certain architected storage tables may be written to automatically by the hardware, e.g., Reference and Change bits in the Page Table. When the

hardware writes to such a table, the following rules are obeyed.

- Unless otherwise stated, no defined field other than the one(s) specifically being updated are modified.
- Contents of reserved fields are either preserved or written as zero.

1.3 General Systems Overview

The hardware contains the sequencing and processing controls for instruction fetch, instruction execution, and interrupt action. Most implementations also contain data and instruction caches. Instructions that the processing unit can execute fall into the following classes:

- instructions executed in the Branch Facility
- instructions executed in the Fixed-Point Facility
- instructions executed in the Floating-Point Facility
- instructions executed in the Vector Facility

Almost all instructions executed in the Branch Facility, Fixed-Point Facility, Floating-Point Facility, and Vector Facility are nonprivileged and are described in Book I. Book II may describe additional nonprivileged instructions (e.g., Book II describes some nonprivileged instructions for cache management). Instructions related to the privileged state, control of hardware resources, control of the storage hierarchy, and all other privileged instructions are described here or are implementation-dependent.

1.4 Exceptions

The following augments the exceptions defined in Book I that can be caused directly by the execution of an instruction:

- the execution of a floating-point instruction when MSR_{FP}=0 (Floating-Point Unavailable interrupt)
- an attempt to modify a hypervisor resource when the thread is in privileged but non-hypervisor state (see Chapter 2), or an attempt to execute a hypervisor-only instruction (e.g., ***tlbie***) when the thread is in privileged but non-hypervisor state
- the execution of a traced instruction (Trace interrupt)
- the execution of a Vector instruction when the vector facility is unavailable (Vector Unavailable interrupt)

1.5 Synchronization

The synchronization described in this section refers to the state of the thread that is performing the synchronization.

1.5.1 Context Synchronization

An instruction or event is *context synchronizing* if it satisfies the requirements listed below. Such instructions and events are collectively called *context synchronizing operations*. The context synchronizing operations are the ***isync*** instruction, the *System Linkage* instructions, the ***mtmsr***[***d***] instructions with L=0, and most interrupts (see Section 6.4).

1. The operation causes instruction dispatching (the issuance of instructions by the instruction fetching mechanism to any instruction execution mechanism) to be halted.
2. The operation is not initiated or, in the case of ***isync***, does not complete, until all instructions that precede the operation have completed to a point at which they have reported all exceptions they will cause.
3. The operation ensures that the instructions that precede the operation will complete execution in the context (privilege, relocation, storage protection, etc.) in which they were initiated, except that the operation has no effect on the context in which the associated Reference and Change bit updates are performed.
4. If the operation directly causes an interrupt (e.g., ***sc*** directly causes a System Call interrupt) or is an interrupt, the operation is not initiated until no exception exists having higher priority than the exception associated with the interrupt (see Section 6.9).
5. The operation ensures that the instructions that follow the operation will be fetched and executed in the context established by the operation. (This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them out-of-order also be discarded, except as described in Section 5.5, “Performing Operations Out-of-Order”.)

Programming Note

A context synchronizing operation is necessarily execution synchronizing; see Section 1.5.2.

Unlike the *Synchronize* instruction, a context synchronizing operation does not affect the order in which storage accesses are performed.

Item 2 permits a choice only for ***isync*** (and ***sync*** and ***ptesync***; see Section 1.5.2) because all other execution synchronizing operations also alter context.

1.5.2 Execution Synchronization

An instruction is *execution synchronizing* if it satisfies items 2 and 3 of the definition of context synchronization (see Section 1.5.1). ***sync*** and ***ptesync*** are treated like ***isync*** with respect to item 2. The execution synchronizing instructions are ***sync***, ***ptesync***, the ***mtmsr***[***d***] instructions with L=1, and all context synchronizing instructions.

Programming Note

Unlike a context synchronizing operation, an execution synchronizing instruction does not ensure that the instructions following that instruction will execute in the context established by that instruction. This new context becomes effective sometime after the execution synchronizing instruction completes and before or at a subsequent context synchronizing operation.

Chapter 2. Logical Partitioning (LPAR) and Thread Control

2.1 Overview

The Logical Partitioning (LPAR) facility permits threads and portions of real storage to be assigned to logical collections called *partitions*, such that a program executing on a thread in one partition cannot interfere with any program executing on a thread in a different partition. This isolation can be provided for both problem state and privileged non-hypervisor state programs, by using a layer of trusted software, called a *hypervisor* program (or simply a "hypervisor"), and the resources provided by this facility to manage system resources. (A hypervisor is a program that runs in hypervisor state; see below.)

The number of partitions supported is implementation-dependent.

A thread is assigned to one partition at any given time. A thread can be assigned to any given partition without consideration of the physical configuration of the system (e.g., shared registers, caches, organization of the storage hierarchy), except that threads that share certain hypervisor resources may need to be assigned to the same partition; see Section 2.6. The registers and facilities used to control Logical Partitioning are listed below and described in the following subsections.

Except in the following subsections, references to the "operating system" in this document include the hypervisor unless otherwise stated or obvious from context.

2.2 Logical Partitioning Control Register (LPCR)

The contents of the LPCR control a number of aspects of the operation of the thread with respect to a logical partition. Below are shown the bit definitions for the LPCR.

Bit **Description**

0:3 ***Virtualization Control*** (VC)

Controls the virtualization of partition memory for partitions that use HPT translation. This field contains three subfields, VPM, ISL, and KBV. Accesses that are initiated in hypervisor state (i.e., $MSR_{HV\ PR}$=0b10) are performed as if VC=0b0000.

0 Reserved

1 ***Virtualized Partition Memory*** (VPM)

Controls whether VPM mode is enabled when address translation is enabled as specified below.

0 - VPM mode disabled
1 - VPM mode enabled

When address translation is disabled, VPM mode is enabled. See Section 5.7.2, "Virtualized Partition Memory (VPM) Mode", and Section 5.7.3.3, "Virtual Real Mode Addressing Mechanism", for additional information on VPM mode.

Programming Note

VPM must be set to zero by hypervisors that use HPT translation and want to receive storage interrupts from applications running directly under them as DSIs and ISIs (instead of HDSIs and HISIs).

2 ***Ignore SLB Large Page Specification*** (ISL)

Controls whether ISL mode is enabled as specified below.

0 - ISL mode disabled
1 - ISL mode enabled

When ISL mode is enabled and address translation is enabled, address translation is performed as if the contents of $SLB_{L||LP}$ and $PRTE_{STPS}$ were 0b000. When address translation is disabled, the setting of the ISL

bit has no effect. ISL mode has no effect on SLB, TLB, and ERAT entry invalidations caused by ***slbie***, ***slbieg***, ***slbia***, ***slbiag***, ***tlbie***, and ***tlbiel***.

Programming Note

Specifying that L||LP=0b000 in $PATE_{PS}$ has the same effect on address translation when translation is disabled as enabling ISL mode when translation is enabled.

ISL mode is needed when translation is enabled because translation uses the SLB, and the contents of the SLB are controlled by the operating system and should not be modified by the hypervisor. ISL mode is not needed when translation is disabled since Virtual Real Mode address translation uses $PATE_{PS}$, which is not visible to the operating system and is in complete control of the hypervisor.

3 ***Key-Based Virtualization*** (KBV)

Controls whether Key-Based Virtualization is enabled as specified below.

0 - KBV is disabled
1 - KBV is enabled

When KBV is enabled and $MSR_{HV||PR}$≠0b10, Virtual Page Class Key Storage Protection exceptions that occur on storage operand accesses when VPM=0 cause Hypervisor Data Storage interrupts.

Programming Note

Key-Based Virtualization provides an efficient means for the hypervisor to intercept storage references, e.g. MMIO, that must be emulated. (The corresponding behavior for instruction fetching is not desired.) Virtual Page Class Key Storage Protection exceptions not handled by the hypervisor should be reflected to the operating system at its Data Storage interrupt vector with the hypervisor having set $DSISR_{42}$.

4:8 Reserved

9:11 ***Default Prefetch Depth*** (DPFD)

The DPFD field is used as the default prefetch depth for data stream prefetching when $DSCR_{DPFD}$=0; see page 842.

12:16 Reserved

17:19 ***Power-saving mode Exit Cause Enable (Upper Section)*** ($PECE_U$)

17 **Hypervisor Virtualization Exit Enable**

0 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Hypervisor Virtualization exceptions are not enabled to cause exit from power-saving mode.
1 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Hypervisor Virtualization exceptions are enabled to cause exit from power-saving mode.

18:19 Reserved

20:37 Reserved

38 ***Interrupt Little-Endian*** (ILE)

The contents of the ILE bit are copied into MSR_{LE} by interrupts that set MSR_{HV} to 0 (see Section 6.5), to establish the Endian mode for the interrupt handler.

39:40 ***Alternate Interrupt Location*** (AIL)

Controls the effective address offset, or alternate effective address for System Call Vectored, of the interrupt handler and the relocation mode in which it begins execution for all interrupts except those subject to the overrides described below.

0 The interrupt is taken with $MSR_{IR\ DR}$ = 0b00 and no effective address offset or alternate effective address.
1 Reserved
2 The interrupt is taken with $MSR_{IR\ DR}$ = 0b11. If the interrupt is not System Call Vectored , an effective address offset of 0x0000_0000_0001_8000 is applied. System Call Vectored does not use an alternate effective address.
3 The interrupt is taken with $MSR_{IR\ DR}$ = 0b11. If the interrupt is not System Cal Vectored, an effective address offset of 0xc000_0000_0000_4000 is applied. System Call Vectored uses an alternate effective address of 0xc000_0000_0000_3 || LEV || 0b0_0000.

Machine Check, System Reset, and Hypervisor Maintenance interrupts are taken as if $LPCR_{AIL}$=0. In the remainder of this definition, “other interrupts” means interrupts other than these three.

Other interrupts that occur when MSR_{IR}=0 or MSR_{DR}=0, are taken as if $LPCR_{AIL}$=0.

When the hypervisor receiving the other interrupts uses HPT translation and the interrupts have caused a transition from MSR_{HV}=0 to

MSR_{HV}=1, the interrupts are taken as if $LPCR_{AIL}$=0.

Programming Note

One of the purposes of the AIL field is to provide relocation for interrupts that occur while an application is running with $MSR_{HV\ PR}$=0b11 under a "bare metal" operating system (i.e., an operating system that runs in hypervisor state), such as KVM.

41 ***Use Process Table*** (UPRT)

Controls whether Process Tables are used. For a radix-using partition, UPRT must be set to 1. For a paravirtualized HPT partition, UPRT is set to 1 when the operating system does not require the use of the legacy software-managed SLB.

0 Process Table is not used. (Software-managed SLB in use, for paravirtualized HPT partition.)
1 Process Table is used. (Segment Table in use, for paravirtualized HPT partition.)

Programming Note

The POWER9 processor operates as though $LPCR_{UPRT}$=0 for partitions that use HPT translation, requiring operating systems to fully manage the SLB in software. Nonetheless, operating systems may need to maintain segment tables for use by accelerators.

42 ***Enhanced Virtualization*** (EVIRT)

Controls whether Enhanced Virtualization is enabled, as specified below.

0 Enhanced Virtualization is disabled: attempts to access hypervisor resources or execute hypervisor privileged instructions in privileged but non-hypervisor state cause a Privileged Instruction type Program interrupt; attempts to access undefined SPR numbers (using **mtspr** or **mfspr**) other than 0, 4, 5, and 6 in privileged state are treated as no-ops.
1 Enhanced Virtualization is enabled: attempts to access hypervisor resources or execute hypervisor privileged instructions in privileged but non-hypervisor state cause a Hypervisor Emulation Assistance interrupt; attempts to access undefined SPR numbers (using **mtspr** or **mfspr**) other than 0, 4, 5, and 6 in privileged state cause a Hypervisor Emulation Assistance interrupt.

Programming Note

Running with $LPCR_{EVIRT}$=1 facilitates support of nested hypervisors (hypervisors that run with $MSR_{HV\ PR}$=0b00 and have their use of hypervisor resources virtualized by a higher level hypervisor); see the relevant Programming Note in Section 6.5.18, "Hypervisor Emulation Assistance Interrupt". It also permits emulation of new SPRs on designs that do not support them in hardware.

All accesses to the reserved noop SPRs (808-811) are always treated as noops, independent of the value of EVIRT.

43 ***Host Radix*** (HR)

Indicates whether the partition uses Radix Tree translation, as specified below.

0 Hypervisor does not use Radix Tree translation.
1 Hypervisor uses Radix Tree translation.

Programming Note

The hypervisor must program HR to match the Host Radix bit in the appropriate Partition Table Entry. If the values do not match, the results are undefined.

HR is duplicated in the LPCR because there are times such as immediately after a partition swap when it is difficult for hardware to quickly access the PATE.

44 Reserved

45 ***Online*** (ONL)

0 The PURR and SPURR do not increment.
1 The PURR and SPURR increment.

Programming Note

Typically, the hypervisor sets the ONL bit to 0 when the thread is not in a power saving mode, is not performing useful work, and is available for use. The hypervisor may take the state of the ONL bit into account when making course-grain load balancing and power management decisions.

46 ***Large Decrementer*** (LD)

0 Large Decrementer mode is not enabled.
1 Large Decrementer mode is enabled.

See Section 7.4 for additional information.

47:51 ***Power-saving mode Exit Cause Enable (Lower Section)*** ($PECE_L$)

47 **Privileged Doorbell Exit Enable**

0 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Directed Privileged Doorbell exceptions are not enabled to cause exit from power-saving mode

1 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Directed Privileged Doorbell exceptions are enabled to cause exit from power-saving mode.

48 **Hypervisor Doorbell Exit Enable**

0 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Directed Hypervisor Doorbell exceptions are not enabled to cause exit from power-saving mode

1 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Directed Hypervisor Doorbell exceptions are enabled to cause exit from power-saving mode.

49 **External Exit Enable**

0 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, External exceptions are not enabled to cause exit from power-saving mode.

1 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, External exceptions are enabled to cause exit from power-saving mode.

50 **Decrementer Exit Enable**

0 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Decrementer exceptions are not enabled to cause exit from power-saving mode.

1 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Decrementer exceptions are enabled to cause exit from power-saving mode. (Decrementer exceptions do not occur if the state of the Decrementer is not maintained and updated as if the thread was not in power-saving mode.)

51 **Other Exit Enable**

0 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Machine Check, Hypervisor Maintenance, and certain implementation-specific exceptions are not enabled to cause exit from power-saving mode.

1 When the ***stop*** instruction is executed with $PSSCR_{EC}$=1, Machine Check, Hypervisor Maintenance, and certain implementation-specific exceptions are enabled to cause exit from power-saving mode.

If the state of the PECE field is lost during power-saving mode, implementations must provide the means to exit power-saving mode upon the occurrence of a System Reset exception and any of the exceptions that were enabled by the PECE field when the ***stop*** instruction was executed. In addition, they may also exit power-saving mode on exceptions that were disabled by the PECE field as well. See Section 6.5.1 and Section 6.5.2 for additional information about exit from power-saving mode.

52 ***Mediated External Exception Request*** (MER)

0 A Mediated External exception is not requested.

1 A Mediated External exception is requested.

The exception effects of this bit are said to be consistent with the contents of this bit if one of the following statements is true.

- $LPCR_{MER}$ = 1 and a Mediated External exception exists.
- $LPCR_{MER}$ = 0 and a Mediated External exception does not exist.

A context synchronizing instruction or event that is executed or occurs when $LPCR_{MER}$ = 0 ensures that the exception effects of $LPCR_{MER}$ are consistent with the contents of $LPCR_{MER}$. Otherwise, when an instruction changes the contents of $LPCR_{MER}$, the exception effects of $LPCR_{MER}$ become consistent with the new contents of $LPCR_{MER}$ reasonably soon after the change.

Programming Note

$LPCR_{MER}$ provides a means for the hypervisor to direct an external exception to a partition independent of the partition's MSR_{EE} setting. (When MSR_{EE}=0, it is inappropriate for the hypervisor to deliver the exception.) Using $LPCR_{MER}$, the partition can be interrupted upon enabling external interrupts. Without using $LPCR_{MER}$, the hypervisor must check the state of MSR_{EE} whenever it gets control, which will result in less timely delivery of the exception to the partition.

53 ***Guest Translation Shootdown Enable*** (GTSE)

Controls whether the operating system is permitted to use ***tlbie***, ***slbieg***, and ***slbiag*** directly, or must issue a system call to the hypervisor.

0 Guest is not permitted to use ***tlbie***, ***slbieg***, ***slbiag***, ***tlbsync***, and ***slbsync***.

1 Guest is permitted to use ***tlbie***, ***slbieg***, ***slbiag***, ***tlbsync***, and ***slbsync***.

Programming Note

An operating system that uses HPT translation must know whether VPM is active in order to invalidate the translation for a specific page using ***tlbie***[***l***]. See the related Programming Notes in the descriptions of ***tlbie*** and ***tlbiel***.

54 ***Translation Control*** (TC)

0 The secondary Page Table search is enabled.
1 The secondary Page Table search is disabled.

55:58 Reserved

59 ***Hypervisor External Interrupt Control*** (HEIC)

0 Direct External interrupts can occur in Hypervisor state.
1 Direct External interrupts cannot occur in hypervisor state.

Programming Note

By setting HEIC=1, the Hypervisor Interrupt Virtualization handler can prevent External interrupts from occurring during the Hypervisor Virtualization interrupt handler. See Section 6.5.7.1.

60 ***Logical Partitioning Environment Selector*** (LPES)

0 External interrupts set the HSRRs, set MSR_{HV} to 1, and leave MSR_{RI} unchanged.
1 External interrupts set the SRRs, set MSR_{RI} to 0, and leave MSR_{HV} unchanged.

Programming Note

LPES = 1 should be used by operating systems not running under a hypervisor, so that external interrupts are directed to the SRRs rather than to the HSRRs.

Programming Note

In versions of the architecture that precede Version 2.07, LPES was a two-bit field, in which the second bit controlled significant aspects of storage accessing and interrupt handling.

61 Reserved

62 ***Hypervisor Virtualization Interrupt Conditionally Enable*** (HVICE)

0 Hypervisor Virtualization interrupts are disabled.
1 Hypervisor Virtualization interrupts are enabled if permitted by MSR_{EE}, MSR_{HV}, and MSR_{PR}; see Section 6.5.21.

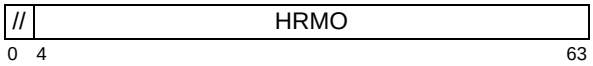
63 ***Hypervisor Decrementer Interrupt Conditionally Enable*** (HDICE)

0 Hypervisor Decrementer interrupts are disabled.
1 Hypervisor Decrementer interrupts are enabled if permitted by MSR_{EE}, MSR_{HV}, and MSR_{PR}; see Section 6.5.12 on page 1077.

See Section 6.5 on page 1063 for a description of how the setting of LPES affects the processing of interrupts.

2.3 Hypervisor Real Mode Offset Register (HRMOR)

The layout of the Hypervisor Real Mode Offset Register (HRMOR) is shown in Figure 1 below.



Bits	Name	Description
4:63	HRMO	Real Mode Offset

Figure 1. Hypervisor Real Mode Offset Register

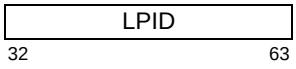
All other fields are reserved.

The supported HRMO values are the non-negative multiples of 2^r, where r is an implementation-dependent value and $12 \le r \le 26$.

The contents of the HRMOR affect how some storage accesses are performed as described in Section 5.7.3 on page 984 and Section 5.7.5 on page 987.

2.4 Logical Partition Identification Register (LPIDR)

The layout of the Logical Partition Identification Register (LPIDR) is shown in Figure 2 below.



Bits	Name	Description
32:63	LPID	Logical Partition Identifier

Figure 2. Logical Partition Identification Register

The contents of the LPIDR identify the partition to which the thread is assigned, affecting some aspects of translation and interrupt delivery. The number of LPIDR bits supported is implementation-dependent.

Programming Note

Radix tree translation assigns special meaning to LPID=0, specifically indicating the hypervisor's own partition. When HR=1, LPIDR should not be set to zero except when MSR_{HV}=1.

HPT translation provides special functionality for LPID=0 when HV=1, as described in Section 5.9.3, to support the execution of a "bare metal" operating system (an operating system that runs in hypervisor state). Speculative Segment Table walks are prohibited when MSR_{HV}=1 in other partitions because adjunct translations are bolted. A partition that uses HPT translation and requires the services of an adjunct should not be assigned LPID=0.

Programming Note

The aspect of interrupt delivery that the LPIDR affects is the delivery of certain external interrupts. Some platforms make LPIDR/PIDR/TIDR available so that specific threads can be targeted for interrupt delivery. This function is most commonly used to communicate the disposition of accelerator-related processing back to the initiating thread.

2.5 Processor Compatibility Register (PCR)

The layout of the Processor Compatibility Register (PCR) is shown in Figure 3 below.

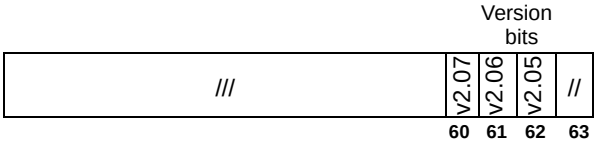


Figure 3. Processor Compatibility Register

Each defined bit in the PCR controls whether certain instructions, SPRs, and other related facilities are available in problem state. Except as specified elsewhere in this section, the PCR has no effect on facilities when the thread is not in problem state. Facilities that are made unavailable by the PCR are treated as follows when the thread is in problem state.

- Instructions are treated as illegal instructions.
- SPRs are treated as if they were not defined for the implementation.
- The "reserved SPRs" (see Section 1.3.3 of Book I) are treated as not defined for the implementation.
- Fields in instructions are treated as if they were 0s.
- Unless the second item of this list applies, bits in system registers read back 0s for ***mfspr*** and ***mtspr*** operations have no effect on their values, except as described immediately below for bits 44:45 of the XER.

  For bits 44:45 of the XER, two pairs of bits are provided, an "OV32-CA32" bit pair for XER_{OV32} and XER_{CA32} and a "reserved" bit pair for legacy XER bits 44:45 behavior.

  Which bit pair is read by ***mfxer*** is controlled by the PCR. ***mtxer*** writes to both bit pairs, independent of the PCR. ***mcrxr*** reads the "OV32-CA32" bit pair.

  Each bit in the "OV32-CA32" bit pair is implicitly set by instructions that implicitly set their respective XER_{OV} or XER_{CA}, independent of the PCR. The "reserved" bit pair for bits 44:45 of the XER are not altered by these instructions, independent of the PCR.

  The ***txer***, ***selii***[.], ***selir***[.], ***selri***[.], and ***selrr***[.] instructions read bits 44:45 of the XER as 0s, independent of the PCR.

  Programming Note

  The "reserved" bit pair does not conform to the usual rules for reading (***mfspr***) reserved bits in registers (see Section 1.3.3 of Book I) because some early implementations used bits 44:45 of the XER for implementation-specific purposes. On these implementations, and on subsequent implementations that implemented versions of the architecture that precede V. 3.0, ***mfxer*** returned the contents of the bits, despite that the bits were defined as reserved.

A defined bit in the PCR may also control whether certain instructions, SPRs, and other related facilities are available in a privileged state (MSR_{PR}=0). Affected facilities will be specifically annotated.

Programming Note

When a bit in a system register is made unavailable by the PCR, ***mtspr*** operations performed on the register in problem state have no effect on the value of the bit regardless of the privilege state in which the register may subsequently be read.

A PCR bit may also determine how an instruction field value is interpreted or may define other behavior as specified in the bit definitions below.

The PCR has no effect on the setting of the MSR and [H]SRR1 by interrupts (and of the Count Register by the System Call Vectored interrupt), and by the ***rfscv***,

[h]rfid and ***mtmsr[d]*** instructions, except as specified elsewhere in this section.

When facilities that have enable bits in the MSR, FSCR, HFSCR, or MMCR0 are made unavailable by the value in the PCR, they become unavailable in problem state as specified above regardless of whether they are enabled by the corresponding MSR, FSCR, HFSCR, or MMCR0 bit; facility availability interrupts (e.g. [Hypervisor] Facility Available, Vector Unavailable, etc.) do not occur as a result of problem state accesses even if the corresponding field in the MSR, [H]FSCR, or MMCR0 makes them unavailable in problem state.

Programming Note

Facilities that can be disabled in problem state by the PCR that also have enable bits in either the MSR or [H]FSCR include Transactional Memory, the BHRB instructions, event-based branch instructions, TAR, DSCR at SPR 3, SIER, MMCR2, the event-based branch instructions, and certain Floating-Point, Vector, and VSX instructions. When any of these facilities are made unavailable in problem state by the PCR, the corresponding [Hypervisor] Facility Unavailable, Floating-Point Unavailable, Vector, or VSX unavailable interrupts do not occur when the facility is accessed in problem state. Note, however, that the PCR does not affect privileged accesses, and thus any Hypervisor Facility Unavailable, Floating-Point Unavailable, Vector unavailable, or VSX unavailable interrupts that are specified to occur as a result of privileged accesses occur regardless of the PCR value.

The bit definitions for the PCR are shown below.

Bit	Description
0:59	Reserved
60	**Version 2.07** (v2.07) When MSR_{PR}=1 (i.e., problem state), this bit controls the availability of the following instructions, facilities, and behaviors that were newly available in the version of the architecture subsequent to Version 2.07. – The instructions listed in Table 1 – ***scv*** – The splitting out of footprint overflows in which other threads contributed to the problem to set $TEXASR_{17}$ and indicate a transient failure instead of setting $TEXASR_{10}$ and indicating a persistent failure. 0 The instructions, behaviors, and facilities listed above are available. ***mfxer*** reads the contents of the "OV32-CA32" bit pair for XER bits 44:45. 1 The instructions, behaviors, and facilities listed above are unavailable. ***mfxer*** reads the contents of the "reserved" bit pair for XER bits 44:45. When MSR_{PR}=0 (i.e., privileged or hypervisor-privileged state), this bit controls the availability of the ***mcrxrx*** instruction and which bit pair is read by ***mfxer*** for XER bits 44:45. 0 ***mcrxrx*** is available. ***mfxer*** reads the contents of the "OV32-CA32" bit pair for XER bits 44:45. 1 ***mcrxrx*** is unavailable. ***mfxer*** reads the contents of the "reserved" bit pair for XER bits 44:45.

Mnemonic	Instruction Name
addpcis	Add PC Immediate Shifted Prefix
bcdcfn.	Decimal Convert From National
bcdcfsq.	Decimal Convert From Signed Qword
bcdcfz.	Decimal Convert From Zoned
bcdcpsgn	Decimal CopySign
bcdctn.	Decimal Convert To National
bcdctsq.	Decimal Convert To Signed Qword
bcdctz.	Decimal Convert To Zoned
bcds.	Decimal Shift
bcdsetsgn.	Decimal Set Sign
bcdsr.	Decimal Shift and Round
bcdtrunc.	Decimal Truncate
bcdus.	Decimal Unsigned Shift

Table 1: Instructions Controlled by the V 2.07 Bit

Mnemonic	Instruction Name
bcdutrunc.	Decimal Unsigned Truncate
cmpeqb	Compare Equal Byte
cmprb	Compare Ranged Byte
cnttzd[.]	Count Trailing Zeros Dword
cnttzw[.]	Count Trailing Zeros Word
copy	Copy
cpabort	Copy-Paste Abort
darn	Deliver a Random Number
dtstsfi	DFP Test Significance Immediate
dtstsfiq	DFP Test Significance Immediate Quad
extswsli[.]	Extend Sign Word and Shift Left Immediate
ldat	Load Doubleword Atomic
lwat	Load Word Atomic
lxsd	Load VSX Scalar Dword
lxsibzx	Load VSX Scalar as Integer Byte & Zero Indexed
lxsihzx	Load VSX Scalar as Integer Hword & Zero Indexed
lxssp	Load VSX Scalar Single
lxv	Load VSX Vector
lxvb16x	Load VSX Vector Byte*16 Indexed
lxvh8x	Load VSX Vector Halfword*8 Indexed
lxvl	Load VSX Vector with Length
lxvll	Load VSX Vector Left-justified with Length
lxvwsx	Load VSX Vector Word & Splat Indexed
lxvx	Load VSX Vector Indexed
maddhd	Multiply-Add High Dword
maddhdu	Multiply-Add High Dword Unsigned
maddld	Multiply-Add Low Dword
mcrxrx	Move XER to CR Extended
mffsce	Move From FPSCR & Clear Enables
mffscdrn	Move From FPSCR Control & set DRN
mffscdrni	Move From FPSCR Control & set DRN Immediate
mffscrn	Move From FPSCR Control & set RN
mffscrni	Move From FPSCR Control & set RN Immediate
mffsl	Move From FPSCR Lightweight
mfvsrld	Move From VSR Lower Dword
modsd	Modulo Signed Dword
modsw	Modulo Signed Word
modud	Modulo Unsigned Dword
moduw	Modulo Unsigned Word
mtvsrdd	Move To VSR Double Dword
mtvsrws	Move To VSR Word & Splat
paste.	Paste
setb	Set Boolean
stdat	Store Doubleword Atomic
stwat	Store Word Atomic
stxsd	Store VSX Scalar Dword
stxsibx	Store VSX Scalar as Integer Byte Indexed
stxsihx	Store VSX Scalar as Integer Hword Indexed

Table 1: Instructions Controlled by the V 2.07 Bit

Mnemonic	Instruction Name
stxssp	Store VSX Scalar Single
stxv	Store VSX Vector
stxvb16x	Store VSX Vector Byte*16 Indexed
stxvh8x	Store VSX Vector Halfword*8 Indexed
stxvl	Store VSX Vector with Length
stxvll	Store VSX Vector Left-justified with Length
stxvx	Store VSX Vector Indexed
vabsdub	Vector Absolute Difference Unsigned Byte
vabsduh	Vector Absolute Difference Unsigned Hword
vabsduw	Vector Absolute Difference Unsigned Word
vbpermd	Vector Bit Permute Dword
vclzlsbb	Vector Count Leading Zero Least-Significant Bits Byte
vcmpneb[.]	Vector Compare Not Equal Byte
vcmpneh[.]	Vector Compare Not Equal Hword
vcmpnew[.]	Vector Compare Not Equal Word
vcmpnezb[.]	Vector Compare Not Equal or Zero Byte
vcmpnezh[.]	Vector Compare Not Equal or Zero Hword
vcmpnezw[.]	Vector Compare Not Equal or Zero Word
vctzb	Vector Count Trailing Zeros Byte
vctzd	Vector Count Trailing Zeros Dword
vctzh	Vector Count Trailing Zeros Hword
vctzlsbb	Vector Count Trailing Zero Least-Significant Bits Byte
vctzw	Vector Count Trailing Zeros Word
vextractd	Vector Extract Dword
vextractub	Vector Extract Unsigned Byte
vextractuh	Vector Extract Unsigned Hword
vextractuw	Vector Extract Unsigned Word
vextsb2d	Vector Extend Sign Byte To Dword
vextsb2w	Vector Extend Sign Byte To Word
vextsh2d	Vector Extend Sign Hword To Dword
vextsh2w	Vector Extend Sign Hword To Word
vextsw2d	Vector Extend Sign Word To Dword
vextublx	Vector Extract Unsigned Byte Left-Indexed
vextubrx	Vector Extract Unsigned Byte Right-Indexed
vextuhlx	Vector Extract Unsigned Hword Left-Indexed
vextuhrx	Vector Extract Unsigned Hword Right-Indexed
vextuwlx	Vector Extract Unsigned Word Left-Indexed
vextuwrx	Vector Extract Unsigned Word Right-Indexed
vinsertb	Vector Insert Byte
vinsertd	Vector Insert Dword
vinserth	Vector Insert Hword
vinsertw	Vector Insert Word
vmul10cuq	Vector Multiply-by-10 & write Carry Unsigned Qword
vmul10ecuq	Vector Multiply-by-10 Extended & write Carry Unsigned Qword
vmul10euq	Vector Multiply-by-10 Extended Unsigned Qword
vmul10uq	Vector Multiply-by-10 Unsigned Qword
vnegd	Vector Negate Dword
vnegw	Vector Negate Word

Table 1: Instructions Controlled by the V 2.07 Bit

Mnemonic	Instruction Name
vpermr	Vector Permute Right-indexed
vprtybd	Vector Parity Byte Dword
vprtybq	Vector Parity Byte Qword
vprtybw	Vector Parity Byte Word
vrldmi	Vector Rotate Left Dword then Mask Insert
vrldnm	Vector Rotate Left Dword then AND with Mask
vrlwmi	Vector Rotate Left Word then Mask Insert
vrlwnm	Vector Rotate Left Word then AND with Mask
vslv	Vector Shift Left Variable
vsrv	Vector Shift Right Variable
wait	Wait
xsabsqp	VSX Scalar Quad-Precision Absolute
xsaddqp[o]	VSX Scalar Quad-Precision Add [& round to Odd]
xscmpexpdp	VSX Scalar Double-Precision Compare Exponents
xscmpexpqp	VSX Scalar Quad-Precision Compare Exponents
xscmpoqp	VSX Scalar Quad-Precision Compare Ordered
xscmpuqp	VSX Scalar Quad-Precision Compare Unordered
xscpsgnqp	VSX Scalar Quad-Precision CopySign
xscvdpqp	VSX Scalar Quad-Precision Convert From Double-Precision
xscvhpsp	VSX Scalar Convert Half-Precision to Double-Precision
xscvqpdp[o]	VSX Scalar round & Convert Quad-Precision to Double-Precision [using round to Odd]
xscvqpsdz	VSX Scalar truncate & Convert Quad-Precision to Signed Dword
xscvqpswz	VSX Scalar truncate & Convert Quad-Precision to Signed Word
xscvqpudz	VSX Scalar truncate & Convert Quad-Precision to Unsigned Dword
xscvqpuwz	VSX Scalar truncate & Convert Quad-Precision to Unsigned Word
xscvsdqp	VSX Scalar Convert Signed Dword format to Quad-Precision format
xscvsphp	VSX Scalar round & Convert Double-Precision to Half-Precision
xscvudqp	VSX Scalar Convert Unsigned Dword format to Quad-Precision format
xsdivqp[o]	VSX Scalar Quad-Precision Divide [& round to Odd]
xsiexpdp	VSX Scalar Double-Precision Insert Exponent
xsiexpqp	VSX Scalar Quad-Precision Insert Exponent
xsmaddqp[o]	VSX Scalar Quad-Precision Multiply-Add [& round to Odd]
xsmsubqp[o]	VSX Scalar Quad-Precision Multiply-Subtract [& round to Odd]
xsmulqp[o]	VSX Scalar Quad-Precision Multiply [& round to Odd]
xsnabsqp	VSX Scalar Quad-Precision Negative Absolute
xsnegqp	VSX Scalar Quad-Precision Negate
xsnmaddqp[o]	VSX Scalar Quad-Precision Negative Multiply-Add [& round to Odd]
xsnmsubqp[o]	VSX Scalar Quad-Precision Negative Multiply-Subtract [& round to Odd]
xsrqpi	VSX Scalar Round to Quad-Precision Integer
xsrqpxp	VSX Scalar Quad-Precision Round to Double-Extended-Precision
xssqrtqp[o]	VSX Scalar Quad-Precision Square Root [& round to Odd]
xssubqp[o]	VSX Scalar Quad-Precision Subtract [& round to Odd]
xststdcdp	VSX Scalar Double-Precision Test Data Class
xststdcqp	VSX Scalar Quad-Precision Test Data Class
xststdcsp	VSX Scalar Single-Precision Test Data Class
xsxexpdp	VSX Scalar Double-Precision Extract Exponent

Table 1: Instructions Controlled by the V 2.07 Bit

Mnemonic	Instruction Name
xsxexpqp	VSX Scalar Quad-Precision Extract Exponent
xsxsigdp	VSX Scalar Double-Precision Extract Significand
xsxsigqp	VSX Scalar Quad-Precision Extract Significand
xvcvhpsp	VSX Vector Convert Half-Precision to Single-Precision
xvcvsphp	VSX Vector round & Convert Single-Precision to Half-Precision
xviexpdp	VSX Vector Double-Precision Insert Exponent
xviexpsp	VSX Vector Single-Precision Insert Exponent
xvtstdcdp	VSX Vector Double-Precision Test Data Class
xvtstdcsp	VSX Vector Single-Precision Test Data Class
xvxexpdp	VSX Vector Double-Precision Extract Exponent
xvxexpsp	VSX Vector Single-Precision Extract Exponent
xvxsigdp	VSX Vector Double-Precision Extract Significand
xvxsigsp	VSX Vector Single-Precision Extract Significand
xxbrd	VSX Vector Byte-Reverse Dword
xxbrh	VSX Vector Byte-Reverse Hword
xxbrq	VSX Vector Byte-Reverse Qword
xxbrw	VSX Vector Byte-Reverse Word
xxextractuw	VSX Vector Extract Unsigned Word
xxinsertw	VSX Vector Insert Word
xxperm	VSX Vector Permute
xxpermr	VSX Vector Permute Right-indexed
xxspltib	VSX Vector Splat Immediate Byte

Table 1: Instructions Controlled by the V 2.07 Bit

61 **Version 2.06** (v2.06)

This bit controls the availability, in problem state, of the following instructions, facilities, and behaviors that were newly available in problem state in the version of the architecture subsequent to Version 2.06.

- ***icbt***
- ***lq, stq lbarx, lharx, stbcx, sthcx***
- ***lqarx., stqcx.***
- ***clrbhrb, mfbhrbe***
- ***rfebb, bctar[l]***
- The entire Transactional Memory facility
- The instructions in Table 2
- The reserved no-op instructions (see Section 1.9.3 of Book I)
- The reserved SPRs (see Section 1.3.3 of Book I)
- PPR32
- DSCR at SPR number 3
- SIER and MMCR2
- $MMCR0_{42:47,\ 51:55}$ and $MMCRA_{0:63}$.

Programming Note

The specified bits of MMCR0 and MMCRA above cannot be changed by ***mtspr*** instructions and ***mfspr*** instructions return 0s for these bits.

- BESCR, EBBHR, and TAR
- The ability of the ***or 31,31,31*** and ***or 5,5,5*** instructions to change the value of $PPR_{PRI.}$
- The ability of ***mtspr*** instructions that attempt to set PPR_{PRI} to 001 or 101 to change the value of PPR_{PRI}.

0 The instructions, facilities, and behaviors listed above are available in problem state.

1 The listed instructions, facilities, and behaviors listed above are unavailable in problem state.

If this bit is set to 1, then the V 2.07 bit must also be set to 1.

Mnemonic	Instruction Name
bcdadd.	Decimal Add Modulo
bcdsub.	Decimal Subtract Modulo
fmrgew	Floating Merge Even Word
fmrgow	Floating Merge Odd Word
lxsiwax	Load VSX Scalar as Integer Word Algebraic Indexed
lxsiwzx	Load VSX Scalar as Integer Word and Zero Indexed
lxsspx	Load VSX Scalar Single-Precision Indexed
mfvsrd	Move From VSR Doubleword
mfvsrwz	Move From VSR Word and Zero
mtvsrd	Move To VSR Doubleword
mtvsrwa	Move To VSR Word Algebraic
mtvsrwz	Move To VSR Word and Zero
stxsiwx	Store VSX Scalar as Integer Word Indexed
stxsspx	Store VSX Scalar Single-Precision Indexed
vaddcuq	Vector Add & write Carry Unsigned Quadword
vaddecuq	Vector Add Extended & write Carry Unsigned Quadword
vaddeuqm	Vector Add Extended Unsigned Quadword Modulo
vaddudm	Vector Add Unsigned Doubleword Modulo
vadduqm	Vector Add Unsigned Quadword Modulo
vbpermq	Vector Bit Permute Quadword
vcipher	Vector AES Cipher
vcipherlast	Vector AES Cipher Last
vclzb	Vector Count Leading Zeros Byte
vclzd	Vector Count Leading Zeros Doubleword
vclzh	Vector Count Leading Zeros Halfword
vclzw	Vector Count Leading Zeros Word
vcmpequd[.]	Vector Compare Equal To Unsigned Doubleword
vcmpgtsd[.]	Vector Compare Greater Than Signed Doubleword
vcmpgtud[.]	Vector Compare Greater Than Unsigned Doubleword
veqv	Vector Logical Equivalence
vgbbd	Vector Gather Bits by Bytes by Doubleword
vmaxsd	Vector Maximum Signed Doubleword
vmaxud	Vector Maximum Unsigned Doubleword
vminsd	Vector Minimum Signed Doubleword
vminud	Vector Minimum Unsigned Doubleword
vmrgew	Vector Merge Even Word
vmrgow	Vector Merge Odd Word
vmulesw	Vector Multiply Even Signed Word
vmuleuw	Vector Multiply Even Unsigned Word
vmulosw	Vector Multiply Odd Signed Word
vmulouw	Vector Multiply Odd Unsigned Word
vmuluwm	Vector Multiply Unsigned Word Modulo
vnand	Vector Logical NAND

Table 2: VSX and Vector Instructions Controlled by the v2.06 Bit

Mnemonic	Instruction Name
vncipher	Vector AES Inverse Cipher
vncipherlast	Vector AES Inverse Cipher Last
vorc	Vector Logical OR with Complement
vpermxor	Vector Permute and Exclusive-OR
vpksdss	Vector Pack Signed Doubleword Signed Saturate
vpksdus	Vector Pack Signed Doubleword Unsigned Saturate
vpkudum	Vector Pack Unsigned Doubleword Unsigned Modulo
vpkudus	Vector Pack Unsigned Doubleword Unsigned Saturate
vpmsumb	Vector Polynomial Multiply-Sum Byte
vpmsumd	Vector Polynomial Multiply-Sum Doubleword
vpmsumh	Vector Polynomial Multiply-Sum Halfword
vpmsumw	Vector Polynomial Multiply-Sum Word
vpopcntb	Vector Population Count Byte
vpopcntd	Vector Population Count Doubleword
vpopcnth	Vector Population Count Halfword
vpopcntw	Vector Population Count Word
vrld	Vector Rotate Left Doubleword
vsbox	Vector AES S-Box
vshasigmad	Vector SHA-512 Sigma Doubleword
vshasigmaw	Vector SHA-256 Sigma Word
vsld	Vector Shift Left Doubleword
vsrad	Vector Shift Right Algebraic Doubleword
vsrd	Vector Shift Right Doubleword
vsubcuq	Vector Subtract & write Carry Unsigned Quadword
vsubecuq	Vector Subtract Extended & write Carry Unsigned Quadword
vsubeuqm	Vector Subtract Extended Unsigned Quadword Modulo
vsubudm	Vector Subtract Unsigned Doubleword Modulo
vsubuqm	Vector Subtract Unsigned Quadword Modulo
vupkhsw	Vector Unpack High Signed Word
vupklsw	Vector Unpack Low Signed Word
xsaddsp	VSX Scalar Add Single-Precision
xscvdpspn	Scalar Convert Double-Precision to Single-Precision format Non-signalling
xscvdpspn	Scalar Convert Single-Precision to Double-Precision format Non-signalling
xscvsxdsp	VSX Scalar Convert Signed Fixed-Point Doubleword to Single-Precision
xscvsxdsp	VSX Scalar round and Convert Signed Fixed-Point Doubleword to Single-Precision format
xscvuxdsp	VSX Scalar Convert Unsigned Fixed-Point Doubleword to Single-Precision
xscvuxdsp	VSX Scalar round and Convert Unsigned Fixed-Point Doubleword to Single-Precision format
xsdivsp	VSX Scalar Divide Single-Precision
xsmaddasp	VSX Scalar Multiply-Add Type-A Single-Precision
xsmaddmsp	VSX Scalar Multiply-Add Type-M Single-Precision
xsmsubasp	VSX Scalar Multiply-Subtract Type-A Single-Precision
xsmsubmsp	VSX Scalar Multiply-Subtract Type-M Single-Precision
xsmulsp	VSX Scalar Multiply Single-Precision

Table 2: VSX and Vector Instructions Controlled by the v2.06 Bit

Mnemonic	Instruction Name
xsnmaddasp	VSX Scalar Negative Multiply-Add Type-A Single-Precision
xsnmaddmsp	VSX Scalar Negative Multiply-Add Type-M Single-Precision
xsnmsubasp	VSX Scalar Negative Multiply-Subtract Type-A Single-Precision
xsnmsubmsp	VSX Scalar Negative Multiply-Subtract Type-M Single-Precision
xsresp	VSX Scalar Reciprocal Estimate Single-Precision
xsrsp	VSX Scalar Round to Single-Precision
xsrsqrtesp	VSX Scalar Reciprocal Square Root Estimate Single-Precision
xssqrtsp	VSX Scalar Square Root Single-Precision
xssubsp	VSX Scalar Subtract Single-Precision
xxleqv	VSX Logical Equivalence
xxlnand	VSX Logical NAND
xxlorc	VSX Logical OR with Complement

Table 2: VSX and Vector Instructions Controlled by the v2.06 Bit

62 **Version 2.05** (v2.05)

This bit controls the availability, in problem state, of the following instructions, facilities, and behaviors that were newly available in problem state in the version of the architecture subsequent to Version 2.05.

- AMR access using SPR 13
- ***addg6s***
- ***bperm***
- ***cdtbcd, cbcdtd***
- ***dcffix[.]***
- ***divde[o][.], divdeu[o][.], divwe[o][.], divweu[o][.]***
- ***isel***
- ***lfiwzx***
- ***fctidu[.], fctiduz[.], fctiwu[.], fctiwuz[.], fcfids[.], fcfidu[.], fcfidus[.], ftdiv, ftsqrt***
- ***ldbrx, stdbrx***
- ***popcntw, popcntd***
- All facilities in the VSX facility

0 The instructions, facilities, and behaviors listed above are available in problem state.

1 The instructions, facilities, and behaviors listed above are unavailable in problem state.

If this bit is set to 1, then the v2.06 bit must also be set to 1.

63 Reserved

The initial state of the PCR is all 0s.

Programming Note

Because the PCR has no effect on privileged instructions except as specified above, privileged instructions that are available on newer implementations but not available on older implementations will behave differently when the thread is in problem state. On older implementations, either an Illegal Instruction type Program interrupt or a Hypervisor Emulation Assistance interrupt will occur because the instruction is undefined; on newer implementations, a Privileged Instruction type Program interrupt will occur because the instruction is implemented. (On older implementations the interrupt will be an Illegal Instruction type Program interrupt if the implementation complies with a version of the architecture that precedes V. 2.05, or complies with V. 2.05 and does not support the Hypervisor Emulation Assistance interrupt, and will be a Hypervisor Emulation Assistance interrupt otherwise.)

In future versions of the architecture, in general the lowest-order reserved bit of the PCR will be used to control the availability of the instructions and related resources that are new in that version of the architecture; the name of the bit will correspond to the previous version of the architecture (i.e., the newest version in which the instructions and related resources were not available).

In these future versions of the architecture, there will be a requirement that if any bit of the low-order defined bits is set to 1 then all higher-order bits of the defined low-order bits must also be set to 1, and the architecture version with which the implementation appears to comply, in problem state, will be the version corresponding to the name of the lowest-order 1 bit in the set of defined low-order PCR bits, or the current architecture version if none of these bits are 1. Also, in general the highest-order reserved bits will be used to control the availability of sets of instructions and related resources having the requirement that their availability be independent of versions of the architecture.

2.6 Other Hypervisor Resources

In addition to the resources described in the preceding sections, all hypervisor privileged instructions as well as the following resources are hypervisor resources, accessible to software only when the thread is in hypervisor state except as noted below.

- All implementation-specific resources except for privileged non-hypervisor implementation-specific SPRs. (See Section 4.4.4 for the list of the implementation-specific SPRs that are allowed to be privileged non-hypervisor SPRs.) Implementation-specific registers include registers (e.g., "HID" registers) that control hardware functions or affect the results of instruction execution. Examples include resources that disable caches, disable hardware error detection, set breakpoints, control power management, or significantly affect performance.
- ME bit of the MSR
- SPRs defined as hypervisor-privileged in Section 4.4.4. (Note: Although the Time Base, the PURR, and the SPURR can be altered only by a hypervisor program, the Time Base can be read by all programs and the PURR and SPURR can be read when the thread is in privileged state.)

The contents of a hypervisor resource can be modified by the execution of an instruction (e.g., ***mtspr***) only in hypervisor state ($MSR_{HV\ PR}$ = 0b10). An attempt to modify the contents of a given hypervisor resource, other than MSR_{ME}, in privileged but non-hypervisor state ($MSR_{HV\ PR}$ = 0b00) causes a Privileged Instruction type Program Interrupt when $LPCR_{EVIRT}$=0 and a Hypervisor Emulation Assistance interrupt when $LPCR_{EVIRT}$=1. An attempt to modify MSR_{ME} in privileged but non-hypervisor state is ignored (i.e., the bit is not changed).

Programming Note

Because the SPRs listed above are privileged for writing, an attempt to modify the contents of any of these SPRs in problem state (MSR_{PR}=1) using ***mtspr*** causes a Privileged Instruction type Program exception, and similarly for MSR_{ME}.

2.7 Sharing Hypervisor Resources

Shared SPRs are SPRs that are accessible to multiple threads. Changes to shared SPRs made by one thread are immediately readable (using ***mfspr***) by all other threads sharing the SPR.

The LPIDR and DPDES must appear to software to be shared among threads of a sub-processor (see Section 2.8). If the implementation does not support sub-processors, the LPIDR and DPDES must be shared among all threads of the multi-threaded processor.

Certain additional hypervisor resources may be shared among threads. Programs that modify these resources must be aware of this sharing, and must allow for the fact that changes to these resources may affect more than one thread.

The following additional resources may be shared among threads.

- HRMOR (see Section 2.3)
- LPIDR (see Section 2.4)
- PCR (see Section 2.5)

- PVR (see Section 4.3.1)
- RPR (see Section 4.3.9)
- PTCR (see Section 5.7.6.1)
- AMOR (see Section 5.7.13.1)
- HMEER (see Section 6.2.10)
- Time Base (see Section 7.2)
- Virtual Time Base (see Section 7.3)
- Hypervisor Decrementer (see Section 7.5)
- certain implementation-specific registers or implementation-specific fields in architected registers

The set of resources that are shared is implementation-dependent.

Threads that share any of the resources listed above, with the exception of the PTCR, the PVR and the HRMOR, must be in the same partition.

For each field of the LPCR, except the AIL, EVIRT, ONL, HDICE, MER,PECE, HEIC, and HVICE fields, software must ensure that the contents of the field are identical among all threads that are in the same partition and are not in hypervisor state.

2.8 Sub-Processors

Hardware is allowed to sub-divide a multi-threaded processor into "sub-processors" that appear to privileged programs as multi-threaded processors with fewer threads. Such a multi-threaded processor appears to the hypervisor as a processor with a number of threads equal to the sum of all sub-processor threads, and in which the LPIDR for each sub-processor must appear to be shared among all threads of that sub-processor.

2.9 Thread Identification Register (TIR)

The TIR is a 64-bit read-only register that contains the thread number, which is a binary number corresponding to the thread.

For implementations that do not support sub-processors, the thread number of a thread is unique among all thread numbers of threads on the multi-threaded processor.

For implementations that support sub-processors, the value of this register depends on whether it is read in hypervisor or privileged, non-hypervisor state as follows.

- When this register is read in privileged, non-hypervisor state, the thread number is unique among all thread numbers of threads on the sub-processor.
- When this register is read in hypervisor state, the thread number is unique among all thread numbers of threads on the multi-threaded processor.

Threads are numbered sequentially, with valid values ranging from 0 to t-1, where t is the number of threads implemented. A thread for which TIR = n is referred to as "thread n."

The layout of the TIR is shown below.

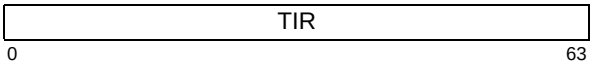


Figure 4. Thread Identification Register

Access to the TIR is privileged.

Since the thread number contained in this register is different if it is read in hypervisor from when it is read in privileged, non-hypervisor state in implementations that support sub-processors, the following conventions are used.

- The value returned in privileged, non-hypervisor state is referred to as the "privileged thread number."
- The value returned in hypervisor state is referred to as the "hypervisor thread number."

2.10 Hypervisor Interrupt Little-Endian (HILE) Bit

The Hypervisor Interrupt Little-Endian (HILE) bit is a bit in an implementation-dependent register or similar mechanism. The contents of the HILE bit are copied into MSR_{LE} by interrupts that set MSR_{HV} to 1 (see Section 6.5), to establish the Endian mode for the interrupt handler. The HILE bit is set, by an implementation-dependent method, only during system initialization.

The contents of the HILE bit must be the same for all threads under the control of a given instance of the hypervisor; otherwise all results are undefined.

Chapter 3. Branch Facility

3.1 Branch Facility Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Branch Facility that are not covered in Book I.

3.2 Branch Facility Registers

3.2.1 Machine State Register

The Machine State Register (MSR) is a 64-bit register. This register defines the state of the thread. On interrupt, the MSR bits are altered in accordance with Figure 65 on page 1064. The MSR can also be modified by the ***mtmsr***[***d***], ***rfscv***, ***rfid***, and ***hrfid*** instructions. It can be read by the ***mfmsr*** instruction.

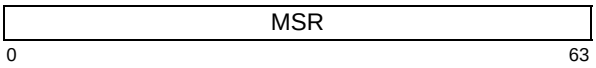


Figure 5. Machine State Register

Below are shown the bit definitions for the Machine State Register.

Bit **Description**

0 ***Sixty-Four-Bit Mode*** (SF)

- 0 The thread is in 32-bit mode.
- 1 The thread is in 64-bit mode.

1:2 Reserved

3 ***Hypervisor State*** (HV)

- **0** The thread is not in hypervisor state.
- **1** If MSR_{PR}=0 the thread is in hypervisor state; otherwise the thread is not in hypervisor state.

Programming Note

The privilege state of the thread is determined by MSR_{HV} and MSR_{PR}, as follows.

HV	PR	
0	0	privileged
0	1	problem
1	0	hypervisor
1	1	problem

Hypervisor state is also a privileged state (MSR_{PR} = 0). All references to “privileged state” in the Books include hypervisor state unless otherwise stated or obvious from context.

MSR_{HV} can be set to 1 only by the *System Call* instruction and some interrupts. It can be set to 0 only by ***rfid*** and ***hrfid***.

It is possible to run an operating system in an environment that lacks a hypervisor, by always having MSR_{HV} = 1 and using $MSR_{HV\ PR}$ = 10 for the operating system (effectively, the OS runs in hypervisor state) and $MSR_{HV\ PR}$ = 11 for applications.

4 Reserved

5 Software must ensure that this bit contains 0; otherwise the results of executing all instructions are boundedly undefined.

Programming Note

This bit is initialized to 0 by hardware at system bringup. The handling of this bit by interrupts and by the ***rfid***, ***hrfid***, and ***rfscv*** instructions is such that, unless software deliberately sets the bit to 1, the bit will continue to contain 0.

6:28 Reserved

29:30 ***Transaction State*** (TS)

- 00 Non-transactional
- 01 Suspended
- 10 Transactional
- 11 Reserved

Changes to MSR_{TS} that are caused by Transactional Memory instructions, and by invocation of the transaction's failure handler, take effect immediately (even though these instructions and events are not context synchronizing).

31 ***Transactional Memory Available*** (TM)

0 The thread cannot execute any Transactional Memory instructions or access any Transactional Memory registers.
1 The thread can execute Transactional Memory instructions and access Transactional Memory registers unless the Transactional Memory facility has been made unavailable by some other register.

32:37 Reserved

38 ***Vector Available*** (VEC)

0 The thread cannot execute any vector instructions, including vector loads, stores, and moves.
1 The thread can execute vector instructions unless they have been made unavailable by some other register.

39 Reserved

40 ***VSX Available*** (VSX)

0 The thread cannot execute any VSX instructions, including VSX loads, stores, and moves.
1 The thread can execute VSX instructions unless they have been made unavailable by some other register.

Programming Note

An application binary interface defined to support Vector-Scalar operations should also specify a requirement that MSR_{FP} and MSR_{VEC} be set to 1 whenever MSR_{VSX} is set to 1.

41:47 Reserved

48 ***External Interrupt Enable*** (EE)

0 External, Decrementer, Performance Monitor, and Privileged Doorbell interrupts are disabled.
1 External, Decrementer, Performance Monitor, and Privileged Doorbell interrupts are enabled.

This bit also affects whether Hypervisor Decrementer, Hypervisor Maintenance, and Directed Hypervisor Doorbell interrupts are enabled; see Section 6.5.12 on page 1077, Section 6.5.19 on page 1086, and Section 6.5.20 on page 1086.

49 ***Problem State*** (PR)

0 The thread is in privileged state.
1 The thread is in problem state.

Programming Note

Any instruction that sets MSR_{PR} to 1 also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

50 ***Floating-Point Available*** (FP)

0 The thread cannot execute any floating-point instructions, including floating-point loads, stores, and moves.
1 The thread can execute floating-point instructions unless they have been made unavailable by some other register.

51 ***Machine Check Interrupt Enable*** (ME)

0 Machine Check interrupts are disabled.
1 Machine Check interrupts are enabled.

This bit is a hypervisor resource; see Chapter 2., “Logical Partitioning (LPAR) and Thread Control”, on page 927.

Programming Note

The only instructions that can alter MSR_{ME} are ***rfid*** and ***hrfid***.

52 ***Floating-Point Exception Mode 0*** (FE0)

See below.

53:54 ***Trace Enable*** (TE)

00 Trace Disabled: The thread executes instructions normally.
01 Branch Trace: The thread generates a Branch type Trace interrupt after completing the execution of a branch instruction, whether or not the branch is taken.
10 Single Step Trace: The thread generates a Single-Step type Trace interrupt after successfully completing the execution of the next instruction, unless that instruction is an ***hrfid, rfid***, ***rfscv***, or a *Power-Saving Mode* instruction, all of which are never traced. Successful completion means that the instruction caused no other interrupt and, if the processor is in the Transactional state, is not a disallowed instruction (e.g., ***dcbf***) or an ***mtspr*** specifying an SPR that is not part of the checkpointed registers and is not the GSR (see Section 5.3.1 of Book II).
11 Reserved.

Branch tracing need not be supported. If the function is not implemented, the 0b01 bit encoding is treated as reserved.

55 ***Floating-Point Exception Mode 1*** (FE1)

See below.

56:57 Reserved

58 ***Instruction Relocate*** (IR)

0 Instruction address translation is disabled.
1 Instruction address translation is enabled.

Programming Note

See the Programming Note in the definition of MSR_{PR}.

59 ***Data Relocate*** (DR)

0 Data address translation is disabled. Effective Address Overflow (EAO) (see Book I) does not occur.
1 Data address translation is enabled. EAO causes a Data Storage interrupt.

Programming Note

See the Programming Note in the definition of MSR_{PR}.

60 Reserved

61 ***Performance Monitor Mark*** (PMM)

This bit is used by software in conjunction with the Performance Monitor, as described in Chapter 9.

Programming Note

Software can use this bit as a process-specific marker which, in conjunction with $MMCR0_{FCM0\ FCM1}$ (see Section 9.4.4) and MMCR2 (see Section 9.4.6), permits events to be counted on a process-specific basis. (The bit is saved by interrupts and restored by ***rfid***.)

Common uses of the PMM bit include the following.

- All counters count events for a few selected processes. This use requires the following bit settings.
  - MSR_{PMM}=1 for the selected processes, MSR_{PMM}=0 for all other processes
  - $MMCR0_{FCM0}$=1
  - $MMCR0_{FCM1}$=0
  - MMCR2 = 0x0000
- All counters count events for all but a few selected processes. This use requires the following bit settings.
  - MSR_{PMM}=1 for the selected processes, MSR_{PMM}=0 for all other processes
  - $MMCR0_{FCM0}$=0
  - $MMCR0_{FCM1}$=1
  - MMCR2 = 0x0000

Notice that for both of these uses a mark value of 1 identifies the “few” processes and a mark value of 0 identifies the remaining “many” processes. Because the PMM bit is set to 0 when an interrupt occurs (see Figure 65 on page 1064), interrupt handlers are treated as one of the “many”. If it is desired to treat interrupt handlers as one of the “few”, the mark value convention just described would be reversed.

If only a specific counter n is to be frozen, $MMCR0_{FCM0\ FCM1}$ is set to 0b00, and $MMCR2_{FCnM0}$ and $MMCR2_{FCnM1}$ instead of $MMCR0_{FCM0}$ and $MMCR0_{FCM1}$ are set to the values described above.

62 ***Recoverable Interrupt*** (RI)

0 Interrupt is not recoverable.
1 Interrupt is recoverable.

Additional information about the use of this bit is given in Sections 6.4.3, “Interrupt Processing” on page 1059, 6.5.1, “System Reset Interrupt” on page 1065, and 6.5.2, “Machine Check Interrupt” on page 1067.

63 ***Little-Endian Mode*** (LE)

0 The thread is in Big-Endian mode.
1 The thread is in Little-Endian mode.

Programming Note

The only instructions that can alter MSR_{LE} are ***rfid*** and ***hrfid,*** and ***rfscv***.

The Floating-Point Exception Mode bits FE0 and FE1 are interpreted as shown below. For further details see Book I.

FE0	FE1	Mode
0	0	Ignore Exceptions
0	1	Imprecise Nonrecoverable
1	0	Imprecise Recoverable
1	1	Precise

3.2.2 State Transitions Associated with the Transactional Memory Facility

Updates to MSR_{TS} and MSR_{TM} caused by ***rfebb***, ***rfid***, ***rfscv***, ***hrfid***, or ***mtmsrd*** occur as described in Table 3. The value written, and whether or not the instruction causes an interrupt, are dependent on the current values of MSR_{TS} and MSR_{TM}, and the values being written to these fields. When the setting of MSR_{TS} causes an illegal state transition, a TM Bad Thing type Program interrupt is generated.

Programming Note

The transition rules are the same for ***mtmsrd*** as for the ***rfid***-type instructions because if a transition were illegal for ***mtmsrd*** but allowed for ***rfid***, or vice versa, software could use the instruction for which the transition is allowed to achieve the effect of the other instruction.

Table 3 shows all the transaction state transitions that can be requested by ***rfebb***, ***rfid***, ***rfscv***, ***hrfid***, and ***mtmsrd***. If $PCR_{v2.06}$=1 and the instruction requests a transition to problem state, transaction state transitions that the table shows as legal and as resulting in the thread being in Transactional or Suspended state instead cause a TM Bad Thing type Program interrupt; see Section 6.5.9. (The preceding sentence does not apply to ***rfebb***, because ***rfebb*** cannot cause a change of privilege state, and cannot be executed in problem state when $PCR_{v2.06}$=1.) In the table, the contents of MSR_{TS} and MSR_{TM} are abbreviated in the form AB, where A represents MSR_{TS} (N, T or S) and B represents MSR_{TM} (0 or 1). “x” in the “B” position means that the entry covers both MSR_{TM} values, with the same value applying in all columns of a given row for a given instance of the transition. (E.g., the first row means that the transition from N0 to N0 is allowed and results in N0, and that the transition from N0 to N1 is allowed and results in N1.) “Input $MSR_{TS}MSR_{TM}$” in the second column refers to the MSR_{TS} and MSR_{TM} values supplied by CTR for ***rfscv***, BESCR for ***rfebb*** (just the TS value), SRR1 for ***rfid***, HSRR1 for ***hrfid***, or register RS for ***mtmsrd***.

Current $MSR_{TS}MSR_{TM}$	Input $MSR_{TS}MSR_{TM}$	Resulting MSR_{TS} MSR_{TM}	Comments
N0	Nx	Nx	May occur in the context of a Transactional Memory type of Facility Unavailable interrupt handler, enabling/disabling transactions for user-level applications.
	All others - Illegal[1]	N0	
T0	N/A		Unreachable state
S0	N0[2]	S0	Operating system code that is not TM aware may attempt to set TS and TM to zero, thinking they're reserved bits. Change is suppressed.
	T1	T1	May occur at an ***rfid*** returning to an application whose transaction was suspended on interrupt.
	Sx	Sx	This case may occur for an ***rfid*** returning to an application whose suspended transaction was interrupted.
	All others - Illegal[1]	S0	
N1	Nx	Nx	After a ***treclaim***, the OS dispatches Nx program.
	All others -IIllegal[1]	N0	
T1	all	N1	Disallowed instructions in Transactional state
S1	T1	T1	May occur after ***trechkpt.*** when returning to an application.
	Sx	Sx	
	All others - Illegal[1]	S0	

Notes:
1. Generate TM Bad Thing type Program interrupt. "All others" includes all attempts to set MSR_{TS} to 0b11 (reserved value).
2. Instruction completes, change to MSR_{TM} suppressed, except when attempted by ***rfebb***, in which case the result is a TM Bad Thing type Program interrupt.

Table 3: Transaction state transitions that can be requested by ***rfebb***, ***rfid***, ***rfscv***, ***hrfid***, and ***mtmsrd***.

Programming Note

For ***rfscv, [h]rfid***, and ***mtmsrd***, the attempted transition from S0 to N0 is suppressed in order that interrupt handlers that are "unaware" of transactional memory, and load an MSR value that has not been updated to take account of transactional memory, will continue to work correctly. (If the interrupt occurs when a transaction is running or suspended, the interrupt will set $MSR_{TS||TM}$ to S0. If the interrupt handler attempts to load an MSR value that has not been updated to take account of transactional memory, that MSR value will have TS || TM = N0. It is desirable that the interrupt handler remain in state S0, so that it can return normally to the interrupted transaction.)

The problem solved by suppressing this transition does not apply to ***rfebb***, so for ***rfebb*** an attempt to transition from S0 to N0 is not suppressed, and instead causes a TM Bad Thing type Program interrupt.

3.2.3 Processor Stop Status and Control Register (PSSCR)

The layout of the PSSCR is shown below.

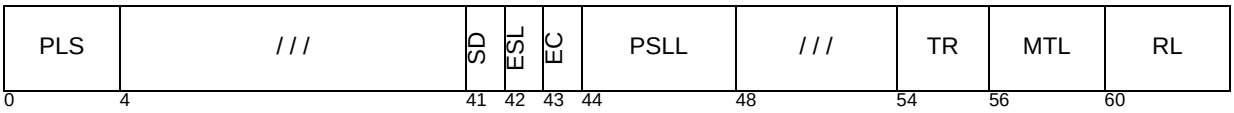


Figure 6. Processor stop Status and Control Register

The contents of the PSSCR control the operation of the ***stop*** instruction and provide status indicating the level of power saving that was entered while in power-saving mode.

All fields of this register can be read and written by the hypervisor using either hypervisor SPR 855 or privileged SPR 823. A subset of the fields of this register can be read and written in privileged non-hypervisor state using privileged SPR 823, as specified below. Fields that can only be read or written by the hypervisor are indicated below; all other fields can be read or written in either privileged non-hypervisor or hypervisor states. When a field that is accessible only to the hypervisor is accessed in privileged non-hypervisor state, writes have no effect and reads return 0s regardless of the value of the field.

The bits and their meanings are as follows.

0:3 **Power-Saving Level Status (PLS)**

Hardware sets this field to the highest power-saving level that the thread entered between the time when the ***stop*** instruction is executed and when the thread exits power-saving mode. See the description of the SD field for the value returned in this field when the PSSCR is read.

Programming Note

Since the power-saving level entered during power-saving mode may vary with time, the PLS field may not indicate the power-saving level that existed at exit from power-saving mode.

4:40 Reserved

41 **Status Disable** (SD)

This field is accessible only to the hypervisor.

0 The current value of the PLS field is returned in the PLS field when reading the PSSCR (using ***mfspr***).
1 0's are returned in the PLS field when reading the PSSCR (using ***mfspr***).

Programming Note

Before dispatching an OS, the hypervisor may initialize this field to 1 in order to prevent the OS from reading the Power-Saving Level Status (PLS) field. This may be necessary in secure environments since an OS may be capable of detecting the presence of another OS on the same processor by observing the state of the PLS field after exiting power-saving mode.

42 **Enable State Loss (ESL)**

This field is accessible only to the hypervisor.

0 State loss while in power-saving mode is controlled by the RL, MTL, and PSLL fields.
1 Non-hypervisor state loss is allowed while in power-saving mode in addition to state loss controlled by the RL, MTL, and PSLL fields.

If this field is set to 1 when the ***stop*** instruction is executed in privileged non-hypervisor state, a Hypervisor Facility Unavailable interrupt occurs. See Section 6.5.26.

For power-saving levels that allow loss of the LPCR, implementations must provide the means to exit power-saving mode upon the occurrence of a System Reset exception and any of the exceptions that were enabled by the PECE field when the ***stop*** instruction was executed. For this case, the implementation is also allowed to exit on the occurrence of any exceptions that were disabled by the PECE as well.

Programming Note

When state loss occurs, thread resources such as SPRs, GPRs, address translation resources, etc. may be powered off or allocated to other threads during power-saving mode. The amount of state loss for various combinations of ESL, RL, and MTL values is implementation dependent, subject to the restrictions specified in Section 3.3.2.

43 **Exit Criterion** (EC)

This field is accessible only to the hypervisor.

0 Hardware will exit power-saving mode when the exception corresponding to any system-caused interrupt occurs.
Power-saving mode is exited either at the instruction following the stop (if MSR_{EE}=0) or in the corresponding interrupt handler (if MSR_{EE}=1).

1 Provided $LPCR_{PECE}$ is not lost, hardware will exit power-saving mode only when a System Reset exception or one of the events specified in $LPCR_{PECE}$ occurs. If the event is a Machine Check exception, then a Machine Check interrupt occurs; otherwise a System Reset interrupt occurs, and the contents of SRR1 indicate the event that caused exit from power-saving mode.

When the ***stop*** instruction is executed in hypervisor state, the hypervisor must set the ESL field to the same value as this field. Also, if the RL or MTL fields are set to values that allow state loss, then fields ESL and EC must both be set to 1. Other combinations of the values of the ESL, EC, RL, and MTL fields are reserved for future use.

Architecture Note

Other combinations of the values of the ESL, EC, RL, and MTL fields may be allowed in a future version of the architecture in order to provide additional functionality.

If this field is set to 1 when the ***stop*** instruction is executed in privileged non-hypervisor state, a Hypervisor Facility Unavailable interrupt occurs. See Section 6.5.26.

Programming Note

In order to enable an OS to enter power-saving mode without hypervisor involvement, both the EC and ESL bits must be set to 0s. When this is done, OS execution of the ***stop*** instruction will not cause hypervisor involvement provided that bits RL and and MTL are less than or equal to PSLL. See Section 6.5.26 for details.

44:47 **Power-Saving Level Limit** (PSLL)

This field is accessible only to the hypervisor.

This field limits the power-saving level that may be entered or transitioned into when the ***stop*** instruction is executed in privileged non-hypervisor state; when the ***stop*** instruction is executed in hypervisor state, this field is ignored.

48:53 Reserved

54:55 **Transition Rate** (TR)

This field is used to specify the relative rate at which the power-saving level increases during power-saving mode. The rate of power-saving level increase corresponding to each value is implementation-dependent, and monotonically increasing with the value specified.

56:59 **Maximum Transition Level** (MTL)

If the value of this field is greater than the value of the Power-Saving Level Limit (PSLL) field when ***stop*** is executed in privileged non-hypervisor state, a Hypervisor Facility Unavailable interrupt occurs. See Section 6.5.26 of Book III.

Otherwise, if the value of this field is greater than the value of the RL field, the power-saving level is allowed to increase from the value in the RL field up to the value of this field during power-saving mode.

If this field is less than or equal to the value of the PSLL field when ***stop*** is executed in privileged non-hypervisor state, this field is used to specify the maximum power-saving level that can be reached during power-saving mode provided that the value of this field is greater than the value of the RL field. If this field is less than the Requested Level (RL) field when ***stop*** is executed hardware is not allowed to increase the power-saving level during power-saving mode beyond the value indicated in the RL field.

60:63 **Requested Level** (RL)

This field is used to specify the power-saving level that is to be entered when the ***stop*** instruction is executed.

If the value of this field is greater than the value of the Power-Saving Level Limit (PSLL) field when ***stop*** is executed in privileged non-hypervisor state, a Hypervisor Facility Unavailable interrupt occurs.

Programming Note

The Hypervisor Facility Unavailable interrupt occurs when a privileged non-hypervisor program executes ***stop*** when $PSSCR_{RL} > PSSCR_{PSLL}$ so that the Hypervisor may decide whether or not to allow the requested loss of state to occur.

If the hypervisor decides that some loss of state is acceptable, it may choose to re-execute ***stop*** after either setting $PSSCR_{MTL}$ to a value that causes state loss, or setting both $PSSCR_{RL}$ and $PSSCR_{MTL}$ to values that cause state loss. When the thread exits power-saving mode, the hypervisor can quickly determine whether any resources were actually lost and need to be restored.

3.3 Branch Facility Instructions

3.3.1 System Linkage Instructions

These instructions provide the means by which a program can call upon the system to perform a service, and by which the system can return from performing a service or from processing an interrupt.

The *System Call* instruction is described in Book I, but only at the level required by an application programmer. A complete description of this instruction appears below.

System Call *SC-form*

sc LEV

17	///	///	//	LEV	//	1	/
0	6	11	16	20	27	30	31

```
SRR0 ←iea CIA + 4
SRR1 33:36 42:47 ← 0
SRR1 0:32 37:41 48:63 ← MSR 0:32 37:41 48:63
MSR ← new_value (see below)
NIA ← 0x0000_0000_0000_0C00
```

The effective address of the instruction following the *System Call* instruction is placed into SRR0. Bits 0:32, 37:41, and 48:63 of the MSR are placed into the corresponding bits of SRR1, and bits 33:36 and 42:47 of SRR1 are set to zero.

Then a System Call interrupt is generated. The interrupt causes the MSR to be set as described in Section 6.5, “Interrupt Definitions” on page 1063. The setting of the MSR is affected by the contents of the LEV field. LEV values greater than 1 are reserved. Bits 0:5 of the LEV field (instruction bits 20:25) are treated as a reserved field.

The interrupt causes the next instruction to be fetched from effective address 0x0000_0000_0000_0C00.

This instruction is context synchronizing.

Special Registers Altered:

SRR0 SRR1 MSR

Programming Note

If LEV=1 the hypervisor is invoked. This is the only way that executing an instruction can cause hypervisor state to be entered.

Because this instruction is not privileged, it is possible for application software to invoke the hypervisor. However, such invocation should be considered a programming error.

Programming Note

sc serves as both a basic and an extended mnemonic. The Assembler will recognize an ***sc*** mnemonic with one operand as the basic form, and an ***sc*** mnemonic with no operand as the extended form. In the extended form the LEV operand is omitted and assumed to be 0.

System Call Vectored *SC-form*

scv LEV

17	///	///	//	LEV	//	0	1
0	6	11	16	20	27	30	31

```
LR ← CIA + 4
CTR33:36 42:47 ← undefined
CTR0:32 37:41 48:63 ← MSR0:32 37:41 48:63
MSR ← new_value (see below)
NIA ← (see below)
```

The effective address of the instruction following the *System Call Vectored* instruction is placed into the Link Register. Bits 0:32, 37:41, and 48:63 of the MSR are placed into the corresponding bits of Count Register, and bits 33:36 and 42:47 of Count Register are set to undefined values.

Then a System Call Vectored interrupt is generated. The interrupt causes the MSR to be altered as described in Section 6.5.

The interrupt causes the next instruction to be fetched as specified in $LPCR_{AIL}$ (see to Section 2.2).

The SRRs are not affected.

This instruction is context synchronizing.

Special Registers Altered:

LR CTR MSR

Return From System Call Vectored *XL-form*

rfscv

19	///	///	///	82	/
0	6	11	16	21	31

```
if (MSR29:31 ¬= 0b010 | CTR29:31 ¬= 0b000) then
   MSR29:31 ← CTR29:31
MSR48 ← CTR48 | CTR49
MSR58 ← CTR58 | CTR49
MSR59 ← CTR59 | CTR49
MSR0:2 4:28 32 37:41 49:50 52:57 60:63←CTR0:2 4:28 32 37:41 49:50 52:57
60:63
NIA ←iea LR0:61 || 0b00
```

If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of the Count Register are not equal to 0b000, then the value of bits 29 through 31 of the Count Register is placed into bits 29 through 31 of the MSR. The result of ORing bits 48 and 49 of the Count Register is placed into MSR_{48}. The result of ORing bits 58 and 49 of the Count Register is placed into MSR_{58}. The result of ORing bits 59 and 49 of the Count Register is placed into MSR_{59}. Bits 0:2, 4:28, 32, 37:41, 49:50, 52:57, and 60:63 of the Count Register are placed into the corresponding bits of the MSR.

If the instruction attempts to cause an illegal transaction state transition or, when TM is made unavailable in problem state by the PCR, attempts to cause a transition to problem state and also a transaction state transition that Table 3 on page 947 shows as legal and as resulting in the thread being in Transactional or Suspended state, a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the ***rfscv*** instruction. Otherwise, if the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $LR_{0:61}$ || 0b00 (when SF=1 in the new MSR value) or $^{32}0$ || $LR_{32:61}$ || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0 or HSRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is privileged and context synchronizing.

Special Registers Altered:

MSR

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

I

Return From Interrupt Doubleword XL-form

rfid

19	///	///	///	18	/
0	6	11	16	21	31

```
MSR51 ← (MSR3 & SRR151) | ((¬MSR3) & MSR51)
MSR3 ← MSR3 & SRR13
if (MSR29:31 ¬= 0b010 | SRR129:31 ¬= 0b000) then
   MSR29:31 ← SRR129:31
MSR48 ← SRR148 | SRR149
MSR58 ← SRR158 | SRR149
MSR59 ← SRR159 | SRR149
MSR0:2 4:28 32 37:41 49:50 52:57 60:63←SRR10:2 4:28 32 37:41 49:50 52:57
60:63
NIA ←iea SRR00:61 || 0b00
```

If MSR_3=1 then bits 3 and 51 of SRR1 are placed into the corresponding bits of the MSR. If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of SRR1 are not equal to 0b000, then the value of bits 29 through 31 of SRR1 is placed into bits 29 through 31 of the MSR. The result of ORing bits 48 and 49 of SRR1 is placed into MSR_{48}. The result of ORing bits 58 and 49 of SRR1 is placed into MSR_{58}. The result of ORing bits 59 and 49 of SRR1 is placed into MSR_{59}. Bits 0:2, 4:28, 32, 37:41, 49:50, 52:57, and 60:63 of SRR1 are placed into the corresponding bits of the MSR.

If the instruction attempts to cause an illegal transaction state transition or, when TM is made unavailable in problem state by the PCR, attempts to cause a transition to problem state and also a transaction state transition that Table 3 on page 947 shows as legal and as resulting in the thread being in Transactional or Suspended state, a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the ***rfid*** instruction. Otherwise, if the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $SRR0_{0:61}$ || 0b00 (when SF=1 in the new MSR value) or $^{32}0$ || $SRR0_{32:61}$ || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0 or HSRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is privileged and context synchronizing.

Special Registers Altered:

MSR

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

Hypervisor Return From Interrupt Doubleword *XL-form*

hrfid

19	///	///	///	274	/
0	6	11	16	21	31

```
if (MSR29:31 ¬= 0b010 | HSRR129:31 ¬= 0b000) then
   MSR29:31 ← HSRR129:31
MSR48 ← HSRR148 | HSRR149
MSR58 ← HSRR158 | HSRR149
MSR59 ← HSRR159 | HSRR149
MSR0:28 32 37:41 49:57 60:63 ← HSRR10:28 32 37:41 49:57 60:63
NIA ←iea HSRR00:61 || 0b00
```

If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of HSRR1 are not equal to 0b000, then the value of bits 29 through 31 of HSRR1 is placed into bits 29 through 31 of the MSR. The result of ORing bits 48 and 49 of HSRR1 is placed into MSR_{48}. The result of ORing bits 58 and 49 of HSRR1 is placed into MSR_{58}. The result of ORing bits 59 and 49 of HSRR1 is placed into MSR_{59}. Bits 0:28, 32, 37:41, 49:57, and 60:63 of HSRR1 are placed into the corresponding bits of the MSR.

If the instruction attempts to cause an illegal transaction state transition or, when TM is made unavailable in problem state by the PCR, attempts to cause a transition to problem state and also a transaction state transition that Table 3 on page 947 shows as legal and as resulting in the thread being in Transactional or Suspended state, a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the *hrfid* instruction. Otherwise, if the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $HSRR0_{0:61}$ || 0b00 (when SF=1 in the new MSR value) or $^{32}0$ || $HSRR0_{32:61}$ || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0 or HSRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is hypervisor privileged and context synchronizing.

Special Registers Altered:

MSR

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

3.3.2 Power-Saving Mode

Power-Saving Mode is a mode in which the thread does not execute instructions and may consume less power than it would if it were not in power-saving mode.

There are 16 levels of power savings, designated as levels 0-15. For each power-saving level, the power consumed may be less than or equal to the power consumed in the next-lower level, and the time required for the thread to exit power-saving mode and resume execution may be greater than or equal that of the next-lower level.

When the thread is in power-saving mode, some resource state may be lost. The state that may be lost while in each power-saving level is implementation dependent, with the following restrictions.

- For $PSSCR_{ESL}$ = 0 and power-saving level 0000, no thread state is lost.
- There must be a power-saving level in which the Decrementer and all hypervisor resources are maintained as if the thread was not in power-saving mode, and in which sufficient information is maintained to allow the hypervisor to resume execution.
- The amount of state loss in a given level is less than or equal to the amount of state loss in the next higher level.
- The state of all read-only resources and the HRMOR is always maintained.

Programming Note

For the power-saving level corresponding to the second item above, if the state of the Decrementer were not maintained and updated as if the thread was not in power-saving mode, Decrementer exceptions would not reliably cause exit from this power-saving level even if Decrementer exceptions were enabled to cause exit.

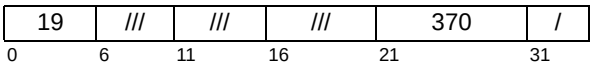
The thread can be put in power-saving mode by executing the ***stop*** instruction. As specified below, this instruction stops execution immediately after the stop instruction is executed, and the thread is put into power-saving mode. The power-saving level that is entered depends on the contents of the PSSCR (see Section 3.2.3).

3.3.2.1 Power-Saving Mode Instruction

The ***stop*** instruction is used to stop instruction fetching and execution and put the thread into power-saving mode. The thread remains in power-saving mode until a system reset exception or an event that is enabled to cause exit from power-saving mode occurs. (See the definition of $PSSCR_{EC}$ in Section 3.2.3.)

stop ***XL-form***

stop



The thread is placed into power-saving mode and execution is stopped.

The power-saving level that is entered is determined by the contents of the PSSCR (see Section 3.2.3). The thread state that is maintained depends on the power-saving level that is entered. The thread state that is maintained at each power-saving level is implementation-dependent, subject to the restrictions specified in Section 3.3.2.MSR_{EE}=0) or in the corresponding interrupt handler (if MSR_{EE}=1).

Programming Note

If stop was executed when $PSSCR_{EC}$=0, then $PSSCR_{ESL}$ must also be set to 0 and $PSSCR_{RL\ MTL}$ must be set to values that do not allow state loss. (See the definition of the EC bit description in Section 3.3.2.) This guarantees that the state of MSR_{EE} is not lost.

Programming Note

If stop was executed when $PSSCR_{EC}$=0 and MSR_{EE}=0 (in order to avoid the hang condition described in the above Programming Note), MSR_{EE} should be set to 1 after power-saving mode is exited in order to take the interrupt corresponding to the exception that caused exit from power-saving mode.

The thread remains in power-saving mode until either a System Reset exception or certain other events occur. The events that may cause exit from power-saving mode are specified by $PSSCR_{EC}$ and $LPCR_{PECE}$. If the event that causes the exit is a System Reset, Machine Check, or Hypervisor Maintenance exception, resource state that would be lost if the exception occurred when the thread was not in power-saving mode may be lost.

An attempt to execute this instruction in Suspended state will result in a TM Bad Thing type Program interrupt.

This instruction is privileged and context synchronizing.

Special Registers Altered:

None

3.3.2.2 Entering and Exiting Power-Saving Mode

Before software executes the ***stop*** instruction, the PSSCR is initialized. If the ***stop*** instruction is to be used by the OS, the hypervisor initializes the fields that are accessible only to the hypervisor before dispatching the OS. These fields include the SD, ESL, EC, and PSLL fields. See the Programming Notes for these fields in Section 3.2.3 for additional information.

If the ***stop*** instruction is to be executed by the hypervisor when $PSSCR_{EC}$=1, the $LPCR_{PECE}$ must be set to the desired value (see Section 2.2). Depending on the implementation and the power-saving level to be entered, it may also be necessary to save the state of certain resources and perform synchronization procedures to ensure that all stores have been performed with respect to other threads or mechanisms that use the storage areas before executing the ***stop***. See the the User's Manual for the implementation for details.

Software must also specify the requested and maximum power-saving level limit fields (i.e RL and MTL fields), and the Transition Rate (TR) field in the PSSCR in order to bound the range of power-saving modes that can be entered. If the value of the RL field is greater than or equal to the value of the MTL field, the power-saving level will not increase from the initial level during power-saving mode.

Programming Note

If MSR_{EE}=1 when the stop instruction is executed, then the interrupt corresponding to the exception that was expected to cause exit from power-saving mode may occur immediately prior to execution of the ***stop*** instruction. If this occurs, the result may be a software hang condition since the exception that was expected to cause exit from power-saving mode has already occurred.

The above software hang condition can be prevented by setting MSR_{EE}=0 prior to executing ***stop***.

After the thread has entered power-saving mode with $PSSCR_{EC}$=0, any exception may cause exit from power-saving mode. When an exception occurs, power-saving mode is exited either at the instruction following the stop (if After the thread has entered power-saving mode with $PSSCR_{EC}$=1, only the System Reset or Machine Check exceptions and the exceptions enabled in $LPCR_{PECE}$ will cause exit. If the event

that causes exit is a Machine Check exception, then a Machine Check interrupt occurs; otherwise a System Reset interrupt occurs, and the contents of SRR1 indicate the exception that caused exit from power-saving mode.

If the hypervisor has set $PSSCR_{SD}$=0 prior to when the ***stop*** instruction is executed, the instruction following the ***stop*** may typically be a ***mfspr*** in order to read the contents of $PSSCR_{PLS}$ to determine the maximum power-saving level that was entered during power-saving mode.

3.4 Event-Based Branch Facility and Instruction

The Event-Based Branch facility is described in Chapter 7 of Book II, but only at the level required by the application program.

Event-based branches can only occur in problem state and when event-based branches and exceptions have been enabled in the FSCR and HFSCR, and BESCR$_{GE}$=1. Additionally, the following additional bits must be set to one in order to enable EBB exceptions specific to a given function to occur.

- MMCR0$_{EBE}$ and BESCR$_{PME}$ must be set to 1 to enable Performance Monitor event-based exceptions.
- BESCR$_{EE}$ must be set to 1 to enable External event-based exceptions.

If an event-based exception exists (as indicated by BESCR$_{PMEO}$=1 or BESCR$_{EEO}$=1) when MSR$_{PR}$=0, the corresponding event-based branch will occur when MSR$_{PR}$=1, FSCR$_{EBB}$=1, HFSCR$_{EBB}$=1, and BESCR$_{GE}$=1.

Programming Note

Software EBB handlers should ensure that previous exceptions have been cleared (by setting BESCR$_{PMEO}$ and/or BESCR$_{EEO}$ to 0) before re-enabling event-based branches (by setting BESCR$_{GE}$ to 1 or executing rfebb 1) in order to prevent earlier exceptions from causing additional EBBs.

If the ***rfebb*** instruction attempts to cause an illegal transaction state transition (see Section 3.2.2), a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism is the address of the ***rfebb*** instruction.

Chapter 4. Fixed-Point Facility

4.1 Fixed-Point Facility Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Fixed-Point Facility that are not covered in Book I.

4.2 Special Purpose Registers

Special Purpose Registers (SPRs) are read and written using the ***mfspr*** (page 975) and ***mtspr*** (page 974) instructions. Most SPRs are defined in other chapters of this book; see the index to locate those definitions.

4.3 Fixed-Point Facility Registers

4.3.1 Processor Version Register

The Processor Version Register (PVR) is a 32-bit read-only register that contains a value identifying the version and revision level of the implementation. The contents of the PVR can be copied to a GPR by the ***mfspr*** instruction. Read access to the PVR is privileged; write access is not provided.

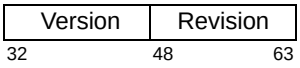


Figure 7. Processor Version Register

The PVR distinguishes between implementations that differ in attributes that may affect software. It contains two fields.

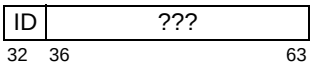
Version A 16-bit number that identifies the version of the implementation. Different version numbers indicate major differences between implementations.

Revision A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between implementations having the same version number, such as clock rate and Engineering Change level.

Version numbers are assigned by the Power ISA process. Revision numbers are assigned by an implementation-defined process.

4.3.2 Chip Information Register

The Chip Information Register (CIR) is a 32-bit read-only register that contains a value identifying the manufacturer and other characteristics of the chip on which the processor is implemented. The contents of the CIR can be copied to a GPR by the ***mfspr*** instruction. Read access to the CIR is privileged; write access is not provided.

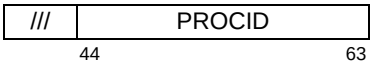


Bit	Description
32:35	**Manufacturer ID** (ID) A four-bit field that identifies the manufacturer of the chip.
36:63	Implementation-dependent.

Figure 8. Chip Information Register

4.3.3 Processor Identification Register

The Processor Identification Register (PIR) is a 32-bit register that contains a 20-bit PROCID field that can be used to distinguish the thread from other threads in the system. The contents of the PIR can be copied to a GPR by the ***mfspr*** instruction. Read access to the PIR is privileged; write access is not provided.



Bits	Name	Description
32:43		Reserved
44:63	PROCID	Thread ID

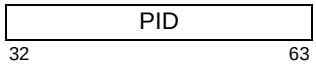
Figure 9. Processor Identification Register

The means by which the PIR is initialized are implementation-dependent.

The PIR is a hypervisor resource; see Chapter 2.

4.3.4 Process Identification Register

The layout of the Process Identification Register (PIDR) is shown in Figure 10 below.



Bit(s)	Name	Description
32:63	PID	Process Identifier

Figure 10. Process Identification Register

The contents of the PIDR identify the process to which the thread is assigned. The value is used to perform translation and manage the caching of translations. The number of PIDR bits supported is implementation-dependent.

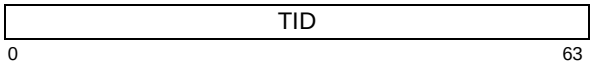
Access to the PIDR is privileged.

Programming Note

Radix tree translation assigns special meaning to PID=0, specifically indicating the operating system's kernel process. When GR=1, PIDR should not be set to zero except when MSR_{PR}=0.

4.3.5 Thread ID Register

The Thread ID Register (TIDR) is a 64-bit register that holds an identifier for the thread that is unique among threads with the same Process ID that are using accelerators. The layout of the Thread Identification Register (TIDR) is shown in Figure 11 below.



Bit(s)	Name	Description
0:63	TID	Thread Identifier

Figure 11. Thread Identification Register

An implementation may opt to implement only the least-significant *n* bits of the Thread ID Register, where $0 \le n \le 64$. The most-significant 64–*n* bits of the Thread ID Register are treated as reserved.

Access to the TIDR is privileged.

Programming Note

The TIDR is used by platform hardware to deliver a notification signal that will complete ***wait*** on the appropriate thread. This "platform notify" signal commonly reports the completion of processing by an accelerator. See Section 4.6.4, "Wait Instruction", in Book II for additional details. See platform documentation for possible synchronization requirements for changing the TID.

4.3.6 Control Register

The Control Register (CTRL) is a 32-bit register as shown below.

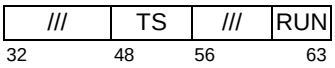


Figure 12. Control Register

The field definitions for the CTRL are shown below.

Bit(s)	Description
32:47	Reserved
48:55	**Thread State** (TS) Problem State Access Reserved Privileged Non-hypervisor State Access Bits 0:7 of this field are read-only bits that indicate the state of $CTRL_{RUN}$ for threads with privileged thread numbers 0 through 7, respectively; bits corresponding to privileged thread numbers higher than the maximum privileged thread number supported are set to 0s. Hypervisor State Access Bits 0:7 of this field are read-only bits that indicate the state of $CTRL_{RUN}$ for threads with hypervisor thread numbers 0 through 7, respectively; bits corresponding to hypervisor thread numbers higher than the maximum hypervisor thread number supported are set to 0s.
56:62	Reserved
63	RUN This bit controls an external I/O pin. This signal may be used for the following:

- driving the RUN Light on a system operator panel
- Direct External exception routing
- Performance Monitor Counter incrementing (see Chapter 9)

The RUN bit can be used by the operating system to indicate when the thread is doing useful work.

Write access to the CTRL is privileged. Reads can be performed in privileged or problem state.

4.3.7 Program Priority Register

Privileged programs may set a wider range of program priorities in the PRI field of PPR and PPR32 than may be set by problem state programs (see Chapter 3 of Book II). Problem state programs may only set values in the range of 0b001 to 0b100 unless the Problem State Priority Boost register (see Section 4.3.8) allows the value 0b101. Privileged programs may set values in the range of 0b001 to 0b110. Hypervisor software may also set 0b111. For all priorities except 0b101, if a program attempts to set a value that is not allowed for its privilege level, the PRI field remains unchanged. If a problem state program attempts to set its priority value to 0b101 when this priority value is not allowed for problem state programs, the priority is set to 0b100. The values and their corresponding meanings are as follows.

Bit(s)	Description
11:13	***Program Priority*** (PRI) 001 very low 010 low 011 medium low 100 medium 101 medium high 110 high 111 very high

4.3.8 Problem State Priority Boost Register

The Problem State Priority Boost (PSPB) register is a 32-bit register that controls whether problem state programs have access to program priority medium high. (See Section 3.1 of Book II.)

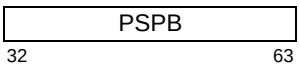


Figure 13. Problem State Priority Boost Register

A problem state program is able to set the program priority to medium high only when the PSPB of the thread contains a non-zero value.

The maximum value to which the PSPB can be set must be a power of 2 minus 1. Bits that are not required to represent this maximum value must return 0s when read regardless of what was written to them.

When the PSPB is set to a value less than its maximum value but greater than 0, its contents decrease monotonically at the same rate as the SPURR until its contents minus the amount it is to be decreased are 0 or less when a problem state program is executing on the thread at a priority of medium high.When the contents of the PSPB minus the amount it is to be decreased are 0 or less, its contents are replaced by 0.

When the PSPB is set to its maximum value or 0, its contents do not change until it is set to a different value.

Whenever the priority of a thread is medium high and either of the following conditions exist, hardware changes the priority to medium:

- the PSPB counts down to 0, or
- PSPB=0 and the privilege state of the thread is changed to problem state (MSR_{PR}=1).

4.3.9 Relative Priority Register

The Relative Priority Register (RPR) is a 64-bit register that allows the hypervisor to control the relative priorities corresponding to each valid value of PPR_{PRI}.

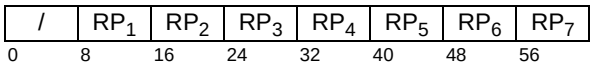


Figure 14. Relative Priority Register

Each RP_n field is defined as follows.

Bits	Meaning
0:1	Reserved
2:7	**Relative priority of priority level n:** Specifies the relative priority that corresponds to the priority corresponding to PPR_{PRI}=n, where a value of 0 indicates the lowest relative priority and a value of 0b111111 indicates the highest relative priority.

Programming Note

The hypervisor must ensure that the values of the RP_n fields increase monotonically for each n and are of different enough magnitudes to ensure that each priority level provides a meaningful difference in priority.

4.3.10 Software-use SPRs

Software-use SPRs are 64-bit registers provided for use by software.

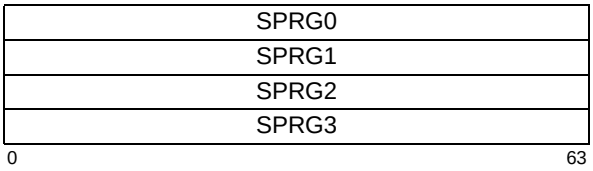


Figure 15. Software-use SPRs

SPRG0, SPRG1, and SPRG2 are privileged registers. SPRG3 is a privileged register except that the contents may be copied to a GPR in Problem state when accessed using the ***mfspr*** instruction.

Programming Note

Neither the contents of the SPRGs, nor accessing them using ***mtspr*** or ***mfspr***, has a side effect on the operation of the thread. One or more of the registers is likely to be needed by non-hypervisor interrupt handler programs (e.g., as scratch registers and/or pointers to per thread save areas).

Operating systems must ensure that no sensitive data are left in SPRG3 when a problem state program is dispatched, and operating systems for secure systems must ensure that SPRG3 cannot be used to implement a “covert channel” between problem state programs. These requirements can be satisfied by clearing SPRG3 before passing control to a program that will run in problem state.

HSPRG0 and HSPRG1 are 64-bit registers provided for use by hypervisor programs.

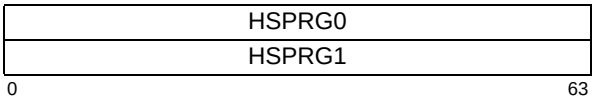


Figure 16. SPRs for use by hypervisor programs

Programming Note

Neither the contents of the HSPRGs, nor accessing them using ***mtspr*** or ***mfspr***, has a side effect on the operation of the thread. One or more of the registers is likely to be needed by hypervisor interrupt handler programs (e.g., as scratch registers and/or pointers to per thread save areas).

4.4 Fixed-Point Facility Instructions

4.4.1 Fixed-Point Load and Store Caching Inhibited Instructions

The storage accesses caused by the instructions described in this section are performed as though the specified storage location is Caching Inhibited and Guarded. The instructions can be executed only in hypervisor state. Software must ensure that the specified storage location is not in the caches. If the specified storage location is in a cache, the results are undefined.

The *Fixed-Point Load and Store Caching Inhibited* instructions must be executed only when MSR_{DR}=0. The storage location specified by the instructions must not be in storage specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded. If either of these conditions is violated, the result is a Data Storage interrupt.

Programming Note

The instructions described in this section can be used to permit a control register on an I/O device to be accessed without permitting the corresponding storage location to be copied into the caches.

The *Fixed-Point Load and Store Caching Inhibited* instructions are fixed-point *Storage Access* instructions; see Section 3.3.1 of Book I.

Load Byte and Zero Caching Inhibited Indexed *X-form*

lbzcix RT,RA,RB

31	RT	RA	RB	853	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
RT ← 56 0 || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Load Halfword and Zero Caching Inhibited Indexed *X-form*

lhzcix RT,RA,RB

31	RT	RA	RB	821	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
RT ← 48 0 || MEM(EA, 2)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Load Word and Zero Caching Inhibited Indexed *X-form*

lwzcix RT,RA,RB

31	RT	RA	RB	789	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
RT ← 32 0 || MEM(EA, 4)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Load Doubleword Caching Inhibited Indexed *X-form*

ldcix RT,RA,RB

31	RT	RA	RB	885	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
RT ← MEM(EA, 8)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). The doubleword in storage addressed by EA is loaded into RT.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Store Byte Caching Inhibited Indexed X-form

stbcix RS,RA,RB

31	RS	RA	RB	981	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 1) ← (RS)56:63
```

Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Store Halfword Caching Inhibited Indexed X-form

sthcix RS,RA,RB

31	RS	RA	RB	949	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 2) ← (RS)48:63
```

Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Store Word Caching Inhibited Indexed X-form

stwcix RS,RA,RB

31	RS	RA	RB	917	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 4) ← (RS)32:63
```

Let the effective address (EA) be the sum (RA|0)+ (RB). $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Store Doubleword Caching Inhibited Indexed X-form

stdcix RS,RA,RB

31	RS	RA	RB	1013	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b + (RB)
MEM(EA, 8) ← (RS)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). (RS) is stored into the doubleword in storage addressed by EA.

The storage access caused by this instruction is performed as though the specified storage location is Caching Inhibited and Guarded.

This instruction is hypervisor privileged.

Special Registers Altered:

None

4.4.2 OR Instruction

or Rx,Rx,Rx can be used to set PPR_{PRI} (see Section 4.3.7) as shown in Figure 17. For all priorities except medium high, PPR_{PRI} remains unchanged if the privilege state of the thread executing the instruction is lower than the privilege indicated in the figure. For priority medium high, PPR_{PRI} is set to medium if the thread executing the instruction is in problem state and medium high priority is not allowed for problem state programs. (The encodings available to problem state programs, as well as encodings for additional shared resource hints not shown here, are described in Chapter 3 of Book II.)

Rx	PPR_{PRI}	Priority	Privileged
31	001	very low	no
1	010	low	no
6	011	medium low	no
2	100	medium	no
5	101	medium high	no/yes[1]
3	110	high	yes
7	111	very high	hypv

[1]This value is privileged unless the Problem State Priority Boost register allows the priority value 0b101 (See Section 4.3.8.)

Figure 17. Priority levels for *or Rx,Rx,Rx*

4.4.3 Transactional Memory Instructions

Privileged software that makes the Transactional Memory Facility available to applications takes on the responsibility of managing the facility's resources and the application's transaction state during interrupt handling, service calls, task switches, and its own use of TM. In addition to the existing instructions like ***rfid*** and problem state TM instructions that play a role in this management, ***treclaim*** and ***trechkpt.*** may be used, as described below. See Section 3.2.2 for additional information about managing the TM facility and associated state transitions.

Transaction Reclaim *X-form*

treclaim. RA

31	///	RA	///	942	1
0	6	11	16	21	31

```
CR0 ← 0 || MSR_TS || 0

if MSR_TS = 0b10 | MSR_TS = 0b01 then
      #Transactional or Suspended
   if RA = 0 then cause <- 0x00000001
   else          cause <- GPR(RA)_56:63 || 0x000001
   if TEXASR_FS = 0 then
     Discard transactional footprint
   TMRecordFailure(cause)
   checkpointed registers ←
                    contents from checkpoint area
   Discard all resources related to current transaction

MSR_TS <- 0b00                    #Non-transactional
```

The *treclaim.* instruction frees the transactional facility for use by a new transaction. It sets condition register field 0 to 0 || MSR_{TS} || 0. If the transactional facility is in the Transactional state or Suspended state, failure recording is performed as defined in Section 5.3.2 of Book II. If RA is 0, the failure cause is set to 0x00000001, otherwise it is set to $GPR(RA)_{56:63}$ || 0x000001. The checkpointed registers are restored from the checkpoint area, and all resources related to the current transaction are discarded, including the transactional footprint (if it wasn't already discarded for a pending failure).

The transaction state is set to Non-transactional.

If an attempt is made to execute ***treclaim.*** in Non-transactional state, a TM Bad Thing type Program interrupt will be generated.

This instruction is privileged.

Special Registers Altered:

CR0TEXASR TFIAR TS

Programming Note

The *treclaim.* instruction can be used by an interrupt handler to deallocate the current thread's transactional resources in preparation for subsequent use of the facility by a new transaction. (An abort is not appropriate for this use, because (a) the interrupt handler is in Suspended state and an abort in Suspended state leaves the thread in Suspended state, and (b) an abort in Suspended state does not restore the checkpointed registers.) After ***treclaim.*** is executed, the interrupt handler should save the contents of the checkpointed registers to storage. When the interrupted program is next dispatched it should be resumed by first restoring the contents of the checkpointed registers from storage and then using ***trechkpt.*** to copy the contents of the checkpointed registers to the checkpoint area. (This saving and restoring of the checkpointed register state is in addition to the normal saving and restoring of the entire current register state.) The result of this use of ***treclaim.*** and ***trechkpt.*** is to restore the pre-transactional register values into the checkpoint area and to cause the thread to transition from Suspended state to Non-transactional state and back again. Failure handling for the program will occur when the program next attempts to execute an instruction in the Transactional state, which will cause the failure handler to be invoked because TDOOMED will be 1. (This will be immediate if the program was in the Transactional state when the interrupt occurred, or will be after ***tresume.*** is executed if the program was in the Suspended state when the interrupt occurred.)

Transaction Recheckpoint ***X-form***

trechkpt.

31	///	///	///	1006	1
0	6	11	16	21	31

```
CR0 ← 0 || MSR_TS || 0

MSR_TS ← 0b01
TDOOMED ← 1
checkpoint area ←(checkpointed registers)
```

The ***trechkpt.*** instruction copies the contents of the checkpointed registers to the checkpoint area. It sets condition register field 0 to 0 || MSR_{TS} || 0. The current values of the checkpointed registers are loaded into the checkpoint area. TDOOMED is set to 0b1.

The transaction state is set to Suspended.

If an attempt is made to execute this instruction in Transactional or Suspended state or when TEXAS-R_{FS}=0, a TM Bad Thing type Program interrupt will be generated.

This instruction is privileged.

Special Registers Altered:

CR0 TS

4.4.4 Move To/From System Register Instructions

The *Move To Special Purpose Register* and *Move From Special Purpose Register* instructions are described in Book I, but only at the level available to an application programmer. For example, no mention is made there of registers that can be accessed only in privileged state. The descriptions of these instructions given below extend the descriptions given in Book I, but do not list Special Purpose Registers that are implementation-dependent. In the descriptions of these instructions given in below, the “defined” SPR numbers are the SPR numbers shown in the Figure 18 for the instruction and the implementation-specific SPR numbers that are implemented, and similarly for “defined” registers. All other SPR numbers are undefined for the instruction. (Implementation-specific SPR numbers that are not implemented are considered to be undefined.) When an SPR is defined for ***mtspr*** and undefined for ***mfspr***, or vice versa, a hyphen appears in the column for the instruction for which the SPR number is undefined.

SPR 158, identified in Figure 18 as GSR, is a special SPR in that it retains no state and exists only to identify a performance optimization opportunity. ***mtspr*** specifying the GSR (Group Start Register) is used to identify the start of a sequence of ***mtspr*** instructions that may be optimized to have their SPR changes synchronized once as a group, rather than independently. The sequence is ended by any instruction other than a ***mtspr*** and also by an implicit redirection of instruction fetching, including those caused by interrupts and transaction failure. This function may be useful when restoring a number of SPRs. If any of the ***mtspr*** instructions in the sequence requires explicit context synchronization, a context synchronizing instruction must follow the sequence. See Chapter 11 of Book III for more details. Write access to the GSR is privileged; read access is not provided.

SPR numbers that are not shown in Figure 18 and are in the ranges shown below are reserved for implementation-specific uses.

848 - 863
880 - 895
976 - 991
1008 - 1023

Implementation-specific registers must be privileged. SPR numbers for implementation-specific SPRs should be registered in advance with the Power ISA architects.

Figure 18. SPR encodings (Sheet 1 of 3)

decimal	SPR[1]	Register Name	Privileged		Length (bits)	Extended Mnemonics*	
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr		mtspr	mfspr
1	00000 00001	XER	no	no	64	mtxer Rx	mfxer Rx
3	00000 00011	DSCR	no	no	64	mtudscr	mfudscr
8	00000 01000	LR	no	no	64	mtlr Rx	mflr Rx
9	00000 01001	CTR	no	no	64	mtctr Rx	mfctr Rx
13	00000 01101	AMR	no[4]	no	64	mtuamr Rx	mfuamr Rx
17	00000 10001	DSCR	yes	yes	64	mtdscr Rx	mfdscr Rx
18	00000 10010	DSISR	yes	yes	32	mtdsisr Rx	mfdsisr Rx
19	00000 10011	DAR	yes	yes	64	mtdar Rx	mfdar Rx
22	00000 10110	DEC	yes	yes	64	mtdec Rx	mfdec Rx
26	00000 11010	SRR0	yes	yes	64	mtsrr0 Rx	mfsrr0 Rx
27	00000 11011	SRR1	yes	yes	64	mtsrr1 Rx	mfsrr1 Rx
28	00000 11100	CFAR	yes	yes	64	mtcfar Rx	mfcfar Rx
29	00000 11101	AMR	yes[4]	yes	64	mtamr Rx	mfamr Rx
48	00001 10000	PIDR	yes[13]	yes	32	mtpidr Rx	mfpidr Rx
61	00001 11101	IAMR	yes[7]	yes	64	mtiamr Rx	mfiamr Rx
128	00100 00000	TFHAR	no	no	64	mttfhar Rx	mftfhar Rx
129	00100 00001	TFIAR	no	no	64	mttfiar Rx	mftfiar Rx
130	00100 00010	TEXASR	no	no	64	mttexasr Rx	mftexasr Rx
131	00100 00011	TEXASRU	no	no	32	mttexasru Rx	mftexasru Rx
136	00100 01000	CTRL	-	no	32	-	mfctrl Rx
144	00100 10000	TIDR	yes	yes	64	mttidr Rx	mftidr Rx
152	00100 11000	CTRL	yes	-	32	mtctrl Rx	-
153	00100 11001	FSCR	yes	yes	64	mtfscr Rx	mffscr Rx
157	00100 11101	UAMOR	yes[5]	yes	64	mtuamor Rx	mfuamor Rx
158	00100 11110	GSR	yes	-	[9]	mtgsr Rx	-
159	00100 11111	PSPB	yes	yes	32	mtpspb Rx	mfpspb Rx
176	00101 10000	DPDES	hypv[2]	yes	64	mtdpdes Rx	mfdpdes Rx
180	00101 10100	DAWR0	hypv[2]	hypv[2]	64	mtdawr0 Rx	mfdawr0 Rx
186	00101 11010	RPR	hypv[2]	hypv[2]	64	mtrpr Rx	mfrpr Rx
187	00101 11011	CIABR	hypv[2]	hypv[2]	64	mtciabr Rx	mfciabr Rx
188	00101 11100	DAWRX0	hypv[2]	hypv[2]	32	mtdawrx0 Rx	mfdawrx0 Rx
190	00101 11110	HFSCR	hypv[2]	hypv[2]	64	mthfscr Rx	mfhfscr Rx
256	01000 00000	VRSAVE	no	no	32	mtvrsave Rx	mfvrsave Rx
259	01000 00011	SPRG3	-	no	64	-	mfusprg3
268	01000 01100	TB	-	no	64	-	mftb Rx[11]
269	01000 01101	TBU	-	no	32	-	mftbu Rx[11]
272-275	01000 100xx	SPRG[n] n=0-3	yes	yes	64	mtspgrn Rx	mfspgrn Rx
283	01000 11011	CIR	-	yes	32	-	mfcir Rx
284	01000 11100	TBL	hypv[2]	-	32	mttbl Rx	-
285	01000 11101	TBU	hypv[2]	-	32	mttbu Rx	-
286	01000 11110	TBU40	hypv	-	64	mttbu40 Rx	-
287	01000 11111	PVR	-	yes	32	-	mfpvr Rx
304	01001 10000	HSPRG0	hypv[2]	hypv[2]	64	mthsprg0 Rx	mfhsprg0 Rx
305	01001 10001	HSPRG1	hypv[2]	hypv[2]	64	mthsprg1 Rx	mfhsprg1 Rx
306	01001 10010	HDSISR	hypv[2]	hypv[2]	32	mthdisr Rx	mfhdisr Rx
307	01001 10011	HDAR	hypv[2]	hypv[2]	64	mthdar Rx	mfhdar Rx
308	01001 10100	SPURR	hypv[2]	yes	64	mtspurr Rx	mfspurr Rx
309	01001 10101	PURR	hypv[2]	yes	64	mtpurr Rx	mfpurr Rx
310	01001 10110	HDEC	hypv[2]	hypv[2]	64	mthdec Rx	mfhdec Rx
313	01001 11001	HRMOR	hypv[2]	hypv[2]	64	mthrmor Rx	mfhrmor Rx
314	01001 11010	HSRR0	hypv[2]	hypv[2]	64	mthsrr0 Rx	mfhsrr0 Rx
315	01001 11011	HSRR1	hypv[2]	hypv[2]	64	mthsrr1 Rx	mfhsrr1 Rx

Figure 18. SPR encodings (Sheet 2 of 3)

decimal	SPR[1]	Register Name	Privileged		Length (bits)	Extended Mnemonics*	
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr		mtspr	mfspr
318	01001 11110	LPCR	hypv[2]	hypv[2]	64	mtlpcr Rx	mflpcr Rx
319	01001 11111	LPIDR	hypv[2,14]	hypv[2]	32	mtlpidr Rx	mflpidr Rx
336	01010 10000	HMER	hypv[2,3]	hypv[2]	64	mthmer Rx	mfhmer Rx
337	01010 10001	HMEER	hypv[2]	hypv[2]	64	mthmeer Rx	mfhmeer Rx
338	01010 10010	PCR	hypv[2]	hypv[2]	64	mtpcr Rx	mfpcr Rx
339	01010 10011	HEIR	hypv[2]	hypv[2]	32	mtheir Rx	mfheir Rx
349	01010 11101	AMOR	hypv[2]	hypv[2]	64	mtamor Rx	mfamor Rx
446	01101 11110	TIR	-	yes	64	-	mftir Rx
464	01110 10000	PTCR	hypv[2]	hypv[2]	64	mtptcr Rx	mfptcr Rx
768	11000 00000	SIER	-	no[6]	64	-	mfusier Rx mfsier Rx
769	11000 00001	MMCR2	no[6]	no[6]	64	mtummcr2 Rx mtmmcr2 Rx	mfummcr2 Rx mfmmcr2 Rx
770	11000 00010	MMCRA	no[6]	no[6]	64	mtummcra Rx	mfummcra Rx mfmmcra Rx
771	11000 00011	PMC1	no[6]	no[6]	32	mtupmc1 Rx	mfupmc1 Rx mfpmc1 Rx
772	11000 00100	PMC2	no[6]	no[6]	32	mtupmc2 Rx	mfupmc2 Rx mfpmc2 Rx
773	11000 00101	PMC3	no[6]	no[6]	32	mtupmc3 Rx	mfupmc3 Rx mfpmc3 Rx
774	11000 00110	PMC4	no[6]	no[6]	32	mtupmc4 Rx	mfupmc4 Rx mfpmc4 Rx
775	11000 00111	PMC5	no[6]	no[6]	32	mtupmc5 Rx	mfupmc5 Rx mfpmc5 Rx
776	11000 01000	PMC6	no[6]	no[6]	32	mtupmc6 Rx	mfupmc6 Rx mfpmc6 Rx
779	11000 01011	MMCR0	no[6]	no[6]	64	mtummcr0 Rx	mfummcr0 Rx mfmmcr0 Rx
780	11000 01100	SIAR	-	no[6]	64	-	mfusiar Rx mfsiar Rx
781	11000 01101	SDAR	-	no[6]	64	-	mfusdar Rx mfsdar Rx
782	11000 01110	MMCR1	-	no[6]	64	-	mfummcr1 Rx mfmmcr1 Rx
784	11000 10000	SIER	yes	yes	64	mtsier Rx	[12]
785	11000 10001	MMCR2	yes	yes	64	[12]	[12]
786	11000 10010	MMCRA	yes	yes	64	mtmmcra Rx	[12]
787	11000 10011	PMC1	yes	yes	32	mtpmc1 Rx	[12]
788	11000 10100	PMC2	yes	yes	32	mtpmc2 Rx	[12]
789	11000 10101	PMC3	yes	yes	32	mtpmc3 Rx	[12]
790	11000 10110	PMC4	yes	yes	32	mtpmc4 Rx	[12]
791	11000 10111	PMC5	yes	yes	32	mtpmc5 Rx	[12]
792	11000 11000	PMC6	yes	yes	32	mtpmc6 Rx	[12]
795	11000 11011	MMCR0	yes	yes	64	mtmmcr0 Rx	[12]
796	11000 11100	SIAR	yes	yes	64	mtsiar Rx	[12]
797	11000 11101	SDAR	yes	yes	64	mtsdar Rx	[12]
798	11000 11110	MMCR1	yes	yes	64	mtmmcr1 Rx	[12]
800	11001 00000	BESCRS	no	no	64	mtbescrs Rx	mfbescrs Rx
801	11001 00001	BESCRSU	no	no	32	mtbescrsu Rx	mfbescrsu Rx
802	11001 00010	BESCRR	no	no	64	mtbescrr Rx	mfbescrr Rx
803	11001 00011	BESCRRU	no	no	32	mtbescrru Rx	mfbescrru Rx
804	11001 00100	EBBHR	no	no	64	mtebbhr Rx	mfebbhr Rx
805	11001 00101	EBBRR	no	no	64	mtebbrr Rx	mfebbrr Rx
806	11001 00110	BESCR	no	no	64	mtbescr Rx	mfbescr Rx
808	11001 01000	reserved[8]	no	no	na	-	-

Figure 18. SPR encodings (Sheet 3 of 3)

decimal	SPR[1]	Register Name	Privileged		Length (bits)	Extended Mnemonics*	
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr		mtspr	mfspr
809	11001 01001	reserved[8]	no	no	na	-	-
810	11001 01010	reserved[8]	no	no	na	-	-
811	11001 01011	reserved[8]	no	no	na	-	-
815	11001 01110	TAR	no	no	64	mttar Rx	mftar Rx
816	11001 10000	ASDR	hypv[2]	hypv[2]	64	mtasdr Rx	mfasdr Rx
823	11001 10111	PSSCR	yes	yes	64	mtpsscr Rx	mfpsscr Rx
848	11010 10000	IC	hypv[2]	yes	64	mtic Rx	mfic Rx
849	11010 10001	VTB	hypv[2]	yes	64	mtvtb Rx	mfvtb Rx
855	11010 10111	PSSCR	hypv[3]	hypv[3]	64	mthpsscr Rx	mfhpsscr
896	11100 00000	PPR	no	no	64	mtppr Rx	mfppr Rx
898	11100 00010	PPR32	no	no	32	mtppr32 Rx	mfppr32 Rx
1023	11111 11111	PIR	-	yes	32	-	mfpir Rx

- This register is not defined for this instruction.

1 Note that the order of the two 5-bit halves of the SPR number is reversed.

2 This register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2).

3 This register cannot be directly written. Instead, bits in the register corresponding to 0 bits in (RS) can be cleared using *mtspr SPR,RS*.

4 The value specified in register RS may be masked by the contents of the [U]AMOR before being placed into the AMR; see the ***mtspr*** instruction description.

5 The value specified in register RS may be ANDed with the contents of the AMOR before being placed into the UAMOR; see the ***mtspr*** instruction description.

6 $MMCR0_{PMCC}$ controls the availability of this SPR, and its contents depend on the privilege state in which it is accessed. See Section 9.4.4 for details.

7 The value specified in Register RS may be masked by the contents of the AMOR before being placed into the IAMR; see the ***mtspr*** instruction description.

8 Accesses to these SPRs are no-ops; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I.

9 The length of the GSR is undefined. An ***mtspr*** instruction specifying this SPR affects synchronization of subsequent ***mtspr*** instructions. See the introductory text in this section for more details

10 SPR numbers 777-778, 783, 793-794, and 799 are reserved for the Performance Monitor. All other SPR numbers that are not shown above and are not implementation-specific are reserved.

11 The ***mftb*** instruction is Phased-Out. Assemblers targeting Version 2.03 or later of the architecture should generate an ***mfspr*** instruction for the ***mftb*** and ***mftbu*** extended mnemonics; see the corresponding Assembler Note in the ***mftb*** instruction description (see Section 6.1 of Book II).

12 No extended mnemonic is provided because previous versions of the architecture defined the obvious extended mnemonic as resolving to the non-privileged SPR number, and because there is no software benefit in using the privileged SPR number, rather than the non-privileged SPR number, for this function.

13 ***mtspr*** specifying PIDR performs the implementation specific lookaside invalidation specified by slbia with IH=0b011 when using Radix Tree translation.

14 ***mtspr*** specifying LPIDR performs the implementation specifc lookaside invalidation specified by slbia with IH=0b110 when switching between partitions both of which use Radix Tree translation.

*This figure also defines extended mnemonics for the ***mtspr*** and ***mfspr*** instructions, including the Special Purpose Registers (SPRs) defined in Book I and for the *Move From Time Base* instruction defined in Book II.

The ***mtspr*** and ***mfspr*** instructions specify an SPR as a numeric operand; extended mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as an operand. Similar extended mnemonics are provided for the *Move From Time Base* instruction, which specifies the portion of the Time Base as a numeric operand.

Note: ***mftb*** serves as both a basic and an extended mnemonic. The Assembler will recognize an ***mftb*** mnemonic with two operands as the basic form, and an ***mftb*** mnemonic with one operand as the extended form. In the extended form the TBR operand is omitted and assumed to be 268 (the value that corresponds to TB)

Move To Special Purpose Register XFX-form

mtspr SPR,RS

31	RS	spr	467	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
switch (n)
  case(13): if MSRHV PR = 0b10 then
              SPR(13) ← (RS)
            else
              if MSRHV PR = 0b00 then
                SPR(13) ← ((RS) & AMOR) |
                          ((SPR(13)) & ¬AMOR)
              else
                SPR(13) ← ((RS) & UAMOR) |
                          ((SPR(13)) & ¬UAMOR)
  case(29,61):if MSRHV PR = 0b10 then
              SPR(n) ← (RS)
            else
               SPR(n) ← ((RS) & AMOR) |
                           ((SPR(n)) & ¬AMOR)
  case(48):  SPR(n) ← (RS)
             if PATEHR=1 for the partition then
               All implementation-specific
               lookaside information that was
               created when address translation
               was enabled and for which effPID≠0
               is invalidated.
  case (157): if MSRHV PR = 0b10 then
                SPR(157) ← (RS)
              else
                SPR(157) ← (RS) & AMOR
  case (158):start mtspr sequence optimization
  case(319):  SPR(n) ← (RS)
              if PATEHR=1 for both the originating
                   and destination partitions than
               All implementation-specific
               lookaside information that was
               created when address translation
               was enabled and for which effPID≠0
               or effLPID≠0 is invalidated.
               All implementation-specific
               lookaside information that was
               created when address translation
               was disabled and MSRHV PR was equal
               to 0b00 is invalidated.
  case (336):SPR(336) ← (SPR(336)) & (RS)
  case (808, 809, 810, 811):
  default: if length(SPR(n)) = 64 then
             SPR(n) ← (RS)
           else
             SPR(n) ← (RS)32:63
```

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 18. If the SPR field contains the value 158, the instruction indicates the start of a sequence of ***mtspr*** instructions that may be synchronized as a group. See the introductory material in this section for more information. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, "Reserved Fields, Reserved Values, and Reserved SPRs" in Book I. Otherwise, the contents of register RS are placed into the designated Special Purpose Register, except as described in the next four paragraphs. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RS are placed into the SPR.

When the designated SPR is the Authority Mask Register (AMR), (using SPR 13 or SPR 29), or the designated SPR is the Instruction Authority Mask Register (IAMR), and $MSR_{HV\ PR}$=0b00, the contents of bit positions of register RS corresponding to 1 bits in the Authority Mask Override Register (AMOR) are placed into the corresponding bits of the AMR or IAMR, respectively; the other AMR or IAMR bits are not modified.

When the designated SPR is the AMR, using SPR 13, and MSR_{PR}=1, the contents of bit positions of register RS corresponding to 1 bits in the User Authority Mask Override Register (UAMOR) are placed into the corresponding bits of the AMR; the other AMR bits are not modified.

When the designated SPR is the UAMOR and $MSR_{HV\ PR}$=0b00, the contents of register RS are ANDed with the contents of the AMOR and the result is placed into the UAMOR.

When the designated SPR is the PIDR and the partition uses Radix Tree translation, the implementation specific lookaside invalidation specified by ***slbia*** with IH=0b011 is performed along with the SPR update. When the designated SPR is the LPIDR and both the originating and destination partitions use Radix Tree translation, the implementation specific lookaside invalidation specified by ***slbia*** with IH=0b110 is performed along with the SPR update.

When the designated SPR is the Hypervisor Maintenance Exception Register (HMER), the contents of register RS are ANDed with the contents of the HMER and the result is placed into the HMER.

For this instruction, SPRs TBL and TBU are treated as separate 32-bit registers; setting one leaves the other unaltered.

spr_0=1 if and only if writing the register is privileged. Execution of this instruction specifying an SPR number with spr_0=1 causes a Privileged Instruction type Program interrupt when MSR_{PR}=1 and, if the SPR is a hypervisor resource (see Figure 18) when $MSR_{HV\ PR}$=0b00, causes a Privileged Instruction type Program interrupt if $LPCR_{EVIRT}$=0 and a Hypervisor Emulation Assistance interrupt if $LPCR_{EVIRT}$=1.

Execution of this instruction specifying an SPR number that is undefined for the implementation causes one of the following.

- if spr_0=0:

- if MSR_{PR}=1: Hypervisor Emulation Assistance interrupt
- if MSR_{PR}=0: Hypervisor Emulation Assistance interrupt for SPR 0,4,5, and 6, and no operation (i.e., the instruction is treated as a no-op) when $LPCR_{EVIRT}$=0 and Hypervisor Emulation Assistance interrupt when $LPCR_{EVIRT}$=1 for all other SPRs

■ if spr_0=1:
- if MSR_{PR}=1: Privileged Instruction type Program interrupt
- if MSR_{PR}=0: no operation (i.e., the instruction is treated as a no-op) when $LPCR_{EVIRT}$=0 and Hypervisor Emulation Assistance interrupt when $LPCR_{EVIRT}$=1

If an attempt is made to execute ***mtspr*** specifying a Transactional Memory SPR in other than Non-transactional state, with the exception of TFHAR in suspended state, a TM Bad Thing type Program interrupt is generated.

Special Registers Altered:

See Figure 18

Programming Note

For a discussion of software synchronization requirements when altering certain Special Purpose Registers, see Chapter 11. “Synchronization Requirements for Context Alterations” on page 1133.

Programming Note

Requiring that an attempt to execute an ***mtspr*** or ***mfspr*** instruction with SPR=0 or an attempt to execute an ***mfspr*** instruction with SPR=4, 5, or 6 cause a Hypervisor Emulation Assistance interrupt permits efficient emulation of ***mt/fspr*** specifying the corresponding SPRs as defined in the POWER Architecture.

Requiring that an attempt to execute an ***mtspr*** instruction with SPR=4, 5, or 6 cause a Hypervisor Emulation Assistance interrupt, even in privileged state, makes the behavior be the same for both instructions for all four SPR numbers, thereby simplifying the architecture. (SPRs 4, 5, and 6 were not defined for ***mtspr*** in the POWER Architecture. The corresponding SPRs were privileged for writing, and ***mtspr*** to those SPRs used the corresponding privileged SPR number.)

Move From Special Purpose Register XFX-form

mfspr RT,SPR

31	RT	spr	339	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
switch (n)
  case(129):
    if (MSRHV PR = 0b10) | (TFIARPR=MSRPR=1) |
     ((MSRHV PR = 0b00) & (TFIARHV PR ≠ 0b10)) then
       RT ← SPR(n)
    else
       RT ← 0
  case(808, 809, 810, 811):
  default:
    if length(SPR(n)) = 64 then
      RT ← SPR(n)
    else
      RT ← 320 || SPR(n)
```

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 18. If the designated Special Purpose Register is the TFIAR and TFIAR indicates the failure was recorded in a state more privileged than the current state, register RT is set to zero. If the SPR field contains a value from 808 through 811, the instruction specifies a reserved SPR, and is treated as a no-op; see Section 1.3.3, “Reserved Fields, Reserved Values, and Reserved SPRs” in Book I. Otherwise, the contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

Programming Note

Note that when a problem state transaction’s failure is recorded in hypervisor state and there is a subsequent need for a context switch in privileged, non-hypervisor state, an attempt to save TFIAR will result in zeros being saved. This is harmless because if the original application ever tries to read the TFIAR, it would read zeros anyway, since the failure took place in hypervisor state.

spr_0=1 if and only if reading the register is privileged. Execution of this instruction specifying an SPR number with spr_0=1 causes a Privileged Instruction type Program interrupt when MSR_{PR}=1 and, if the SPR is a hypervisor resource (see Figure 18) when $MSR_{HV\ PR}$=0b00, causes a Privileged Instruction type Program interrupt when $LPCR_{EVIRT}$=0 and a Hypervisor Emulation Assistance interrupt when $LPCR_{EVIRT}$=1.

Execution of this instruction specifying an SPR number that is not defined for the implementation causes one of the following.

- if spr_0=0:
  - if MSR_{PR}=1: Hypervisor Emulation Assistance interrupt
  - if MSR_{PR}=0: Hypervisor Emulation Assistance interrupt for SPRs 0, 4, 5, and 6, and no operation (i.e., the instruction is treated as a no-op) when $LPCR_{EVIRT}$=0 and Hypervisor Emulation Assistance interrupt when $LPCR_{EVIRT}$=1 for all other SPRs
- if spr_0=1:
  - if MSR_{PR}=1: Privileged Instruction type Program interrupt
  - if MSR_{PR}=0: no operation (i.e., the instruction is treated as a no-op) when $LPCR_{EVIRT}$=0 and Hypervisor Emulation Assistance interrupt when $LPCR_{EVIRT}$=1

Special Registers Altered:

None

Note

See the Notes that appear with ***mtspr***.

Move To Machine State Register X-form

mtmsr RS,L

31	RS	///	L	///	146	/
0	6	11	15	16	21	31

```
if L = 0 then
   MSR48 ← (RS)48 | (RS)49
   MSR58 ← (RS)58 | (RS)49
   MSR59 ← (RS)59 | (RS)49
   MSR32:47 49:50 52:57 60:62 ←(RS)32:47 49:50 52:57 60:62
else
   MSR48 62 ← (RS)48 62
```

The MSR is set based on the contents of register RS and of the L field.

L=0:

The result of ORing bits 48 and 49 of register RS is placed into MSR_{48}. The result of ORing bits 58 and 49 of register RS is placed into MSR_{58}. The result of ORing bits 59 and 49 of register RS is placed into MSR_{59}. Bits 32:47, 49:50, 52:57, and 60:62 of register RS are placed into the corresponding bits of the MSR.

L=1:

Bits 48 and 62 of register RS are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

This instruction is privileged.

If L=0 this instruction is context synchronizing. If L=1 this instruction is execution synchronizing; in addition, the alterations of the EE and RI bits take effect as soon as the instruction completes.

Special Registers Altered:

MSR

Except in the ***mtmsr*** instruction description in this section, references to "***mtmsr***" in this document imply either L value unless otherwise stated or obvious from context (e.g., a reference to an ***mtmsr*** instruction that modifies an MSR bit other than the EE or RI bit implies L=0).

Programming Note

If MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

This instruction does not alter MSR_{ME} or MSR_{LE}. (This instruction does not alter MSR_{HV} because it does not alter any of the high-order 32 bits of the MSR.)

If the only MSR bits to be altered are $MSR_{EE\ RI}$, to obtain the best performance L=1 should be used.

Programming Note

If MSR_{EE}=0 and an External, Decrementer, or Performance Monitor exception is pending, executing an ***mtmsrd*** instruction that sets MSR_{EE} to 1 will cause the interrupt to occur before the next instruction is executed, if no higher priority exception exists (see Section 6.9, "Interrupt Priorities" on page 1092). Similarly, if a Hypervisor Decrementer interrupt is pending, execution of the instruction by the hypervisor causes a Hypervisor Decrementer interrupt to occur if HDICE=1.

For a discussion of software synchronization requirements when altering certain MSR bits, see Chapter 11.

Programming Note

mtmsr serves as both a basic and an extended mnemonic. The Assembler will recognize an ***mtmsr*** mnemonic with two operands as the basic form, and an ***mtmsr*** mnemonic with one operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Programming Note

There is no need for an analogous version of the ***mfmsr*** instruction, because the existing instruction copies the entire contents of the MSR to the selected GPR.

Move To Machine State Register Doubleword *X-form*

mtmsrd RS,L

31	RS	///	L	///	178	/
0	6	11	15	16	21	31

```
if L = 0 then

   if (MSR29:31 ¬= 0b010 | RS29:31 ¬= 0b000) then
      MSR29:31 ← RS29:31
   MSR48 ← (RS)48 | (RS)49
   MSR58 ← (RS)58 | (RS)49
   MSR59 ← (RS)59 | (RS)49
   MSR0:2 4:28 32:47 49:50 52:57 60:62
            ← (RS)0:2 4 6:28 32:47 49:50 52:57 60:62
else
   MSR48 62 ← (RS)48 62
```

The MSR is set based on the contents of register RS and of the L field.

L=0:

If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of RS are not equal to 0b000, then the value of bits 29 through 31 of RS is placed into bits 29 through 31 of the MSR.The result of ORing bits 48 and 49 of register RS is placed into MSR_{48}. The result of ORing bits 58 and 49 of register RS is placed into MSR_{58}. The result of ORing bits 59 and 49 of register RS is placed into MSR_{59}. Bits 0:2, 4:28, 32:47, 49:50, 52:57, and 60:62 of register RS are placed into the corresponding bits of the MSR.

L=1:

Bits 48 and 62 of register RS are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

If the instruction attempts to cause an illegal transaction state transition or, when TM is made unavailable in problem state by the PCR, attempts to cause a transition to problem state and also a transaction state transition that Table 3 on page 947 shows as legal and as resulting in the thread being in Transactional or Suspended state, a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the ***mtmsrd*** instruction.

This instruction is privileged.

If L=0 this instruction is context synchronizing. If L=1 this instruction is execution synchronizing; in addition, the alterations of the EE and RI bits take effect as soon as the instruction completes.

Special Registers Altered:

MSR

Except in the ***mtmsrd*** instruction description in this section, references to "***mtmsrd***" in this document imply either L value unless otherwise stated or obvious from context (e.g., a reference to an ***mtmsrd*** instruction that modifies an MSR bit other than the EE or RI bit implies L=0).

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{EE}, MSR_{IR}, and MSR_{DR} to 1.

This instruction does not alter MSR_{LE}, MSR_{ME} or MSR_{HV}.

If the only MSR bits to be altered are $MSR_{EE\ RI}$, to obtain the best performance L=1 should be used.

Programming Note

If MSR_{EE}=0 and an External, Decrementer, or Performance Monitor exception is pending, executing an ***mtmsrd*** instruction that sets MSR_{EE} to 1 will cause the interrupt to occur before the next instruction is executed, if no higher priority exception exists (see Section 6.9, "Interrupt Priorities" on page 1092). Similarly, if a Hypervisor Decrementer interrupt is pending, execution of the instruction by the hypervisor causes a Hypervisor Decrementer interrupt to occur if HDICE=1.

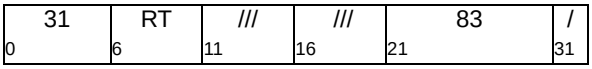
For a discussion of software synchronization requirements when altering certain MSR bits, see Chapter 11.

Programming Note

mtmsrd serves as both a basic and an extended mnemonic. The Assembler will recognize an ***mtmsrd*** mnemonic with two operands as the basic form, and an ***mtmsrd*** mnemonic with one operand as the extended form. In the extended form the L operand is omitted and assumed to be 0.

Move From Machine State Register
X-form

mfmsr RT



RT ← MSR

The contents of the MSR are placed into register RT.

This instruction is privileged.

Special Registers Altered:

None

Chapter 5. Storage Control

5.1 Overview

A program references storage using the effective address computed by the hardware when it executes a *Load*, *Store*, *Branch*, or *Cache Management* instruction, or when it fetches the next sequential instruction. The effective address is translated to a real address according to procedures described in Section 5.7.3, in Section 5.7.7 and in the following sections. The real address is what is presented to the storage subsystem.

For a complete discussion of storage addressing and effective address calculation, see Section 1.11 of Book I.

5.2 Storage Exceptions

A *storage exception* results when the sequential execution model requires that a storage access be performed but the access is not permitted (e.g., is not permitted by the storage protection mechanism), the access cannot be performed because the effective address cannot be translated to a real address, or the access matches some tracking mechanism criteria (e.g., Data Address Watchpoint).

In certain cases a storage exception may result in the "restart" of (re-execution of at least part of) a Load or Store instruction. See Section 2.2 of Book II, and Section 6.6 in this Book.

5.3 Instruction Fetch

Instructions are fetched under control of MSR_{IR}.

MSR_{IR}=0

The effective address of the instruction is interpreted as described in Section 5.7.3.

MSR_{IR}=1

The effective address of the instruction is translated by the Address Translation mechanism described beginning in Section 5.7.7.

5.3.1 Implicit Branch

Explicitly altering certain MSR bits (using ***mtmsr***[***d***]), or explicitly altering SLB entries, Page Table Entries, or certain System Registers (including the HRMOR, and possibly other implementation-dependent registers), may have the side effect of changing the addresses, effective or real, from which the current instruction stream is being fetched. This side effect is called an *implicit branch*. For example, an ***mtmsrd*** instruction that changes the value of MSR_{SF} may change the effective addresses from which the current instruction stream is being fetched. The MSR bits and System Registers (excluding implementation-dependent registers) for which alteration can cause an implicit branch are indicated as such in Chapter 11. "Synchronization Requirements for Context Alterations" on page 1133. Implicit branches are not supported by the Power ISA. If an implicit branch occurs, the results are boundedly undefined.

5.3.2 Address Wrapping Combined with Changing MSR Bit SF

If the current instruction is at effective address $2^{32} - 4$ and is an ***mtmsrd*** instruction that changes the contents of MSR_{SF}, the effective address of the next sequential instruction is undefined.

Programming Note

If the thread is in 32-bit mode, the current instruction is at effective address $2^{32} - 4$, and an interrupt occurs that is defined to set SRR0 or HSRR0 (or LR, for the System Call Vectored interrupt) to the effective address of the next sequential instruction, the contents of SRR0 or HSRR0 (or LR), as appropriate to the interrupt, are undefined.

5.4 Data Access

Data accesses are controlled by MSR_{DR}.

MSR_{DR}=0

The effective address of the data is interpreted as described in Section 5.7.3.

MSR_{DR}=1

The effective address of the data is translated by the Address Translation mechanism described in Section 5.7.7.

5.5 Performing Operations Out-of-Order

An operation is said to be performed "in-order" if, at the time that it is performed, it is known to be required by the sequential execution model. An operation is said to be performed "out-of-order" if, at the time that it is performed, it is not known to be required by the sequential execution model.

Operations are performed out-of-order on the expectation that the results will be needed by an instruction that will be required by the sequential execution model. Whether the results are really needed is contingent on everything that might divert the control flow away from the instruction, such as *Branch*, *Trap*, *System Call*, and *Return From Interrupt* instructions, and interrupts, and on everything that might change the context in which the instruction is executed.

Typically, operations are performed out-of-order when resources are available that would otherwise be idle, so the operation incurs little or no cost. If subsequent events such as branches or interrupts indicate that the operation would not have been performed in the sequential execution model, any results of the operation are abandoned (except as described below).

In the remainder of this section, including its subsections, "*Load* instruction" includes the *Cache Management* and other instructions that are stated in the instruction descriptions to be "treated as a *Load*", and similarly for "*Store* instruction".

A data access that is performed out-of-order may correspond to an arbitrary *Load* or *Store* instruction (e.g., a *Load* or *Store* instruction that is not in the instruction stream being executed). Similarly, an instruction fetch that is performed out-of-order may be for an arbitrary instruction (e.g., the aligned word at an arbitrary location in instruction storage).

Most operations can be performed out-of-order, as long as the machine appears to follow the sequential execution model. Certain out-of-order operations are restricted, as follows.

- Stores
  Stores are not performed out-of-order (even if the Store instructions that caused them were executed out-of-order).
- Accessing Guarded Storage
  The restrictions for this case are given in Section 5.8.1.1.

The only permitted side effects of performing an operation out-of-order are the following.

- A Machine Check or Checkstop that could be caused by in-order execution may occur out-of-order.
- Reference and Change bits may be set as described in Section 5.7.12.
- Non-Guarded storage locations that could be fetched into a cache by in-order fetching or execution of an arbitrary instruction may be fetched out-of-order into that cache.
- SPRs that are specified by ***mtspr*** instructions within a sequence initiated by ***mtgsr*** (see Chapter 11 of Book III) may be modified in arbitrary relative order, except that if an interrupt occurs within the sequence, all ***mtspr*** instructions in the sequence prior to the point of interruption appear to have been executed before the first instruction of the interrupt handler is executed.

5.6 Invalid Real Address

A storage access (including an access that is performed out-of-order; see Section 5.5) may cause a Machine Check if the accessed storage location contains an uncorrectable error or does not exist.

In the case that the accessed storage location does not exist, the Checkstop state may be entered. See Section 6.5.2 on page 1067.

Programming Note

In configurations supporting multiple partitions, hypervisor software must ensure that a storage access by a program in one partition will not cause a Checkstop or other system-wide event that could affect the integrity of other partitions (see Chapter 2). For example, such an event could occur if a real address placed in a Page Table Entry does not exist.

5.7 Storage Addressing

Storage Control Overview

- Host real address space size is 2^m bytes, m≤60; see Note 1.
- Guest real address space size is 2^m bytes, m≤60; see Notes 1 and 2.
- Real page size is 2^{12} bytes (4 KB).
- Effective address space size is 2^{64} bytes.
- For HPT translation, an effective address is translated to a virtual address via a segment descriptor that was either bolted into the Segment Lookaside Buffer (SLB) by software or found and installed into the SLB via a hardware walk of the Segment Table. After that, the virtual address is translated to a host real address via a hardware walk of the Page Table.
  - Virtual address space size is 2^n bytes, 65≤n≤78; see Note 3.
  - Segment size is 2^s bytes, s=28 or 40.
  - 2^{n-40} ≤ number of virtual segments ≤ 2^{n-28}; see Note 3.
  - Virtual page size is 2^p bytes, where 12≤p, and 2^p is no larger than either the size of the biggest segment or the real address space; a size of 4 KB, 64 KB, and an implementation-dependent number of other sizes are supported; see Note 4. The Page Table specifies the virtual page size. The SLB specifies the base virtual page size, which is the smallest virtual page size that the segment can contain. The base virtual page size is 2^b bytes.
  - Segments contain pages of a single size, a mixture of 4 KB and 64 KB pages, or a mixture of page sizes that include implementation-dependent page sizes.
  -
- For Radix Tree translation, an effective address is translated to a (guest or host) real address via a hardware walk of the Page Table..
  - Virtual page size is 2^p bytes, where 12≤p, and 2^p is no larger than the size of the real address space; a size of 4 KB, 64 KB, 2MB, and an implementation-dependent number of other sizes are supported; see Note 4. The virtual page size is determined by the location of the Page Table Entry in the Radix Tree.

Notes:

1. The value of m is implementation-dependent (subject to the maximum given above). When used to address storage or to represent a guest real address, the high-order 60-m bits of the "60-bit" real address must be zeros.
2. The hypervisor may assign a guest real address space size for each partition that uses Radix Tree translation. Accesses to guest real storage outside this range but still mappable by the second level Radix Tree will cause an HISI or HDSI. Accesses to storage outside the mappable range will have boundedly undefined results.
3. The value of n is implementation-dependent (subject to the range given above). In references to 78-bit virtual addresses elsewhere in this Book, the high-order 78-n bits of the "78-bit" virtual address are assumed to be zeros.
4. The supported values of p for the larger virtual page sizes are implementation-dependent (subject to the limitations given above).

Programming Note

Note that without some of the reserved bits in the Radix PTE, the RPN field cannot address the full 60-bit real address space. Similarly without some of the reserved bits in the HPT PTE, the ARPN field cannot address the full 60-bit real address space.

Note that without some of the reserved bits in the HPT PTE, the AVA field cannot resolve the full 78-bit virtual address.

5.7.1 32-Bit Mode

The computation of the 64-bit effective address is independent of whether the thread is in 32-bit mode or 64-bit mode. In 32-bit mode (MSR_{SF}=0), the high-order 32 bits of the 64-bit effective address are treated as zeros for the purpose of addressing storage. This applies to both data accesses and instruction fetches. It applies independent of whether address translation is enabled or disabled. This truncation of the effective address is the only respect in which storage accesses in 32-bit mode differ from those in 64-bit mode.

Programming Note

Treating the high-order 32 bits of the effective address as zeros effectively truncates the 64-bit effective address to a 32-bit effective address such as would have been generated on a 32-bit implementation of the Power ISA. Thus, for example, the ESID in 32-bit mode is the high-order four bits of this truncated effective address; the ESID thus lies in the range 0-15. When address translation is enabled, these four bits would select a Segment Register on a 32-bit implementation of the Power ISA. The SLB entries that translate these 16 ESIDs can be used to emulate these Segment Registers.

5.7.2 Virtualized Partition Memory (VPM) Mode

VPM mode enables the hypervisor to reassign all or part of a partition's memory transparently so that the reassignment is not visible to the partition. When this is done, the partition's memory is said to be "virtualized." This mode is only available within Paravirtualized HPT translation mode. Radix Tree translation mode provides equivalent function by providing two levels of translation with separate Page Tables for the operating system and the hypervisor. (See Section 5.7.7 for a more complete overview of the translation modes.) The VPM field in the LPCR enables VPM mode when address translation is enabled. VPM is always enabled when address translation is disabled.

If the thread is not in hypervisor state, and either address translation is enabled and VPM=1, or address translation is disabled, conditions that would have caused a Data Storage or an Instruction Storage interrupt if the affected memory were not virtualized instead cause a Hypervisor Data Storage or a Hypervisor Instruction Storage interrupt respectively. Because the Hypervisor Data Storage and Hypervisor Instruction Storage interrupts always put the thread in hypervisor state, they permit the hypervisor to handle the condition if appropriate (e.g., to restore the contents of a page that was reassigned), and to reflect it to the operating system's Data Storage or Instruction Storage interrupt handler otherwise.

When address translation is enabled, VPM mode has no effect on address translation. When address translation is disabled, addressing is controlled as specified in Section 5.7.3.

5.7.3 Hypervisor Real And Virtual Real Addressing Modes

When a storage access is an instruction fetch performed when instruction address translation is disabled, or is a data access performed when data address translation is disabled, it is said to be performed in "hypervisor real addressing mode" if the thread is in hypervisor state. If the thread is not in hypervisor state, the access is said to be performed in "virtual real addressing mode." Storage accesses in hypervisor real and virtual real addressing modes are performed in a manner that depends on the contents of MSR_{HV}, $PATE_{HR}$, $PATE_{PS}$, HRMOR (see Chapter 2), bit 0 of the effective address (EA_0), and the state of the Real Mode Storage Control Facility as described below. Bits 1:3 of the effective address are ignored.

MSR_{HV}=1

- If EA_0=0, the Hypervisor Offset Real Mode Address mechanism, described in Section 5.7.3.1, controls the access.
- If EA_0=1, bits 4:63 of the effective address are used as the real address for the access.

MSR_{HV}=0

- If $PATE_{HR}$=0, the Virtual Real Mode Addressing mechanism, described in Section 5.7.3.3, controls the access.
- If $PATE_{HR}$=1, partition-scoped translation is performed on the effective address. (See Section 5.7.11.3, "Obtaining Host Real Address, Radix on Radix".)

5.7.3.1 Hypervisor Offset Real Mode Address

If MSR_{HV} = 1 and EA_0 = 0, the access is controlled by the contents of the Hypervisor Real Mode Offset Register, as follows.

Hypervisor Real Mode Offset Register (HRMOR)

Bits 4:63 of the effective address for the access are ORed with the 60-bit offset represented by the contents of the HRMOR, and the 60-bit result is used as the real address for the access. The supported offset values are all values of the form $i \times 2^r$, where $0 \le i < 2^j$, and j and r are implementation-dependent values having the properties that $12 \le r \le 26$ (i.e., the minimum offset granularity is 4 KB and the maximum offset granularity is 64 MB) and j+r = m, where the real address size supported by the implementation is m bits.

Programming Note

$EA_{4:63-r}$ should equal $^{60-r}0$. If this condition is satisfied, ORing the effective address with the offset produces a result that is equivalent to adding the effective address and the offset.

If m<60, $EA_{4:63-m}$ and $HRMOR_{4:63-m}$ must be zeros.

5.7.3.2 Storage Control Attributes for Accesses in Hypervisor Real Addressing Mode

Storage accesses in hypervisor real addressing mode are performed as though all of storage had the following storage control attributes, except as modified by the Hypervisor Real Mode Storage Control facility (see Section 5.7.3.2.1). (The storage control attributes are defined in Book II.)

- not Write Through Required
- not Caching Inhibited, for instruction fetches
- not Caching Inhibited, for data accesses except those caused by the *Load/Store Caching Inhibited* instructions; Caching Inhibited, for data accesses caused by the *Load/Store Caching Inhibited* instructions
- Memory Coherence Required, for data accesses

- Guarded
- not SAO

Additionally, storage accesses in hypervisor real addressing mode are performed as though all storage was not No-execute.

Programming Note

Because storage accesses in hypervisor real addressing mode do not use the SLB or the Page Table, accesses in this mode bypass all checking and recording of information contained therein (e.g., storage protection checks that use information contained therein are not performed, and reference and change information is not recorded).

5.7.3.2.1 Hypervisor Real Mode Storage Control

The Hypervisor Real Mode Storage Control facility provides a means of specifying portions of real storage that are treated as non-Guarded in hypervisor real addressing mode ($MSR_{HV\ PR}$=0b10, and MSR_{IR}=0 or MSR_{DR}=0, as appropriate for the type of access). The remaining portions are treated as Guarded in hypervisor real addressing mode. The means is a hypervisor resource (see Chapter 2), and may also be system-specific.

The facility divides real storage into history blocks, in implementation-specific sizes. The history for instruction fetches is tracked separately from that for data accesses. If there is no instruction fetch history for a block and it is the target of an instruction fetch, the access is performed as though the block is Guarded, but the block is treated as non-Guarded for subsequent instruction fetches on a best effort basis, limited by the amount of history that the facility can maintain. If there is no data access history for a block and it is accessed using a *Load/Store Caching Inhibited* instruction, the access is performed as though the block is Guarded, and the block is treated as Guarded for subsequent accesses on a best effort basis, limited by the amount of history that the facility can maintain. If there is no data access history for a block and it is accessed using any other *Load* or *Store* instruction, the access is performed as though the block is Guarded, but the block is treated as non-Guarded for subsequent accesses on a best effort basis, limited by the amount of history that the facility can maintain.

The storage location specified by a *Load/Store Caching Inhibited* instruction must not be in storage that is specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded. The storage location specified by any other *Load* or *Store* instruction must not be in storage that is specified by the Hypervisor Real Mode Storage Control facility to be treated as Guarded. ("specified by the Hypervisor Real Mode Storage Control facility" means "specified in a history block".) The history can be erased using an slbia instruction; see Section 5.9.3.2.

Programming Note

There are two cautions about mixing different types of accesses (i.e.*Load/Store Caching Inhibited* instructions vs. any other *Load* or *Store* instruction vs. instruction fetches). The first, as indicated above, is to avoid confusing the history mechanism, and the granularity for concern is a history block. For this caution, instruction fetches are irrelevant because they have their own history mechanism and are always intended to be non-guarded.

The second caution is to avoid storage paradoxes that result from a Caching Inhibited access to a location that is held in a cache. The nature of this caution and its solution are described in Section 5.8.2.2, “Altering the Storage Control Bits”. The minimum granularity for concern is the history block, but may be larger, depending on extant translations to the storage in question. Since the consistency of instruction storage is managed by software and hypervisor real mode instruction fetches are always not Caching Inhibited, instruction fetches are also irrelevant to this caution.

The facility does not apply to implicit accesses to the Page Table performed during address translation or in recording reference and change information. These accesses are performed as described in Section 5.7.3.4.

Programming Note

The preceding capability can be used to improve the performance of hypervisor software that runs in hypervisor real addressing mode, by causing accesses to instructions and data that occupy well-behaved storage to be treated as non-Guarded.

5.7.3.3 Virtual Real Mode Addressing Mechanism

If MSR_{HV}=0, the partition is using Paravirtualized HPT translation ($PATE_{HR}$=0), and MSR_{DR}=0 or MSR_{IR}=0 as appropriate for the type of access, the access is said to be made in virtual real addressing mode and is controlled by the mechanism specified below. The set of storage locations accessible by code is referred to as the Virtualized Real Mode Area (VRMA).

In virtual real addressing mode, address translation, storage protection, and reference and change recording are handled as follows.

- Address translation and storage protection are handled as if address translation were enabled, except that translation of effective addresses to virtual addresses use the SLBE values in Figure 19 instead of the entry in the SLB corresponding to the ESID. In this translation, bits 0:23 of the effective address are ignored (i.e., treated as if they

were 0s), bits 24:63-m may be ignored if m < 40, and the Virtual Page Class Key Protection mechanism does not apply.

Programming Note

The Virtual Page Class Key Protection mechanism does not apply because the authority mask that an OS has set for application programs executing with address translation enabled may not be the same as the authority mask required by the OS when address translation is disabled, such as when first entering an interrupt handler.

- Reference and change recording are handled as if address translation were enabled.

Field	Value
ESID	$^{36}0$
V	1
B	0b01 - 1 TB
VSID	0b00 || 0x0_01FF_FFFF
K_s	0
K_p	undefined
N	0
L	$PATE_{PS[0]}$
C	0
LP	$PATE_{PS[1:2]}$

Figure 19. SLBE for VRMA

Programming Note

The C bit in Figure 19 is set to 0 because the implementation-specific lookaside information associated with the VRMA is expected to be long-lived. See the Programming Note about Class in Section 5.7.8.1.

Programming Note

The 1 TB VSID 0x0_01FF_FFFF should not be used by the operating system for purposes other than mapping the VRMA when address translation is enabled.

Programming Note

Software should specify PTE_B = 0b01 for all Page Table Entries that map the VRMA in order to be consistent with the values in Figure 19.

5.7.3.4 Storage Control Attributes for Implicit Storage Accesses

Implicit accesses to the Partition Table and to a partition-scoped Page Table during address translation and in recording reference and change information are performed as though the storage occupied by the tables had the following storage control attributes.

- not Write Through Required
- not Caching Inhibited
- Memory Coherence Required
- not Guarded
- not SAO

Implicit accesses to a Process Table, Segment table, or process-scoped Page Table during address translation and in recording reference and change information are performed using the storage control attributes in the partition-scoped Page Table Entry that maps the other In-Memory Table Entry or the process-scoped Page Table Entry that is being accessed. The storage control attributes must be those described above.

5.7.4 Definitions

process-scoped: Refers to translation performed using tables pointed to by Process Table Entries: guest Radix Tree translation, host Radix Tree translation for quadrants 0 and 3 when MSR_{HV}=1 , or Segment translation.

partition-scoped: Refers to translation performed using table(s) found using the first doubleword of Partition Table Entries, either host Radix Tree translation or HPT translation.

fully-qualified address: Refers to the address to be translated, when qualified by the effective LPID and effective PID.

guest real address: Refers to the input to the partition-scoped translation process when using nested Radix Tree translation.

virtual address: Refers to the output of Segment translation and input to HPT translation.

host real address: Refers to the output of the partition-scoped translation process in nested Radix Tree translation or the output of the process-scoped translation in nested Radix Tree translation for quadrants 0 and 3 when MSR_{HV}=1 . The simpler “real address” may be used interchangeably.

Page Directory: A table within the Radix Tree translation structure that contains elements (“Page Directory Entries”) that point to other tables, instead of containing just Page Table Entries. The Page Directory that is at the root of the Radix Tree is called the “Root Page Directory.”

effLPID, effPID: This is shorthand for effective LPID and effective PID. In certain circumstances, the value

used for the LPID and/or the PID is specified to be zero instead of the actual register contents. “Effective” or “eff” is used to indicate the possibility of such a substitution. This value substitution happens only in Radix Tree translation, and is based on the value of $EA_{0:1}$ (see Section 5.7.5.1, “Effective Address Space Structure for Radix-using Partitions”). Value substitution does not happen in HPT translation. When a guest uses Radix Tree translation, PID substitution may take place. When a host uses Radix Tree translation, both PID and LPID substitution may take place. When a host uses HPT translation, the only special significance associated with LPIDR=0 is with regard to Segment Table walk when MSR_{HV}=1, as described later.

adjunct: An adjunct is a software entity that resides in a partition along with an operating system and its applications in order to efficiently provide services (e.g. device drivers) for the partition. The adjunct is managed by the hypervisor. It runs in problem state with $MSR_{HV\ PR}$=0b11, thereby restricting the resources it can modify (MSR_{PR}=1) and causing its interrupts to go to the hypervisor (MSR_{HV}=1). It shares an HPT with the partition it serves. The adjunct’s storage is kept separate from the client partition’s storage using Virtual Page Class Key protection. (The adjunct’s lightness of weight derives from not requiring a full partition context switch (SLB flush, TLB flush, LPID/PID change, etc.) when the client partition invokes the services of the adjunct.) Each hardware thread may have its own unique translations for an adjunct. As a result, adjunct segment descriptors cannot exist in the process’s Segment Table and must instead be bolted in the SLB manually. The adjunct construct exists only with a hypervisor that uses HPT translation and only for LPID≠0. The adjunct has its own 64-bit EA space. Entry to an adjunct is only possible from hypervisor state. Prior to dispatching the adjunct, the hypervisor must invalidate SLB entries that map the effective address range that will be used by the adjunct. Similarly, on exit from the adjunct, the hypervisor must invalidate its SLB entries

5.7.5 Address Ranges Having Defined Uses

The address ranges described below have uses that are defined by the architecture.

- Fixed interrupt vectors

  Except for the first 256 bytes, which are reserved for software use, the real page beginning at real address 0x0000_0000_0000_0000 is either used for interrupt vectors or reserved for future interrupt vectors.

- Implementation-specific use

  The two contiguous real pages beginning at real address 0x0000_0000_0000_1000 are reserved for implementation-specific purposes.

- Offset Real Mode interrupt vectors

  The real page beginning at the real address specified by the HRMOR is used similarly to the page for the fixed interrupt vectors.

- Relocated interrupt vectors

  Depending on the values of $MSR_{IR\ DR}$ and $LPCR_{AIL}$ and on whether the specific interrupt will cause MSR_{HV} to change, either the virtual page containing the byte addressed by effective address 0x0000_0000_0001_8000 or the virtual page containing the byte addressed by effective address 0xC000_0000_0000_4000 may be used similarly to the page for the fixed interrupt vectors. (See Section 2.2.)

- *System Call Vectored* interrupt vectors

  Depending on the value of $LPCR_{AIL}$, the virtual page containing the effective address 0x0000_0000_0001_7000 or 0xc000_0000_0000_3000 contains the interrupt vectors that are invoked by the *System Call Vectored* instruction.

- Partition Table

  A contiguous sequence of real pages beginning at the real address specified by the PTCR contains the Partition Table.

- Page Table

  A contiguous sequence of real pages beginning at the real address specified by the first doubleword of the Partition Table Entry when HR=0 contains the Page Table.

5.7.5.1 Effective Address Space Structure for Radix-using Partitions

When Radix Tree translation is in use but translation is disabled (MSR_{PR}=0), MSR_{HV} selects between partition-scoped translation of the real mode guest real address, formed by treating $EA_{0:1}$ as 0b00, and hypervisor real mode (see Section 5.7.3). When Radix Tree translation is in use and translation is enabled, $EA_{0:1}$ together with MSR_{HV} are used to select one of as many as four Radix Trees with which to perform process-scoped translation, as a technique to make system calls and interrupts more efficient by avoiding the need to immediately change the contents of the PIDR and LPIDR. (See Figure 20 for an illustration of the mappings.) Since there’s nothing to prevent a process from generating any address in the 64b EA space, the exceptional cases are defined as follows. When a quadrant of the EA space has no associated Radix Tree, access to it results in an Instruction Segment exception or Data Segment exception, as appropriate

for the type of access. Similarly, reference to any portion of these quadrants or the real mode guest real address described above that is not mapped by a Radix Tree (versus mapped by an invalid entry) will cause an Instruction or Data Segment exception.

Programming Note

Note that the quadrant structure is only available to software running in 64b mode. 32b software will only be able to access storage mapped by its own Radix Tree.

Programming Note

Warning: The functionality described in this section, e.g. directing most hypervisor interrupts to the LPID=0 translation tables, places great importance on the correctness of the format of and mappings in Partition Table Entry 0 and the tables it anchors. An error in any of these structures could have severe consequences including system checkstops and hangs.

Programming Note

The intent is that the PIDR and LPIDR contents indicate the process and partition on behalf of which execution is taking place. For example, when a guest process interrupts to the hypervisor, execution to service the interrupt will generally be on behalf of the guest partition. When execution changes to be purely managing hypervisor resources that are not directly tied to any partition, the hypervisor should set LPIDR to 0.

For guest and host applications and the guest operating system, quadrant 0 ($EA_{0:1}$=0b00) addresses the Radix Tree for the application and quadrant 3 ($EA_{0:1}$=0b11) addresses the direct supervisor of the application. For the guest and host applications, it will frequently be the case that page protection is used to prevent access to quadrant 3, but partition-wide shared text and/or data may also be located there. Quadrants 1 and 2 have no associated Radix Tree.

Programming Note

Outboard accelerators may commonly be limited to accessing quadrants 0 and 3 as a matter of platform architecture. In such platforms, references to quadrants 1 and 2 may be regarded as errors.

For the hypervisor, quadrants 0 and 3 are as described above. Quadrant 1 ($EA_{0:1}$=0b01) addresses the guest application and quadrant 2 ($EA_{0:1}$=0b10) addresses the guest operating system, one of which experienced a hypervisor interrupt or performed a system call to the hypervisor. It will rarely be the case that quadrants 0 and 1 will be in use concurrently. A new value will usually be put in PIDR between accesses to quadrants 0 and 1.

When MSR_{HV}=1 and $EA_{0:1}$=0b00 or 0b11, only process-scoped translation is performed. When MSR_{HV}=0 and $MSR_{IR/DR}$=0, only partition-scoped translation is performed. Otherwise, nested process- and partition-scoped translations are performed.

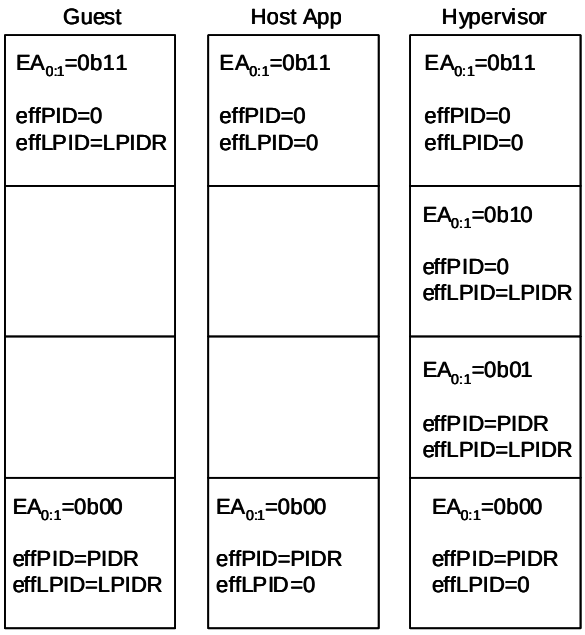


Figure 20. Effective address space structure when using Radix Tree translation

5.7.6 In-Memory Tables

The In-Memory Tables are used to find the tables that are used in the actual translation process for the partition and process that are executing. They enable hardware, including accelerator hardware separate and distinct from the Power ISA processors in the platform, to perform the translation process largely without software intervention. Description of the In-Memory Table structure follows. Hardware may cache the contents of the In-Memory Tables. Variants of ***tlbie[l]*** may be used to manage the caching even though the In-Memory Table contents are not cached in the TLB. When "thread" is used in descriptions of the ordering of accesses and operations (e.g. invalidations) related to translation cache management, it should be understood to include execution streams in accelerators unless otherwise stated or obvious from context.

When an address in the In-Memory Table structure is specified to be a virtual or guest real address, the access to that address is considered to be performed with translation on. For a host using HPT translation, a

base page size is specified for each such access to be used in the HPT search. The hypervisor can override the Segment Table Page Size in the Process Table Entry ($PRTE_{STPS}$, see Figure 23) using $LPCR_{ISL}$. The base page size for the Process Table ($PATE_{PRTPS}$) can be safely altered by the hypervisor since the OS does not have direct access to the Partition Table Entry. All accesses to the In-Memory Tables, the Segment Tables, and the guest Radix Tables that are performed with translation on, including for instruction address translation, are data accesses performed as if MSR_{PR}=0 for the purpose of determing storage protection, although instruction side translation exceptions cause [H]ISI. (A specific example of the implications of this is that tables used to translate instruction fetches may be located in guarded or no-execute storage.)

Programming Note

The descriptors in the entries in this section and its subsections contain addresses that are properly aligned so that no shifting is required. For example, the minimum size of the Partition Table is 4KB, so PATB has the thirteenth least significant address bit as its least significant bit. To construct the real address for a 4KB table, 12 zeros are appended on the right, and an appropriate number of address bits are removed from the left to match the real address size (m) supported by the implementation. For an aligned 8K table, bit 51 of the PTCR would be disregarded, and 13 zeros would be appended.

5.7.6.1 Partition Table

The Partition Table Control Register (PTCR) is a hypervisor privileged SPR that contains the host real address of the base of the Partition Table and specifies its size. Software must ensure that the contents of the PTCR are the same for all processors in the system prior to enabling translation or transferring control to a partition.

///	PATB	//	PATS
0 3	51	58	63

Partition Descriptor

Bit(s)	Name	Description
4:51	PATB	Partition Table Base
59:63	PATS	Partition Table Size=$2^{12+PATS}$ PATS≤24

All other fields are reserved.

Figure 21. Partition Table Control Register

Programming Note

If it becomes necessary to shrink the Partition Table or to change PATB to point to a table that is not identical to the existing one, it is necessary to issue ***tlbie*** with RIC=2 to invalidate caching of outdated In-Memory Table Entries.

The Partition Table is composed of a pair of doublewords per partition. The first doubleword indicates whether the partition uses HPT or Radix Tree translation, and contains the base of the host's translation table structure in host real memory. The first doubleword also contains the size of the table structure and the size of the Root Page Directory for a hypervisor using Radix Tree translation, or the base page size for the VRMA for Paravirtualized HPT translation. Additional details about the parameters for HPT translation follow.

The HTABORG field contains the high-order 42 bits of the 60-bit real address of the Page Table. The Page Table is thus constrained to lie on a 2^{18} byte (256 KB) boundary. At least 11 bits from the hash function (see Figure 30) are used to index into the Page Table. The minimum size Page Table is 256 KB (2^{11} PTEGs of 128 bytes each).

The Page Table can be any size 2^n bytes where $18 \le n \le 46$. As the table size is increased, more bits are used from the hash to index into the table.

The HTABSIZE field contains an integer giving the number of bits (in addition to the minimum of 11 bits) from the hash that are used in the Page Table index. This number must not exceed 28. HTABSIZE is used to generate a mask of the form 0b00...011...1, which is a string of 28 - HTABSIZE 0-bits followed by a string of HTABSIZE 1-bits. The 1-bits determine which additional bits (beyond the minimum of 11) from the hash are used in the index (see Figure 30).

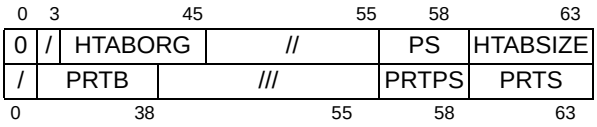
On implementations that support a real address size of only m bits, m<60, bits 0:59-m of the HTABORG field are treated as reserved bits, and software must set them to zeros.

Programming Note

Let n equal the virtual address size (in bits) supported by the implementation. If n<67, software should set the HTABSIZE field to a value that does not exceed n-39. Because the high-order 78-n bits of the VSID are assumed to be zeros, the hash value used in the Page Table search will have the high-order 67-n bits either all 0s (primary hash; see Section 5.7.9.2) or all 1s (secondary hash). If HTABSIZE > n-39, some of these hash value bits will be used to index into the Page Table, with the result that certain PTEGs will not be searched.

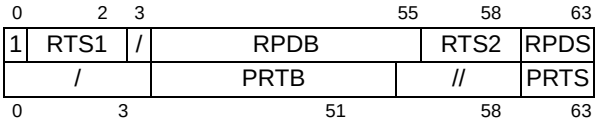
Example:

Suppose that the Page Table is 16,384 (2^{14}) 128-byte PTEGs, for a total size of 2^{21} bytes (2 MB). A 14-bit index is required. Eleven bits are provided from the hash to start with, so 3 additional bits from the hash must be selected. Thus the value in HTABSIZE must be 3. The HPT may begin on any 256KB boundary.



Paravirtualized HPT Partition Table Entry

Bit(s)	Name	Description
0	HR	Host Radix 0b0- hypervisor uses HPT translation for this partition 0b1- hypervisor uses Radix Tree translation for this partition
4:45	HTABORG	Hashed Page Table Base
56:58	PS	Page Size (uses L||LP encoding as in SLBE)
59:63	HTABSIZE	HPT size = $2^{HTABSIZE+18}$ HTABSIZE ≤ 28
1:38	PRTB	Process Table Base (when UPRT=1)
56:58	PRTPS	Process Table Page SIze (when UPRT=1) (uses L||LP encoding as in SLBE)
59:63	PRTS	Process Table Size = $2^{12+PRTS}$ PRTS≤24 (when UPRT=1)



Radix on Radix Partition Table Entry

Bit(s)	Name	Description
0	HR	Host Radix 0b0- hypervisor uses HPT translation for this partition 0b1- hypervisor uses Radix Tree translation for this partition
1:2	RTS1	Radix Tree Size[0:1]
4:55	RPDB	Root Page Directory Base
56:58	RTS2	Radix Tree Size[2:4] (number of address bits mapped), size=2^{RTS+31}
59:63	RPDS	Root Page Directory Size = 2^{RPDS+3}, RPDS≥5
4:51	PRTB	Process Table Base
59:63	PRTS	Process Table Size = $2^{12+PRTS}$ PRTS≤24 (when UPRT=1)

All other fields are reserved.

Figure 22. Partition Table Entry Variants

The second doubleword of the Partition Table Entry contains the base of the partition's Process Table, which is a guest real address (or effective address when effective LPID=0) for radix hypervisor and virtual address for HPT hypervisor, and the size of the Process Table. The Process Table is assumed to be

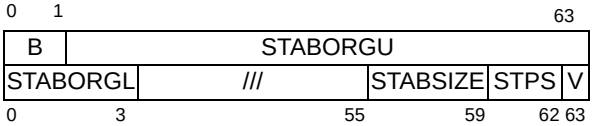
aligned. Software that uses Radix Tree translation must set the low order PRTS bits of PRTB to 0s. When Segment Tables are provided, the Process Table base address is specified as a VSID with the assumption that the Process Table is located at zero offset in the segment, and also includes the base page size used for the HPT search, with the rest of the implied segment descriptor being B=0b01 (1TB segment), Ks=Kp=0, N=0, C=0, and virtual page class key protection does not apply. The Partition Table Entry variants are illustrated in Figure 22.Notethat a configuration with HR=1 for a non-zero LPID and HR=0 for LPID=0 is considered an unsupported MMU configuration because it would attempt to perform HPT translation in quadrants 0 and 3 when MSR_{HV}=1. In addition, LPID=0 with Radix Tree translation is an unsupported MMU configuration when MSR_{HV}=0.

Programming Note

The size of the Process Table is provided to simplify hardware design and testing. The size enables the hardware to mask address bits instead of providing an adder. No size checking is provided for these tables. (An out-of-range LPID or PID will not produce an exception simply because of its size.) Hypervisor software may help detect such errors by the OS by not providing a translation for virtual / guest real addresses for a page or two beyond the end of the Process Table.

5.7.6.2 Process Table

The Process Table is composed of a quadword Process Table Entry per process in the partition. For partitions that use HPT translation, the Process Table Entry contains a Segment Table descriptor, which is composed of the origin of the Segment Table in virtual address space, the size of the segment and pages that hold the table, the size of the table, and a valid bit that is turned off while changes are made to the entry and Segment Table. The translation of the base address of the Segment Table is completed using an implied segment descriptor with Ks=Kp=0, N=0, C=0, and virtual page class key protection does not apply. For partitions that use Radix Tree translation, the Process Table Entry contains a Radix Tree root descriptor. When running on a host that uses Radix Tree translation, there are two cases. When effLPID=0, the RPDB is a host real address. Otherwise, the address is a guest real address and must undergo translation using the hypervisor's Radix Tree for the partition (i.e. the "partition-scoped" tables, as defined later).



DW	Bit(s)	Name	Description
0	0:1	B	Segment Table Segment Size
	2:63	STABORGU	Segment Table Origin Upper
1	0:3	STABORGL	Segment Table Origin Lower
	56:59	STABSIZE	Segment Table Size = $2^{12+STABSIZE}$, STABSIZE ≤ 12
	60:62	STPS	Segment Table Page Size (uses L||LP encoding as in SLBE)
	63	V	Valid

DW	Bit(s)	Name	Description
0	1:2	RTS1	Radix Tree Size[0:1]
	3	/	Reserved
	4:55	RPDB	Root Page Directory Base
	56:58	RTS2	Radix Tree Size[2:4] (number of address bits mapped), size=2^{RTS+31}
	59:63	RPDS	Root Page Directory Size = 2^{RPDS+3}, RPDS$\geq$5

All other fields are reserved.

Figure 23. Process Table Entry Variants

5.7.7 Address Translation Overview

The effective address (EA) is the address generated by the hardware for an instruction fetch or for a data access. If address translation is enabled, this address is passed to the Address Translation mechanism, which attempts to convert the address to a real address which is then used to access storage. If the effective address cannot be translated, a storage exception (see Section 5.2) occurs.

The architecture defines segment translation and two types of page translation. Segment translation is paired with HPT translation. The other supported "pairing" is two level Radix Tree translation. Either of these pairings can be used to translate an effective address into a host real address. The In-Memory Tables described above determine the translation mode used by a partition, as well as the locations of the Page Tables and Segment Tables, and the base page size for the Segment Tables. When MSR_{HV}=1 and/or MSR_{IR}=0 or MSR_{DR}=0 (as appropriate for the type of access), the steps taken for a given mode vary. See Sections 5.7.11.3 and 5.7.11.4 for details.

The pairing of Segment translation and Hashed Page Table (HPT) translation applies Segment translation to an effective address to produce a virtual address as described in Section 5.7.8, and HPT translation to the virtual address to produce a host real address as described in Section 5.7.9. Segment translations can be established by both the guest and the hypervisor, but the HPT translation is always managed by the hypervisor with the guest typically giving direction via system calls to the hypervisor in a paravirtualization relationship. This mode is commonly referred to as Paravirtualized HPT translation. The segment translation is managed on a per-process ("process-scoped") basis, mapping a smaller effective address space into a large "partition-scoped" virtual address space, where the segment can be used as a shared memory object. There is also the possibility of thread-unique mappings. In the basic version of HPT translation, storage exceptions are directed to the operating system, which in turn issues system calls to the hypervisor. When Virtualized Partition Memory is enabled, storage exceptions are directed to the hypervisor, enabling a higher degree of memory overcommitment as the hypervisor transparently steals pages from the partition. Figure 24 gives an overview of the address translation process.

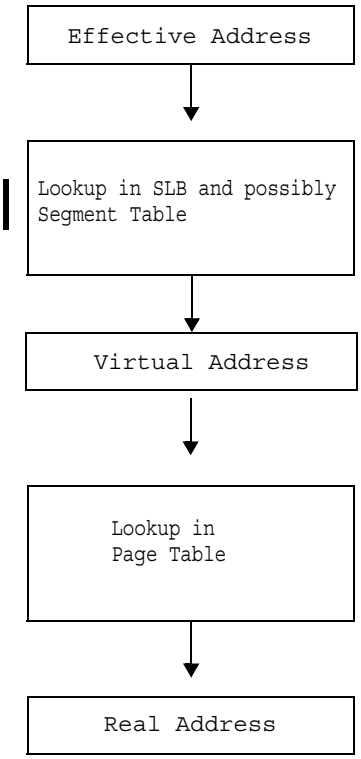


Figure 24. Address translation overview

In Paravirtualized HPT mode, the hypervisor also uses the segment/HPT pairing, and can create a process called an "adjunct". To do so, it eliminates any potentially conflicting guest segment mappings and creates adjunct mappings prior to dispatching the adjunct.

In the other pairing, Radix Tree translation is used for both the process-scoped and partition-scoped mappings. This mode is sometimes referred to as nested Radix or Radix on Radix translation. Figure 25 gives an overview of the address translation process for Radix on Radix translation. Note that each level of the guest Radix Tree produces a guest real address that must itself undergo partition-scoped translation. See Figure 36 for a detailed illustration of the entire process.

Storage exceptions for process-scoped translation are directed to the operating system, and storage exceptions for partition-scoped translation are directed to the hypervisor. (In this categorization, single level translation is considered process-scoped translation except when VPM is active, in which case it is treated like partition-scoped translation.) As a result, for Radix on Radix translation, the hypervisor can use the partition-scoped mapping to limit the size of the guest real address space, and Virtualized Partition Memory is not necessary to enable a higher degree of memory overcommitment. If in Radix on Radix mode the guest real address is outside the range covered by the partition-scoped Radix Tree, the results are boundedly undefined.

The address specified in ASDR is the guest real address or VSID for which translation has most immediately failed except when the translation fails too early to produce that value. HDAR will generally contain the EA or lower VA bits for which translation has most immediately failed. For example, in the case of a Page Directory being paged out, the ASDR will contain the guest real address of the Page Directory Entry (down to bit 51), rather than the GRA of the datum being accessed. Exceptions may be manifest in unexpected ways. For example, an instruction fetch can fail to set a Change bit in the host PTE mapping the guest PTE. Similarly, the Reference bit update might fail for lack of write authority on the PTE.

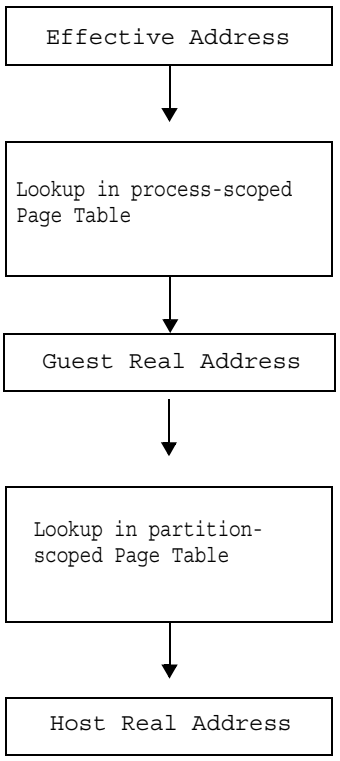


Figure 25. Address translation overview, Radix on Radix

Translation Lookaside Buffer

Conceptually, the Page Table is searched by the address relocation hardware to translate every reference. For performance reasons, the hardware usually keeps a Translation Lookaside Buffer (TLB) that holds PTEs that have recently been used. The TLB is searched prior to searching the Page Table. As a consequence, when software makes changes to the Page Table it must perform the appropriate TLB invalidate operations to maintain the consistency of the TLB with the Page Table (see Section 5.10).

An implementation may associate each of its TLB entries with the partition for which the TLB entry was created, so that the entries can be retained while other partitions are executing. In this case, when a valid TLB entry is created, the LPID value from LPIDR is written into the TLB entry.

Programming Notes

1. Page Table Entries may or may not be cached in a TLB.
2. It is possible that the hardware implements more than one TLB, such as one for data and one for instructions. In this case the size and shape of the TLBs may differ, as may the values contained therein.
3. Use the ***tlbie*** instruction to ensure that the TLB no longer contains a mapping for a particular virtual page.

5.7.8 Segment Translation

Conversion of a 64-bit effective address to a virtual address is done by searching the Segment Lookaside Buffer (SLB) as shown in Figure 26. If no matching translation is found in the SLB, $LPCR_{UPRT}$=1, and either MSR_{HV}=0 or LPID=0, the Segment Table is searched. For implicit accesses, implicit segment descriptors are provided, as described elsewhere in this chapter.

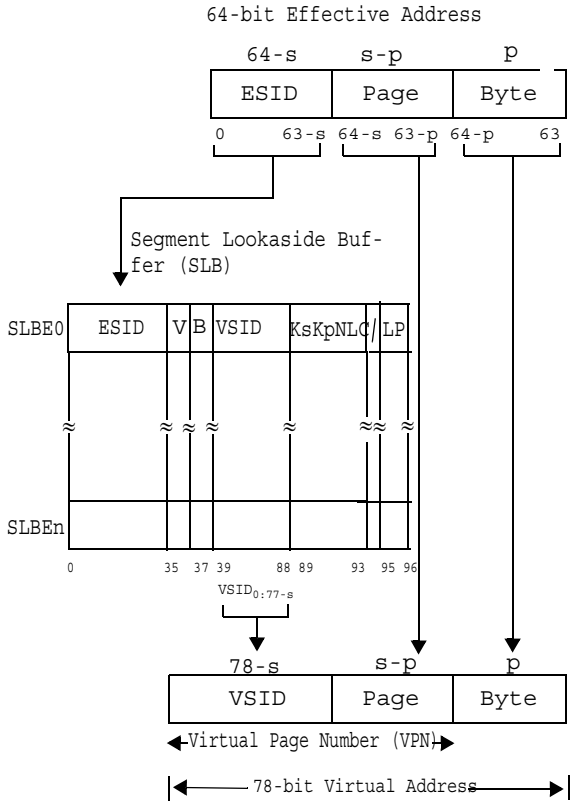


Figure 26. Translation of 64-bit effective address to 78 bit virtual address

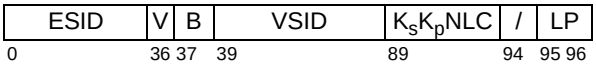
5.7.8.1 Segment Lookaside Buffer (SLB)

The Segment Lookaside Buffer (SLB) specifies the mapping between Effective Segment IDs (ESIDs) and Virtual Segment IDs (VSIDs). The number of SLB entries is implementation-dependent, except that all implementations provide at least 32 entries.

The first four entries, and when $LPCR_{UPRT}$=0 all of the entries, of the SLB are managed by software, using the instructions described in Section 5.9.3.2. See Chapter 11. “Synchronization Requirements for Context Alterations” on page 1133 for the rules that software must follow when updating the SLB.

SLB Entry

Each SLB entry (SLBE, sometimes referred to as a “segment descriptor”) maps one ESID to one VSID. Figure 27 shows the layout of an SLB entry



Bit(s)	Name	Description
0:35	ESID	Effective Segment ID
36	V	Entry valid (V=1) or invalid (V=0)
37:38	B	Segment Size Selector 0b00 - 256 MB (s=28) 0b01 - 1 TB (s=40) 0b10 - reserved 0b11 - reserved
39:88	VSID	Virtual Segment ID
89	K_s	Supervisor (privileged) state storage key (see Section 5.7.13.2)
90	K_p	Problem state storage key (See Section 5.7.13.2.)
91	N	No-execute segment if N=1
92	L	Virtual page size selector bit 0
93	C	Class
95:96	LP	Virtual page size selector bits 1:2

All other fields are reserved. B_0 ($SLBE_{37}$) is treated as a reserved field.

Figure 27. SLB Entry

Instructions cannot be executed from a No-execute (N=1) segment.

Segments may contain a mixture of page sizes. The L and LP bits specify the base virtual page size for the segment. The $SLB_{L||LP}$ encodings are those shown in Figure 28. The base virtual page size (also referred to as the “base page size”) is the smallest virtual page size that can be used to map a given access, and in most cases is the smallest virtual page size for the segment. (The exception is that multiple base virtual page sizes can occur within the same segment when the base page size specified for a given implicit access (e.g. of one segment table) does not match the base page size specified for another implicit access (e.g. of a different segment table or the process table) or for explicit accesses. References to the base page size for a segment will be understood not to preclude or functionally conflict with this possibility.) The base virtual page size is 2^b bytes. The actual virtual page size (also referred to as the “actual page size” or “virtual page size”) is specified by $PTE_{L\ LP}$.

encoding	base page size
0b000	4 KB
0b101	64 KB
additional values[1]	2^b bytes, where b > 12 and b may differ among encoding values

[1] The "additional values" are implementation-dependent, as are the corresponding base virtual page sizes. Any values that are not supported by a given implementation are reserved in that implementation.

Figure 28. Page Size Encodings

For each SLB entry, software must ensure the following requirements are satisfied.

- L||LP contains a value supported by the implementation.
- The base virtual page size selected by the L and LP fields does not exceed the segment size selected by the B field.
- If s=40, the following bits of the SLB entry contain 0s.
  - $ESID_{24:35}$
  - $VSID_{38:49}$

  The bits in the above two items are ignored by the hardware.

The Class field of the SLBE is used in conjunction with the ***slbia*** instruction (see Section 5.9.3.2). "Class" refers to a grouping of SLB entries and implementation-specific lookaside information so that only entries in a certain group need be invalidated and others might be preserved. The Class value assigned to an implementation-specific lookaside entry derived from an SLB entry must match the Class value of that SLB entry.

Software must ensure that the SLB contains at most one entry that translates a given effective address, and that if the SLB contains an entry that translates a given effective address, then any previously existing translation of that effective address has been invalidated. An attempt to create an SLB entry that violates this requirement may cause a Machine Check.

Programming Note

Class values should be assigned such that Class 0 is used for translations that are expected to be long-lived and Class 1 is used for translations that are expected to be short-lived. This assignment facilitates use of the ***slbia*** instruction, for which several IH values cause preferential invalidation of Class 1 SLB entries and lookaside information entries.

Programming Note

It is permissible for software to replace the contents of a valid SLB entry without invalidating the translation specified by that entry provided the specified restrictions are followed. See Chapter 11 Note 10.

5.7.8.2 SLB Search

When the hardware searches the SLB, all entries are tested for a match with the EA. For a match to exist, the following conditions must be satisfied for indicated fields in the SLBE.

- V=1
- $ESID_{0:63-s}$=$EA_{0:63-s}$, where the value of s is specified by the B field in the SLBE being tested

If no match is found, the search fails. If one match is found, the search succeeds. If more than one match is found, one of the matching entries is used as if it were the only matching entry, or a Machine Check occurs.

If the SLB search succeeds, the virtual address (VA) is formed from the EA and the matching SLB entry fields as follows.

$$VA=VSID_{0:77-s} \,||\, EA_{64-s:63}$$

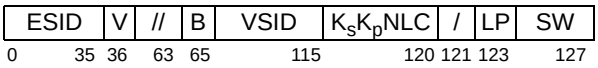
The Virtual Page Number (VPN) is bits 0:77-p of the virtual address. The value of p is the actual virtual page size specified by the PTE used to translate the virtual address (see Section 5.7.9.1). If $SLBE_N$ = 1, the N (No-execute) value used for the storage access is 1.

If the SLB search fails and the state is not such that a Segment Table search will be performed, a *segment fault* occurs. This is an Instruction Segment exception or a Data Segment exception, depending on whether the effective address is for an instruction fetch or for a data access.

5.7.8.3 Segment Table Description and Search

The Segment Table is an aligned structure composed of 16B segment descriptors organized into 128 byte Segment Table Entry Groups (STEGs). Let q = STABSIZE+12, $\log_2$(size of the Segment Table). The base of the Segment Table in virtual address space is $STABORG_{0:77-q} \,||\, {}^{q}0$. Software must set the low order q-12 bits of STABORG to 0s. Primary and secondary hashes are defined for 256MB and 1TB segments, each mapping the ESID to an STEG. The appropriate number (for the size of the Segment Table) of low order ESID bits (their inverse, for the secondary hash) directly select the STEG. The order of STEG specification in the following subsections is the preferred order for a serial search. Implementations may search the STEGs in parallel. If no match is found, a segment fault occurs. If a serial search is done, the search may stop when a match has been found. If more than one

match is found, one of the matching entries is used as if it were the only matching entry.



Bit(s)	Name	Description
0:35	ESID	Effective Segment ID
36	V	Entry valid (V=1) or invalid (V=0)
64:65	B	Segment Size Selector 0b00 - 256 MB (s=28) 0b01 - 1 TB (s=40) 0b10 - reserved 0b11 - reserved
66:115	VSID	Virtual Segment ID
116	K_s	Supervisor (privileged) state storage key (see Section 5.7.13.2)
117	K_p	Problem state storage key (See Section 5.7.13.2.)
118	N	No-execute segment if N=1
119	L	Virtual page size selector bit 0
120	C	Class
122:123	LP	Virtual page size selector bits 1:2
124:127	SW	available for software use

All other fields are reserved.

Figure 29. Segment Table Entry

5.7.8.3.1 Primary Hash for 256MB Segment

The STEG is located at host VA
$STABORG_{0:77-q} || EA_{43-q:35} ||$ 0b0000000.
Each of the 8 STEs are searched to find a valid entry (V=1, B=0b00) that matches the ESID ($STE_{ESID[0:35]} = EA_{0:35}$) of the access being translated.

5.7.8.3.2 Primary Hash for 1TB Segment

The STEG is located at host VA
$STABORG_{0:77-q} || EA_{31-q:23} ||$ 0b0000000.
Each of the 8 STEs are searched to find a valid entry (V=1, B=0b01) that matches the ESID ($STE_{ESID[0:23]} = EA_{0:23}$) of the access being translated.

5.7.8.3.3 Secondary Hash for 256MB Segment

The STEG is located at host VA
$STABORG_{0:77-q} || \neg EA_{43-q:35} ||$ 0b0000000.
Each of the 8 STEs are searched to find a valid entry (V=1, B=0b00) that matches the ESID ($STE_{ESID[0:35]} = EA_{0:35}$) of the access being translated.

5.7.8.3.4 Secondary Hash for 1TB Segment

The STEG is located at host VA
$STABORG_{0:77-q} || \neg EA_{31-q:23} ||$ 0b0000000.
Each of the 8 STEs are searched to find a valid entry (V=1, B=0b01) that matches the ESID ($STE_{ESID[0:23]} = EA_{0:23}$) of the access being translated.

5.7.9 Hashed Page Table Translation

In Paravirtualized HPT mode, conversion of a 78-bit virtual address to a real address is done by searching the Page Table as shown in Figure 30.

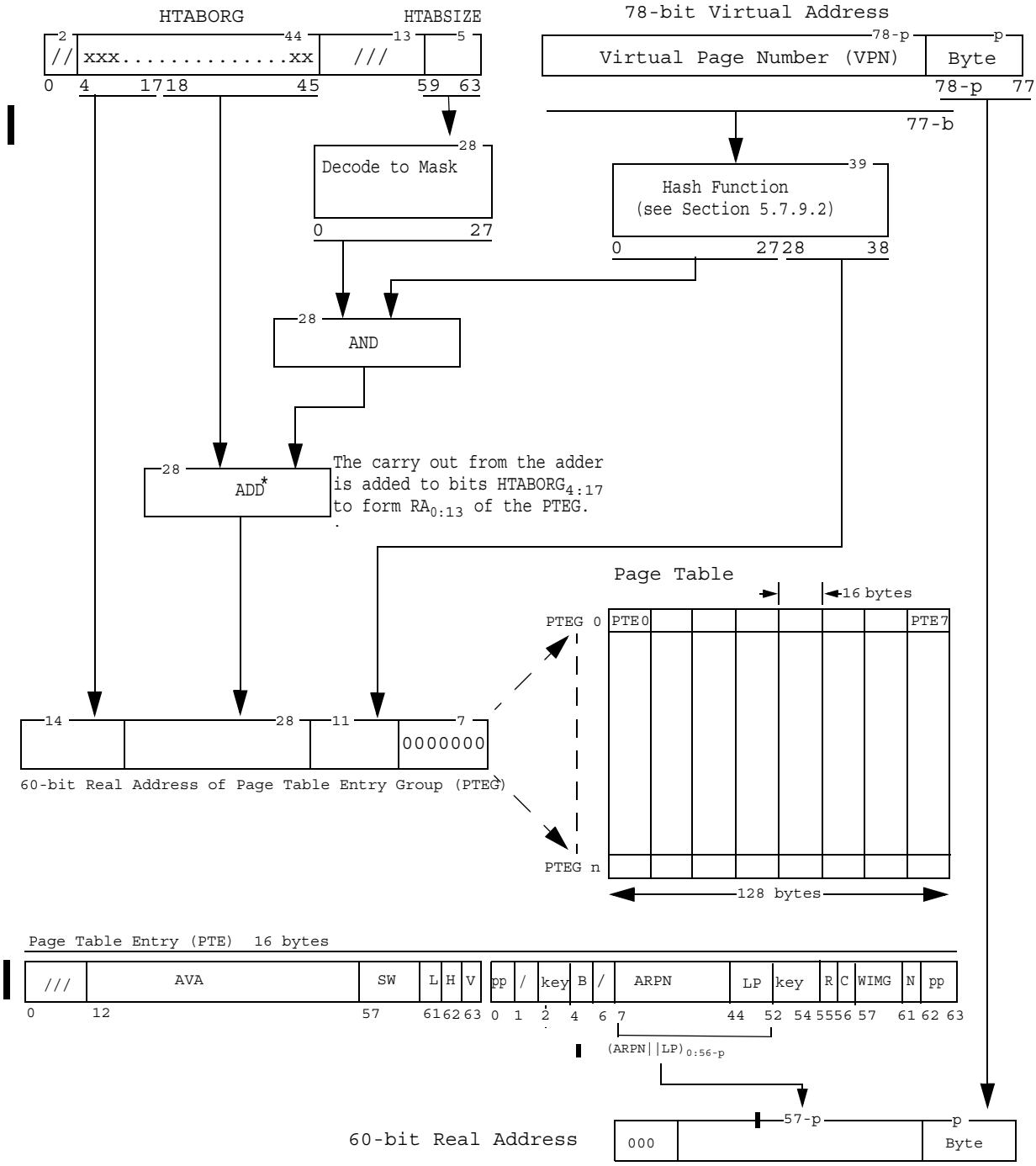


Figure 30. Translation of 78-bit virtual address to 60-bit real address

5.7.9.1 Hashed Page Table

The Hashed Page Table (HTAB) is a variable-sized data structure that specifies the mapping between virtual page numbers and real page numbers, where the real page number of a real page is bits 0:47 of the address of the first byte in the real page. The HTAB's size can be any size 2^n bytes where $18 \le n \le 46$. The HTAB must be located in storage having the storage control attributes that are used for implicit accesses to it (see Section 5.7.3.4). The starting address must be a multiple of 2^{18} bytes.

The HTAB contains Page Table Entry Groups (PTEGs). A PTEG contains 8 Page Table Entries (PTEs) of 16 bytes each; each PTEG is thus 128 bytes long. PTEGs are entry points for searches of the Page Table.

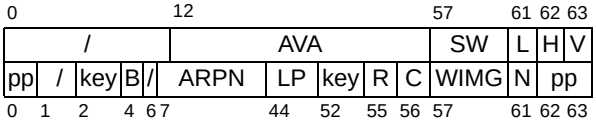
See Section 5.10 for the rules that software must follow when updating the Page Table.

Programming Note

The Page Table must be treated as a hypervisor resource (see Chapter 2), and therefore must be placed in real storage to which only the hypervisor has write access. Moreover, the contents of the Page Table must be such that non-hypervisor software cannot modify storage that contains hypervisor programs or data.

Page Table Entry

Each Page Table Entry (PTE) maps one VPN to one RPN. Figure 31 shows the layout of a PTE. This layout is independent of the Endian mode of the thread.



Dword	Bit(s)	Name	Description
0	12:56	AVA	Abbreviated Virtual Address
	57:60	SW	Available for software use
	61	L	Virtual page size 0b0 - 4 KB 0b1 - greater than 4KB (large page)
	62	H	Hash function identifier
	63	V	Entry valid (V=1) or invalid (V=0)
1	0	pp	Page Protection bit 0
	2:3	key	KEY bits 0:1
	4:5	B	Segment Size 0b00 - 256 MB 0b01 - 1 TB 0b10 - reserved 0b11 - reserved
	7:43	ARPN	Abbreviated Real Page Number
	44:51	LP	Large page size selector
	52:54	key	KEY bits 2:4
	55	R	Reference bit
	56	C	Change bit
	57:60	WIMG	Storage control bits
	61	N	No-execute page if N=1
	62:63	pp	Page Protection bits 1:2

All other fields are reserved.

Figure 31. Page Table Entry

Programming Note

The H bit in the Page Table Entry should not be set to one unless the secondary Page Table search has been enabled.

Because the length of the Abbreviated Virtual Address (AVA) field is only 45 bits, on implementations of this version of the architecture the virtual address size cannot exceed 68 bits ($n \le 68$). On implementations for which $n<68$, bits 0:67-n of the AVA field must be zeros.

If $b \le 23$, the AVA field contains bits 10:54 of the VA. Otherwise bits 0:67-b of the AVA field contain bits 10:77-b of the VA, and bits 68-b:44 of the AVA field must be zero.

Programming Note

The AVA field omits the low-order 23 bits of the VA. These bits are not needed in the PTE, because the low-order b of these bits are part of the byte offset into the virtual page and, if $b<23$, the high-order 23-b of these bits are always used in selecting the PTEGs to be searched (see Section 5.7.9.2).

A virtual page is mapped to a sequence of 2^{p-12} contiguous real pages such that the low-order p-12 bits of the real page number of the first real page in the sequence are 0s.

$PTE_{L\ LP}$ specify both a base virtual page size (henceforth referred to as the "base page size") and an actual virtual page size (henceforth referred to as the "actual page size" or "virtual page size"). The actual page size is the size of the virtual page mapped by the PTE. The base page size is the smallest actual page size that a segment can contain for explicit accesses or for a given implicit access, and plays a role in the placement of the PTE in the HPT.

If $PTE_L=0$, the base virtual page size and actual virtual page size are 4KB, and ARPN concatenated with LP (ARPN||LP) contains the page number of the real page that maps the virtual page described by the entry.

If PTE_L=1, the base page size and actual page size are specified by PTE_{LP}. In this case, the contents of PTE_{LP} have the format shown in Figure 32. Bits labelled “r” are bits of the real page number. Bits labelled “z” specify the base page size and actual page size. The values of the “z” bits used to specify each size are implementation-dependent. The values of the “z” bits used to specify each size, along with all possible values of “r” bits in the LP field, must result in LP values distinct from other LP values for other sizes. Actual page sizes 4KB and 64KB are always supported; other actual page sizes are implementation-dependent. If PTE_L=1, the actual page size must be greater than 4 KB. Which combinations of different base page size and actual page size are supported is implementation-dependent, except that the combination of a base page size of 4 KB with an actual page size of 64 KB is always supported.

PTE_{LP}	actual page size
r r r r _r r r z	≥8 KB
r r r r _r r z z	≥16 KB
r r r r _r z z z	≥32 KB
r r r r _z z z z	≥64 KB
r r r z _z z z z	≥128 KB
r r z z _z z z z	≥256 KB
r z z z _z z z z	≥512 KB
z z z z _z z z z	≥1 MB

Figure 32. Format of PTE_{LP} when PTE_L=1

There are at least 2 formats of PTE_{LP} that specify a 64 KB page. One format is used with $SLBE_{L||LP}$ = 0b000 and one format is used with $SLBE_{L||LP}$ = 0b101.

The actual page size selected by the LP field must not exceed the segment size selected by the B field. Forms of PTE_{LP} not supported by a given implementation are treated as reserved values for that implementation.

The concatenation of the ARPN field and bits labeled “r” in the LP field contain the high-order bits of the real page number of the real page that maps the first 4KB of the virtual page described by the entry.

The low-order p-12 bits of the real page number contained in the ARPN and LP fields must be 0s and are ignored by the hardware.

Programming Note

The actual page size specified by a given PTE_{LP} format is at least $2^{12+(8-c)}$, where c is the number of r bits in the format.

Programming Note

Implementations often have TLBs and implementation-specific lookaside buffers (e.g. ERATs) used to cache translations of recently used storage addresses. Mapping virtual storage to large pages may increase the effectiveness of such lookaside buffers, improving performance, because it is possible for such buffers to translate a larger range of addresses, reducing the frequency that the Page Table must be searched to translate an address.

Instructions cannot be executed from a No-execute (N=1) page.

Page Table Size

The number of entries in the Page Table directly affects performance because it influences the hit ratio in the Page Table and thus the rate of page faults. If the table is too small, it is possible that not all the virtual pages that actually have real pages assigned can be mapped via the Page Table. This can happen if too many hash collisions occur and there are more than 16 entries for the same primary/secondary pair of PTEGs (when the secondary Page Table search is enabled) or more than 8 entries for the same primary PTEG (when the secondary Page Table search is disabled).

While this situation cannot be guaranteed not to occur for any size Page Table, making the Page Table larger than the minimum size (see Section 5.7.6.1) will reduce the frequency of occurrence of such collisions.

Programming Note

If large pages are not used, it is recommended that the number of PTEGs in the Page Table be at least half the number of real pages to be accessed. For example, if the amount of real storage to be accessed is 2^{31} bytes (2 GB), then we have $2^{31-12}=2^{19}$ real pages. The minimum recommended Page Table size would be 2^{18} PTEGs, or 2^{25} bytes (32 MB).

5.7.9.2 Page Table Search

When the hardware searches the Page Table, the accesses are performed as described in Section 5.7.3.4.

An outline of the HTAB search process is shown in Figure 30. Up to two hash functions are used to locate a PTE that may translate the given virtual address.

1. A 39-bit hash value is computed from the VA. The value of s is the value specified in the SLBE that was used to generate the virtual address; the value of b is equal to $\log_2$(base page size specified

in the SLBE that was used to translate the address).**Primary Hash:**

If s=28, the hash value is computed by Exclusive ORing $VA_{11:49}$ with $(^{11+b}0||VA_{50:77-b})$

If s=40, the hash value is computed by Exclusive ORing the following three quantities: $(VA_{24:37}||^{25}0)$, $(0||VA_{0:37})$, and $(^{b-1}0||VA_{38:77-b})$

The 60-bit real address of a PTEG is formed by concatenating the following values:

- Bits 0:27 of the 39-bit appropriate primary or secondary hash value ANDed with the mask generated from bits 59:63 of the first doubleword of the Partition Table Entry (HTABSIZE) and then added to the value of bits 4:45 of the first doubleword of the Partition Table Entry (HTABORG).
- Bits 28:38 of the 39-bit hash value.
- Seven 0-bits.

This operation identifies a particular PTEG, called the “primary PTEG”, whose eight PTEs will be tested.

2. **Secondary Hash:**

   If the secondary Page Table search is enabled ($LPCR_{TC}$=0), perform the secondary hash function as follows; otherwise do not perform step 2 and proceed to step 3 below.

   If s=28, the hash value is computed by taking the ones complement of the Exclusive OR of $VA_{11:49}$ with $(^{11+b}0||VA_{50:77-b})$

   If s=40, the hash value is computed by taking the ones complement of the Exclusive OR of the following three quantities: $(VA_{24:37}||^{25}0)$, $(0||VA_{0:37})$, and $(^{b-1}0||VA_{38:77-b})$

   The 60-bit real address of a PTEG is formed by concatenating the following values:

   - Bits 0:27 of the 39-bit appropriate primary or secondary hash value ANDed with the mask generated from bits 59:63 of the first doubleword of the Partition Table Entry (HTABSIZE) and then added to the value of bits 4:45 of the first doubleword of the Partition Table Entry (HTABORG).
   - Bits 28:38 of the 39-bit hash value.
   - Seven 0-bits.

   This operation identifies the “secondary PTEG”.

3. As many as 8 PTEs in the primary PTEG and, if the secondary Page Table search is enabled, 8 PTEs in the secondary PTEG are tested to determine if any translate the given virtual address. Let q = minimum(54, 77-b). For a match to exist, the following conditions must be satisfied, where SLBE is the SLBE used to form the virtual address.
   - PTE_H=0 for the primary PTEG, 1 for the secondary PTEG
   - PTE_V=1
   - PTE_B=$SLBE_B$
   - $PTE_{AVA[0:q-10]}$=$VA_{10:q}$
   - if b = 12 then
     (PTE_L = 0) | (PTE_{LP} specifies the 4KB base page size)
     else
     (PTE_L = 1) & (PTE_{LP} specifies the base page size specified by $SLBE_{L||LP}$)

   If no match is found, the search fails. The result is a page fault -- a [Hypervisor] Instruction Storage exception or a [Hypervisor] Data Storage exception, depending on whether the effective address is for an instruction fetch or for a data access. If one match is found, the search succeeds. If more than one match is found, one of the matching entries is used as if it were the only matching entry, or a Machine Check occurs.

If the Page Table search succeeds, the real address (RA) is formed by concatenating the following values, where the p value is the $\log_2$ (actual page size specified by $PTE_{L\ LP}$).

- three 0 bits
- bits 0:56-p of ARPN||LP from the matching PTE
- bits 64-p:63 of the effective address (the byte offset)

$$RA = 0b000 \ || \ (ARPN \ || \ LP)_{0:56-p} \ || \ EA_{64-p:63}$$

A TLB entry may be created as a result of the successful HPT translation. Depending on the specific TLB implementation, the scope of the entry may be the base page size, the virtual page size, or any size in between. In the absence of a TLB, software would be required to create a PTE for each base page sized piece of storage within the virtual page. The number of PTEs actually created to map a virtual page will depend on the scopes supported for TLB entries, the access pattern, and the lifetime of the TLB entries. Hardware generally will not create more than one TLB entry to translate a given virtual address. Multiple matching TLB entries may be created only if the Page Table contains PTEs that map different-sized virtual pages that overlap in the virtual address space. If a TLB search finds multiple matching TLB entries created from such PTEs, one of the matching TLB entries is used as if it were the only matching entry, or a Machine Check occurs. Software should scrupulously avoid creating such mappings.

Programming Note

If PTE_L = 0, the actual page size (and base page size) are 4 KB. Otherwise the actual page size and base page size are specified by PTE_{LP}.

Since hardware searches the Page Table using a value of b equal to $\log_2$ (base page size specified in the SLBE that was used to translate the address) regardless of the actual page size, the hardware Page Table search will identify different PTEs for VAs in different 2^b-byte blocks of the virtual page if the actual page size is larger than the base page size. Therefore, there may need to be a valid PTE corresponding to each 2^b-byte block of the virtual page that is referenced. For an actual page size that is larger than 2^{23} (8 MB), the PTE_{AVA} will differ among some or all of these PTEs. Depending on the Page Table size, some or all of these PTEs may be in the same PTEG. Any such PTEs that are in the same PTEG will differ in the value of PTE_H or PTE_{AVA} or both.

All PTEs for the same virtual page should have the same values in the Page Protection, KEY, ARPN, WIMG, and N fields. A set of values from any one of the PTEs that maps the virtual page may be used for an access in the virtual page since lookaside buffer information may be used to translate the virtual address.

To avoid creating multiple matching PTEs, software should not create PTEs for each of two different virtual pages that overlap in the virtual address space. If the virtual page sizes differ, two virtual pages overlap if the values of virtual address bits 0:77-p for both virtual pages are the same, where 2^p is the actual virtual page size of the larger page.

Programming Note

Because a segment may contain pages of different sizes, the Page Table search uses the segment's base page size (which is the same for all virtual pages in the segment).

- The value of b used when searching the Page Table to identify the PTEGs to be checked for a match is $\log_2$(segment's base page size).
- A PTE (in the selected PTEGs) satisfies the Page Table search only if the base page size specified in the PTE is equal to the segment's base page size.

The matching PTE supplies the actual page size, 2^p; this value of p is used in forming the real address.

A virtual page of 2^p bytes in a segment with a base page size of 2^b bytes may be mapped by as many as $2^{(p-b)}$ PTEs.

Programming Note

To obtain the best performance, Page Table Entries should be allocated beginning with the first empty entry in the primary PTEG, or with the first empty entry in the secondary PTEG if the primary PTEG is full and the secondary Page Table search is enabled ($LPCR_{TC}$=0).

In Paravirtualized HPT mode, the N (No-execute) value used for the storage access is the result of ORing the N bit from the matching PTE with the N bit from the SLB entry that was used to translate the effective address.

5.7.10 Radix Tree Translation

Radix Tree translation uses a nested set of tables to map storage with increasing granularity. Although there is no requirement for an individual table to have uniform content, Page Directories generally contain pointers to other Page Directories or Page Tables (Page Directory Entries, PDEs), while Page Tables are the leaf tables that contain PTEs. Each Page Directory Entry and Page Table Entry in the Radix Tree is 8 bytes long. A Radix Tree root descriptor (RTRD) specifies the size of the address being translated, the size of the root table, and its location. RTRDs appear in variants of the Partition and Process Table Entries. (See Figures 22 and 23.) The Root Page Directory Size (RPDS) is specified as $\log_2$ (number of entries in the table). That number of bits is taken from the most significant end of the portion of the address being translated, as an index to choose an element in the Root Page Directory. The entries in the Root Page Directory each point to another page of entries, and give its size in the Next Level Size field, PDE_{NLS}. The next most significant NLS bits are taken from the address to choose an entry in that table. The process continues until an entry is found that has its Leaf bit set, indicating it is a Page Table Entry. The base size of the page mapped by the PTE is determined by the number of bits remaining in the address after removing the bits used to select the Page Directory and Page Table Entries. An example with RPDS = 13 and PDE_{NLS} = 9 in each Page Directory is shown in Figure 33.

The sizes of table supported at each level of the Radix Tree, as well as the ultimate page sizes supported, are implementation specific with the following exceptions. Implementations must support two Radix Tree configurations that map 52 bit effective addresses: each starting with a 64KB root page size followed by 2 levels of 4KB tables, ending with either a 256 byte table or a 4KB table. The former produces a page size of 64KB and the latter a 4KB page size. In both cases, a leaf node in the next to last level of table produces a 2MB page size.

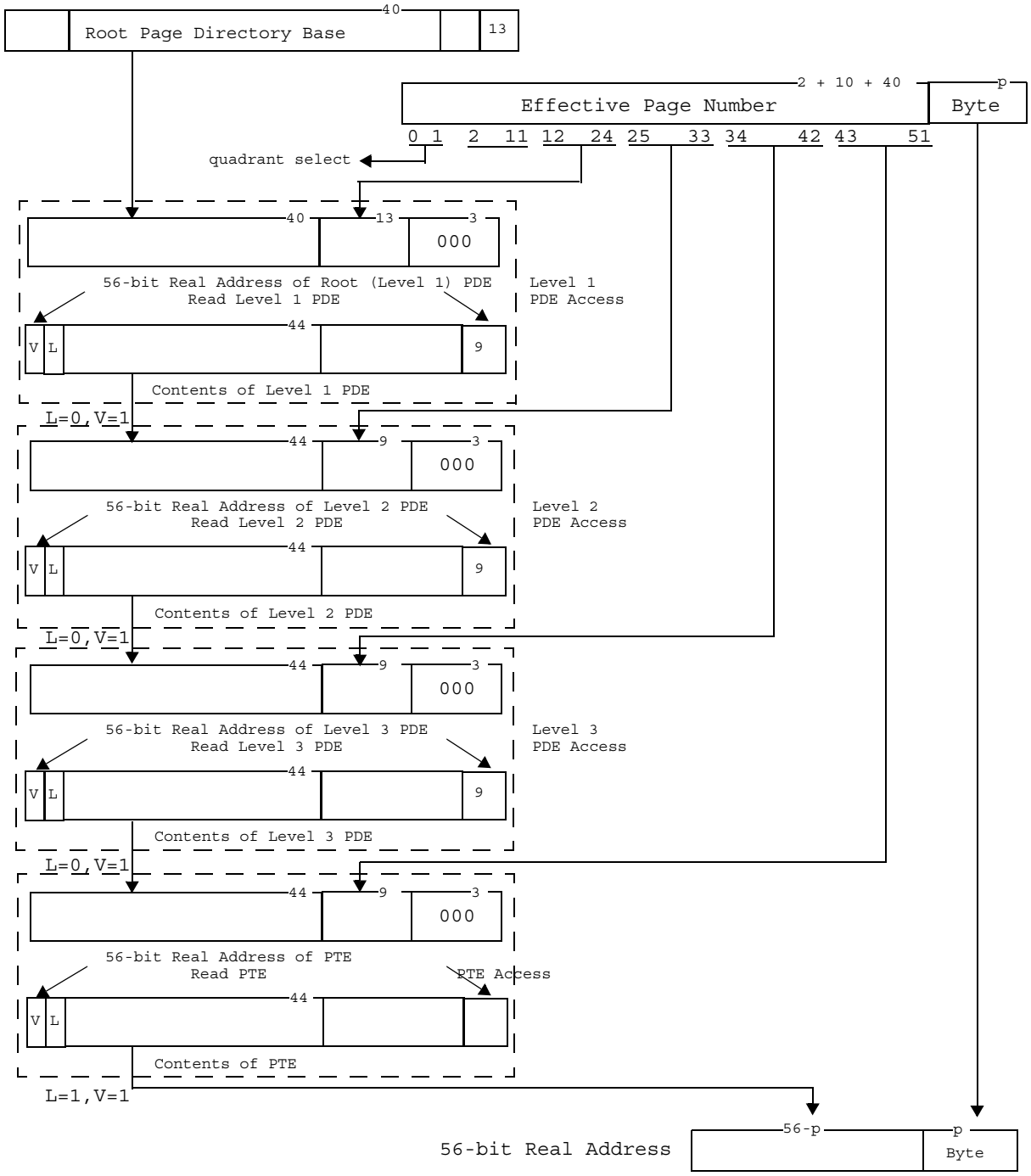
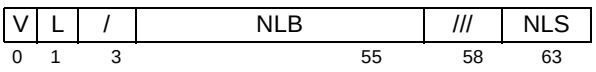


Figure 33. Four level Radix Tree walk translating a 52b EA with NLS=13 in the root PDE and NLS=9 in the other PDEs.

5.7.10.1 Radix Tree Page Directory Entry

.



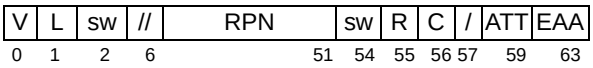
Bit(s)	Name	Description
0	V	Valid
1	L	Leaf (entry is a PTE)
4:55	NLB	Next Level Base
59:63	NLS	Next Level Size (size of next level of table is 2^{NLS+3}), NLS≥5

All other fields are reserved.

Figure 34. Radix Tree Page Directory Entry

5.7.10.2 Radix Tree Page Table Entry

.



Bit(s)	Name	Description
0	V	Valid
1	L	Leaf (entry is a PTE)
2	sw	SW bit 0
7:51	RPN	Real Page Number
52:54	sw	SW bits 1:3
55	R	Reference
56	C	Change
58:59	Att	Attributes (equivalent WIMG value) 0b00- normal memory (0010) 0b01- SAO (1110) 0b10- non-idempotent I/O (0111) 0b11- tolerant I/O (0110)
60:63	EAA	Encoded Access Authority
	0	Privilege (applies only to process-scoped translation) 0 - problem state access permitted; privileged access controlled by key 0 of the [I]AMR 1 - privileged access only
	1	Read 0 - loads not permitted 1 - loads permitted
	2	Read/Write 0 - loads and stores not permitted 1 - loads and stores permitted
	3	Execute 0 - instruction execution not permitted 1 - instruction execution permitted

All other fields are reserved.

Figure 35. Radix Tree Page Table Entry

5.7.10.3 Nested Translation

When MSR_{HV}=0 and translation is enabled, each guest real address must undergo partition-scoped translation using the hypervisor's Radix Tree for the partition. See Figure 36.

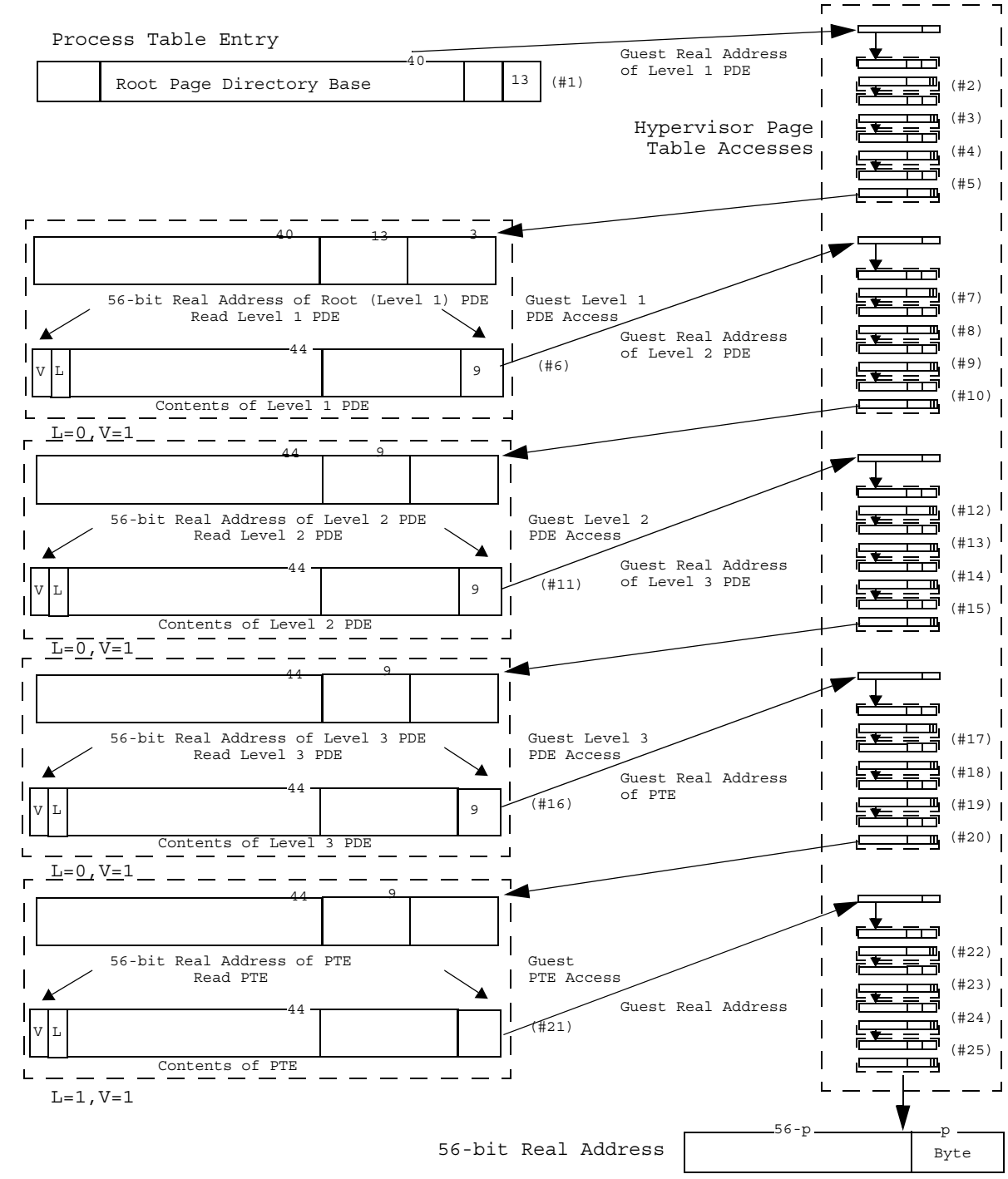


Figure 36. Radix on Radix Page Table search for a 52-bit EA depicting memory reads 1-24 numbered in sequence

When nested translation is being performed, there is the potential for two different sets of protection settings and two different sets of storage attributes. For protection settings, the least permissive values take effect. For read, write, and execute authority, each is controlled independently based on the least permissive setting of the two translation mechanisms (including all component authority mechanisms within each of them). For storage ordering, the SAO attribute takes effect when both SAO and normal memory attributes are specified. (The hypervisor will typically specify “normal memory” and the OS may override that with SAO.) The

Guarded attribute is controlled by the process-scoped PTE. Mismatches of the Caching Inhibited attribute have the following behavior. If the process-scoped PTE specifies I=1 when the partition-scoped PTE specifies I=0, the result is I=0. The reverse mismatch raises a data storage or instruction storage exception, as appropriate for the access. The results of these rules are shown in Table 4. Together these rules can produce the WIMG=0b0011 state that any individual Att value cannot express.

Unless otherwise stated or obvious from context, references elsewhere in the Books to storage control attributes for nested Radix Tree translations apply to the result of combining the guest and host storage control attributes as specified above. For example, the description of TM disallowed access types in Section 5.3.1 applies to the results of the combining.

	partition-scoped Att	00	01	10	11
process-scoped Att	*SAO/I/G*	000	100	011	010
00	000	000	100	Att mismatch	Att mismatch
01	100	100	100	Att mismatch	Att mismatch
10	011	001	001	011	011
11	010	000	000	010	010

Table 4: Effective SAO, I and G attributes for nested translation

Programming Note

The mismatched Caching Inhibited attribute in the lower left quadrant above is given defined behavior instead of excepting in order to support frame buffer emulation. For frame buffer emulation, the guest believes it is writing to a frame buffer (I=1) in address space that the hypervisor maps to normal memory (I=0).

Reference and Change bit recording is done in both the process-scoped and partition-scoped Page Table Entries. Recording is done as described in Section 5.7.12, “Reference and Change Recording”.

For performance reasons, the result of each walk of a Radix Tree may be cached in a TLB. Logically, the result of each walk is cached separately. For nested translation, the effective to guest real (process-scoped) translation may be cached, as well as the partition-scoped translation for each guest real address produced by the translation process. A minimum of two TLB accesses is required to complete a nested translation: one for the effective to guest real address and one for the guest real to host real address. (An implementation may optimize the process, as long as the optimization can be managed correctly using the ***tlbie*** instructions that software will use to manage the logical model.)

5.7.11 Translation Process

As previously described, in its most complicated form the translation process includes the following steps:

- use of the PTCR to find the required Partition Table Entry
- use of the Partition Table Entry to find the partition-scoped Page Table
- use of the Partition Table Entry and the partition-scoped Page Table to find the required Process Table Entry
- use of the Process Table Entry and partition-scoped Page Table to find the required Segment Table Entry or walk the process-scoped Page Table (i.e. translate the effective address to a virtual or guest real address), and
- use of the partition-scoped Page Table to translate the virtual or guest real address.

Depending on the translation mode and process state, some of these steps may be skipped. The following subsections enumerate the cases and explain the steps in more detail.

5.7.11.1 Fully-Qualified Address

The storage control facilities enable hardware to perform the entire translation process given a “fully-qualified address” and context that makes it a unique input. In addition to its normal use, the term “effective address” is sometimes used as shorthand for the fully-qualified address, and the architecture should be read with this possibilty in mind. The following are the components of the fully-qualified address.

- effLPID
- effPID
- EA

The additional context required to perform a translation or match a cached translation may include the following.

- $PATE_{HR}$ (selected using the value in LPIDR, not effLPID)
- $MSR_{HV\ PR\ IR\ DR}$

At a high level, the translation mode is selected by the Host Radix bit found in the Partition Table Entry. The Host Radix bit indicates whether the partition is using HPT or Radix Tree translation. Given the overall process, $MSR_{HV\ PR\ IR\ DR}$ determine where and how the process is entered.

5.7.11.2 Finding the Page Tables

[The following description assumes that no legacy mode is active, i.e. $LPCR_{UPRT}$=1.]

The components of the fully-qualified address are used to determine the table(s) used in the translation process. The effective LPID and effective PID are used to find the appropriate Page Table base address(es) using the In-Memory Table structures. Some types of translation use process-scoped Page Tables, some use partition-scoped Page Tables, and some use both.

Process-scoped table descriptors are found in the Process Tables as follows. The Partition Table Entry (PATE) host real address is calculated by adding the Partition Table Base Address (PATB||^{12}0) in the PTCR with 16 times the effective LPID. The second doubleword of the entry contains the base address of the Process Table for the partition. The Process Table is assumed to be aligned in effective (HR=1, effLPID=0), virtual, or guest real address space. The Process Table Entry (PRTE) host real address is calculated by ORing the Process Table Base Address (PRTB||^{40}0 for for an HPT host and PRTB||^{12}0 for a radix host) in the PATE with 16 times the effective PID and then performing partition-scoped translation. (If the table is not aligned or is not large enough to support the PID value, an unreported error will most likely result.) The Process Table Entry at that location contains a process-scoped table base address, which is a guest real address for a radix guest (HV=0), a host real address for a radix host (HV=1), or a virtual address (all cases using HPT translation). The virtual or guest real address must be translated via the appropriate partition-scoped table.

Programming Note

The guest real or virtual address of the Process Table, for a radix or HPT guest, respectively, may be set via an hcall. The radix guest may choose to map the Process Table into its own effective address space. These matters are not visible to the architecture.

Programming Note

Note that the sole purpose of partition-scoped Page Table descriptor when LPID=0 for a radix host is to translate the effective addresses of the Process Table Entries for LPID=0. (If the Process Table Base address for LPID=0 was a real address, the Process Table would have to be in contiguous real storage.) This descriptor will commonly be the same as the descriptor found in the LPID=0, PID=0 Process Table Entry, both pointing to the hypervisor's own page trable, but it may be set up to point to a table used solely to translate the addresses of Process Table Entries.

Partition-scoped Page Table descriptors are found in the Partition Table as follows. The Partition Table Base Address is found in the PTCR. The effective LPID (times 16 bytes per partition) is used to index off the Partition Table Base Address to find the appropriate Partition Table Entry. The first doubleword of the entry contains the base address of the Page Table.

5.7.11.3 Obtaining Host Real Address, Radix on Radix

The following cases exist.

- Guest access to quadrant 0 with translation on: process-scoped translation is performed on LPIDR||PIDR||EA, with the result subject to partition-scoped translation with effective LPID=LPIDR.
- Guest access to quadrant 3 with translation on: process-scoped translation is performed on LPIDR||0||EA, with the result subject to partition-scoped translation with effective LPID=LPIDR.
- Hypervisor access to quadrant 1 with translation on: process-scoped translation is performed on LPIDR||PIDR||EA, with the result subject to partition-scoped translation with effective LPID=LPIDR if LPIDR≠0.
- Hypervisor access to quadrant 2 with translation on: process-scoped translation is performed on LPIDR||0||EA, with the result subject to partition-scoped translation with effective LPID=LPIDR if LPIDR≠0.
- Guest OS access with translation off: partition-scoped translation is performed with effective LPID = LPIDR.
- Hypervisor or host application access to quadrant 0 with translation on: process-scoped translation is performed on 0||PIDR||EA.
- Hypervisor or host application access to quadrant 3 with translation on: process-scoped translation is performed with 0||0||EA.
- Hypervisor real mode access: subject to HRMOR and EA_0 as described in Section 5.7.3.1.

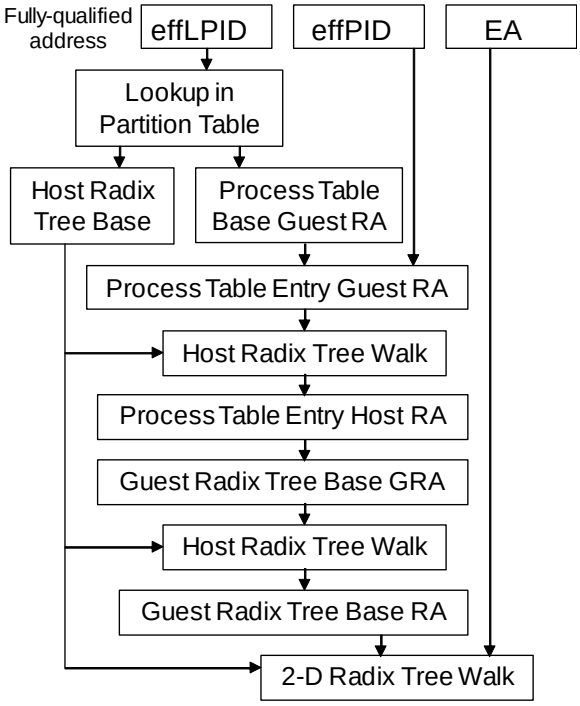


Figure 37. Radix on Radix translation, general case

5.7.11.4 Obtaining Host Real Address, HPT

There are two scenarios for Paravirtualized HPT translation. The first is the legacy scenario with a native HPT hypervisor. The second scenario is for a Radix Tree translation hypervisor providing a Paravirtualized HPT environment for the guest. In this latter scenario, the LPID=0 Partition Table Entry will have HR=1. For both scenarios when MSR_{HV}=1, the LPID value is always taken from LPIDR and the PID value is always taken from PIDR. In the latter scenario, the hypervisor will explicitly set LPIDR=0 when it wants to use its Radix Tree(s).

When using Paravirtualized HPT translation, the process-scoped Page Tables are replaced by Segment Tables, and the description in Section 5.7.11.2, "Finding the Page Tables" can be read with that substitution in mind. The process-scoped translation is the effective-to-virtual translation described in Section 5.7.8. In-Memory Table walks are processed via the LPID=LPIDR partition-scoped HPT.

As with the previous enumerations, this is done from a hardware point of view. As a result, it does not differentiate the software cases for which Segment translation should only be satisfied by bolted translations

The following cases exist.

- Guest access with translation on: process-scoped translation is performed on LPIDR||PIDR||EA with the result subject to partition-scoped translation using parameters from the matching segment descriptor.
- Hypervisor or adjunct access with translation on and LPID≠0: process-scoped translation, limited to an SLB search with no Segment Table walk, is performed on LPIDR||PIDR||EA, with the result subject to partition-scoped translation using parameters from the matching segment descriptor.
- Hypervisor or adjunct access with translation on and LPID=0: process-scoped translation (with Segment Table walk) is performed on LPIDR||PIDR||EA, with the result subject to partition-scoped translation using parameters from the matching segment descriptor.
- Guest OS access with translation off: subject to VPM, as described in Section 5.7.3.3.
- Hypervisor real mode access: subject to HRMOR and EA_0 as described in Section 5.7.3.1.

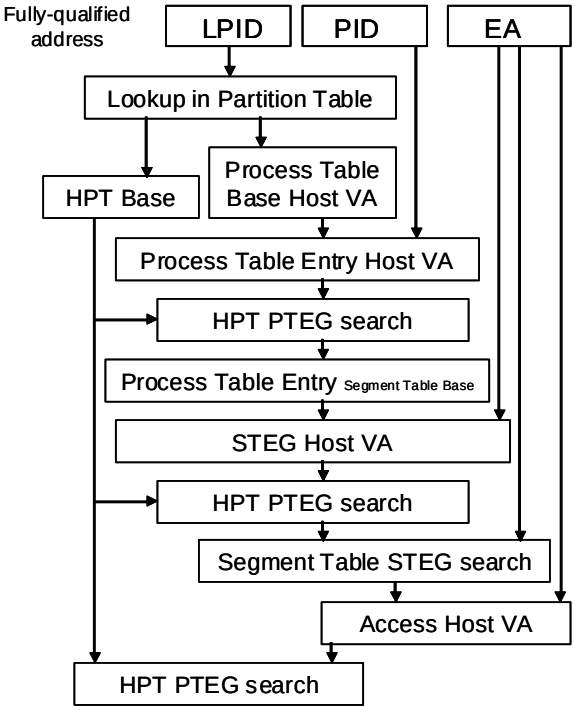


Figure 38. Paravirtualized HPT translation

5.7.12 Reference and Change Recording

When operating in Paravirtualized HPT mode, Reference (R) and Change (C) bits are updated in any one of what could be multiple (because of the multiple base size PTEs mapping a virtual page) Page Table Entries that map the virtual page that is being accessed. When operating in Radix on Radix mode, Reference (R) and Change (C) bits may be updated in multiple Page Table

Entries that are accessed as part of the translation process. (For example, each access to a guest's Page Directory or Page Table Entry potentially sets a Reference bit in the partition-scoped table mapping it.) If the storage operand of a *Load* or *Store* instruction crosses a virtual page boundary, the accesses to the components of the operand in each page are treated as separate and independent accesses to each of the pages for the purpose of setting the Reference and Change bits.

For Radix Tree translation, the Reference and Change bits are set atomically, as though the PTE was read to perform the translation using a *Load And Reserve* instruction, and conditional on the translation being valid and correct (and on the existence of the reservation), the appropriate bit(s) are set as though with a *Store Conditional* instruction. ("as though" indicates that the reservation(s) held for this purpose are distinct from one another and from the reservation established by a Load And Reserve instruction.) For HPT translation, Reference and Change bits are set as though the PTE was read to perform the update using a (simple) *Load* instruction and the appropriate bit(s) are set as though with a (simple) *Store* instruction. Setting the bits need not be atomic with respect to performing the access that causes the bits to be updated. The Reference bit must contain 1 in order to load from the corresponding page. The Change bit must contain 1 in order to store to the corresponding page. If hardware is unable to set the bit(s) atomically for Radix Tree translation, a [Hypervisor] Data Storage or [Hypervisor] Instruction Storage interrupt will be caused.

Programming Note

The interrupt indicates to software that it must set the appropriate bit(s) itself. Note that an instruction fetch can cause a Change bit to be set, for example in the host Page Table Entry that maps the guest Page Table Entry if the instruction fetch causes the Reference bit to be set in the guest Page Table Entry.

Programming Note

The atomic setting of the Reference and Change bits enables an optimized sampling of them, for example when determining what pages to reclaim for other uses. To accurately sample the bits under HPT translation, it is necessary to first invalidate the PTE and the corresponding TLB entries. The optimized sequence eliminates the requirement for the relatively expensive invalidation of the TLB entries before sampling the bits. Instead, software may simply load the PTE using a *Load And Reserve* instruction, and then set the PTE invalid using a *Store Conditional* instruction. The TLB invalidation may be deferred indefinitely and grouped into cluster bombs for improved performance. The Reference and Change bits sampled in this manner are accurate (if the store conditional succeeds) because with the PTE marked invalid, it will be impossible to access a page for which the appropriate bit is not already set.

Programming Note

In nested Radix Tree translation, as many as three Change bits may be set: in the process-scoped and partition-scoped PTEs for the access itself, and in the partition-scoped PTE that maps the process-scoped PTE. Similarly, a large number of Reference bits may be set, including for each partition-scoped PTE that maps a process-scoped PDE or PTE.

Reference and Change bits are set by the hardware as described below. An attempt to access storage may cause one or more of the bits to be set (as described below) even if the access is not performed. The bits are updated in the Page Table Entry if the new value would otherwise be different from the old value for the virtual page, as determined by examining either the Page Table Entry or any lookaside information for the virtual page (e.g., TLB) maintained by the hardware.

Reference Bit

The Reference bit is set to 1 if the corresponding access (load, store, implicit access, or instruction fetch) is required by the sequential execution model and is performed. Otherwise the Reference bit may be set to 1 if the corresponding access is attempted, either in-order or out-of-order, even if the attempt causes an exception, except that the Reference bit is not set to 1 for the access caused by an indexed *Move Assist* instruction for which the XER specifies a length of zero.

Change Bit

The Change bit is set to 1 if a *Store* instruction is executed and the store is performed or if an

implicit update is performed. Otherwise in general the Change bit may be set to 1 if a *Store* instruction is executed and the store is permitted by the storage protection mechanism and, if the *Store* instruction is executed out-of-order, the instruction would be required by the sequential execution model in the absence of the following kinds of interrupts:

- system-caused interrupts (see Section 6.4 on page 1057)
- Floating-Point Enabled Exception type Program interrupts when the thread is in an Imprecise mode.

The only exceptions to the preceding statement are that the Change bit is not set to 1 if the instruction is a *Store String Indexed* instruction for which the XER specifies a length of zero, if the instruction is a *Load Atomic* or *Store Atomic* instruction with an invalid function code, or if the instruction is a *Store Caching Inhibited* instruction executed when MSR_{DR}=1.

Programming Note

A virtual page in a segment with a smaller base page size may be mapped by multiple PTEs. For each access of a virtual page, hardware may search the Page Table to update the R and C bits. If lookaside buffer information for the virtual page already indicates that all such bits to be set have already been set in a PTE that maps the virtual page, hardware need not make an update. Consider the following sequence of events:

1. A virtual page is mapped by 2 PTEs A and B and the R and C bits in both PTEs are 0.
2. A Load instruction accesses the virtual page and the R bit is updated in PTE A.
3. A Load instruction accesses the virtual page and the R bit is updated in PTE B.
4. A Store instruction accesses the virtual page and the C bit is updated in PTE B.
5. The virtual page is paged out. Software must examine both PTE A and B to get the state of the R and C bits for the virtual page.

Furthermore, if in event 2, PTE A was not found, a Data Storage interrupt or Hypervisor Data Storage interrupt may occur. Subsequently, if in event 3 or 4, PTE B was not found, a Data Storage interrupt or Hypervisor Data Storage interrupt may occur.

Programming Note

Even though the execution of a *Store* instruction causes the Change bit to be set to 1, the store might not be performed or might be only partially performed in cases such as the following.

- A *Store Conditional* instruction (***stwcx.*** or ***stdcx.***) or a Load Atomic or Store Atomic instruction (e.g. Fetch and Increment Bounded, Store Twin) is executed, but no store is performed.
- The *Store* instruction causes a Data Storage exception (all cases except *Load Atomic* or *Store Atomic* with an invalid function code, *Store Caching Inhibited* executed when MSR_{DR}=1, EAO, or storage protection violation, which do not store and are not permitted to set the Change bit).
- The *Store* instruction causes an Alignment exception.
- The Page Table Entry that translates the virtual address of the storage operand is altered such that the new contents of the Page Table Entry preclude performing the store (e.g., the PTE is made invalid, or the PP bits are changed).

  For example, when executing a *Store* instruction, the thread may search the Page Table for the purpose of setting the Change bit and then re-execute the instruction. When reexecuting the instruction, the thread may search the Page Table a second time. If the Page Table Entry has meanwhile been altered, by a program executing on another thread, the second search may obtain the new contents, which may preclude the store.
- A system-caused interrupt occurs before the store has been performed.

When the hardware updates the Reference and Change bits in a Page Table Entry, the accesses are performed as described in Section 5.7.3.4, “Storage Control Attributes for Implicit Storage Accesses” on page 986. These Reference and Change bit updates are not necessarily immediately visible to software. Executing a ***sync*** instruction ensures that all Reference and Change bit updates associated with address translations that were performed, by the thread executing the ***sync*** instruction, before the ***sync*** instruction is executed will be performed with respect to that thread before the ***sync*** instruction’s memory barrier is created. There are additional requirements for synchronizing Reference and Change bit updates in multi-threaded systems; see Section 5.10, “Translation Table Update Synchronization Requirements” on page 1043.

Programming Note

Because the ***sync*** instruction is execution synchronizing, the set of Reference and Change bit updates that are performed with respect to the thread executing the ***sync*** instruction before the memory barrier is created includes all Reference and Change bit updates associated with instructions preceding the ***sync*** instruction.

If software refers to a Page Table Entry when MSR_{DR}=1 or MSR_{HV}=0, the Reference and Change bits in the associated Page Table Entry are set as for ordinary loads and stores. See Section 5.10 for the rules software must follow when updating Reference and Change bits.

Figure 39 on page 1010 summarizes the rules for setting the Reference and Change bits. The table applies to each atomic storage reference. It should be read from the top down; the first line matching a given situation applies. For example, if ***stwcx.*** fails due to both a storage protection violation and the lack of a reservation, the Change bit is not altered. The figure applies to PTE(s) that map instructions or storage operands of instructions. When Radix Tree translation is in use, Reference and Change bits are set in other, partition-scoped, PTEs as described earlier in this section.

In the figure, the "*Load*-type" instructions are the *Load* instructions described in Books I, II, and III, and the *Cache Management* instructions that are treated as *Loads*. The "*Store*-type" instructions are the *Store* instructions described in Books I, II, and III, and the *Cache Management* instructions that are treated as *Stores*. The *Load Atomic* and *Store Atomic* instructions are considered to be both loads and stores, and as a result could match *"Load-type"* and *"Store-type"* entries in the table. As a result, *"Store-type"* entries precede *"Load-type"* entries in the table so that AMOs match *"Store-type"* entries. The "ordinary" *Load* and *Store* instructions are those described in Books I, II, and III. "set" means "set to 1".

Status of Access	R	C
Indexed *Move Assist* insn w 0 len in XER	No	No
Load or *Store Atomic* instruction with invalid function code, *Load* or *Store Caching Inhibited* executed when MSR_{DR}=1	Acc[1]	No
Storage protection violation	Acc[1]	No
Out-of-order Store-type inst'n, including transactional Store-type inst'n, excluding ***dcbtst***		
Would be required by the sequential execution model in the absence of system-caused or imprecise interrupts[3], or transaction failure	Acc	Acc[1 2]
All other cases	Acc	No
Out-of-order I-fetch or Load-type Inst'n (including transactional Load-type inst'n or ***dcbtst***)	Acc	No
In-order *Load*-type or *Store*-type insn, access not performed[4]		
Store-type insn	Acc	Acc[2]
Load-type insn	Acc	No
Other in-order access		
Other ordinary *Store*, ***dcbz***	Yes	Yes
icbi, ***icbt***, ***dcbt***, ***dcbtst***, ***dcbst***, ***dcbf***[***l***]	Acc	No
I-fetch or ordinary *Load*	Yes	No

"Acc" means that it is acceptable to set the bit.

1 It is preferable not to set the bit.

2 If C is set, R is also set unless it is already set.

3 For Floating-Point Enabled Exception type Program interrupts, "imprecise" refers to the exception mode controlled by $MSR_{FE0\ FE1}$.

4 This case does not apply to the *Touch* instructions, because they do not cause a storage access.

Figure 39. Setting the Reference and Change bits

5.7.13 Storage Protection

The storage protection mechanism provides a means for selectively granting instruction fetch access, granting read access, granting write access, and prohibiting access to areas of storage based on a number of control criteria.

The operation of the storage protection mechanism depends on the contents of one or more of the following.

- MSR bits HV, IR, DR, PR
- the key bits and N bit in the associated SLB entry
- the page protection bits, key bits, N bit, and G attribute in the associated PTE
- the AMR, IAMR, AMOR, and UAMOR

The storage protection mechanism consists of the Virtual Page Class Key Protection mechanism described in Section 5.7.13.1, the Basic Storage Protection mechanism described in Section 5.7.13.2 and Section 5.7.13.3, and the Radix Tree Translation Storage Protection mechanism described in Section 5.7.13.4.

When address translation is enabled for an access, the access is permitted in Paravirtualized HPT mode if and only if the access is permitted by both the Virtual Page Class Key Protection mechanism and the Basic Storage Protection mechanism. When address translation is enabled for a guest access, the access is permitted in Radix on Radix mode if and only if the access is permitted by the Radix Tree Translation Storage Protection mechanism for both the process-scoped and partition-scoped PTEs. When address translation is disabled for a guest access or is enabled for an access with MSR_{HV}=1, the access is permitted in Radix on Radix mode if and only if the access is permitted by the Radix Tree Translation Storage Protection mechanism for the partition-scoped PTE. When address translation is disabled for an access with MSR_{HV}=1, the access is permitted if and only if the access is permitted by the Basic Storage Protection mechanism. If an instruction fetch is not permitted, an Instruction Storage exception or a Hypervisor Instruction Storage exception is generated. If a data access is not permitted, a Data Storage exception or a Hypervisor Data Storage exception is generated.

A *protection domain* is a maximal range of effective addresses, virtual addresses, or guest real addresses for which variables related to storage protection can be independently specified (including by default, as in real and hypervisor real addressing modes), or a maximal range of addresses, effective, virtual, or guest real, for which variables related to storage protection cannot be specified. Examples include: a segment, a virtual page (including for the Virtualized Real Mode Area), the Virtualized Real Mode Area, the effective address range $0:2^{60}$-1 in hypervisor real addressing mode, and a maximal range of effective, virtual, or guest real addresses that cannot be mapped to real addresses. A *protection boundary* is a boundary between protection domains.

5.7.13.1 Virtual Page Class Key Protection

The Virtual Page Class Key Protection mechanism provides the means to assign virtual pages to one of 32 classes, and to modify data access permissions for each class by modifying the Authority Mask Register (AMR), shown in Figure 40, and to modify instruction access permissions for each class by modifying the Instruction Authority Mask Register (IAMR) shown in Figure 41.

Programming Note

If address translation is disabled for a given access, the access is not affected by the Virtual Page Class Key Protection mechanism even if the access is made in virtual real addressing mode.

Authority Mask Register

Key0	Key1	Key2	. . .	Key29	Key30	Key31
0	2	4	6	58	60	62

Bits	Name	Description
0:1	Key0	Access mask for class number 0
2:3	Key1	Access mask for class number 1
…	…	…
2n:2n+1	Keyn	Access mask for class number n
…	…	…
62:63	Key31	Access mask for class number 31

Figure 40. Authority Mask Register (AMR)

The access mask for each class defines the access permissions that apply to loads and stores for which the virtual address is translated using a Page Table Entry that contains a Key field value equal to the class number. The access permissions associated with each class are defined as follows, where AMR_{2n} and AMR_{2n+1} refer to the first and second bits of the access mask corresponding to class number n.

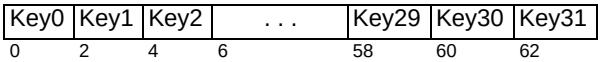
- A store is permitted if AMR_{2n}=0b0; otherwise the store is not permitted.
- A load is permitted if AMR_{2n+1}=0b0; otherwise the load is not permitted.

The AMR can be accessed using either SPR 13 or SPR 29. Access to the AMR using SPR 29 is privileged.

Programming Note

Because the AMR is part of the program context (if address translation is enabled), and because it is desirable for most application programmers not to have to understand the software synchronization requirements for context alterations (or the nuances of address translation and storage protection), operating systems should provide a system library program that application programs can use to modify the AMR.

Instruction Authority Mask Register



Bits	Name	Description
0:1	Key0	Access mask for class number 0
2:3	Key1	Access mask for class number 1
…	…	…
2n:2n+1	Keyn	Access mask for class number n
…	…	…
62:63	Key31	Access mask for class number 31

Figure 41. Instruction Authority Mask Register (IAMR)

The access mask for each class defines the access permissions that apply to instruction fetches for which the virtual address is translated using a Page Table Entry that contains a Key field value equal to the class number. The access permission associated with each class is defined as follows, where $IAMR_{2n+1}$ refers to the bit of the access mask corresponding to class number n.

- An instruction fetch is permitted if $IAMR_{2n+1}$=0b0; otherwise the instruction fetch is not permitted.

Bit 0 of each key field is reserved

Access to the IAMR is privileged.

The Authority Mask Override Register (AMOR) and the User Authority Mask Override Register (UAMOR), shown in Figure 42 and Figure 43 respectively, can be used to restrict modifications (***mtspr***) of the AMR. Also, the AMOR can be used to restrict modifications of the UAMOR and IAMR. Access to both the AMOR and UAMOR is privileged. The AMOR is a hypervisor resource.

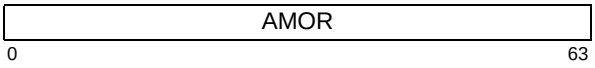


Figure 42. Authority Mask Override Register (AMOR)

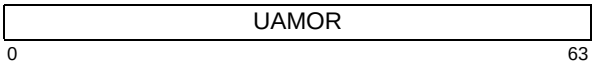


Figure 43. User Authority Mask Override Register (UAMOR)

The bits of the AMOR and UAMOR are in 1-1 correspondence with the bits of the AMR (i.e., $[U]AMOR_i$ corresponds to AMR_i). The AMOR affects modifications of the AMR and UAMOR in privileged but non hypervisor state; the UAMOR affects modifications of the AMR in problem state.

Similarly, the odd bits of the AMOR are in 1-1 correspondence with the odd bits of the IAMR (i.e., $AMOR_{2j+1}$ corresponds to $IAMR_{2j+1}$). The AMOR affects modifications of the IAMR in privileged but non hypervisor state; the IAMR cannot be accessed in problem state.

- When ***mtspr*** specifying the AMR (using either SPR 13 or SPR 29) or the IAMR is executed in privileged but non-hypervisor state, the AMOR is used as a mask that controls which bits of the resulting AMR or IAMR contents come from register RS and which AMR or IAMR bits are not modified.
- Similarly, when ***mtspr*** specifying the AMR (using SPR 13) is executed in problem state, the UAMOR is used as a mask that controls which bits of the resulting AMR contents come from register RS and which AMR bits are not modified.
- When ***mtspr*** specifying the UAMOR is executed in privileged but non-hypervisor state, the AMOR is ANDed with the contents of register RS and the result is placed into the UAMOR; the AMOR thereby controls which bits of the resulting UAMOR contents come from register RS and which UAMOR bits are set to zero.

A complete description of these effects can be found in the description of the ***mtspr*** instruction in Section 4.4.4.

Software must ensure that both bits of each even/odd bit pair of the AMOR contain the same value. — i.e., the contents of register RS for ***mtspr*** specifying the AMOR must be such that $(RS)_{2n} = (RS)_{2n+1}$ for every n in the range 0:31 — and likewise for the UAMOR. If this

requirement is violated for the UAMOR the results of accessing the UAMOR (including implicitly by the hardware as described in the second item of the preceding list) are boundedly undefined; if the requirement is violated for the AMOR the results of accessing the AMOR (including implicitly by the hardware as described in the first and third items of the list) are undefined.

Programming Note

The preceding requirement permits designs to implement the AMOR and/or UAMOR as 32-bit registers — specifically, to implement only the even-numbered bits (or only the odd-numbered bits) of the register — in a manner such that the reduction, from the architecturally-required 64 bits to 32 bits, is not visible to (correct) software. This implementation technique saves space in the hardware. (A design that uses this technique does the appropriate “fan in/out” when the register is accessed, to provide the appearance, to (correct) software, of supporting all 64 bits of the register.)

Permitting designs to implement the [U]AMOR as 32-bit registers by virtue of the software requirement specified above, rather than by defining the [U]AMOR as 32-bit registers, permits the architecture to be extended in the future to support controlling modification of the “read access” AMR bits (the odd-numbered bits) independently from the “write access” AMR bits (the even-numbered bits), if that proves desirable. If this independent control does prove desirable, the only architecture change would be to eliminate the software requirement.

Programming Note

When modifying the AMOR and/or UAMOR, the hypervisor should ensure that the two registers are consistent with one another before giving control to a non-hypervisor program. In particular, the hypervisor should ensure that if $AMOR_i$=0 then $UAMOR_i$=0, for all i in the range 0:63. (Having $AMOR_i$=0 and $UAMOR_i$=1 would permit problem state programs, but not the operating system, to modify AMR bit i.)

Programming Note

The Virtual Page Class Key Protection mechanism replaces the Data Address Compare mechanism that was defined in versions of the architecture that precede Version 2.04 (e.g., the two facilities use some of the same resources, as described below). However, the Virtual Page Class Key Protection mechanism can be used to emulate the Data Address Compare mechanism. Moreover, programs that use the Data Address Compare mechanism can be modified in a manner such that they will work correctly both on implementations that comply with versions of the architecture that precede Version 2.04 (and hence implement the Data Address Compare mechanism) and on implementations that comply with Version 2.04 of the architecture or with any subsequent version (and hence instead implement the Virtual Page Class Key Protection mechanism). The technique takes advantage of the facts that the SPR number for privileged access to the AMR (29) is the same as the SPR number for the Data Address Compare mechanism's ACCR (Address Compare Control Register), that KEY_4 occupies the same bit in the PTE as the Data Address Compare mechanism's AC (Address Compare) bit, and that the definition of $ACCR_{62:63}$ is very similar to the definition of each even-odd pair of AMR bits. The technique is as follows, where PTE1 refers to doubleword 1 of the PTE.

- Set bits 2:3 and 62:63 of SPR 29 (which is either the ACCR or the AMR) to x, where x is the desired 2-bit value for controlling Data Address Compare matches, and set bits 0:1 to 0s.
- Set $PTE1_{54}$ (which is either the AC bit or KEY_4) to the same value that the AC bit would be set to, and set $PTE1_{2:3}$ (which are either RPN bits, that correspond to a real address size larger than the size supported by any implementation that supports the Data Address Compare mechanism, or $KEY_{0:1}$) and $PTE1_{52:53}$ (which are either reserved bits or $KEY_{2:3}$) to 0s.
- Use PTE_{KEY} values 0 and 1 only for purposes of emulating the Data Address Compare mechanism, except that PTE_{KEY} value 0 may also be used for any virtual pages for which it is desired that the Virtual Page Class Key Protection mechanism permit all accesses. Do not use PTE_{KEY} =31.
- When a Hypervisor Data Storage interrupt occurs, if $HDSISR_{42}$=1 then ignore the interrupt for *Cache Management* instructions other than ***dcbz***. (These instructions can cause a virtual page class key protection violation but cannot cause a Data Address Compare match.) Otherwise forward the interrupt to the operating system, which will treat the interrupt as if a Data Address Compare match had occurred. (Note: Cases for which it is undefined whether a Data Address Compare match occurs do not necessarily cause a virtual page class key protection violation.)

(Because privileged software can access the AMR using either SPR 13 or SPR 29, it might seem that, when SPR 13 was added to the architecture (in Version 2.06), SPR 29 should have been removed. SPR 29 is retained for two reasons: first, to avoid requiring privileged software to change to use the newer SPR number; and second, to retain the ability to emulate the Data Address Compare mechanism as described above.)

Programming Note

An example of the use of the AMOR (and UAMOR) is to support adjuncts (see Section 5.7.4, “Definitions”), The hypervisor could use KEY value j for all data virtual pages that only the adjunct must be able to access. Before dispatching the partition for the first time, the hypervisor would initialize the three registers as follows.

AMR: all 0s except bits 2j and 2j+1, which would contain 1s

UAMOR: all 0s

AMOR: all 1s except bits 2j and 2j+1, which would contain 0s

Before dispatching the adjunct, the hypervisor would set UAMOR to all 0s, and would set the AMR to all 1s except bits 2j and 2j+1, which would be set to 0s. (Because the adjunct would run in problem state, there is no need for the hypervisor to modify the AMOR, and the adjunct cannot modify the UAMOR.) In addition, the hypervisor would prevent the partition from modifying or deleting PTEs that contain translations used by the adjunct.

(It may be desirable to avoid using KEY values 0, 1, and 31 for storage that only the adjunct can access, because these KEY values may be needed by the partition to emulate the Data Address Compare mechanism, as described above. Also, old software, that was written for an implementation that complies with a version of the architecture that precedes Version 2.04 (the version in which virtual page class keys were added), effectively uses KEY 0 for all virtual pages.)

Programming Note

Initialization of the UAMOR to all 0s, by the hypervisor before dispatching a partition for the first time, as described in the preceding Programming Note, permits operating systems (in partitions that run in a compatibility mode corresponding to Version 2.06 of the architecture or a subsequent version) to migrate gradually to supporting problem state access to the AMR — specifically, to avoid having to be changed immediately to modify the UAMOR and to save the AMR contents when an interrupt occurs from problem state. Relatedly, having the UAMOR contain all 0s while an application program is running protects old application programs that are “AMR-unaware”. In the absence of programming errors, such application programs would not attempt to read or modify the AMR. However, having the UAMOR contain all 0s protects such programs against modifying the AMR inadvertently.

Permitting an “AMR-unaware” application program to modify the AMR (inadvertently) is potentially harmful for the obvious reasons. (The program might set to 1 an AMR bit corresponding to accesses that are necessary in order for the program to work correctly.) Moreover, even for an operating system that includes support for problem state modification of the AMR, having the UAMOR contain all 0s allows the operating system to avoid saving and restoring the AMR for “AMR-unaware” application programs. Such an operating system would provide a system service program that allows an application program to declare itself to be “AMR-aware” — i.e., potentially to need to modify the AMR. When an application program invokes this service, the operating system would set the UAMOR to the non-zero value appropriate to the access authorities (load and/or store, for one or more key values) that the application program is allowed to modify, and thereafter would save and restore the AMR (and preserve the UAMOR) for this application program. (Having the UAMOR contain all 0s does not prevent an “AMR-unaware” program from reading the AMR, but inadvertent reading of the AMR is likely to be much less harmful than inadvertently modifying it.)

(For partitions that run in a compatibility mode corresponding to a version of the architecture that precedes Version 2.06, the PCR provides sufficient protection to application programs.)

5.7.13.2 Basic Storage Protection, Address Translation Enabled

When address translation is enabled, , the Basic Storage Protection mechanism is controlled by the following.

- MSR_{PR}, which distinguishes between supervisor (privileged) state and problem state
- K_s and K_p, the supervisor (privileged) state and problem state storage key bits in the SLB entry used to translate the effective address
- PP, page protection bits 0:2 in the Page Table Entry used to translate the effective address
- For instruction fetches only:
  - the N (No-execute) value used for the access (see Sections 5.7.8.1 and 5.7.9.2)
  - PTE_G, the G (Guarded) bit in the Page Table Entry used to translate the effective address

Using the above values, the following rules are applied.

1. For an instruction fetch, the access is not permitted if the N value is 1 or if PTE_G=1.
2. For any access except an instruction fetch that is not permitted by rule 1, a "Key" value is computed using the following formula:

   $Key \leftarrow (K_p \ \& \ MSR_{PR}) \ | \ (K_s \ \& \ \neg MSR_{PR})$

   Using the computed Key, Figure 44 is applied. An instruction fetch is permitted for any entry in the figure except "no access". A load is permitted for any entry except "no access". A store is permitted only for entries with "read/write".

Key	PP	Access Authority
0	000	read/write
0	001	read/write
0	010	read/write
0	011	read only
0	110	read only
1	000	no access
1	001	read only
1	010	read/write
1	011	read only
1	110	no access
All PP encodings not shown above are reserved. The results of using reserved PP encodings are boundedly undefined.		

Figure 44. PP bit protection states, address translation enabled

5.7.13.3 Basic Storage Protection, Address Translation Disabled

When address translation is disabled, the Basic Storage Protection mechanism is controlled by MSR_{HV}, which (when MSR_{PR}=0) distinguishes between hypervisor state and privileged but non-hypervisor state (see Chapter 2 and Section 5.7.3, "Hypervisor Real And Virtual Real Addressing Modes"). The following rules apply.

1. If MSR_{HV}=0, access authority is determined as described in Section 5.7.3.3.
2. If MSR_{HV}=1, the access is permitted.

5.7.13.4 Radix Tree Translation Storage Protection

For Radix Tree translation, an attempt to fetch instructions from Guarded storage is a storage protection violation. In all other respects, the storage protection mechanism for Radix Tree translation is completely different from what is provided for HPT translation. $EAA_{1:3}$ provide control over read, read/write, and execute access if the process has the appropriate privilege. EAA_0, together with key 0 in the AMR or IAMR, provide three protection configurations for process-scoped translation: (1) a mode that gives equivalent access to privileged and problem state processes, (2) a mode that gives access only to problem state, and (3) a mode that gives access only to privileged processes. (Note that privileged includes hypervisor privileged.) For partition-scoped translation, including translation of table entry addresses, either value of EAA_0 permits the access. See Figure 35 and Figure 45 for details. The choice of whether to limit access to problem state for process-scoped protection of privileged read and write is determined by key 0 of the AMR. When bit 0 is 0, the privileged bit in the PTE is ignored for a privileged store. When bit 0 is 1, the privileged bit must be 1 for a privileged store. Similarly when bit 1 is 0, the privileged bit in the PTE is ignored for a privileged load. When bit 1 is 1, the privileged bit must be 1 for a privileged load. The choice of whether to limit access to problem state for process-scoped protection of execute is determined by key 0 of the IAMR. When bit 1 is 0, the privileged bit in the PTE is ignored for an attempt to execute the instruction in privileged state. When bit 1 is 1, the privileged bit must be 1 to execute the instruction in priivleged state.

Privilege (EAA[0])	Read (EAA[1])	Read/Write (EAA[2])	Execute (EAA[3])	Access Authority problem state (MSR_{PR}=1)	Access Authority privileged state (MSR_{PR}=0)
0	0	0	0	na	na
0	0	0	1	e	e*
0	0	1	0	rw	rw*
0	0	1	1	rwe	rwe*
0	1	0	0	r	r*
0	1	0	1	re	re*
0	1	1	0	rw	rw*
0	1	1	1	rwe	rwe*
1	0	0	0	na	na
1	0	0	1	na	e
1	0	1	0	na	rw
1	0	1	1	na	rwe
1	1	0	0	na	r
1	1	0	1	na	re
1	1	1	0	na	rw
1	1	1	1	na	rwe

```
Key:
na: no access
r : read
w : write
e : execute
* : For partition-scoped translation, including all translation of table entry
    addresses, all accesses in the entry are permitted.
    For process-scoped translation, each access in the entry is permitted if and only
    if the relevant bit of key 0 of the [I]AMR is 0.
```

Figure 45. Encoded Access Authority (aka page protection)

5.8 Storage Control Attributes

This section describes aspects of the storage control attributes that are relevant only to privileged software programmers. The rest of the description of storage control attributes may be found in Section 1.6 of Book II and subsections.

5.8.1 Guarded Storage

Storage is said to be "well-behaved" if the corresponding real storage exists and is not defective, and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. Data and instructions can be fetched out-of-order from well-behaved storage without causing undesired side effects.

Storage is said to be Guarded if any of the following conditions is satisfied.

- MSR bit IR or DR is 1 for instruction fetches or data accesses respectively, or MSR_{HV}=0, and either G=1 or Att=0b010 in the relevant Page Table Entry.
- MSR bit IR or DR is 0 for instruction fetches or data accesses respectively, MSR_{HV}=1, and the storage is outside the range(s) specified by the Hypervisor Real Mode Storage Control facility (see Section 5.7.3.2.1).

In general, storage that is not well-behaved should be Guarded. Because such storage may represent a control register on an I/O device or may include locations that do not exist, an out-of-order access to such storage may cause an I/O device to perform unintended operations or may result in a Machine Check.

The following rules apply to in-order execution of *Load* and *Store* instructions for which the first byte of the storage operand is in storage that is both Caching Inhibited and Guarded.

- *Load* or *Store* instruction that causes an atomic access

  If any portion of the storage operand has been accessed and an External, Decrementer, Hypervisor Decrementer, Performance Monitor, or Imprecise mode Floating-Point Enabled exception is pending, the instruction completes before the interrupt occurs.
- *Load* or *Store* instruction that causes an Alignment exception, or that causes a [Hypervisor] Data Storage exception for reasons other than Data Address Watchpoint match.

The portion of the storage operand that is in Caching Inhibited and Guarded storage is not accessed.

(The corresponding rules for instructions that cause a Data Address Watchpoint match are given in Section 8.4.)

5.8.1.1 Out-of-Order Accesses to Guarded Storage

In general, Guarded storage is not accessed out-of-order. The only exceptions to this rule are the following.

Load Instruction

If a copy of any byte of the storage operand is in a cache then that byte may be accessed in the cache or in main storage.

Instruction Fetch

If $MSR_{HV\ IR}$=0b10 then an instruction may be fetched if any of the following conditions are met.

1. The instruction is in a cache. In this case it may be fetched from the cache or from main storage.
2. The instruction is in a real page from which an instruction has previously been fetched, except that if that previous fetch was based on condition 1 then the previously fetched instruction must have been in the instruction cache.
3. The instruction is in the same real page as an instruction that is required by the sequential execution model, or is in the real page immediately following such a page.

Programming Note

Software should ensure that only well-behaved storage is copied into a cache, either by accessing as Caching Inhibited (and Guarded) all storage that may not be well-behaved, or by accessing such storage as not Caching Inhibited (but Guarded) and referring only to cache blocks that are well-behaved.

If a real page contains instructions that will be executed when MSR_{IR}=0 and MSR_{HV}=1, software should ensure that this real page and the next real page contain only well-behaved storage (or that the Hypervisor Real Mode Storage Control facility specifies that this real page is not Guarded).

5.8.2 Storage Control Bits

When the thread is not in hypervisor real addressing mode, each storage access is performed under the control of the Page Table Entry used to translate the effective address. Each Page Table Entry contains storage control bits that specify the presence or absence of the corresponding storage control for all accesses translated by the entry as shown in Figure 46 and Figure 47. In the following description, references to individual WIMG bits apply to the corresponding Radix Att encoding, or to the result of combining the process-scoped and partition-scoped ATT encodings (see Section 5.7.10.3), except where otherwise stated or obvious from context.

Bit	Storage Control Attribute
W[1,3]	0 - not Write Through Required 1 - Write Through Required
I[3]	0 - not Caching Inhibited 1 - Caching Inhibited
M[2]	0 - not Memory Coherence Required 1 - Memory Coherence Required
G	0 - not Guarded 1 - Guarded

1 Support for the 1 value of the W bit is optional. Implementations that do not support the 1 value treat the bit as reserved and assume its value to be 0.

2 Support for the 0 value of the M bit is optional, implementations that do not support the 0 value assume the value of the bit to be 1, and may either preserve the value of the bit or write it as 1.

3 The combination WIMG = 0b1110 has behavior unrelated to the meanings of the individual bits. See see Section 5.8.2.1, “Storage Control Bit Restrictions” for additional information.

Figure 46. Storage control bits, HPT PTE

Att value	Storage Type
00	normal memory (WIMG=0010)
01[1]	SAO (WIMG=1110)
10	non-idempotent I/O (WIMG=0111)
11	tolerant I/O (WIMG=0110)

W=0 always for Radix Tree translation
M=1 always for Radix Tree translation

1 Behaves like WIMG=0010 but with strong access order.

Figure 47. Storage control bits, Radix PTE

When the thread is not in hypervisor real addressing mode, instructions are not fetched from storage for which the G bit in the Page Table Entry is set to 1; see Section 5.7.13.

When the thread is in hypervisor real addressing mode, the storage control attributes are implicit; see Section 5.7.3.2.

In Sections 5.8.2.1 and 5.8.2.2, "access" includes accesses that are performed out-of-order, and references to W, I, M, and G bits include the values of those bits that are implied when the thread is in hypervisor real addressing mode.

Programming Note

In a system consisting of only a single-threaded processor which has caches, correct coherent execution does not require storage to be accessed as Memory Coherence Required, and accessing storage as not Memory Coherence Required may give better performance.

5.8.2.1 Storage Control Bit Restrictions

All combinations of W, I, M, and G values are permitted except those for which both W and I are 1 and M||G ≠ 0b10.

The combination WIMG = 0b1110 is used to identify the Strong Access Ordering (SAO) storage attribute (see Section 1.7.1, "Storage Access Ordering", in Book II). Because this attribute is not intended for general purpose programming, it is provided only for a single combination of the attributes normally identified using the WIMG bits. That combination would normally be indicated by WIMG = 0b0010.

References to Caching Inhibited storage (or storage with I=1) elsewhere in the Power ISA have no application to SAO storage or its WIMG encoding, despite the encoding using I=1. Conversely, references to storage that is not Caching Inhibited (or storage with I=0) apply to SAO storage or its WIMG encoding. References to Write Through Required storage (or storage with W=1) elsewhere in the Power ISA have no application to SAO storage or its WIMG encoding, despite the fact that the encoding uses W=1. Conversely, references to storage that is not Write Through Required (or storage with W=0) apply to SAO storage or its WIMG encoding.

If a given real page is accessed concurrently as SAO storage and as non-SAO storage, the result may be characteristic of the weakly consistent model.

Programming Note

If an application program requests both the Write Through Required and the Caching Inhibited attributes for a given storage location, the operating system should set the I bit to 1 and the W bit to 0. The operating system should provide a means by which application programs can request SAO storage, in order to avoid confusion with the preceding guideline (since SAO is encoded using WI=0b11).

At any given time, the value of the W bit must be the same for all accesses to a given real page.

At any given time, the value of the I bit must be the same for all accesses to a given real page.

5.8.2.2 Altering the Storage Control Bits

When changing the value of the W bit for a given real page from 0 to 1, software must ensure that no thread modifies any location in the page until after all copies of locations in the page that are considered to be modified in the data caches have been copied to main storage using ***dcbst*** or ***dcbf***[***l***].

When changing the value of the I bit for a given real page from 0 to 1, software must set the I bit to 1 and then flush all copies of locations in the page from the caches using ***dcbf***[***l***] and ***icbi*** before permitting any other accesses to the page. Note that similar cache management is required before using the Fixed-Point Load and Store Caching Inhibited instructions to access storage that has formerly been cached. (See Section 4.4.1 on page 965.)

Programming Note

The storage control bit alterations described above are examples of cases in which the directives for application of statements about the W and I bits to SAO given in the third paragraph of the preceding subsection must be applied. A transition from the typical WIMG=0b0010 for ordinary storage to WIMG=0b1110 for SAO storage does not require the flush described above because both WIMG combinations indicate storage that is not Caching Inhibited.

Programming Note

It is recommended that ***dcbf*** be used, rather than ***dcbfl***, when changing the value of the I or W bit from 0 to 1. (***dcbfl*** would have to be executed on all threads for which the contents of the data cache may be inconsistent with the new value of the bit, whereas, if the M bit for the page is 1, ***dcbf*** need be executed on only one thread in the system.)

When changing the value of the M bit for a given real page, software must ensure that all data caches are consistent with main storage. The actions required to do this are system-dependent.

Programming Note

For example, when changing the M bit in some directory-based systems, software may be required to execute ***dcbf***[***l***] on each thread to flush all storage locations accessed with the old M value before permitting the locations to be accessed with the new M value.

Additional requirements for changing the storage control bits in the Page Table are given in Section 5.10.

5.9 Storage Control Instructions

5.9.1 Cache Management Instructions

This section describes aspects of cache management that are relevant only to privileged software programmers.

For a ***dcbz*** instruction that causes the target block to be newly established in the data cache without being fetched from main storage, the hardware need not verify that the associated real address is valid. The existence of a data cache block that is associated with an invalid real address (see Section 5.6) can cause a delayed Machine Check interrupt or a delayed Checkstop.

Each implementation provides an efficient means by which software can ensure that all blocks that are considered to be modified in the data cache have been copied to main storage before the thread enters any power conserving mode in which data cache contents are not maintained.

5.9.2 Synchronize Instruction

The *Synchronize* instruction is described in Section 4.6.3 of Book II, but only at the level required by an application programmer. This section describes properties of the instruction that are relevant only to operating system and hypervisor software programmers.

The *Synchronize* instruction provides an ordering function for stores that are in set A of the memory barrier created by the *Synchronize* instruction, relative to data accesses caused by instructions that are executed on other threads after the occurrence of the interrupt that is caused by a ***msgsndp*** or ***msgsnd*** instruction that follows the *Synchronize* instruction. The thread that is the target of the ***msgsndp*** or ***msgsnd*** instruction is here called the "target thread".

- For ***msgsndp***, and L = 0, 1, or 2 for the *Synchronize* instruction, the stores are performed with respect to the target thread before any data accesses caused by instructions that are executed on the target thread after the corresponding Directed Privileged Doorbell interrupt has occurred.
- For ***msgsnd***, and L = 0 or 2 for the *Synchronize* instruction (***sync*** or ***ptesync***), the stores are performed with respect to any given other thread before any data accesses caused by instructions that are executed on the given thread after a ***msgsync*** instruction is executed on that thread after the corresponding Directed Hypervisor Doorbell interrupt has occurred on the target thread.

Programming Note

Synchronize with L=1 (l***wsync***) should not be used with ***msgsnd***. (If used, it will not have the desired ordering effect.)

Programming Note

The ***msgsync*** instruction, which is needed when ***msgsnd*** is used, is not needed when ***msgsndp*** is used because ***msgsndp*** targets only threads on the same multi-threaded processor as the thread executing the ***msgsndp***, while ***msgsnd*** can target any thread in the system. (If the target thread for ***msgsnd*** is on the same multi-threaded processor as the thread executing the ***msgsnd***, in principle the ***msgsync*** can be omitted. This optimization is practical only when the ***msgsnd*** topology is appropriately constrained, however, because the Directed Hypervisor Doorbell interrupt provides no indication of which thread executed the ***msgsnd*** that caused the interrupt, so there is no easy way for the interrupt handler to determine whether the ***msgsync*** can be omitted.) ***msgsync*** is not needed or defined in V. 2.07 for a similar reason: ***msgsnd*** in V. 2.07 can target only threads on the same multi-threaded processor as the thread executing the ***msgsnd***.

The ordering done by ***sync*** (and ***ptesync***) provides the appearance of "causality" across a sequence of ***msgsnd*** instructions, as in the following example. "***msgsnd***->T1" means "***msgsnd*** instruction targetting thread T1". "<DHDI 0>" means "occurrence of Directed Hypervisor Doorbell interrupt caused by ***msgsnd*** executed on T0". On T0, register r1 is assumed to contain the value 1.

```
T0              T1              T2
std  r1,X       <DHDI 0>        <DHDI 1>
sync            msgsnd->T2      msgsync
msgsnd->T1                      ld   r1,X
```

In this example, T2's load from X must return 1.

Another variant of the *Synchronize* instruction is described below. It is designated the Page Table Entry Synchronize instruction, and is specified by the

extended mnemonic ***ptesync*** (equivalent to ***sync*** with L=2).

The ***ptesync*** instruction has all of the properties of ***sync*** with L=0 and also the following additional properties.

- The memory barrier created by the ***ptesync*** instruction provides an ordering function for the storage accesses associated with all instructions that are executed by the thread executing the ***ptesync*** instruction and, as elements of set A, for all Reference and Change bit updates associated with additional address translations that were performed, by the thread executing the ***ptesync*** instruction, before the ***ptesync*** instruction is executed. The applicable pairs are all pairs a_i,b_j in which b_j is a data access and a_i is not an instruction fetch.
- The ***ptesync*** instruction causes all Reference and Change bit updates associated with address translations that were performed, by the thread executing the ***ptesync*** instruction, before the ***ptesync*** instruction is executed, to be performed with respect to that thread before the ***ptesync*** instruction's memory barrier is created.
- The memory barrier created by the ***ptesync*** instruction provides an ordering function for all stores to the Partition Table, Process Tables, Segment Tables, Page Directories, and Page Tables caused by *Store* instructions preceding the ***ptesync*** instruction with respect to invalidations, of cached copies of information derived from these tables, caused by ***slbieg***, ***slbiag***, and ***tlbie*** instuctions following the ***ptesync*** instruction. The memory barrier ensures that all searches of these tables by another thread, that are performed after an invalidation caused by such an ***slbieg***, ***slbiag***, or ***tlbie*** instruction has been performed with respect to the other thread and that implicitly load from the target location of such a store, will obtain the value stored (or a value stored subsequently).

  Programming Note

  The next bullet is sufficient to order the stores with respect to the invalidations on the thread executing the ***ptesync*** instruction. That bullet is also sufficient to provide the ordering with respect to invalidations caused by ***slbie***, ***slbia***, and ***tlbiel*** instuctions, which affect only the thread executing them.

- The ***ptesync*** instruction provides an ordering function for all stores to the Partition Table, Process Tables, Segment Tables, Page Directories, and Page Tables caused by *Store* instructions preceding the ***ptesync*** instruction with respect to searches of these tables that are performed, by the thread executing the ***ptesync*** instruction, after the ***ptesync*** instruction completes. Executing a ***ptesync*** instruction ensures that all such searches that implicitly load from the target location of such a store willl obtain the value stored (or a value stored subsequently). Also, the memory barrier created by the ***ptesync*** insruction ensures that all searches of these tables by any other thread, that are performed after a store in set B of the memory barrier has been performed with respect to the other thread and that implicitly load from the target location of such a store, will obtain the value stored (or a value stored subsequently).
- In conjunction with the ***tlbie*** and ***tlbsync*** instructions, the ***ptesync*** instruction provides an ordering function for TLB invalidations and related storage accesses on other threads as described in the ***tlbsync*** instruction description on page 1042.

  Similarly, in conjunction with the ***slbieg*** or ***slbiag*** and ***slbsync*** instructions, the ***ptesync*** instruction provides an ordering function for SLB invalidations and related storage accesses on other threads as described in the ***slbsync*** instruction description on page 1032.

Programming Note

For instructions following a ***ptesync*** instruction, the memory barrier need not order implicit storage accesses for purposes of address translation and reference and change recording.

The functions performed by the ***ptesync*** instruction may take a significant amount of time to complete, so this form of the instruction should be used only if the functions listed above are needed. Otherwise ***sync*** with L=0 should be used (or ***sync*** with L=1, or ***eieio***, if appropriate).

Section 5.10, "Translation Table Update Synchronization Requirements" on page 1043 gives examples of uses of ***ptesync***.

5.9.3 Lookaside Buffer Management

All implementations have a Segment Lookaside Buffer (SLB). Independent of whether the executing partition operates in a mode that uses hardware SLB loading and bolting versus pure software loading (controlled by the value of $LPCR_{UPRT}$), software is responsible for keeping the SLB current with the segment mapping for the process that is executing. Proper management of the SLB across context switches is described in programming notes.

For performance reasons, most implementations also cache other information that is used in address translation. These caches may include: a Translation Loo-

kaside Buffer (TLB) which is a cache of recently used Page Table Entries (PTEs); a cache of recently used translations of effective addresses to real addresses; a Page Walk Cache for Radix Tree translation; caching of the In-Memory Tables; or any combination of these. Lookaside information, including the SLB, is managed using the instructions described in the subsections of this section unless additional requirements are provided in implementation-specific documentation.

To simplify lookaside buffer management, hardware will only perform speculative translation for the context that is executing, in particular using the current effective values of LPID and PID. Except when LPIDR=0, no translations will be created and cached speculatively when HR=0 and MSR_{HV}=1. Furthermore, no translations will be created and cached speculatively in hypervisor real addressing mode. The limitation of speculative behavior in these situations is to cache a PATE when LPIDR is loaded and a PRTE when PIDR is loaded.

Lookaside information derived from PTEs is not necessarily kept consistent with the Page Table. When software alters the contents of a PTE, in general it must also invalidate all corresponding TLB entries and implementation-specific lookaside information; exceptions to this rule are described in Section 5.10.1.2.

The effects of the ***slbie***, ***slbieg***, ***slbia***, ***slbiag***, and *TLB Management* instructions on address translations, as specified in Sections 5.9.3.2 for the SLB and 5.9.3.3 for the TLB, Page Walk Cache, and In-Memory Table caches, apply to all implementation-specific lookaside information that is used in address translation. Unless otherwise stated or obvious from context, references to SLB entry invalidation and TLB entry invalidation elsewhere in the Books apply also to invalidation of Page Walk Cache content, In-Memory Table cache content, and all implementation-specific lookaside information that is derived from SLB entries and PTEs, respectively.

All implementations provide a means by which software can invalidate all implementation-specific lookaside information that is derived from PTEs.

Implementation-specific lookaside information that contains translations of effective addresses to real addresses may include “translations” that apply in real addressing mode. Because such “translations” are affected by the contents of the LPCR and HRMOR, when software alters the (relevant) contents of these registers it must also invalidate the corresponding implementation-specific lookaside information. Software can invalidate all such lookaside information by using the ***slbia*** instruction with IH=0b000. However, performance is likely to be better if other, appropriate, IH values are used to limit the amount of lookaside information that is invalidated.

All implementations that have such lookaside information provide a means by which software can invalidate all such lookaside information.

For simplicity, elsewhere in the Books it is assumed that the TLB exists.

Programming Note

Because the instructions used to manage TLBs, SLBs, Page Walk Caches, caches of Partition and Process Table Entries, and implementation-specific lookaside information may be changed in a future version of the architecture, it is recommended that software “encapsulate” their use into subroutines.

Programming Note

The function of all the instructions described in Sections 5.9.3.2 - 5.9.3.3 is independent of whether address translation is enabled or disabled.

For a discussion of software synchronization requirements when invalidating SLB and TLB entries, see Chapter 11.

5.9.3.1 Thread-Specific Segment Translations

It is necessary to provide thread-specific temporary ESID to VSID translations. These translations cannot be placed in valid entries in the Segment Table because the Segment Table has a process scope rather than a thread scope. Instead, software will use ***slbmte*** to install such translations in the SLB. All SLB entries created using ***slbmte*** are considered to be “software created.” Software created entries will only translate accesses from the hardware thread by which they are installed. When $LPCR_{UPRT}$=1, they are also considered to be “bolted.” Each thread has the ability to bolt four entries.

5.9.3.2 SLB Management Instructions

Software establishes translations in the SLB using ***slbmte***. Care must be taken to avoid creating multiple effective-to-virtual translations for any given effective address. Software-created entries will remain in the SLB until invalidated using ***slbie*** or ***slbia*** (which also invalidate related implementation-specific lookaside information) or overwritten using ***slbmte***. After updating a Segment Table Entry, software must use an ***slbie*** or ***slbieg*** instruction to remove lookaside information associated with the old contents of the entry. ***slbie*** may be used to invalidate software-created entries, but will not invalidate outboard translation caches. ***slbieg*** does not invalidate software-created entries, but is the only way to invalidate outboard translation caches. When taking a PID out of service with the intent of reusing it, software should use ***slbiag*** to remove stale translations from SLBs and ERATs in the “nest.” (Nest

refers to the platform external to the processor cores. Here the reference is to translations cached for use by accelerators.) ***slbsync*** will establish order between ***slbieg*** and ***slbiag*** instructions and a subsequent ***ptesync***. ***ptesync*** must also be used to synchronize the Segment Table update prior to performing the lookaside management. When performing a context switch, software must use an ***slbia*** instruction to remove lookaside information associated with the old context. ***slbmfee*** and ***slbmfev*** may be used by the hypervisor to save software-created entries. ***slbmte*** is used to restore software-created entries. ***slbfee*** has no function when $LPCR_{UPRT}$=1 for the partition that is running.

Programming Note

Accesses to a given SLB entry caused by the instructions described in this section obey the sequential execution model with respect to the contents of the entry and with respect to data dependencies on those contents. That is, if an instruction sequence contains two or more of these instructions, when the sequence has completed, the final contents of the SLB entry and of General Purpose Registers is as if the instructions had been executed in program order.

However, software synchronization is required in order to ensure that any alterations of the entry take effect correctly with respect to address translation; see Chapter 11.

Programming Note

Changes to the segment mappings in the presence of active transactions may compromise transactional semantics if the transaction has accessed a segment that is assigned a new VSID. Consequently, when modifying segment mappings, it is the responsibility of the OS or hypervisor to ensure that any transaction that may have touched the modified segment is terminated, using a ***tabort.*** or ***treclaim.*** instruction.

SLB Invalidate Entry *X-form*

slbie RB

31	///	///	RB	434	/
0	6	11	16	21	31

```
ea0:35 ← (RB)0:35
if, for SLB entry that translates
  or most recently translated ea,

    entry_seg_size = size specified in (RB)37:38
then for SLB entry (if any) that translates ea
  SLBEV ← 0
  all other fields of SLBE ← undefined
else
   s ← log_base_2(entry_seg_size)
   esid ← (RB)0:63-s
   u ← undefined 1-bit value
   if u then
     if an SLB entry translates esid
        SLBEV ← 0
        all other fields of SLBE ← undefined
```

Let the Effective Address (EA) be any EA for which $EA_{0:35} = (RB)_{0:35}$. Let the segment size be equal to the segment size specified in $(RB)_{37:38}$; the allowed values of $(RB)_{37:38}$, and the correspondence between the values and the segment size, are the same as for the B field in the SLBE (see Figure 27 on page 994).

The segment size must be the same as the segment size in the SLB entry that translates the EA, or the values that were in the SLB entry that most recently translated the EA if the translation is no longer in the SLB; if these values are not the same, it is implementation-dependent whether the SLB entry (or implementation-dependent translation information) that translates the EA is invalidated, and the next paragraph need not apply.

If the SLB contains only a single entry that translates the EA, then that is the only SLB entry that is invalidated, except that it is implementation-dependent whether an implementation-specific lookaside entry for a real mode address “translation” is invalidated. If the SLB contains more than one such entry, then zero or more such entries are invalidated, and similarly for any implementation-specific lookaside information used in address translation; additionally, a machine check may occur.

SLB entries are invalidated by setting the V bit in the entry to 0, and the remaining fields of the entry are set to undefined values.

This instruction terminates any Segment Table walks being performed on behalf of the thread that executes it.

The hardware ignores the contents of RB listed below and software must set them to 0s.

- $(RB)_{37}$

- $(RB)_{39}$
- $(RB)_{40:63}$
- If s = 40, $(RB)_{24:35}$

If this instruction is executed in 32-bit mode, $(RB)_{0:31}$ must be zeros.

This instruction is privileged.

Special Registers Altered:

None

Programming Note

slbie does not affect SLBs on other threads.

Programming Note

The B value in register RB may be needed for invalidating ERAT entries corresponding to the translation being invalidated.

Programming Note

When switching to execute an adjunct, a hypervisor will disable translation and use ***slbie*** to be sure there is no SLB entry mapping the effective address space that will be used by the incoming adjunct. It will then bolt an entry for the incoming adjunct and transfer control to that adjunct. While the thread is in hypervisor real addressing mode and during adjunct execution, no speculative Segment Table walks will be performed.

SLB Invalidate Entry Global *X-form*

slbieg RS,RB

31	RS	///	RB	466	/
0	6	11	16	21	31

```
target_PID = RS0:31
if MSRHV=1 then target_LPID = RS32:63
else target_LPID = LPIDR
ea0:35 ← (RB)0:35
for each thread with LPIDR=target_LPID and
                     PIDR=target_PID
  if, for each SLB entry that
  translates or most recently translated ea
    entry_seg_size = size specified in (RB)37:38
  then for SLB entry (if any)
  that translates ea and is not software-created
    SLBEV ← 0
    all other fields of SLBE ← undefined
  else
     s ← log_base_2(entry_seg_size)
     esid ← (RB)0:63-s
     u ← undefined 1-bit value
     if u then
       if an SLB entry translates esid and the entry
            is not software-created
          SLBEV ← 0
          all other fields of SLBE ← undefined
```

The operation performed by this instruction is based on the contents of registers RS and RB. The contents of these registers are shown below.

RS

PID	LPID
0	32 63

RB

ESID	C	B	0s
0	36	37	39 63

$RS_{0:31}$ PID
$RS_{32:63}$ LPID
$RB_{0:35}$ ESID
RB_{36} must be 0b0
$RB_{37:38}$ B
$RB_{39:63}$ must be 0b0 || 0x000000

Let the target PID be $RS_{0:31}$. If the instruction is executed in hypervisor state, let the target LPID be $RS_{32:63}$; otherwise let the target LPID be the contents of LPIDR. Let the Effective Address (EA) be any EA for which $EA_{0:35} = (RB)_{0:35}$. Let the segment size be equal to the segment size specified in $(RB)_{37:38}$; the allowed values of $(RB)_{37:38}$, and the correspondence between

the values and the segment size, are the same as for the B field in the SLBE (see Figure 27 on page 994).

Only SLBs for threads running on behalf of target_LPID and target_PID are searched. Software-created entries are ignored. The segment size must be the same as the segment size in the SLB entry that translates the EA, or the values that were in the SLB entry that most recently translated the EA if the translation is no longer in the SLB; if these values are not the same, it is implementation-dependent whether the SLB entry (or implementation-dependent translation information) that translates the EA is invalidated, and the next paragraph need not apply.

If the SLB contains only a single entry that translates the EA, then that is the only SLB entry that is invalidated, except that it is implementation-dependent whether an implementation-specific lookaside entry for a real mode address "translation" is invalidated. If the SLB contains more than one such entry, then zero or more such entries are invalidated, and similarly for any implementation-specific lookaside information used in address translation; additionally, a machine check may occur.

SLB entries are invalidated by setting the V bit in the entry to 0, and the remaining fields of the entry are set to undefined values.

The hardware ignores the contents of RB listed below and software must set them to 0s.

- $(RB)_{36}$
- $(RB)_{37}$
- $(RB)_{39}$
- $(RB)_{40:63}$
- If s = 40, $(RB)_{24:35}$

If this instruction is executed in 32-bit mode, $(RB)_{0:31}$ must be zeros.

The operation performed by this instruction is ordered by the ***eieio*** (or ***sync*** or ***ptesync***) instruction with respect to a subsequent ***slbsync*** instruction executed by the thread executing the ***slbieg*** instruction. The operations caused by ***slbieg*** and ***slbsync*** are ordered by ***eieio*** as a fifth set of operations, which is independent of the other four sets that ***eieio*** orders.

This instruction is privileged except when $LPCR_{G-TSE}$=0, making it hypervisor privileged.

Special Registers Altered:

None

Programming Note

slbieg does affect SLBs on other threads.

Programming Note

The B value in register RB may be needed for invalidating ERAT entries corresponding to the translation being invalidated.

Programming Note

Use of ***slbieg*** to invalidate software-created segment descriptors is a programming error. The architecture requires that bolted entries be ignored by the instruction.

SLB Invalidate All *X-form*

slbia IH

31	//	IH	///	///	498	/
0	6	8	11	16	21	31

```
switch (IH)
  case (0b000, 0b001, 0b010, 0b110):
    for each SLB entry except SLB entry 0
      SLBE_V ← 0
      all other fields of SLBE ← undefined
  case (0b011):
    for each SLB entry such that SLBE_Class = 1
      SLBE_V ← 0
      all other fields of SLBE ← undefined
  case (0b100):
    for each SLB entry
      SLBE_V ← 0
      all other fields of SLBE ← undefined
  case (0b111):
```

slbia invalidates the contents of the SLB, and of implementation-specific lookaside information for effective to real address translations, based on the contents of the IH field as described below. SLB entries are invalidated by setting the V bit in the entry to 0. When an SLB entry is invalidated, the remaining fields of the entry are set to undefined values.

In the description of the IH values, "implementation-specific lookaside information" is shorthand for "implementation-specific lookaside information for effective to real address translations," and "when address translation was enabled" is shorthand for "when MSR_{IR} was equal to 1 or MSR_{DR} was equal to 1, as appropriate for the type of access," and correspondingly for "when address translation was disabled." The descriptions specify which entries must be invalidated; additional entries may be invalidated except where the description states that certain SLB entries are not invalidated.

0b000 All SLB entries except entry 0 are invalidated; SLB entry 0 is not invalidated.
All implementation-specific lookaside information is invalidated.

0b001 All SLB entries except entry 0 are invalidated; SLB entry 0 is not invalidated.
All implementation-specific lookaside information that was created when address translation was enabled and satisfies either of the following conditions is invalidated.
- The information is for an SLB-derived translation and has a Class value of 1.
- The information is for a Radix Tree-derived translation for which effPID≠0.

0b010 All SLB entries except entry 0 are invalidated; SLB entry 0 is not invalidated.
All implementation-specific lookaside information that was created when address translation was enabled is invalidated.

0b011 All SLB entries having a Class value of 1 are invalidated; SLB entry 0 is not invalidated if it has a Class value of 0.
All implementation-specific lookaside information that was created when address translation was enabled and satisfies either of the following conditions is invalidated.
- The information is for an SLB-derived translation and has a Class value of 1.
- The information is for a Radix Tree-derived translation for which effPID≠0.

0b100 All SLB entries are invalidated.
All implementation-specific lookaside information is invalidated.

0b110 All SLB entries except entry 0 are invalidated; SLB entry 0 is not invalidated.
All implementation-specific lookaside information that satisfies any of the following conditions is invalidated.
- The information is for an SLB-derived or SLS translation.
- The information is for a Radix Tree-derived translation for which effLPID≠0 or effPID≠0.
- The information was created when address translation was disabled and $MSR_{HV\ PR}$ was equal to 0b00.

0b111 No SLB entries are invalidated.
All implementation-specific lookaside information is invalidated.

Programming Note

In the preceding description, "SLB-derived translation" excludes any SLS translation, since SLS translation does not use segmentation.

All other IH values are reserved. If the IH field contains a reserved value, the hint provided by the IH field is undefined.

When IH=0b000, execution of this instruction has the side effect of clearing the storage access history associated with the Hypervisor Real Mode Storage Control facility. See Section 5.7.3.2.1, "Hypervisor Real Mode Storage Control" for more details.

This instruction terminates any Segment Table walks being performed on behalf of the thread that executes it, and ensures that any new table walks will be performed using the current PIDR value.

This instruction is privileged.

Special Registers Altered:

None

Programming Note

When performing a context switch between processes, an HPT operating system will use ***mtPIDR*** followed by ***slbia***. The synchronization of the PID value and termination of outstanding Segment Table walks ensures that SLB will not contain multiple entries mapping the same EA range (i.e. from the former and new PIDs). Note that if this sequence is performed with translation enabled, care must be taken to avoid an implicit branch. (i.e. the same translation(s) for the locations containing the context switch routine must be valid for both processes.)

For the corresponding situation when changing partitions from or to a partition using HPT translation, hypervisor software should get all the affected threads into real mode, execute ***mtLPIDR***, and then perform the ***slbia*** on all the affected threads. (If the affected threads were not in real mode, avoiding implicit branches due to the ***mtLPIDR*** would be very difficult.)

Programming Note

slbia does not affect SLBs on other threads.

Programming Note

If ***slbia*** is executed when instruction address translation is enabled, software can ensure that attempting to fetch the instruction following the ***slbia*** does not cause an Instruction Segment interrupt by placing the ***slbia*** and the subsequent instruction in the effective segment mapped by SLB entry 0. (The preceding assumes that no other interrupts occur between executing the ***slbia*** and executing the subsequent instruction. It also assumes that IH values other than 0b011 and 0b100 are used.)

Programming Note

Examples of the intended use of the IH values follow.

0b000 This setting should be used by a bare metal operating system or hypervisor to make extensive translation changes with translation on or using $LPCR_{UPRT}$=0.

0b001 This setting should be used by an operating system that uses HPT translation and manages the Class bit but doesn't trust its use to manage the SLB (an extension to the longer-standing base function that could have compatibility implications) when switching tasks. Operating systems that use Radix Tree translation may also use this setting.

0b010 This setting should be used by an operating system that uses HPT translation and does not manage the Class bit when switching tasks.

0b011 This setting should be used by an operating system that uses HPT translation, manages the Class bit and trusts its use to manage the SLB when switching tasks. Operating systems that use Radix Tree translation may also use this setting.

0b100 This setting should be used by a bare metal operating system or hypervisor to make extensive translation changes with translation off or using $LPCR_{UPRT}$=1.

0b110 This setting should be used by the hypervisor when switching partitions.

0b111 This setting is provided mainly for use prior to product shipment, but may provide benefit in an environment that uses Radix Tree translation if SLB invalidation is much slower than ERAT invalidation.

Programming Note

slbia serves as both a basic and an extended mnemonic. The Assembler will recognize an ***slbia*** mnemonic with one operand as the basic form, and an ***slbia*** mnemonic with no operand as the extended form. In the extended form the IH operand is omitted and assumed to be 0.

SLB Invalidate All Global *X-form*

slbiag RS

31	RS	///	///	850	/
0	6	11	16	21	31

```
target_PID = RS0:31
if MSRHV=1 then target_LPID = RS32:63
else target_LPID = LPIDR
for each nest SLB
  for each SLBE with LPID=target_LPID and
                     PID=target_PID
    SLBEV ← 0
    all other fields of SLBE ← undefined
```

The operation performed by this instruction is based on the contents of register RS. The contents of this register is shown below.

RS

PID	LPID
0	32 63

$RS_{0:31}$ PID
$RS_{32:63}$ LPID

Let the target PID be $RS_{0:31}$. If the instruction is executed in hypervisor state, let the target LPID be $RS_{32:63}$; otherwise let the target LPID be the contents of LPIDR.

All nest SLBs are searched. Each SLBE for process PID in partition LPID is invalidated.

SLB entries are invalidated by setting the V bit in the entry to 0, and the remaining fields of the entry are set to undefined values.

Implementation specific lookaside information associated with the invalidated SLB entries is invalidated. Additional implementation specific lookaside information may be invalidated.

The operation performed by this instruction is ordered by the ***eieio*** (or ***sync*** or ***ptesync***) instruction with respect to a subsequent ***slbsync*** instruction executed by the thread executing the ***slbiag*** instruction. The operations caused by ***slbiag*** and ***slbsync*** are ordered by ***eieio*** as a fifth set of operations, which is independent of the other four sets that ***eieio*** orders.

This instruction is privileged except when $LPCR_{G\text{-}TSE}$=0, making it hypervisor privileged.

Special Registers Altered:

None

Programming Note

slbiag does not affect SLBs on processor threads.

"g" (Global) in the name of the instruction reflects the fact that a future version of the architecture may extend the definition of ***slbiag*** to allow programs to specify additional sets of SLBs that the instruction affects, possibly including SLBs on processor threads.

SLB Move To Entry *X-form*

slbmte RS,RB

31	RS	///	RB	402	/
0	6	11	16	21	31

When $LPCR_{UPRT}$=0, this instruction is the sole means for specifying Segment translations to the hardware. When $LPCR_{UPRT}$=1, Segment Table walks populate the SLB, and this instruction is used only to bolt thread-specific Segment translations.

The SLB entry specified by bits 52:63 of register RB is loaded from register RS and from the remainder of register RB. The contents of these registers are interpreted as shown in Figure 48.

RS

B	VSID	K_sK_pNLC	0	LP	0s
0	2	52	57	58	60 63

RB

ESID	V	0s	index
0	36	37 52	63

$RS_{0:1}$ B
$RS_{2:51}$ VSID
RS_{52} K_s
RS_{53} K_p
RS_{54} N
RS_{55} L
RS_{56} C
RS_{57} must be 0b0
$RS_{58:59}$ LP
$RS_{60:63}$ must be 0b0000
$RB_{0:35}$ ESID
RB_{36} V
$RB_{37:51}$ must be 0b000 || 0x000
$RB_{52:63}$ index, which selects the SLB entry

Figure 48. GPR contents for slbmte

On implementations that support a virtual address size of only n bits, n<78, $(RS)_{2:79-n}$ must be zeros.

When $LPCR_{UPRT}$=1, the value of index must not exceed 3. $(RB)_{52:61}$ are ignored.

High-order bits of $(RB)_{52:63}$ that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

The hardware ignores the contents of RS and RB listed below and software must set them to 0s.

- $(RS)_{57}$
- $(RS)_{60:63}$
- $(RB)_{37:51}$

If this instruction is executed in 32-bit mode, $(RB)_{0:31}$ must be zeros (i.e., the ESID must be in the range 0:15).

This instruction must not be used to load a segment descriptor that is in the Segment Table when $LPCR_{UPRT}$=1, and cannot be used to invalidate the translation contained in an SLB entry.

This instruction is privileged.

Special Registers Altered:

None

Programming Note

The reason ***slbmte*** must not be used to load segment descriptors that are in the Segment Table is that there could be a race condition with hardware loading the same segment descriptor, resulting in duplicate SLB entries. Software must not allow duplicate SLB entries to be created; see Section 5.7.8.2, “SLB Search”.

The reason ***slbmte*** cannot be used to invalidate an SLB entry is that it does not necessarily affect implementation-specific address translation lookaside information. ***slbie*** (or ***slbia***) must be used for this purpose.

SLB Move From Entry VSID *X-form*

slbmfev RT,RB

31	RT	///	L	RB	851	/
0	6	11	15	16	21	31

This instruction is used to read software-loaded SLB entries. When $LPCR_{UPRT}$=0, the entry is specified by bits 52:63 of register RB. When $LPCR_{UPRT}$=1, only the first four entries can be read, so bits 52:61 of register RB are ignored. If the specified entry is valid (V=1), the contents of the B, VSID, K_s, K_p, N, L, C, and LP fields of the entry are placed into register RT. The contents of these registers are interpreted as shown in Figure 49.

RT

B	VSID	K_sK_pNLC	0	LP	0s
0	2	52	57	58	60 63

RB

0s	index
0	52 63

$RT_{0:1}$	B
$RT_{2:51}$	VSID
RT_{52}	K_s
RT_{53}	K_p
RT_{54}	N
RT_{55}	L
RT_{56}	C
RT_{57}	set to 0b0
$RT_{58:59}$	LP
$RT_{60:63}$	set to 0b0000

$RB_{0:51}$	must be 0x0_0000_0000_0000
$RB_{52:63}$	index, which selects the SLB entry

Figure 49. GPR contents for slbmfev

On implementations that support a virtual address size of only n bits, n<78, $RT_{2:79-n}$ are set to zeros.

If the SLB entry specified by bits 52:63 of register RB is invalid (V=0), the contents of register RT are set to 0.

High-order bits of $(RB)_{52:63}$ that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

The hardware ignores the contents of $RB_{0:51}$.

This instruction is privileged.

The use of the L field is implementation specific.

Special Registers Altered:

None

SLB Move From Entry ESID *X-form*

slbmfee RT,RB

31	RT	///	L	RB	915	/
0	6	11	15	16	21	31

This instruction is used to read software-loaded SLB entries. When $LPCR_{UPRT}$=0, the entry is specified by bits 52:63 of register RB. When $LPCR_{UPRT}$=1, only the first four entries can be read, so bits 52:61 of register RB are ignored. If the specified entry is valid (V=1), the contents of the ESID and V fields of the entry are placed into register RT. If $LPCR_{UPRT}$=1, the value of the BO field of the entry is also placed into register RT. The contents of these registers are interpreted as shown in Figure 50.

RT

ESID	V	BO	0s
0	36	37 38	63

RB

0s	index
0	52 63

$RT_{0:35}$ ESID
RT_{36} V
RT_{37} BO, entry is bolted
$RT_{38:63}$ set to 0b000 || 0x00_0000
$RB_{0:51}$ must be 0x0_0000_0000_0000
$RB_{52:63}$ index, which selects the SLB entry

Figure 50. GPR contents for slbmfee

If the SLB entry specified by bits 52:63 of register RB is invalid (V=0), the contents of register RT are set to 0.

High-order bits of $(RB)_{52:63}$ that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

The hardware ignores the contents of $RB_{0:51}$.

This instruction is privileged.

The use of the L field is implementation specific.

Special Registers Altered:

None

SLB Find Entry ESID *X-form*

slbfee. RT,RB

31	RT	///	RB	979	1
0	6	11	16	21	31

The SLB is searched for an entry that matches the effective address specified by register RB. When $LPCR_{UPRT}$=1, this instruction is nonfunctional. The search is performed as if it were being performed for purposes of address translation. That is, in order for a given entry to satisfy the search, the entry must be valid (V=1), and $(RB)_{0:63-s}$ must equal $SLBE[ESID_{0:63-s}]$ (where 2^s is the segment size selected by the B field in the entry).If exactly one matching entry is found, the contents of the B, VSID, K_s, K_p, N, L, C, and LP fields of the entry are placed into register RT. If no matching entry is found, register RT is set to 0. If more than one matching entry is found, either one of the matching entries is used, as if it were the only matching entry, or a Machine Check occurs. If a Machine Check occurs, register RT, and CR Field 0 are set to undefined values, and the description below of how this register and this field is set does not apply.

The contents of registers RT and RB are interpreted as shown in Figure 51.

RT

B	VSID	K_sK_pNLC	0	LP	0s
0 2		52	57	58 60	63

RB

ESID	0000	0s
0	36 40	63

$RT_{0:1}$ B
$RT_{2:51}$ VSID
RT_{52} K_s
RT_{53} K_p
RT_{54} N
RT_{55} L
RT_{56} C
RT_{57} set to 0b0
$RT_{58:59}$ LP
$RT_{60:63}$ set to 0b0000
$RB_{0:35}$ ESID
$RB_{36:39}$ must be 0b0000
$RB_{40:63}$ must be 0x000000

Figure 51. GPR contents for slbfee.

If s > 28, $RT_{80-s:51}$ are set to zeros. On implementations that support a virtual address size of only n bits, n < 78, $RT_{2:79-n}$ are set to zeros.

CR Field 0 is set as follows. j is a 1-bit value that is equal to 0b1 if a matching entry was found. Otherwise, j is 0b0. When $LPCR_{UPRT}\neq0$, j=0b0.

$CR0_{LT\ GT\ EQ\ SO}$ = 0b00 || j || XER_{SO}

The hardware ignores the contents of $RB_{36:38\ 40:63}$.

If this instruction is executed in 32-bit mode, $(RB)_{0:31}$ must be zeros (i.e., the ESID must be in the range 0-15).

This instruction is privileged.

Special Registers Altered:

CR0

SLB Synchronize *X-form*

slbsync

31	///	///	///	338	/
0	6	11	16	21	31

The ***slbsync*** instruction provides an ordering function for the effects of all ***slbieg*** and ***slbiag*** instructions executed by the thread executing the ***slbsync*** instruction, with respect to the memory barrier created by a subsequent ***ptesync*** instruction executed by the same thread. Executing a ***slbsync*** instruction ensures that all of the following will occur.

- All SLB invalidations caused by ***slbieg*** and ***slbiag*** instructions preceding the ***slbsync*** instruction will have completed on any other thread before any data accesses caused by instructions following the ***ptesync*** instruction are performed with respect to that thread.
- All storage accesses by other threads for which the address was translated using the translations being invalidated will have been performed with respect to the thread executing the ***ptesync*** instruction, to the extent required by the associated Memory Coherence Required attributes, before the ***ptesync*** instruction's memory barrier is created.

The operation performed by this instruction is ordered by the ***eieio*** (or ***sync*** or ***ptesync***) instruction with respect to preceding ***slbieg*** and ***slbiag*** instructions executed by the thread executing the ***slbsync*** instruction. The operations caused by ***slbieg*** or ***slbiag*** and ***slbsync*** are ordered by ***eieio*** as a fifth set of operations, which is independent of the other four sets that ***eieio*** orders.

The ***slbsync*** instruction may complete before operations caused by ***slbieg*** or ***slbiag*** instructions preceding the ***slbsync*** instruction have been performed.

This instruction is privileged except when $LPCR_{G\text{-}TSE}$=0, making it hypervisor privileged.

See Section 5.10 for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Programming Note

slbsync should not be used to synchronize the completion of ***slbie***.

5.9.3.3 TLB Management Instructions

In addition to managing the TLB, ***tlbie*** and ***tlbiel*** are also used to manage the Page Walk Cache, In-Memory Table caching, and implementation-specific lookaside information that depends on the values of the PTEs. The parameters described below specify the type of translations to invalidate and the scope of the invalidation to be performed.

Radix Invalidation Control (RIC) specifies whether to invalidate the TLB, the Page Walk Cache, or both together with partition and Process Table caching. The RIC values and functions are as follows.

0 Just invalidate TLB.

1 Invalidate just Page Walk Cache.

2 Invalidate TLB, Page Walk Cache, and any caching of Partition and Process Table Entries.

3 Invalidate a series of consecutive translations (just in the TLB).

Process Scoped (PRS) specifies whether the translation(s) to be invalidated are partition scoped or process scoped including, for RIC=2, whether process or Partition Table caching is being invalidated.

0 Invalidate partition-scoped translation(s).

1 Invalidate process-scoped translations.

Radix (R) specifies whether the translations to be invalidated are Radix Tree translations or HPT translations. If the R value is incorrect for the target partition, the results of the operation are boundedly undefined. (R is ignored for invalidates with IS=3 and MSR_{HV}=1 because they have the potential to target translations for multiple partitions.)

0 Invalidate HPT translation(s).

1 Invalidate Radix Tree translations.

Invalidation Selector (IS) (found in RB) specifies the scope of the context to be invalidated.

0 Invalidate just the target VA.

1 Invalidate matching PID.

2 Invalidate matching LPID.

3 If MSR_{HV}=1, invalidate all entries, otherwise invalidate matching LPID.

The IS≠0 RIC=2 variants of ***tlbie*** and ***tlbiel*** perform the same TLB invalidations as the corresponding RIC=0 variants, but in addition invalidate Page Walk Cache Entries and partition or Process Table caching associated with the specified LPID or LPID/PID. When RIC=1 and IS≠0, the Page Walk Cache Entries for the specified LPID or LPID/PID are invalidated while leaving the corresponding TLB entries intact. The ability to target an individual Page Walk Cache Entry or the set of entries associated with a given Page Table Entry (i.e. IS=0 for RIC=1 or RIC=2) is not supported by the Power ISA. When RIC=3 and IS=0, ***tlbie*** invalidates a series of consecutive translations for HPT translation. The IS≠0 ***tlbiel*** variants operate on a specified congruence class, requiring a software loop where ***tlbie*** operates on the entire TLB. For IS=0 invalidations of Radix Tree translations, the use of ***tlbie[l]*** is limited to translations for quadrant 0.

When reassigning an LPID or PID, after updating the Partition and/or Process Table(s) software must use a ***tlbie*** instruction to remove lookaside information associated with the old parition or process.

To invalidate TLB entries, software must supply an effective page number for process-scoped Radix Tree translations, a guest real page number for partition-scoped Radix Tree translations, and an abbreviated virtual page number for HPT translations. The RTL, RB illustration, and verbal description for R=1 require the reader to make the appropriate mental substitution for partition-scoped invalidation. Note also that where page size is specified to be a function of L and AP, it may also be a function of L and LP. The architecture allows for three independent sets of page sizes, one for R=1, one for RIC=3 (requires R=0), and one for all other cases. An implementation may choose to have a single set of encodings work consistenty between any two or all three states.

Programming Note

Changes to the Page Table in the presence of active transactions may compromise transactional semantics if a page accessed by a translation is remapped within the lifetime of the transaction. Through the use of a ***tlbie*** instruction to the unmapped page, an operating system or hypervisor can ensure that any transaction that has touched the affected page is terminated.

Changes to local translation lookaside buffers, through the ***tlbiel*** instruction, have no effect on transactions. Consequently, if these instructions are used to invalidate TLB entries after the unmapping of a page, it is the responsibility of the OS or hypervisor to ensure that any transaction that may have touched the modified page is terminated, using a ***tabort.*** or ***treclaim*** instruction.

TLB Invalidate Entry *X-form*

tlbie RB,RS,RIC,PRS,R

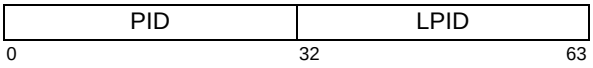
31	RS	/	RIC	PRS	R	RB	306	/
0	6	11	12	14	15	16	21	31

```
IS ← (RB)52:53
if MSRHV=1 then search_LPID=RS32:63
else search_LPID=LPIDRLPID
switch(IS)
  case (0b00):
    If RIC=0
      if R=0 then
        L ← (RB)63
        if L = 0
          then
            base_pg_size = 4K
            actual_pg_size =
              page size specified in (RB)56:58
            i = 51
          else
            base_pg_size =
              base page size specified in (RB)44:51
            actual_pg_size =
             actual page size specified in (RB)44:51
            b ← log_base_2(base_pg_size)
            p ← log_base_2(actual_pg_size)
            i = max(min(43,63-b),63-p)
        sg_size ←segment size specified in (RB)54:55
        for each thread
          for each TLB entry
            if (entry_VA14:i+14 = (RB)0:i) &
               (entry_sg_size = sg_size) &
              (entry_base_pg_size = base_pg_size) &
               (entry_actual_pg_size =
                 actual_pg_size) &
               (entry_LPID = search_LPID) &
               (entry_process_scoped = 0)
            then
              if ((L = 0)|(b ≥ 20)) then
                TLB entry ← invalid
              else
               if (entry_VA58:77-b = (RB)56:75-b) then
                  TLB entry ← invalid
      else
        actual_pg_size =
          page size specified in (RB)56:58
        p ← log_base_2(actual_pg_size)
        i = 63-p
        for each thread
          for each TLB entry
            if (entry_EA0:i = (RB)0:i) &
               (entry_actual_pg_size =
                  actual_pg_size) &
               (entry_LPID = search_LPID) &
               (entry_process_scoped = PRS) &
               ((PRS = 0) |
               (entry_PID = (RS)0:31))
            then
              TLB entry ← invalid
    else if RIC=3 then
      if RBAP indicates cluster bomb then
        n = implementation-specific series size
            f(L||AP)
        base_pg_size =
          implementation-specific base page
            size f(L||AP)
        trunc=log2(n * base_pg_size)-12
        loop_RB ← RB
        loop_RBAP ← implementation-specific
                    encoding for base_pg_size
        loop_RBVPN ← loop_RBVPN[0:51-trunc] || trunc0
        do i=0 to n-1
          tlbie loop_RB,RS,0,0,0
          loop_RBVPN←loop_RBVPN +(base_pg_size/4096)
case (0b01):
  if RIC=0 | RIC=2 then
    for each TLB entry for each thread
      if (entry_LPID=search_LPID)
         &(entry_PID=RS0:31)
         &(entry_PRS=1)
        then TLB entry ← invalid
  if RIC=1 | RIC=2 then
    for each thread
      invalidate process-scoped radix page walk
      caching associated with process RS0:31 in
      partition search_LPID
  if (RIC=2)&(PRS=1) then
    for each thread
      invalidate Process Table caching associated
      with process RS0:31 in partition search_LPID
case (0b10):
  if RIC=0 | RIC=2 then
    if (PRS=0)&((MSRHV=1)|(R=0)) then
     for each partition-scoped TLB entry for each
      thread
        if entry_LPID=search_LPID
          then TLB entry ← invalid
    if PRS=1 then
      for each process-scoped TLB entry for each
      thread
        if entry_LPID=search_LPID
          then TLB entry ← invalid
  if RIC=1 | RIC=2 then
    for each thread
      if (PRS=0)&(MSRHV=1) then
        for each thread invalidate partition-
        scoped page walk caching associated with
        partition search_LPID
      if PRS=1 then
        for each thread invalidate process-scoped
        page walk caching associated with
        partition search_LPID
  if RIC=2 then
    if (PRS=0)&(MSRHV=1) then
      for each thread invalidate Partition Table
      caching associated with partition
      search_LPID
    if PRS=1 then
      for each thread invalidate Process Table
      caching associated with partition
      search_LPID
case (0b11):
  if RIC=0 | RIC=2 then
    if MSRHV then
      for all threads
        if PRS=0 then
          all partition-scoped TLB entries
               ←invalid
```

```
        else
          all process-scoped TLB entries ←invalid
    if (MSR_HV=0)&(PRS=1) then
      for each process-scoped TLB entry for each
      thread
        if TLBE_LPID=search_LPID
          then TLB entry ← invalid
    if (MSR_HV=0)&(PRS=0)&(R=0) then
      for each partition-scoped TLB entry for
      each thread
        if TLBE_LPID=search_LPID
          then TLB entry ←  invalid
  if RIC=1 | RIC=2 then
    if MSR_HV then
      if PRS=0 then
        for all threads
          invalidate all partition-scoped
          page walk caching
      else
        for all threads
          invalidate all process-scoped
          page walk caching
    if (MSR_HV=0) & (PRS=1) then
        for each thread invalidate process-scoped
        page walk caching associated with
        partition search_LPID
  if RIC=2 then
    if MSR_HV then
      if PRS=0 then
        for each thread
          invalidate all Partition Table caching
      else
        for each thread
          invalidate all Process Table caching
    if (MSR_HV=0) & (PRS=1) then
      for each thread invalidate Process Table
      caching associated with partition
      search_LPID
```

The operation performed by this instruction is based on the contents of registers RS and RB. The contents of these registers are shown below, where IS is $(RB)_{52:53}$ and L is $(RB)_{63}$.

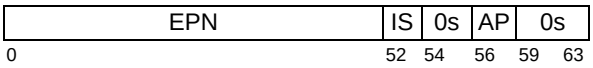
RS:



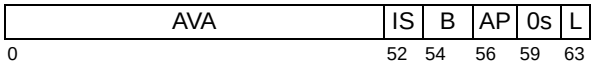
Programming Note

Note that although there is no PID compare for partition-scoped translation, software must still place the PID in RS. It may be used, for example, in the TLB hash.

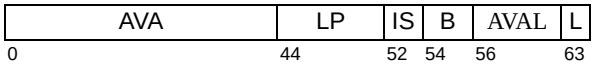
RB for R=1 and IS=0b00::



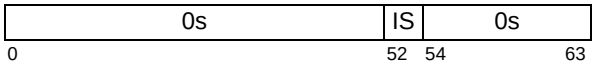
RB for R=0, IS=0b00, and L=0::



RB for R=0, IS=0b00, and L=1:



RB for IS=0b01, 0b10, or 0b11:



If this instruction is executed in hypervisor state, $RS_{32:63}$ contains the partiion ID (LPID) of the partition for which one or more translations are being invalidated. Otherwise, the value in LPIDR is used. The supported $(RS)_{32:63}$ values are the same as the LPID values supported in LPIDR. $RS_{0:31}$ contains a PID value. The supported values of $RS_{0:31}$ are the same as the PID values supported in PIDR.

The following forms are invalid.

- PRS=1, R=0, and RIC≠2 (The only process-scoped HPT caching is of the Process Table.)
- RIC=1 and R=0 (There is no Page Walk Cache for HPT translation.)
- RIC=3 and R=1 (Cluster bombs are only supported for HPT translation.)

The following forms are treated as if the instruction form were invalid.

- RIC=1 and IS=0 (The architecture does not support shootdown of individual translations in the Page Walk Cache.)
- RIC=2 and IS=0 (RIC is for comprehensive invalidation that is not supported at the level of an individual page.)
- RIC=3 and IS≠0 (Cluster bombs are only supported for individual pages.)
- PRS=0 and IS=1 (Partition-scoped translations are not associated with processes.)
- R=0, IS=1, and RIC≠2 (HPT translations are not associated with processes.)
- R=0, RIC=2, PRS=0, HV=0, and IS=2 or 3 (The similar cases with RIC=0 allow the HPT OS to invalidate all of its TLB entries. The only incremental function of these cases is to invalidate partition table caching, which the OS is not permitted to do.)

The results of an attempt to invalidate a translation outside of quadrant 0 for Radix Tree translation (R=1, RIC=0, PRS=1, IS=0, and $EA_{0:1}$≠0b00) are boundedly undefined.

IS field in RB contains 0b00

If RIC=0, this is a search for a single TLB entry. The following relationships must be true and tests and actions are performed to search for an HPT translation.

If the base page size specified by the PTE that was used to create the TLB entry to be invalidated is 4 KB, the L field in register RB must contain 0.

If the L field in RB contains 0, the base page size is 4 KB and $RB_{56:58}$ (AP - Actual Page size field) must be set to the $SLBE_{L||LP}$ encoding for the page size corresponding to the actual page size specified by the PTE that was used to create the TLB entry to be invalidated. Thus, b is equal to 12 and p is equal to $\log_2$ (actual page size specified by $(RB)_{56:58}$). The Abbreviated Virtual Address (AVA) field in register RB must contain bits 14:65 of the virtual address translated by the TLB entry to be invalidated. Variable i is equal to 51.

If the L field in RB contains 1, the following rules apply.

- The base page size and actual page size are specified in the LP field in register RB, where the relationship between $(RB)_{44:51}$ (LP - Large Page size selector field) and the base page size and actual page size is the same as the relationship between PTE_{LP} and the base page size and actual page size, except for the "r" bits (see Section 5.7.9.1 on page 998 and Figure 32 on page 999). Thus, b is equal to $\log_2$ (base page size specified by $(RB)_{44:51}$) and p is equal to $\log_2$ (actual page size specified by $(RB)_{44:51}$). Specifically, $(RB)_{44+c:51}$ must be equal to the contents of bits c:7 of the LP field of the PTE that was used to create the TLB entry to be invalidated, where c is the maximum of 0 and (20-p).
- Variable i is the larger of (63-p) and the value that is the smaller of 43 and (63-b). $(RB)_{0:i}$ must contain bits 14:(i+14) of the virtual address translated by the TLB to be invalidated. If b>20, $RB_{64-b:43}$ may contain any value and are ignored by the hardware.
- If b<20, $(RB)_{56:75-b}$ must contain bits 58:77-b of the virtual address translated by the TLB to be invalidated, and other bits in $(RB)_{56:62}$ may contain any value and are ignored by the hardware.
- If b≥20, $(RB)_{56:62}$ (AVAL - Abbreviated Virtual Address, Lower) may contain any value and are ignored by the hardware.

Let the segment size be equal to the segment size specified in $(RB)_{54:55}$ (B field). The contents of $RB_{54:55}$ must be the same as the contents of the B field of the PTE that was used to create the TLB entry to be invalidated.

$RB_{52:53}$ and $RB_{59:62}$ (when $(RB)_{63}$ = 0) must contain zeros and are ignored by the hardware.

All TLB entries on all threads that have all of the following properties are made invalid.

- The entry translates a virtual address for which all the following are true.
  - $VA_{14:14+i}$ is equal to $(RB)_{0:i}$.
  - L=0 or b≥20 or, if L=1 and b<20, $VA_{58:77-b}$ is equal to $(RB)_{56:75-b}$.
- The segment size of the entry is the same as the segment size specified in $(RB)_{54:55}$.
- Either of the following is true:
  - The L field in RB is 0, the base page size of the entry is 4 KB, and the actual page size of the entry matches the actual page size specified in $(RB)_{56:58}$.
  - The L field in RB is 1, the base page size of the entry matches the base page size specified in $(RB)_{44:51}$, and the actual page size of the entry matches the actual page size specified in $(RB)_{44:51}$.
- $TLBE_{LPID}$ = search_LPID.

Additional TLB entries may also be made invalid if those TLB entries contain an LPID that matches search_LPID.

The following relationships must be true and tests and actions are performed to search for a Radix Tree translation. For a partition-scoped invalidation, references to the effective address are understood to refer to the guest real address.

The page size is encoded in $RB_{56:58}$ (AP - Actual Page size field). Thus p is equal to $\log_2$(page size specified by $RB_{56:58}$). The Effective Page Number (EPN) field in register RB must contain the bits 0:i of the effective address translated by the TLB entry to be invalidated. Variable i is equal to 63-p.

The fields shown as zeros must be set to zero and are ignored by the hardware.

All TLB entries on all threads that have all of the following properties are made invalid.

- The entry translates an effective address for which $EA_{0:i}$ is equal to $(RB)_{0:i}$.
- The page size of the entry matches the page size specified in $(RB)_{56:58}$.
- The entry has the appropriate scope (partition or process).
- The process ID specified in RS matches the process ID in the TLB entry if not invalidating a partition-scoped translation.
- $TLBE_{LPID}$ matches the partiion ID of the partition for which the translation is to be invalidated.

Additional TLB entries may also be made invalid if those TLB entries contain an LPID that matches the partition ID of the partition for which the translation is to be invalidated.

If RIC=3, then an implementation-specific encoding of AP indicates the number and (base) size of a series of sequential virtual pages for which the translations will be invalidated. The pages occupy an aligned region of virtual storage. The address in RB is masked to get the base address of the region. ***tlbie***s are then performed for the virtual pages within the region.

IS field in RB is non-zero

If RIC=0 or RIC=2, all partition-scoped TLB entries when PRS=0 and either MSR_{HV}=1 or R=0, or all process-scoped TLB entries when PRS=1 on all threads for which any of the following conditions are met for the entry are made invalid.

- The IS field in RB contains 0b10 or MSR_{HV}=0 and the IS field contains 0b11, and $TLBE_{LPID}$ matches the partition ID of the partition for which the translation is to be invalidated.
- The IS field in RB contains 0b01, $TLBE_{LPID}$ matches the partition ID of the partition for which the translation is to be invalidated, and $TLBE_{PID}$=$RS_{0:31}$.
- The IS field in RB contains 0b11 and MSR_{HV}=1.

If RIC=1 or RIC=2, if the following conditions are met, the respective partition-scoped contents when PRS=0 and MSR_{HV}=1 or process-scoped contents when PRS=1 of the page walk cache are invalidated.

- If the IS field in RB contains 0b10 or if IS contains 0b11 and MSR_{HV}=0, for all threads, all properly-scoped page walk caching associated with the partition for which the translation is to be invalidated is invalidated.
- If the IS field in RB contains 0b11 and MSR_{HV}=1, the entire properly-scoped page walk caching for each thread is invalidated.
- If the IS field in RB contains 0b01 (and PRS=1), for all threads, all properly-scoped page walk caching associated with process $RS_{0:31}$ in the partition for which the translation is to be invalidated is invalidated.

If RIC=2, if the following conditions are met, the respective partition and Process Table caching are invalidated for all threads.

- If the IS field in RB contains 0b01 and PRS=1, for all threads, caching of Process Table Entries for process $RS_{0:31}$ in the partition for which the translation is to be invalidated is invalidated.
- If the IS field in RB contains 0b10, MSR_{HV}=1, and PRS=0, for all threads, caching of Partition Tables for the partition for which the translation is to be invalidated is invalidated.
- If the IS field in RB contains 0b10 and PRS=1, for all threads, caching of Process Tables for the partition for which the translation is to be invalidated is invalidated.
- if the IS field in RB contains 0b11, MSR_{HV}=1, and PRS=0, for all threads, all Partition Table caching is invalidated.
- if the IS field in RB contains 0b11, MSR_{HV}=1, and PRS=1, for all theads, all Process Table caching is invalidated.
- If the IS field in RB contains 0b11, MSR_{HV}=0, and PRS=1, for all threads, caching of Process Tables for the partition for which the translation is to be invalidated is invalidated.

When i>40, $RB_{40:i-1}$ may contain any value and are ignored by the hardware.

For all IS values

For all threads, any implementation specific lookaside information that is based on any TLB entry that would be invalidated by this instruction will also be invalidated.

MSR_{SF} must be 1 when this instruction is executed; otherwise the results are undefined.

If the value specified in $RS_{0:31}$, $RS_{32:63}$, $RB_{54:55}$ when R=0, $RB_{56:58}$ when RB_{63}=0, or $RB_{44:51}$ when RB_{63}=1 is not supported by the implementation, the instruction is treated as if the instruction form were invalid.

The operation performed by this instruction is ordered by the ***eieio*** (or ***sync*** or ***ptesync***) instruction with respect to a subsequent ***tlbsync*** instruction executed by the thread executing the ***tlbie*** instruction. The operations caused by ***tlbie*** and ***tlbsync*** are ordered by ***eieio*** as a fourth set of operations, which is independent of the other four sets that ***eieio*** orders.

This instruction is privileged except when $LPCR_{GTSE}$=0 or when PRS=0 and HR=1, making it hypervisor privileged.

See Section 5.10, “Translation Table Update Synchronization Requirements” for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonic for ***tlbie***::

Extended:		Equivalent to:	
tlbie	RB,RS	tlbie	RB,RS,0,0,0

Special Registers Altered:

None

Programming Note

For ***tlbie***[***l***] instructions in which $(RB)_{63}$=0, the AP value in RB is provided to make it easier for the hardware to locate address translations, in lookaside buffers, corresponding to the address translation being invalidated.

For ***tlbie***[***l***] instructions the AP specification is not binary compatible with versions of the architecture that precede Version 2.06. As an example, for an actual page size of 64 KB AP=0b101, whereas software written for an implementation that complies with a version of the architecture that precedes V. 2.06 would have AP=100 since AP was a 1 bit value followed by 0s in $RB_{57:58}$. If binary compatibility is important, for a 64 KB page software can use AP=0b101 on these earlier implementations since these implementations were required to ignore $RB_{57:58}$.

Programming Note

For ***tlbie***[***l***] instructions the AVA and AVAL fields in RB contain different VA bits from those in PTE_{AVA}.

Programming Note

An operating system that uses HPT translation should only use ***tlbie*** to invalidate the translation for a specific page when it knows whether VPM is active, and more specifically, what page size is actually in use for the target translation. The address comparison performed by ***tlbie*** is not sensitive to whether VPM is active. As a result, the operating system must supply an AVA value that is appropriate for the page size that is in use.

TLB Invalidate Entry Local *X-form*

tlbiel RB,RS,RIC,PRS,R

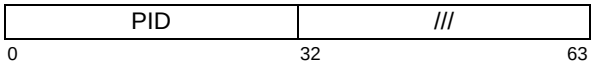
31	RS	/	RIC	PRS	R	RB	274	/
0	6	11	12	14	15	16	21	31

```
IS ← (RB)52:53
search_LPID=LPIDRLPID
switch(IS)
  case (0b00):
    If RIC=0
      If R=0
        L ← (RB)63
        if L = 0  then
          base_pg_size = 4K
          actual_pg_size =
            page size specified in (RB)56:58
          i = 51
        else
          base_pg_size = base page size specified
            in (RB)44:51
          actual_pg_size =
            actual page size specified in (RB)44:51
          b ← log_base_2(base_pg_size)
          p ← log_base_2(actual_pg_size)
          i = max(min(43,63-b),63-p)
        sg_size←segment size specified in (RB)54:55
        for each TLB entry
          if (entry_VA14:i+14 = (RB)0:i) &
             (entry_sg_size = segment_size) &
             (entry_base_pg_size = base_pg_size) &
            (entry_actual_pg_size =actual_pg_size)&
             (TLBELPID=search_LPID) &
             (entry_process_scoped=0)
            then
              if ((L = 0)|(b ≥ 20)) then
                  TLB entry ← invalid
              else
               if (entry_VA58:77-b = (RB)56:75-b) then
                    TLB entry ← invalid
      else
        pg_size = page size specified in (RB)56:58
        p ← log_base_2(pg_size)
        i = 63-p
        for each TLB entry
          if (entry_EA0:i = (RB)0:i) &
             (entry_pg_size = pg_size) &
             (entry_LPID = search_LPID) &
             (entry_process_scoped = PRS) &
             ((PRS = 0) |
             (entry_PID = (RS)0:31))
          then
            TLB entry ← invalid
  case (0b01):
    if RIC=0 | RIC=2 then
      i ← implementation-dependent number, 40≤i≤51
      for each TLB entry in set (RB)i:51
        if (entry_LPID=search_LPID)
           &(entry_PID=RS0:31)
           &(entry_PRS=1)
          then TLB entry ← invalid
    if RIC=1 | RIC=2 then
      invalidate process-scoped radix page walk
      caching associated with process RS0:31 in
      partition search_LPID
    if (RIC=2)&(PRS=1) then
      invalidate Process Table caching associated
      with process RS0:31 in partition search_LPID
  case (0b10):
    if RIC=0 | RIC=2 then
      i←implementation-dependent number, 40≤i≤51
      if (PRS=0)&((MSRHV=1)|(R=0)) then
        for each partition-scoped TLB entry in set
        (RB)i:51
          if entry_LPID=search_LPID
            then TLB entry ← invalid
      if PRS=1 then
        for each process-scoped TLB entry in
        set (RB)i:51
          if entry_LPID=search_LPID
            then TLB entry ← invalid
```

```
  if RIC=1 | RIC=2 then
    for each thread
      if (PRS=0)&(MSR_HV=1) then
        invalidate partition-scoped page walk
        caching associated with partition
        search_LPID
      if PRS=1 then
        invalidate process-scoped page walk
        caching associated with partition
        search_LPID
  if RIC=2 then
    if (PRS=0)&(MSR_HV=1) then
      invalidate Partition Table caching
      associated with partition search_LPID
    if PRS=1 then
      invalidate Process Table caching associated
      with partition search_LPID
case (0b11):
  if RIC=0 | RIC=2 then
    i←implementation-dependent number, 40≤i≤51
    if MSR_HV then
      if PRS=0 then
        all partition-scoped TLB entries in
        set (RB)_i:51 ←invalid
      else
        all process-scoped TLB entries in
        set (RB)_i:51 ←invalid
    if (MSR_HV=0)&(PRS=1) then
      for each process-scoped TLB entry in
      set (RB)_i:51
        if entry_LPID=search_LPID
          then TLB entry ← invalid
    if (MSR_HV=0)&(PRS=0)&(R=0) then
      for each partition-scoped TLB entry in
      set (RB)_i:51
        if entry_LPID=search_LPID
          then TLB entry ←  invalid
  if RIC=1 | RIC=2 then
    if MSR_HV then
      if PRS=0 then
        invalidate all partition-scoped
        page walk caching
      else
        invalidate all process-scoped
        page walk caching
    if (MSR_HV=0) & (PRS=1) then
        invalidate process-scoped page walk
        caching associated with partition
        search_LPID
  if RIC=2 then
    if MSR_HV then
      if PRS=0 then
        invalidate all Partition Table caching
      else
        invalidate all Process Table caching
    if (MSR_HV=0) & (PRS=1) then
      invalidate Process Table caching associated
      with partition search_LPID
```

The operation performed by this instruction is based on the contents of registers RS and RB. The contents of these registers are shown below, where IS is $(RB)_{52:53}$ and L is $(RB)_{63}$.

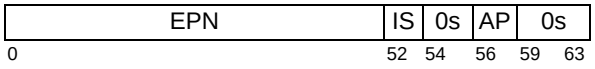
RS:



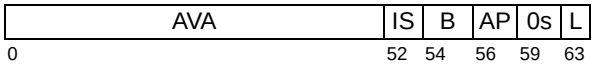
Programming Note

Note that although there is no PID compare for partition-scoped translation, software must still place the PID in RS. It may be used, for example, in the TLB hash.

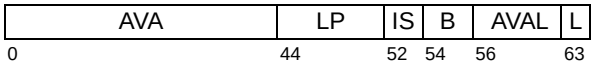
RB for R=1 and IS=0b00::



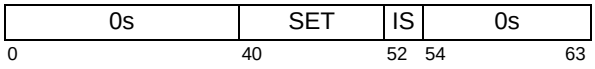
RB for R=0, IS=0b00, and L=0:



RB for R=0, IS=0b00, and L=1:



RB for IS=0b01, 0b10, or 0b11:



LPIDR contains the partiion ID (LPID) of the partition for which the translation is being invalidated. $RS_{0:31}$ contains a PID value. The supported values of $RS_{0:31}$ are the same as the PID values supported in PIDR.

The following forms are invalid.

- PRS=1, R=0, and RIC≠2 (The only process-scoped HPT caching is of the Process Table.)
- RIC=1 and R=0 (There is no Page Walk Cache for HPT translation.)
- RIC=3 (Cluster bombs are not supported for tlbiel.)

The following forms are treated as though the instruction form was invalid.

- RIC=1 and IS=0 (The architecture does not support shootdown of individual translations in the Page Walk Cache.)
- RIC=2 and IS=0 (RIC is for comprehensive invalidation that is not supported at the level of an individual page.)
- PRS=0 and IS=1 (Partition-scoped translations are not associated with processes.)

- R=0, IS=1, and RIC≠2 (HPT translations are not associated with processes.)
- R=0, RIC=2, PRS=0, HV=0, and IS=2 or 3 (The similar cases with RIC=0 allow the HPT OS to invalidate all of its TLB entries. The only incremental function of these cases is to invalidate partition table caching, which the OS is not permitted to do.)

The results of an attempt to invalidate a translation outside of quadrant 0 for Radix Tree translation (R=1, RIC=0, PRS=1, IS=0, and $EA_{0:1}$≠0b00) are boundedly undefined.

IS field in RB contains 0b00

If RIC=0, this is a search for a single TLB entry. The following relationships must be true and tests and actions are performed to search for an HPT translation.

If the base page size specified by the PTE that was used to create the TLB entry to be invalidated is 4 KB, the L field in register RB must contain 0.

If the L field in RB contains 0, the base page size is 4 KB and $RB_{56:58}$ (AP - Actual Page size field) must be set to the $SLBE_{L||LP}$ encoding for the page size corresponding to the actual page size specified by the PTE that was used to create the TLB entry to be invalidated. Thus, b is equal to 12 and p is equal to $\log_2$ (actual page size specified by $(RB)_{56:58}$). The Abbreviated Virtual Address (AVA) field in register RB must contain bits 14:65 of the virtual address translated by the TLB entry to be invalidated. Variable i is equal to 51.

If the L field in RB contains 1, the following rules apply.

- The base page size and actual page size are specified in the LP field in register RB, where the relationship between $(RB)_{44:51}$ (LP - Large Page size selector field) and the base page size and actual page size is the same as the relationship between PTE_{LP} and the base page size and actual page size, except for the "r" bits (see Section 5.7.9.1 on page 998 and Figure 32 on page 999). Thus, b is equal to $\log_2$ (base page size specified by $(RB)_{44:51}$) and p is equal to $\log_2$ (actual page size specified by $(RB)_{44:51}$). Specifically, $(RB)_{44+c:51}$ must be equal to the contents of bits c:7 of the LP field of the PTE that was used to create the TLB entry to be invalidated, where c is the maximum of 0 and (20-p).
- Variable i is the larger of (63-p) and the value that is the smaller of 43 and (63-b). $(RB)_{0:i}$ must contain bits 14:(i+14) of the virtual address translated by the TLB to be invalidated. If b>20, $RB_{64-b:43}$ may contain any value and are ignored by the hardware.
- If b<20, $(RB)_{56:75-b}$ must contain bits 58:77-b of the virtual address translated by the TLB to be invalidated, and other bits in $(RB)_{56:62}$ may contain any value and are ignored by the hardware.
- If b≥20, $(RB)_{56:62}$ (AVAL - Abbreviated Virtual Address, Lower) may contain any value and are ignored by the hardware.

Let the segment size be equal to the segment size specified in $(RB)_{54:55}$ (B field). The contents of $RB_{54:55}$ must be the same as the contents of the B field of the PTE that was used to create the TLB entry to be invalidated.

All TLB entries that have all of the following properties are made invalid on the thread executing the ***tlbiel*** instruction.

- The entry translates a virtual address for which all the following are true.
  - $VA_{14:14+i}$ is equal to $(RB)_{0:i}$.
  - L=0 or b≥20 or, if L=1 and b<20, $VA_{58:77-b}$ is equal to $(RB)_{56:75-b}$.
- The segment size of the entry is the same as the segment size specified in $(RB)_{54:55}$.
- Either of the following is true:
  - The L field in RB is 0, the base page size of the entry is 4 KB, and the actual page size of the entry matches the actual page size specified in $(RB)_{56:58}$.
  - The L field in RB is 1, the base page size of the entry matches the base page size specified in $(RB)_{44:51}$, and the actual page size of the entry matches the actual page size specified in $(RB)_{44:51}$.
- $TLBE_{LPID} = LPIDR_{LPID}$.

The following relationships must be true and tests and actions are performed to search for a Radix Tree translation. For a partition-scoped invalidation, references to the effective address are understood to refer to the guest real address.

The page size is encoded in $RB_{56:58}$ (AP - Actual Page size field). Thus p is equal to $\log_2$(page size specified by $RB_{56:58}$). The Effective Page Number (EPN) field in register RB must contain the bits 0:i of the effective address translated by the TLB entry to be invalidated. Variable i is equal to 63-p.

The fields shown as zeros must be set to zero and are ignored by the hardware.

All TLB entries that have all of the following properties are made invalid on the thread executing the ***tlbiel*** instruction..

- The entry translates an effective address for which $EA_{0:i}$ is equal to $(RB)_{0:i}$.
- The page size of the entry matches the page size specified in $(RB)_{56:58}$.
- The entry has the appropriate scope (partition or process).
- The process ID specified in RS matches the process ID in the TLB entry if not invalidating a partition-scoped translation.

- $TLBE_{LPID}$ matches the partiion ID of the partition for which the translation is to be invalidated.

IS field in RB is non-zero

If RIC=0 or RIC=2, $(RB)_{i:51}$ (bits i-40:11 of the SET field in (RB)) specify a set of TLB entries, where i is an implementation-dependent value in the range 40:51. Each partition-scoped entry when PRS=0 and either MSR_{HV}=1 or R=0, or each process-scoped entry when PRS=1 in the set is invalidated if any of the following conditions are met for the entry.

- The IS field in RB contains 0b10, or MSR_{HV}=0 and the IS field contains 0b11, and $TLBE_{LPID}$ = $LPIDR_{LPID}$.
- The IS field in RB contains 0b01, $TLBE_{LPID}$=$LPIDR_{LPID}$, and $TLBE_{PID}$=$RS_{0:31}$.
- The IS field in RB contains 0b11 and MSR_{HV}=1.

How the TLB is divided into the 2^{52-i} sets is implementation-dependent. The relationship of virtual addresses to these sets is also implementation-dependent. However, if, in an implementation, there can be multiple TLB entries for the same virtual address and same partition, then all these entries must be in a single set.

If RIC=1 or RIC=2, if the following conditions are met, the respective partition-scoped contents when PRS=0 and MSR_{HV}=1 or process-scoped contents when PRS=1 of the page walk cache are invalidated.

- If the IS field in RB contains 0b10 or if IS contains 0b11 and MSR_{HV}=0, all properly-scoped page walk caching associated with partition $LPDIR_{LPID}$ is invalidated.
- If the IS field in RB contains 0b11 and MSR_{HV}=1, the entire properly-scoped page walk caching is invalidated.
- If the IS field in RB contains 0b01 (and PRS=1), all properly-scoped page walk caching associated with process $RS_{0:31}$ in partition $LPIDR_{LPID}$ is invalidated.

If RIC=2, if the following conditions are met, the respective partition and Process Table caching are invalidated.

- If the IS field in RB contains 0b01 and PRS=1, caching of Process Table Entries for process $RS_{0:31}$ in partition $LPIDR_{LPID}$ is invalidated.
- If the IS field in RB contains 0b10, MSR_{HV}=1, and PRS=0, caching of Partition Tables for partition $LPIDR_{LPID}$ is invalidated.
- If the IS field in RB contains 0b10 and PRS=1, caching of Process Tables for partition $LPIDR_{LPID}$ is invalidated.
- if the IS field in RB contains 0b11, MSR_{HV}=1, and PRS=0, all Partition Table caching is invalidated.
- if the IS field in RB contains 0b11, MSR_{HV}=1, and PRS=1, all Process Table caching is invalidated.
- If the IS field in RB contains 0b11, MSR_{HV}=0, and PRS=1, caching of Process Tables for partition $LIDR_{LPID}$ is invalidated.

When i>40, $RB_{40:i-1}$ may contain any value and are ignored by the hardware.

For all IS values

Any implementation specific lookaside information that is based on any TLB entry that would be invalidated by this instruction will also be invalidated.

Depending on the variant of the instruction, $RB_{0:39}$, $RB_{59:62}$, $RB_{59:63}$, $RB_{54:55}$, and $RB_{54:63}$ are the equivalent of reserved fields, should contain 0s, and are ignored by the hardware. $RS_{32:63}$ is always the equivalent of a reserved field, should contain 0s, and is ignored by the hardware.

Only TLB entries, page walk caching, and process and Segment Table caching on the thread executing the ***tlbiel*** instruction are affected.

MSR_{SF} must be 1 when this instruction is executed; otherwise the results are boundedly undefined.

If the value specified in $RS_{0:31}$, $RB_{54:55}$, $RB_{56:58}$, or $RB_{44:51}$, when it is needed to perform the specified operation, is not supported by the implementation, the instruction is treated as if the instruction form were invalid.

This instruction is privileged except when PRS=0 and HR=1, making it hypervisor privileged.

See Section 5.10, “Translation Table Update Synchronization Requirements” on page 1043 for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonic for ***tlbiel***::

Extended:		**Equivalent to:**	
tlbiel	RB	tlbiel	RB,r0,0,0,0

Programming Note

tlbie and ***tlbiel*** serve as both basic and extended mnemonics. The Assembler will recognize a ***tlbie*** or ***tlbiel*** mnemonic with five operands as the basic form, and a ***tlbie*** with two operands or a ***tlbiel*** mnemonic with one operand as the extended form. In the extended form the RIC, PRS, and R operands, and for ***tlbiel*** the RS operand, are omitted and assumed to be 0.

Programming Note

The primary use of this instruction by hypervisor software is to invalidate TLB entries prior to reassigning a thread to a new logical partition.

The primary use of this instruction by operating system software is to invalidate TLB entries that were created by the hypervisor using an implementation-specific hypervisor-managed TLB facility, if such a facility is provided.

tlbiel may be executed on a given thread even if the sequence ***tlbie*** - ***eieio*** - ***tlbsync*** - ***ptesync*** is concurrently being executed on another thread.

See also the Programming Notes with the description of the ***tlbie*** instruction.

Programming Note

An operating system that uses HPT translation should only use ***tlbiel*** to invalidate the translation for a specific page when it knows whether VPM is active, and more specifically, what page size is actually in use for the target translation. The address comparison performed by ***tlbiel*** is not sensitive to whether VPM is active. As a result, the operating system must supply an AVA value that is appropriate for the page size that is in use.

TLB Synchronize *X-form*

tlbsync

31	///	///	///	566	/
0	6	11	16	21	31

The ***tlbsync*** instruction provides an ordering function for the effects of all ***tlbie*** instructions executed by the thread executing the ***tlbsync*** instruction, with respect to the memory barrier created by a subsequent ***ptesync*** instruction executed by the same thread. Executing a ***tlbsync*** instruction ensures that all of the following will occur.

- All TLB invalidations caused by ***tlbie*** instructions preceding the ***tlbsync*** instruction will have completed on any other thread before any data accesses caused by instructions following the ***ptesync*** instruction are performed with respect to that thread.
- All storage accesses by other threads for which the address was translated using the translations being invalidated, and all Reference and Change bit updates associated with address translations that were performed by other threads using the translations being invalidated, will have been performed with respect to the thread executing the ***ptesync*** instruction, to the extent required by the associated Memory Coherence Required attributes, before the ***ptesync*** instruction's memory barrier is created.

The operation performed by this instruction is ordered by the ***eieio*** (or ***sync*** or ***ptesync***) instruction with respect to preceding ***tlbie*** instructions executed by the thread executing the ***tlbsync*** instruction. The operations caused by ***tlbie*** and ***tlbsync*** are ordered by ***eieio*** as a fourth set of operations, which is independent of the other three sets that ***eieio*** orders.

The ***tlbsync*** instruction may complete before operations caused by ***tlbie*** instructions preceding the ***tlbsync*** instruction have been performed.

This instruction is privileged except when $LPCR_{G\text{-}TSE}$=0, making it hypervisor privileged.

See Section 5.10 for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Programming Note

tlbsync should not be used to synchronize the completion of ***tlbiel***.

5.10 Translation Table Update Synchronization Requirements

This section describes rules that software must follow when updating the Translation Tables, and includes suggested sequences of operations for some representative cases. The sequences required for other cases may be deduced from the sequences that are provided and from this accompanying description.

In the sequences of operations shown in the following subsections, the Page Table Entry is assumed to be for a virtual page for which the base page size is equal to the actual page size. If these page sizes are different, multiple ***tlbie*** instructions are needed, one for each PTE corresponding to the virtual page.

In the sequences of operations shown in the following subsections, any alteration of a translation table entry that corresponds to a single line in the sequence is assumed to be done using a *Store* instruction for which the access is atomic. Appropriate modifications must be made to these sequences if this assumption is not satisfied (e.g., if a store doubleword operation is done using two *Store Word* instructions).

Two correctness-related considerations when choosing translation table update sequences are to be safe for multiple asynchronous sources of update (potentially both hardware and software), and to avoid paradoxes that in some cases could show up as multi-hits in the various translation caches. These considerations lead to the simple, contiguous sequences for general case updates that appear later in this section. Good performance is a third consideration that motivates deferring and/or batching invalidations or even omitting synchronization or invalidation from the general case. The viability of these techniques is determined by whether the lack of a single clear state across the system has problematic repercussions. The discussion of atomic Reference and Change bit updates alludes to one such example. (See Section 5.7.12.) Simpler optimizations are illustrated below.

The following are guidelines for safety when multiple sources of asynchronous updates are possible. To interact correctly with hardware that atomically updates Reference and Change bits (as well as with updates from other software threads), software should use atomic updates to modify valid PTEs. Academically speaking, if hardware uses simple loads and stores, software may either use locking and first invalidate the PTE and cached translations, or may attempt to optimize using atomic updates that don't change the values of the bytes containing the Reference and Change bits with the exception of potentially setting those specific bits to 1 or the Reference bit to 0. When modifying only bytes not subject to hardware modification, software may use either locking or atomic updates, subject to the limitations and optimizations described below. The realities of Reference and Change bit placement may severely limit what optimizations are possible when hardware uses normal loads and stores to update those bits.

To simplify verification and avoid paradoxes, non-impactful limitations are placed on translation table update sequence optimizations. One limitation is that software must not have two or more valid overlapping translations at any level of the translation process with different page or segment sizes. This means that one translation must be marked invalid in the translation table and invalidated from any caches prior to instating the second. The other limitation is that software must not have two or more valid translations with different attributes (i.e. WIMG, ATT). The example of I=1 and I=0 is obvious, but in general there is not enough to be gained to attempt to avoid invalidating one attribute setting before establishing another. In both of these cases, the translation cache invalidation may lag indefinitely behind the table entry invalidations and the cache invalidations may be batched, but must precede enabling the new attributes.

To protect software's ability to have reasonable performance, optimizations that hardware must support are also identified. (These optimizations are understood to be limited by the techniques used for hardware and software updates as described above, and by the properties of the table structure itself. A convention for atomic updates will yield more opportunity than locking. Hardware that does not use atomic updates may limit or eliminate the opportunity for software to optimize. The table structure for Radix Tree translation will yield more opportunity than the dual PTEG structure of HPT translation.) Access authority downgrades and setting Change bits to zero may be done without first marking the PTE invalid and invalidating the translation caches. The translation cache invalidation may lag the PTE change indefinitely and be done in bulk. Access authority upgrades and setting Reference and Change bits to 1 may be done without any PTE or translation cache invalidation. Software bits may be changed without any PTE or translation cache invalidation. Finally, any complete change to the RPN (non-overlapping with the original value) does not of itself require synchronization (though other changes to the PTE made at the same time might).

In the following examples, when the same type of sequence works for both types of translation, the HPT PTE is shown because it is more complex. In this description, and in references in subsequent subsections to "safe for multithreaded software," the safety is with respect to the risk of one thread overwriting another's update. There may also be concern for the creation of multiple matching translations, e.g. within a PTEG or pair of PTEGs. When the reservation granule is equal to or larger in size than the structure on which mutual exclusion must be ensured (e.g. PTE for Radix

Tree translation but PTEG for HPT translation), multiple entries will also be prevented. (Secondary hash groups will generally not be covered by the same reservation granule as primary hash groups.)

Updates (by software) to the tables are performed only when they are known to be required by the sequential execution model (see Section 5.5). Because address translation for instructions preceding a given *Store* instruction might cause an interrupt, and thereby prevent the corresponding store from being required by the sequential execution model, address translations for instructions preceding the *Store* instruction must be performed before the corresponding store is performed. As a result, an update to a translation table need not be preceded by a context synchronizing instruction.

All of the sequences require a context synchronizing operation after the sequence if the new contents of the translation table are to be used for address translations associated with subsequent instructions.

As noted in the description of the *Synchronize* instruction in Section 4.6.3 of Book II, address translation associated with instructions which occur in program order subsequent to the *Synchronize* (and this includes the ***ptesync*** variant) may be performed prior to the completion of the *Synchronize*. To ensure that these instructions and data which may have been speculatively fetched are discarded, a context synchronizing operation is required.

Programming Note

In many cases this context synchronization will occur naturally; for example, if the sequence is executed within an interrupt handler the ***rfid***, ***rfscv***, or ***hrfid*** instruction that returns from the interrupt handler may provide the required context synchronization.

Translation table entries must not be changed in a manner that causes an implicit branch.

5.10.1 Translation Table Updates

TLBs are non-coherent caches of the HTABs and Radix Trees. TLB entries must be invalidated explicitly with one of the *TLB Invalidate* instructions. SLBs are non-coherent caches of the Segment Tables, SLB entries must be invalidated explicitly with one of the *SLB Invalidate* instructions. Page Walk Caches are non-coherent caches of the intermediate steps in Radix Tree translation. Non-coherent caching of the Partition and Process Tables is permitted. Provision has been made for the use of the *TLB Invalidate* instructions to manage the types of caching described in the preceding two sentences at a PID or LPID granularity.

Unsynchronized lookups in the Page, Segment, and when HR=0, Process Tables continue even while they are being modified. (For Partition Table Entries, and for Process Table Entries when HR=1, the process or partition affected must be inactive because the entries do not have valid bits.) With the exceptions previously identified for Segment Table walks (see Section 5.9.3, “Lookaside Buffer Management”), any thread, including a thread on which software is modifying any of the set of tables described in the first sentence, may look in those tables at any time in an attempt to translate an address. When modifying an entry in any of the former set of tables, software must ensure that the table entry’s V bit is 0 if the table entry does not correctly specify its portion of the translation (e.g., if the RPN field is not correct for the current AVA field).

For HPT translation, updates of Reference and Change bits by the hardware are not synchronized with the accesses that cause the updates. When modifying doubleword 1 of a PTE, software must take care to avoid overwriting a hardware update of these bits and to avoid having the value written by a *Store* instruction overwritten by a hardware update.

The most basic sequence that will achieve proper system synchronization for PTE updates is the following.

tlbie instruction(s) specifying the same LPID operand value
eieio
tlbsync
ptesync

Other instructions may be interleaved among these instructions. Operating system and hypervisor software that updates Page Table Entries should use this sequence.

Operating systems and nested hypervisors are exposed to being interrupted during this sequence. The interrupting hypervisor is responsible for completing the sequence above. In general this will require the hypervisor to include the following sequence in an interrupt handler.

eieio
tlbsync
ptesync

This sequence itself may be interrupted by a higher level hypervisor. When returning to the interrupted software, the original sequence will be completed. Hardware must tolerate the result of nested interleaving of these sequences. ***tlbie*** and ***tlbsync*** instructions should only be used as part of these sequences.

The corresponding sequence for Segment Table updates uses ***slbieg*** in place of ***tlbie*** and ***slbsync*** in place of ***tlbsync***. Similarly ***slbieg*** and ***slbsync*** should only be used as part of these sequences. In circumstances where a hypervisor may be interrupting either a PTE update or a Segment Table update, it must include both ***tlbsync*** and ***slbsync*** in its completing sequence, in either order. Hardware must tolerate the result of nested interleaving of these additional sequences.

The PTE sequence is also used to synchronize updates to Partition Table Entries, and to Process Table Entries that do not have valid bits. Mutual exclusion must be added if the update processes are multithreaded.

On systems consisting of only a single-threaded processor, the ***eieio*** and ***tlbsync*** or ***slbsync*** instructions can be omitted.

The following subsections illustrate sequences that must be used for translation table updates to tables that are subject to concurrent use by hardware (i.e. that have valid bits in their entries). For Partition Table Entries and for Process Table Entries that do not have valid bits, simpler sequences consisting of just the preceding sequences, perhaps with mutual exclusion if the update processes are multithreaded, is sufficient.

Programming Note

The ***eieio*** instruction prevents the reordering of the preceding ***tlbie***, ***slbieg***, or ***slbiag*** instructions with respect to the subsequent ***tlbsync*** or ***slbsync*** instruction. The ***tlbsync*** or ***slbsync*** instruction and the subsequent ***ptesync*** instruction together ensure that all storage accesses for which the address was translated using the translations being invalidated (by the ***tlbie***, ***slbieg***, or ***slbiag*** instructions), and all Reference and Change bit updates associated with address translations that were performed using the translations being invalidated, will be performed with respect to any thread or mechanism, to the extent required by the associated Memory Coherence Required attributes, before any data accesses caused by instructions following the ptesync instruction are performed with respect to that thread or mechanism.

For Page Table update sequences that mark the PTE invalid (see Section 5.10.1.2, “Modifying a Translation Table Entry”), Reference and Change bit updates cease when the sequence is complete. When the PTE is marked invalid using an atomic update and the *Store Conditional* setting the entry invalid is successful, the Reference and Change bits obtained by the corresponding *Load And Reserve* instruction are stable/final values.

The sequences of operations shown in the following subsections assume a multi-threaded environment. In an environment consisting of only a single-threaded processor, the ***tlbsync*** or ***slbsync*** and the ***eieio*** that separates the ***tlbie*** or ***slbieg*** from the ***tlbsync*** or ***slbsync*** can be omitted. In a multi-threaded environment, when ***tlbiel*** or ***slbie*** is used instead of ***tlbie*** or ***slbieg*** in a Page or Segment Table update, the synchronization requirements are the same as when ***tlbie*** or ***slbieg*** is used in an environment consisting of only a single-threaded processor.

Programming Note

For all of the sequences shown in the following subsections, if it is necessary to communicate completion of the sequence to software running on another thread, the ***ptesync*** instruction at the end of the sequence should be followed by a *Store* instruction that stores a chosen value to some chosen storage location X. The memory barrier created by the ***ptesync*** instruction ensures that if a *Load* instruction executed by another thread returns the chosen value from location X, all subsequent searches of the Page or Segment Table by the other thread, that implicitly load from the PTE or STE specified by the sequence’s stores, will obtain the values stored (or values stored subsequently). The *Load* instruction that returns the chosen value should be followed by a context synchronizing instruction in order to ensure that all instructions following the context synchronizing instruction will be fetched and executed using the values stored by the sequence (or values stored subsequently). (These instructions may have been fetched or executed out-of-order using the old contents of the PTE or STE.)

This Note assumes that the Page or Segment Table and location X are in storage that is Memory Coherence Required.

5.10.1.1 Adding a Page Table Entry

This is the simplest Page Table case. The V bit of the old entry is assumed to be 0. The following sequence can be used to create a PTE, maintain a consistent state, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes. A single quadword store would avoid the need for the ***eieio***. A similar sequence may be used to add a new Segment Table Entry. Mutual exclusion with respect to other software threads may be required, but there is no concern for interaction with hardware updates because the entry is invalid until the last store in the sequence.

```
PTE(pp key B ARPN LP key R C WIMG N pp) ← new values
eieio      /* order 1st update before 2nd */
PTE(AVA SW L H V) ← new values (V=1)
ptesync    /* order updates before next
              Page Table search and before
              next data access              */
```

5.10.1.2 Modifying a Translation Table Entry

General Case (PTE)

If a valid entry is to be modified and the translation instantiated by the entry being modified is to be invalidated, the sequences below can be used to modify the

PTE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes.

The following sequence is to interact correctly with atomic hardware updates. It returns stable Reference and Change bit values for the old translation and is safe for multitheaded software. If the purpose of the sequence is mainly to collect Reference and Change bit values, the part of the sequence beginning with ***tlbie*** may be deferred and performed as a bulk invalidation (e.g. for a range of storage or an entire process) after collecting values for a plurality of pages. A similar seqence (i.e. using *Load And Reserve* and *Store Conditional* instructions) can be used to update a Segment Table Entry but will not interact correctly with non-atomic hardware Reference and Change bit updates.

```
r6←PTE_V L SW RPN R C Att EAA
r4←addr(pte)
loop:
  lqarx r2,0,r4
  if V=0 abort, else /* to interact with locking */
  stqcx r6,0,r4
  bne-  loop
ptesync   /* order update before tlbie and
             before next Page Table search */
tlbie(old_EA0:63-b,old_AP,old_PID,
          old_LPID)
eieio     /* order tlbie before tlbsync     */
tlbsync   /* order tlbie before ptesync     */
ptesync    /*complete the sequence, stores ordered
           /*by first ptesync
```

The corresponding sequence for non-atomic hardware updates is the following. (The sequence is equivalent to deleting the PTE and then adding a new one.) Mutual exclusion with respect to other software threads may be required. The Reference and Change bit values will not be stable until the entire sequence is completed.

```
PTE_V ← 0  /* (other fields don't matter)*/
ptesync   /* order update before tlbie and
             before next Page Table search */
tlbie(old_B,old_VA14:77-b,old_L,old_LP,old_AP,
          old_LPID)
          /*invalidate old translation*/
eieio     /* order tlbie before tlbsync     */
tlbsync   /* order tlbie before ptesync     */
ptesync   /* order tlbie, tlbsync and 1st
             update before 2nd update      */
PTE_ARPN,LP,AC,R,C,WIMG,N,PP ← new values
eieio     /* order 2nd update before 3rd */
PTE_B,AVA,SW,L,H,V ← new values (V=1)
ptesync   /* order 2nd and 3rd updates before
             next Page Table search and
             before next data access      */
```

General Case(STE)

If a valid entry is to be modified and the translation instantiated by the entry being modified is to be invalidated, the following sequence can be used to modify the STE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the effective address translated by the new entry will use the correct virtual address and associated attributes. (The sequence is much like the general case for a change to a PTE that is subject to non-atomic hardware updates, and is equivalent to deleting the STE and then adding a new one.) Mutual exclusion with respect to other software threads may be required. A similar sequence (except using ***tlbie*** with RIC=2 and ***tlbsync***) may be used to modify HR=0 Process Table Entries.

```
STE_V ← 0  /* (other fields don't matter)*/
ptesync   /* order update before slbieg and
             before next Segment Table search */
slbieg(old_B,old_ESID,old_TA,old_PID,old_LPID)
          /*invalidate old translation*/
eieio     /* order slbieg before slbsync     */
slbsync   /* order slbieg before ptesync     */
ptesync   /* order slbieg, slbsync and 1st
             update before 2nd update      */
   /* deletion sequence ends here */
STE_VSID, Ks, Kp, N, L, C, LP, SW ← new values
eieio     /* order 2nd update before 3rd */
STE_ESID,V ← new values (V=1)
ptesync   /* order 2nd and 3rd updates before
             next Segment Table search and
             before next data access      */
```

Resetting the Reference Bit (PTE)

If the only change being made to a valid entry is to set the Reference bit to 0, a simpler sequence suffices because the Reference bit need not be maintained exactly. The byte store is exposed to overwriting another change being performed by multithreaded software, so mutual exclusion may be required.

```
oldR ← PTE_R /* get old R                        */
if oldR = 1 then
  PTE_R ← 0        /* store byte (R=0, other bits
                      unchanged)                */
  tlbie(B,VA14:77-b,L,LP,AP,LPID) /* invalidate
                                     entry     */
  eieio   /* order tlbie before tlbsync          */
  tlbsync /* order tlbie before ptesync          */
  ptesync /* order tlbie, tlbsync, and update
             before next Page Table search
             and before next data access         */
```

Setting a Reference or Change Bit or Upgrading Access Authority (PTE Subject to Atomic Hardware Updates)

If the only change being made to a valid PTE that is subject to atomic hardware updates is to set the Reference or Change bit to 1 or to add access authorities, a simpler sequence suffices because the translation hardware will refetch the PTE if an access is attempted for which the only problems were reference and/or

change bits needing to be set or insufficient access authority. The store is exposed to overwriting another change being performed by multithreaded software, so mutual exclusion may be required.

```
PTE V L SW RPN R C Att EAA ← new values (V=1)
ptesync   /* order update before next Page Table
             search and before next data access  */
```

Modifying the SW field (PTE)

If the only change being made to a valid entry is to modify the SW field, the following sequence suffices, because the SW field is not used by the hardware (i.e. is not cached in the TLB and has no effect on hardware behavior).

```
loop: ldarx r1 ← PTE_dwd_0 /* load dwd 0 of PTE */
     if V=0 abort, else/*to interact with locking*/
      r1 57:60 ← new SW value /* replace SW, in r1 */
      stdcx. PTE_dwd_0 ←  r1 /* store dwd 0 of PTE
             if still reserved (new SW value, other
             fields unchanged) */
      bne-    loop    /* loop if lost reservation */
```

A ***lbarx/stbcx.***, ***lharx***/***sthcx.***, or ***lwarx***/***stwcx.*** pair (specifying the low-order byte, halfword, or word respectively of doubleword 0 of the PTE) can be used instead of the ***ldarx*** /***stdcx.*** pair shown above for HPT translation. The split SW field in the radix PTE cannot be updated with a single smaller atomic update. This sequence interacts correctly with hardware updates and is safe for multithreaded software. A similar sequence (including the possibility of using a smaller atomic update) can be used to update a Segment Table Entry.

Modifying the Effective Address (STE)

If the effective address translated by a valid STE is to be modified and the new effective address hashes to the same STEG as does the old effective address, the following sequence can be used to modify the STE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the effective address translated by the new entry will use the correct virtual address and associated attributes. Mutual exclusion with respect to other software threads may be required. The corresponding change of the virtual address in the PTE for HPT translation can be performed using a similar sequence, interacting correctly with non-atomic hardware table updates, as long as the second doubleword of the PTE is not stored.

```
STE ESID,V ← new values (V=1)
ptesync   /* order update before slbieg and
             before next Segment Table search     */
slbieg(old_B,old_ESID,old_TA,old_PID,old_LPID)
                  /*invalidate old translation*/
eieio     /* order slbieg before slbsync          */
slbsync   /* order slbieg before ptesync          */
ptesync   /* order slbieg, slbsync, and update
             before next data access              */
```

Chapter 6. Interrupts

6.1 Overview

The Power ISA provides an interrupt mechanism to allow the thread to change state as a result of external signals, errors, or unusual conditions arising in the execution of instructions.

System Reset and Machine Check interrupts are not ordered. All other interrupts are ordered such that only one interrupt is reported, and when it is processed (taken) no program state is lost. Since Save/Restore Registers SRR0 and SRR1 are serially reusable resources used by most interrupts, program state may be lost when an unordered interrupt is taken.

6.2 Interrupt Registers

6.2.1 Machine Status Save/ Restore Registers

When various interrupts occur, the state of the machine is saved in the Machine Status Save/Restore registers (SRR0 and SRR1). Section 6.5 describes which registers are altered by each interrupt.

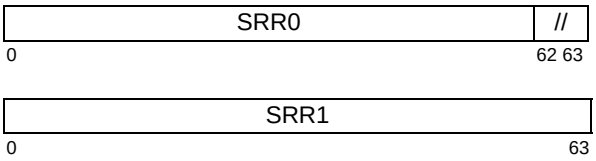


Figure 52. Save/Restore Registers

SRR1 bits may be treated as reserved in a given implementation if they correspond to MSR bits that are reserved or are treated as reserved in that implementation and, for SRR1 bits in the range 33:36, 42:43, and 45:47, they are specified as being set either to 0 or to an undefined value for all interrupts that set SRR1 (including implementation-dependent setting, e.g. by the Machine Check interrupt or by implementation-specific interrupts). $SRR1_{44}$ cannot be treated as reserved, regardless of how it is set by interrupts, because it is used by software, as described in a Programming Note near the end of Section 6.5.9, “Program Interrupt” on page 1074.

6.2.2 Hypervisor Machine Status Save/Restore Registers

When various interrupts occur, the state of the machine is saved in the Hypervisor Machine Status Save/ Restore registers (HSRR0 and HSRR1). Section 6.5 describes which registers are altered by each interrupt.

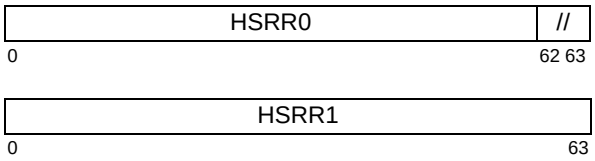


Figure 53. Hypervisor Save/Restore Registers

HSRR1 bits may be treated as reserved in a given implementation if they correspond to MSR bits that are reserved or are treated as reserved in that implementation and, for HSRR1 bits in the range 33:36 and 42:47, they are specified as being set either to 0 or to an undefined value for all interrupts that set HSRR1 (including implementation-dependent setting, e.g. by implementation-specific interrupts).

The HSRR0 and HSRR1 are hypervisor resources; see Chapter 2.

Programming Note

Execution of some instructions, and fetching instructions when MSR_{IR}=1 or MSR_{HV}=0, may have the side effect of modifying HSRR0 and HSRR1; see Section 6.4.4.

6.2.3 Access Segment Descriptor Register

The DAR, HDAR, SRR0, and HSRR0 generally provide the EA for storage exceptions. For hypervisor storage interrupts, additional information is often necessary to enable the hypervisor to handle the interrupt. This information is provided in a 64b SPR called the Access

Segment Descriptor Register (ASDR). When nested Radix Tree translation is taking place, the ASDR will generally provide the guest real address down to bit 51. (The smallest supported page size is 4k.) When using paravirtualized HPT translation, information from the segment descriptor that was used to perform the effective to virtual translation is provided in the ASDR. For exceptions that take place when translating the address of the process table entry or segment table entry group, only the VSID will be provided, because those addresses are specified as virtual addresses and the rest of the segment descriptor is implied. Some instances of the Machine Check interrupt may require the ASDR to be set similarly to how it is set for the hypervisor storage interrupts. The ASDR is set independent of the value of UPRT for the partition that is running.

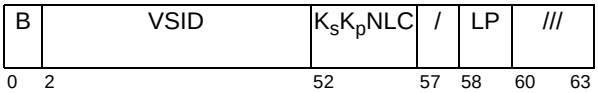


Figure 54. Access Segment Descriptor Register format for a Segment Descriptor

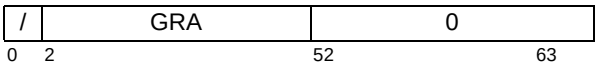


Figure 55. Access Segment Descriptor Register format for a Guest Real Address

6.2.4 Data Address Register

The Data Address Register (DAR) is a 64-bit register that is set by the Machine Check, Data Storage, Data Segment, and Alignment interrupts; see Sections 6.5.2, 6.5.3, 6.5.4, and 6.5.8. In general, when one of these interrupts occurs the DAR is set to an effective address associated with the storage access that caused the interrupt, with the high-order 32 bits of the DAR set to 0 if the interrupt occurs in 32-bit mode.

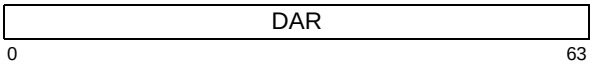


Figure 56. Data Address Register

6.2.5 Hypervisor Data Address Register

The Hypervisor Data Address Register (HDAR) is a 64-bit register that is set by the Hypervisor Data Storage Interrupt; see Section 6.5.16. In general, when this interrupt occurs, the HDAR is set to an effective address associated with the storage access that caused the interrupt, with the high-order 32 bits of the HDAR set to 0 if the interrupt occurs in 32-bit mode.

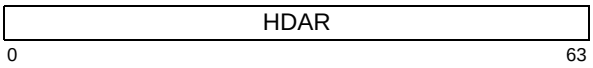


Figure 57. Hypervisor Data Address Register

6.2.6 Data Storage Interrupt Status Register

The Data Storage Interrupt Status Register (DSISR) is a 32-bit register that is set by the Machine Check, Data Storage, and Data Segment interrupts; see Sections 6.5.2, 6.5.3, and 6.5.4.

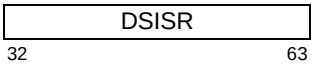


Figure 58. Data Storage Interrupt Status Register

DSISR bits may be treated as reserved in a given implementation if they are specified as being set either to 0 or to an undefined value for all interrupts that set the DSISR.

6.2.7 Hypervisor Data Storage Interrupt Status Register

The Hypervisor Data Storage Interrupt Status Register (HDSISR) is a 32-bit register that is set by the Hypervisor Data Storage interrupt. In general, when one of these interrupts occurs the HDSISR is set to indicate the cause of the interrupt.

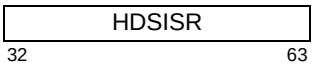


Figure 59. Hypervisor Data Storage Interrupt Status Register

6.2.8 Hypervisor Emulation Instruction Register

The Hypervisor Emulation Instruction Register (HEIR) is a 32-bit register that is set by the Hypervisor Emulation Assistance interrupt; see Section 6.5.18. The image of the instruction that caused the interrupt is loaded into the register.

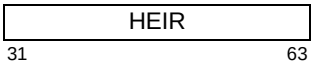


Figure 60. Hypervisor Emulation Instruction Register

6.2.9 Hypervisor Maintenance Exception Register

Each bit in the Hypervisor Maintenance Exception Register (HMER) is associated with one or more causes of the Hypervisor Maintenance exception, and is set when the associated exception(s) occur. If the corresponding bit in the Hypervisor Maintenance Exception Enable Register (HMEER) is set, a Hypervisor Maintenance Interrupt (HMI) may occur. If the thread is in a power-saving mode when the interrupt would have occurred, the thread will exit the power-saving mode; see Section 6.5.19 and Section 3.3.2.

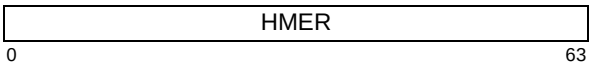


Figure 61. Hypervisor Maintenance Exception Register

The contents of the HMER are as follows:

0	Set to 1 for a Malfunction Alert.
1	Set to 1 when performance is degraded for thermal reasons.
2	Set to 1 when thread recovery is invoked.
Others	Implementation-specific.

When the ***mtspr*** instruction is executed with the HMER as the encoded Special Purpose Register, the contents of register RS are ANDed with the contents of the HMER and the result is placed into the HMER.

The exception bits in the HMER are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an ***mthmer*** instruction.

Programming Note

An access to the HMER is likely to be very slow. Software should access it sparingly.

6.2.10 Hypervisor Maintenance Exception Enable Register

The Hypervisor Maintenance Exception Enable Register (HMEER) is a 64-bit register in which each bit enables the corresponding exception in the HMER to cause the Hypervisor Maintenance interrupt, potentially causing exit from power-saving mode; see Section 6.5.19 and Section 3.3.2.

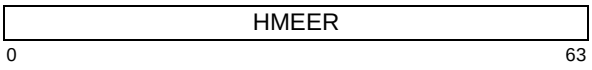


Figure 62. Hypervisor Maintenance Exception Enable Register

6.2.11 Facility Status and Control Register

The Facility Status and Control Register (FSCR) controls the availability of various facilities in problem state and indicates the cause of a Facility Unavailable interrupt.

When the FSCR makes a facility unavailable, attempted usage of the facility in problem state is treated as follows:

- Execution of an instruction causes a Facility Unavailable interrupt.
- Access of an SPR using ***mfspr/mtspr*** causes a Facility Unavailable interrupt.
- ***rfebb***, ***rfid***, ***rfscv***, ***hrfid*** and ***mtmsr[d]*** instructions have the same effect on bits in system registers as they would if the bits were available. The same is true for ***mtspr*** and ***mfspr*** unless the preceding item applies.

The MSR can also make the Transactional Memory facility unavailable in any privilege state, and MMCR0 can make various components of the Performance Monitor unavailable when accessed in problem state. An access to one of these facilities when it is unavailable causes a Facility Unavailable interrupt.

When the PCR makes a facility unavailable in problem state, the facility is treated as not defined in problem state; any Facility Unavailable interrupt that would occur if the facility were not made unavailble by the PCR does not occur.

When a Facility Unavailable interrupt occurs, the unavailable facility that was accessed is indicated in the most-significant byte of the FSCR.

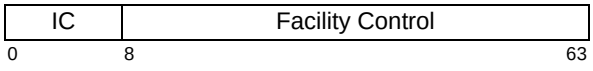


Figure 63. Facility Status and Control Register

The contents of the FSCR are specified below.

Value	Meaning
0:7	**Interruption Cause** (IC) When a Facility Unavailable interrupt occurs, the IC field contains a binary number indicating the facility for which access was attempted. The values and their meanings are specified below. 02 Access to the DSCR at SPR 3 03 Access to a Performance Monitor SPR in group A or B when $MMCR0_{PMCC}$ is set to a value for which the access results in a Facility Unavailable interrupt. (See the

definition of $MMCR0_{PMCC}$ in Section 9.4.4.)
- 04 Execution of a BHRB Instruction
- 05 Access to a Transactional Memory SPR or execution of a Transactional Memory Instruction
- 06 Reserved
- 07 Access to an Event-Based Branch SPR or execution of an Event-Based Branch instruction
- 08 Access to the Target Address Register
- 0C Execution of ***scv***

All other values are reserved.

8:63 **Facility Enable** (FE)

The FE field controls the availability of various facilities in problem state as specified below.

8:50 Reserved

51 ***scv* instruction**

- 0 The ***scv*** instruction is not available.
- 1 The ***scv*** instruction is available.

52:54 Reserved

55 **Target Address Register** (TAR)

- 0 The TAR and ***bctar*** instruction are not available in problem state.
- 1 The TAR and ***bctar*** instruction are available in problem state unless made unavailable by another register.

56 **Event-Based Branch Facility** (EBB)

- 0 The Event-Based Branch facility SPRs and instructions are not available in problem state, and event-based exceptions and branches do not occur.
- 1 The Event-Based Branch facility SPRs and instructions (see Chapter 7 of Book II) are available in problem state unless made unavailable by another register, and event-based exceptions and branches are allowed to occur if enabled by other registers.

57:60 Reserved

Programming Note

$HFSCR_{58:60}$ are used to control the availability of Transactional Memory, the Performance Monitor, and the BHRB in problem and privileged non-hypervisor states. $FSCR_{58:60}$ are reserved since the availability of Transactional Memory is controlled by the MSR, and the availability of the Performance Monitor and BHRB is controlled by MMCR0.

61 **Data Stream Control Register at SPR 3** (DSCR)

- 0 SPR 3 is not available in problem state.
- 1 SPR 3 is available in problem state unless made unavailable by another register.

62:63 Reserved

Programming Note

When an OS has set the FSCR such that a facility is unavailable, the OS should either emulate the facility when it is accessed or provide an application interface that requires the application to request use of the facility before it accesses the facility.

6.2.12 Hypervisor Facility Status and Control Register

The Hypervisor Facility Status and Control Register (HFSCR) controls the availability of various facilities in problem and privileged non-hypervisor states, and indicates the cause of a Hypervisor Facility Unavailable interrupt.

When the HFSCR makes a facility unavailable, attempted usage of the facility in problem or privileged non-hypervisor states is treated as follows:

- Execution of an instruction causes a Hypervisor Facility Unavailable interrupt.
- Access of an SPR using ***mfspr/mtspr*** causes a Hypervisor Facility Unavailable interrupt
- ***rfebb***, ***rfid***, ***rfscv***, ***hrfid*** and ***mtmsr[d]*** instructions have the same effect on bits in system registers as they would if the bits were available. The same is true for ***mtspr*** and ***mfspr*** unless the preceding item applies.

Programming Note

Notice that ***rfebb***, ***rfscv***, ***[h]rfid***, and ***mtmsrd*** instructions can cause a TM Bad Thing type Program interrupt even when executed in a privilege state in which TM is made unavailable by the HFSCR. Here are two examples. Both assume that $HFSCR_{TM}$=0; the second assumes that $HFSCR_{EBB}$=1.

- An operating system, running with $MSR_{TS\ TM}$ = 0b000 (N0), sets $SRR1_{29:31}$ to 0b101 (T1) then executes ***rfid***. The attempted illegal transaction state transition will cause a TM Bad Thing type Program interrupt, despite the fact that TM is made unavailable in privileged non-hypervisor state by the HFSCR.
- An application program, running with $MSR_{TS\ TM}$ = 0b000 (N0), sets $BESCR_{TS}$ to 0b01 (S) then executes ***rfebb***. The attempted illegal transaction state transition will cause a TM Bad Thing type Program interrupt, despite the fact that TM is made unavailable in problem state by the HFSCR.

This anomaly cannot be caused by the PCR.

- ***rfscv***, ***[h]rfid***, and ***mtmsrd*** cannot be executed in the privilege state (problem state) in which TM is made unavailable by the PCR.
- ***rfebb*** can be executed in the privilege state in which TM is made unavailable by the PCR, but the PCR bit that makes TM unavailable (the v2.06 bit) also makes ***rfebb*** unavailable.

Another difference between the HFSCR and the PCR is that $PCR_{v2.06}$=1 prevents a thread from being simultaneously in problem state and in Transactional or Suspended state and $HFSCR_{TM}$=0 does not. However, if the hypervisor always returns to the partition in Non-transactional state when $HFSCR_{TM}$=0, the partition will be unable to enter Transactional or Suspended state.

When the PCR makes a facility unavailable in problem state, the facility is treated as not defined in problem state; any Hypervisor Facility Unavailable interrupt that would occur if the facility were not made unavailble by the PCR does not occur as a result of problem state access. See Section 2.5 for additional information.

When a Hypervisor Facility Unavailable interrupt occurs, the facility that was accessed is indicated in the most-significant byte of the HFSCR.

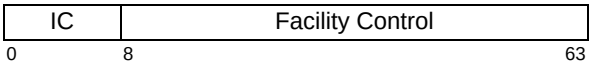


Figure 64. Hypervisor Facility Status and Control Register

The contents of the HFSCR are specified below.

Value	Meaning
0:7	**Interruption Cause** (IC) When a Hypervisor Facility Unavailable interrupt occurs, the IC field contains a binary number indicating the access that was attempted. The values and their meanings are specified below. 00 Access to a Floating Point register or execution of a Floating Point instruction 01 Access to a Vector or VSX register or execution of a Vector or VSX instruction 02 Access to the DSCR at SPRs 3 or 17 03 Read or write access of a Performance Monitor SPR in group A, or read access of a Performance Monitor SPR in group B. (See Section 9.4.1 for a definition of groups A and B.) 04 Execution of a BHRB Instruction 05 Access to a Transactional Memory SPR or execution of a Transactional Memory instruction 06 Reserved 07 Access to an Event-Based Branch SPR or execution of an Event-Based Branch instruction 08 Access to the Target Address Register 09 Access to the ***stop*** instruction in privileged non-hypervisor state when one or more of the following conditions exist. $PSSCR_{EC}$=1 $PSSCR_{ESL}$=1 $PSSCR_{MTL}$>$PSSCR_{PSLL}$ $PSSCR_{RL}$>$PSSCR_{PSLL}$ 0A Access to the ***msgsndp*** or ***msgclrp*** instructions, the TIR or the DPDES Register All other values are reserved.
8:63	**Facility Enable** (FE) The FE field controls the availability of various facilities in problem and privileged non-hypervisor states as specified below.
8:52	Reserved

Programming Note

There is no bit in this register controlling the availability of the ***stop*** instruction because the availability of ***stop*** in privileged non-hypervisor state is controlled by the PSSCR. See Section 3.2.3.

53	***msgsndp*** **instructions and SPRs** (MSGP) 0 The ***msgsndp*** and ***msgclrp*** instructions and the TIR and DPDES registers are not available in privileged non-hypervisor state.

1 The ***msgsndp*** and ***msgclrp*** instructions and the TIR and DPDES registers are available in privileged non-hypervisor state unless made unavailable by another register.

54 Reserved

55 **Target Address Register** (TAR)

0 The TAR and ***bctar*** instruction are not available in problem and privileged non-hypervisor state.
1 The TAR and ***bctar*** instruction are available in problem and privileged states unless made unavailable by another register.

56 **Event-Based Branch Facility** (EBB)

0 The Event-Based Branch facility SPRs and instructions are not available in problem and privileged non-hypervisor states, and event-based exceptions and branches do not occur.
1 The Event-Based Branch facility SPRs and instructions are available in problem and privileged states unless made unavailable by another register, and event-based exceptions and branches are allowed to occur if enabled by other bits.

57 Reserved

58 **Transactional Memory Facility** (TM)

0 The Transactional Memory Facility SPRs and instructions are not available in problem and privileged non-hypervisor states.
1 The Transactional Memory Facility SPRs and instructions are available in problem and privileged states unless made unavailable by another register.

59 **BHRB Instructions** (BHRB)

0 The BHRB instructions (***clrbhrb, mfbhrbe***) are not available in problem and privileged non-hypervisor states.
1 The BHRB instructions (***clrbhrb, mfbhrbe***) are available in problem and privileged states unless made unavailable by another register.

60 **Performance Monitor Facility SPRs** (PM)

0 Read and write operations of Performance Monitor SPRs in group A and read operations of Performance Monitor SPRs in group B are not available in problem and privileged non-hypervisor states; read and write operations to privileged Performance Monitor registers (SPRs 784-792, 795-798) are not available in privileged non-hypervisor state. (See Section 9.4.1 for a definition of groups A and B.) Performance Monitor exceptions do not cause Performance Monitor interrupts to occur when the thread is in problem or privileged states.
1 Read and write operations of Performance Monitor SPRs in group A and read operations of Performance Monitor SPRs in group B are available in problem and privileged states unless made unavailable by another register; read and write operations to privileged Performance Monitor registers (SPRs 784-792, 795-798) are available in privileged state; Performance Monitor interrupts to occur if MSR_{EE}=1 and $MMCR0_{EBE}$=0. See Section 9.2 of Book III for additional information

61 **Data Stream Control Register** (DSCR)

0 SPR 3 is not available in problem or privileged non-hypervisor states and SPR 17 is not available in privileged non-hypervisor state.
1 SPR 3 is available in problem and privileged states and SPR 17 is available in privileged state unless made unavailable by another register.

62 **Vector and VSX Facilities** (VECVSX)

0 The facilities whose availability is controlled by either MSR_{VEC} or MSR_{VSX} are not available in problem and privileged non-hypervisor states.
1 The facilities whose availability is controled by either MSR_{VEC} or MSR_{VSX} are available in problem and privileged states unless made unavailable by another register.

63 **Floating Point Facility** (FP)

0 The facilities whose availability is controlled by MSR_{FP} are not available in problem and privileged non-hypervisor states.
1 The facilities whose availability is controlled by MSR_{FP} are available in problem and privileged states unless made unavailable by another register.

Programming Note

The FSCR can be used to determine whether a particular facility is being used by an application, and the HFSCR can be used to determine whether a particular facility is being used by either an application or by an operating system. This is done by disabling the facility initially, and enabling it in the interrupt handler upon first usage. The information about the usage of a particular facility can be used to determine whether that facility's state must be saved and restored when changing program context.

Programming Note

The following tables summarize the interrupts that occur as a result of accessing the non-privileged Performance Monitor registers in problem state when $MMCR0_{PMCC}$, PCR, and HFSCR are set to various values. (Accesses to privileged Performance Monitor SPRs (SPRs 784-792, 795-798) in problem state result in Privileged Instruction Type Program interrupts.)

			mfspr				***mtspr***			
			PMCC				**PMCC**			
	SPR	**#**	**00**	**01**	**10**	**11**	**00**	**01**	**10**	**11**
Group A	MMCR2[3]	769	HU[4]	FU, HU[4]	HU[4]	HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	HU[4]
	MMCRA	770	HU[4]	FU, HU[4]	HU[4]	HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	HU[4]
	PMC1	771	HU[4]	FU, HU[4]	HU[4]	HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	HU[4]
	PMC2	772	HU[4]	FU, HU[4]	HU[4]	HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	HU[4]
	PMC3	773	HU[4]	FU, HU[4]	HU[4]	HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	HU[4]
	PMC4	774	HU[4]	FU, HU[4]	HU[4]	HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	HU[4]
	PMC5	775	HU[4]	FU, HU[4]	HU[4]	FU, HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	FU, HU[4]
	PMC6	776	HU[4]	FU, HU[4]	HU[4]	FU, HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	FU, HU[4]
	MMCR0	779	HU[4]	FU, HU[4]	HU[4]	HU[4]	HE,HU[4]	FU, HU[4]	HU[4]	HU[4]
Group B	SIER[3]	768	HU[4]	FU, HU[4]	HU[4]	HU[4]	See 2.	See 2.	See 2.	See 2.
	SIAR	780	HU[4]	FU, HU[4]	HU[4]	HU[4]	See 2.	See 2.	See 2.	See 2.
	SDAR	781	HU[4]	FU, HU[4]	HU[4]	HU[4]	See 2.	See 2.	See 2.	See 2.
	MMCR1	782	HU[4]	FU, HU[4]	FU, HU[4]	FU, HU[4]	See 2.	See 2.	See 2.	See 2.

Notes:

1. Terminology:
   FU: Facility Unavailable interrupt
   HE: Hypervisor Emulation Assistance interrupt
   HU: Hypervisor Facility Unavailable interrupt
2. This SPR is read-only, and cannot be written in any privilege state. (See the ***mtspr*** instruction description in Section 4.4.4 for additional information.) FU or HU interrupts do not occur regardless of the value of $MMCR0_{PMCC}$ or $HFSCR_{PM}$.
3. When the PCR indicates a version of the architecture prior to V 2.07, this SPR is treated as undefined in problem state; no FU or HU interrupts occur regardless of the value of $MMCR0_{PMCC}$ or $HFSCR_{PM}$.
4. An HU interrupt occurs if $HFSCR_{PM}$=0 when this SPR is accessed in either problem state or privileged non-hypervisor state.

Programming Note

When an MSR bit makes a facility unavailable, the facility is made unavailable in all privilege states. Examples of this include the Floating Point, Vector, and VSX facilities. The FSCR and HFSCR affect the availability of facilities only in privilege states that are lower than the privilege of the register (FSCR or HFSCR).

6.3 Interrupt Synchronization

When an interrupt occurs, in general SRR0 or HSRR0 is set to point to an instruction such that all preceding instructions have completed execution, no subsequent instruction has begun execution, and the instruction addressed by SRR0 or HSRR0 may or may not have completed execution, depending on the interrupt type. The only exception is that if an ***mtspr*** sequence started by ***mtgsr*** is active when the interrupt occurs, some of the sequence's ***mtspr***s beyond the instruction pointed to by SRR0 or HSRR0 may have been executed; see Chapter 11.

With the exception of System Reset and Machine Check interrupts, all interrupts are context synchronizing as defined in Section 1.5.1. System Reset and Machine Check interrupts are context synchronizing if they are recoverable (i.e., if bit 62 of SRR1 is set to 1 by the interrupt). If a System Reset or Machine Check interrupt is not recoverable (i.e., if bit 62 of SRR1 is set to 0 by the interrupt), it acts like a context synchronizing operation with respect to subsequent instructions. That is, a non-recoverable System Reset or Machine Check interrupt need not satisfy items 1 through 3 of Section 1.5.1, but does satisfy items 4 and 5.

6.4 Interrupt Classes

Interrupts are classified by whether they are directly caused by the execution of an instruction or are caused by some other system exception. Those that are "system-caused" are:

- System Reset
- Machine Check
- External
- Decrementer
- Directed Privileged Doorbell
- Hypervisor Decrementer
- Hypervisor Maintenance
- Hypervisor Virtualization
- Directed Hypervisor Doorbell
- Performance Monitor

External, Decrementer, Hypervisor Decrementer, Directed Privileged Doorbell, Directed Hypervisor Doorbell, Hypervisor Maintenance, and Hypervisor Virtualization interrupts are maskable interrupts. Therefore, software may delay the generation of these interrupts. System Reset and Machine Check interrupts are not maskable.

"Instruction-caused" interrupts are further divided into two classes, *precise* and *imprecise*.

6.4.1 Precise Interrupt

Except for the Imprecise Mode Floating-Point Enabled Exception type Program interrupt, all instruction-caused interrupts are precise.

When the fetching or execution of an instruction causes a precise interrupt, the following conditions exist at the interrupt point.

1. SRR0 addresses either the instruction causing the exception or the immediately following instruction. Which instruction is addressed can be determined from the interrupt type and status bits.
2. An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing thread.
3. The instruction causing the exception may appear not to have begun execution (except for causing the exception), may have been partially executed, or may have completed, depending on the interrupt type.
4. Architecturally, no subsequent instruction has begun execution, except that if an ***mtspr*** sequence started by ***mtgsr*** is active when the interrupt occurs, some of the sequence's ***mtspr***s beyond the interrupt point may have been executed; see Chapter 11 of Book III.

6.4.2 Imprecise Interrupt

This architecture defines one imprecise interrupt, the Imprecise Mode Floating-Point Enabled Exception type Program interrupt.

When an Imprecise Mode Floating-Point Enabled Exception type Program interrupt occurs, the following conditions exist at the interrupt point.

1. SRR0 addresses either the instruction causing the exception or some instruction following that instruction; see Section 6.5.9, "Program Interrupt" on page 1074.
2. An interrupt is generated such that all instructions preceding the instruction addressed by SRR0 appear to have completed with respect to the executing thread.
3. The instruction addressed by SRR0 may appear not to have begun execution (except, in some cases, for causing the interrupt to occur), may have been partially executed, or may have completed; see Section 6.5.9.
4. No instruction following the instruction addressed by SRR0 appears to have begun execution, except that if an ***mtspr*** sequence started by ***mtgsr*** is active when the interrupt occurs, some of the sequence's ***mtspr***s beyond the interrupt point may have been executed; see Chapter 11.

All Floating-Point Enabled Exception type Program interrupts are maskable using the MSR bits FE0 and FE1. Although these interrupts are maskable, they differ significantly from the other maskable interrupts in that the masking of these interrupts is usually controlled by the application program, whereas the masking of all other maskable interrupts is controlled by either the operating system or the hypervisor.

6.4.3 Interrupt Processing

Associated with each kind of interrupt is an *interrupt vector*, which contains the initial sequence of instructions that is executed when the corresponding interrupt occurs.

Interrupt processing consists of saving a small part of the thread's state in certain registers, identifying the cause of the interrupt in other registers, and continuing execution at the corresponding interrupt vector location. When an exception exists that will cause an interrupt to be generated and it has been determined that the interrupt will occur, the following actions are performed. The handling of Machine Check interrupts (see Section 6.5.2) and System Call Vectored interrupts (see Section 6.5.27) differs from the description given below in several respects.

1. SRR0 or HSRR0 is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.
2. Bits 33:36 and 42:47 of SRR1 or HSRR1 are loaded with information specific to the interrupt type.
3. Bits 0:32, 37:41, and 48:63 of SRR1 or HSRR1 are loaded with a copy of the corresponding bits of the MSR.
4. The MSR is set as shown in Figure 65 on page 1064. In particular, MSR bits IR and DR are set as specified by $LPCR_{AIL}$ (see Section 2.2), and MSR bit SF is set to 1, selecting 64-bit mode. The new values take effect beginning with the first instruction executed following the interrupt.
5. Instruction fetch and execution resumes, using the new MSR value, at the effective address specific to the interrupt type. These effective addresses are shown in Figure 66 on page 1065. An offset may be applied to get the effective addresses, as specified by $LPCR_{AIL}$ (see Section 2.2).

Interrupts do not clear reservations obtained with ***lbarx***, ***lharx***, ***lwarx, ldarx***, or ***lqarx***.

Programming Note

In general, when an interrupt occurs, the following instructions should be executed by the interrupt handler before dispatching a "new" program on the thread.

- ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** to clear the reservation if one is outstanding, to ensure that a ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, or ***lqarx*** in the interrupted program is not paired with a ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** on the "new" program.
- "***eieio***, ***tlbsync***, ***slbsync***, ***ptesync***," to complete any outstanding translation table modification sequence and ensure that all storage accesses caused by the interrupted program will be performed with respect to another thread before the program is resumed on that other thread. (If software conventions are such that there is no possibility of a translation table modification sequence being in progress on the thread, a ***sync*** instruction suffices.)
- ***isync*** or ***rfid***, to ensure that the instructions in the "new" program execute in the "new" context.
- ***treclaim***, to ensure that any previous use of the transactional facility is terminated.
- ***cpabort***, to clear state from any previous use of the Copy-Paste Facility.

Programming Note

For instruction-caused interrupts, in some cases it may be desirable for the operating system to emulate the instruction that caused the interrupt, while in other cases it may be desirable for the operating system not to emulate the instruction. The following list, while not complete, illustrates criteria by which decisions regarding emulation should be made. The list applies to general execution environments; it does not necessarily apply to special environments such as program debugging, bring-up, etc.

In general, the instruction should be emulated if:

- The interrupt is caused by a condition for which the instruction description (including related material such as the introduction to the section describing the instruction) implies that the instruction works correctly. Example: Alignment interrupt caused by ***lmw*** for which the storage operand is not aligned, or by ***dcbz*** for which the storage operand is in storage that is Write Through Required or Caching Inhibited.
- The instruction is an illegal instruction that should appear, to the program executing it, as if it were supported by the implementation. Example: A Hypervisor Emulation Assistance interrupt is caused by an instruction that has been phased out of the architecture but is still used by some programs that the operating system supports.

If the instruction is a *Storage Access* instruction, the emulation must satisfy the atomicity requirements described in Section 1.4 of Book II.

In general, the instruction should not be emulated if:

- The purpose of the instruction is to cause an interrupt. Example: System Call interrupt caused by ***sc***.
- The interrupt is caused by a condition that is stated, in the instruction description, potentially to cause the interrupt. Example: Alignment interrupt caused by ***lwarx*** for which the storage operand is not aligned.
- The program is attempting to perform a function that it should not be permitted to perform. Example: Data Storage interrupt caused by ***lwz*** for which the storage operand is in storage that the program should not be permitted to access. (If the function is one that the program should be permitted to perform, the conditions that caused the interrupt should be corrected and the program re-dispatched such that the instruction will be re-executed. Example: Data Storage interrupt caused by ***lwz*** for which the storage operand is in storage that the program should be permitted to access but for which there currently is no PTE that satisfies the Page Table search.)

Programming Note

If a program modifies an instruction that it or another program will subsequently execute and the execution of the instruction causes an interrupt, the state of storage and the content of some registers may appear to be inconsistent to the interrupt handler program. For example, this could be the result of one program executing an instruction that causes a Hypervisor Emulation Assistance interrupt just before another instance of the same program stores an *Add Immediate* instruction in that storage location. To the interrupt handler code, it would appear that a hardware generated the interrupt as the result of executing a valid instruction.

Programming Note

Hardware reports system integrity problems via Machine Check and System Reset interrupts that set $SRR1_{62}$ to 0. All other interrupts that set the SRRs, including Machine Check and System Reset interrupts that do not themselves report integrity problems, copy MSR_{RI} to $SRR1_{62}$. (All interrupts that set the SRRs set MSR_{RI} to 0.) To interact correctly with this behavior, interrupt handlers for interrupts that set the SRRs should do as follows.

- In each such interrupt handler, interpret $SRR1_{62}$ as:
  - 0: interrupt is not recoverable
  - 1: interrupt is recoverable
- In each such interrupt handler, when enough state has been saved that another interrupt that sets the SRRs can be recovered from, set MSR_{RI} to 1.
- In each such interrupt handler, do the following (in order) just before returning.
  1. Set MSR_{RI} to 0.
  2. Set SRR0 and SRR1 to the values to be used by ***rfid***. The new value of SRR1 should have bit 62 set to 1 (which will happen naturally if SRR1 is restored to the value saved there by the interrupt, because the interrupt handler will not be executing this sequence unless the interrupt is recoverable).
  3. Execute ***rfid***.

Programming Note

Because interrupts that set the HSRRs preserve MSR_{RI} instead of setting it to 0 as is done by interrupts that set the SRRs, handlers for interrupts that set the HSRRs must prevent additional such interrupts from occurring until enough state has been saved that another such interrupt can be recovered from, and also when the HSRRs have been restored prior to executing ***hrfid***. Required behavior during those intervals includes the following.

- Keep $MSR_{HV\ EE\ PR}$=0b100. (This state prevents many such interrupts from occurring.)
- Execute only defined instructions that are not in invalid form.
- Pin the first page of the hypervisor's Process Table
- Ensure that the PTE mapping the first page of the hypervisor's Process Table has the Reference bit set and has no other reason to cause an exception.

6.4.4 Implicit alteration of HSRR0 and HSRR1

Executing some of the more complex instructions may have the side effect of altering the contents of HSRR0 and HSRR1. The instructions listed below are guaranteed not to have this side effect. Any omission of instruction suffixes is significant; e.g., ***add*** is listed but ***add.*** is excluded.

1. *Branch* instructions

   b[***l***][***a***], ***bc***[***l***][***a***], ***bclr***[***l***], ***bcctr***[***l***]

2. *Fixed-Point Load* and *Store* Instructions

   lbz, ***lbzx***, ***lhz***, ***lhzx***, ***lwz***, ***lwzx***, ***ld***, ***ldx***, ***stb***, ***stbx***, ***sth***, ***sthx***, ***stw***, ***stwx***, ***std***, ***stdx***

   Execution of these instructions is guaranteed not to have the side effect of altering HSRR0 and HSRR1 only if the storage operand is aligned and $MSR_{HV\ DR}$=0b10.

3. *Arithmetic* instructions

   addi, ***addis***, ***add***, ***subf***, ***neg***

4. *Compare* instructions

   cmpi, ***cmp***, ***cmpli***, ***cmpl***

5. *Logical* and *Extend Sign* instructions

   ori, ***oris***, ***xori***, ***xoris***, ***and***, ***or***, ***xor***, ***nand***, ***nor***, ***eqv***, ***andc***, ***orc, extsb***, ***extsh***, ***extsw***

6. *Rotate and Shift* instructions

   rldicl, ***rldicr***, ***rldic***, ***rlwinm***, ***rldcl***, ***rldcr***, ***rlwnm***, ***rldimi***, ***rlwimi***, ***sld***, ***slw***, ***srd***, ***srw***

7. Other instructions

   isync

   rfid, ***hrfid***

   mtspr, ***mfspr***, ***mtmsrd***, ***mfmsr***

Programming Note

Instructions excluded from the list include the following.

- instructions that set or use XER_{CA}
- instructions that set XER_{OV} or XER_{SO}
- ***andi.***, ***andis.***, and fixed-point instructions with Rc=1 (Fixed-point instructions with Rc=1 can be replaced by the corresponding instruction with Rc=0 followed by a *Compare* instruction.)
- all floating-point instructions
- ***mftb***

These instructions, and the other excluded instructions, may be implemented with the assistance of the Hypervisor Emulation Assistance interrupt, or of implementation-specific interrupts that modify HSRR0 and HSRR1. The included instructions are guaranteed not to be implemented thus. (The included instructions are sufficiently simple as to be unlikely to need such assistance. Moreover, they are likely to be needed in interrupt handlers before HSRR0 and HSRR1 have been saved or after HSRR0 and HSRR1 have been restored.)

Similarly, fetching instructions may have the side effect of altering the contents of HSRR0 and HSRR1 unless $MSR_{HV\ IR}$ = 0b10.

6.5 Interrupt Definitions

Figure 65 shows all the types of interrupts and the values assigned to the MSR for each. Figure 66 shows the effective address of the interrupt vector for each interrupt type. (Section 5.7.5 on page 987 summarizes all architecturally defined uses of effective addresses, including those implied by Figure 66.)

Interrupt Type	**MSR Bit**							
	IR	DR	FE0	FE1	EE	RI	ME	HV
System Reset	0	0	0	0	0	0	p	1
Machine Check	0	0	0	0	0	0	0	1
Data Storage	r	r	0	0	0	0	-	-
Data Segment	r	r	0	0	0	0	-	-
Instruction Storage	r	r	0	0	0	0	-	-
Instruction Segment	r	r	0	0	0	0	-	-
External	r	r	0	0	0	h	-	e
Alignment	r	r	0	0	0	0	-	-
Program	r	r	0	0	0	0	-	-
Floating-Point Unavailable	r	r	0	0	0	0	-	-
Decrementer	r	r	0	0	0	0	-	-
Hypervisor Decrementer	r	r	0	0	0	-	-	1
Directed Privileged Doorbell	r	r	0	0	0	0	-	-
System Call	r	r	0	0	0	0	-	s
Trace	r	r	0	0	0	0	-	-
Hypervisor Data Storage	r	r	0	0	0	-	-	1
Hypervisor Instruction Storage	r	r	0	0	0	-	-	1
Hypervisor Emulation Assistance	r	r	0	0	0	-	-	1
Hypervisor Maintenance	0	0	0	0	0	-	-	1
Directed Hypervisor Doorbell	r	r	0	0	0	-	-	1
Hypervisor Virtualization	r	r	0	0	0	-	-	1
Performance Monitor	r	r	0	0	0	0	-	-
Vector Unavailable	r	r	0	0	0	0	-	-
VSX Unavailable	r	r	0	0	0	0	-	-
Facility Unavailable	r	r	0	0	0	0	-	-
Hypervisor Facility Unavailable	r	r	0	0	0	-	-	1
System Call Vectored	r	r	0	0	-	-	-	-

Interrupt Type	MSR Bit IR DR FE0 FE1 EE RI ME HV

0 bit is set to 0
1 bit is set to 1
- bit is not altered
r for interrupts for which $LPCR_{AIL}$ applies, if $LPCR_{AIL}$=2 or 3, set to 1; otherwise set to 0
p if the interrupt occurred while the thread was in power-saving mode, set to 1; otherwise not altered
e if LPES=0, set to 1; otherwise not altered
h if LPES=1, set to 0; otherwise not altered
s if LEV=1, set to 1; otherwise not altered

Settings for Other Bits

Bits bit 5, TM, VEC, VSX, PR, FP, and PMM are set to 0.

The TE field is set to 0b00.

TM, FP, VEC, VSX, and bit 5 are set to 0.

If the interrupt results in HV being equal to 1, the LE bit is copied from the HILE bit; otherwise the LE bit is copied from the $LPCR_{ILE}$ bit.

The SF bit is set to 1.

If the TS field contained 0b10 (Transactional) when the interrupt occurred, the TS field is set to 0b01 (Suspended); otherwise the TS field is not altered.

Reserved bits are set as if written as 0.

Figure 65. MSR setting due to interrupt

Effective Address[1]	Interrupt Type
00..0000_0100	System Reset
00..0000_0200	Machine Check
00..0000_0300	Data Storage
00..0000_0380	Data Segment
00..0000_0400	Instruction Storage
00..0000_0480	Instruction Segment
00..0000_0500	External
00..0000_0600	Alignment
00..0000_0700	Program
00..0000_0800	Floating-Point Unavailable
00..0000_0900	Decrementer
00..0000_0980	Hypervisor Decrementer
00..0000_0A00	Directed Privileged Doorbell
00..0000_0B00	Reserved
00..0000_0C00	System Call
00..0000_0D00	Trace
00..0000_0E00	Hypervisor Data Storage
00..0000_0E20	Hypervisor Instruction Storage
00..0000_0E40	Hypervisor Emulation Assistance
00..0000_0E60	Hypervisor Maintenance
00..0000_0E80	Directed Hypervisor Doorbell
00..0000_0EA0	Hypervisor Virtualization
00..0000_0EC0	Reserved
00..0000_0EE0	Reserved for implementation-dependent interrupt for performance monitoring
00..0000_0F00	Performance Monitor
00..0000_0F20	Vector Unavailable
00..0000_0F40	VSX Unavailable
00..0000_0F60	Facility Unavailable
00..0000_0F80	Hypervisor Facility Unavailable
00..0000_0FA0	Reserved
. . .	. . .
00..0000_0FFF	Reserved
00..0001_7000	System Call Vectored
00..0001_7020	System Call Vectored
. . .	. . .
00..0001_7FE0	System Call Vectored
00..0001_7FFF	(end of ***scv*** interrupt vectors)

Effective Address[1]	Interrupt Type

[1] The values in the Effective Address column are interpreted as follows.

- 00...0000_0nnn *means* 0x0000_0000_0000_0nnn unless the values of $LPCR_{AIL}$ and $MSR_{HV\ IR\ DR}$ cause the application of an effective address offset. See the description of $LPCR_{AIL}$ in Section 2.2 for more details.
- 0...00_0001_7nnn *means* 0x0000_0000_0001_7nnn unless the values of $LPCR_{AIL}$ and $MSR_{HV\ IR\ DR}$ cause the usage of an alternate effective address. See the description of $LPCR_{AIL}$ in Section 2.2 for details.

[2] Effective addresses 0x0000_0000_0000_0000 through 0x0000_0000_0000_00FF are used by software and will not be assigned as interrupt vectors.

Figure 66. Effective address of interrupt vector by interrupt type

Programming Note

When address translation is disabled, use of any of the effective addresses that are shown as reserved in Figure 66 risks incompatibility with future implementations.

6.5.1 System Reset Interrupt

If a System Reset exception causes an interrupt that is not context synchronizing or causes the loss of a Machine Check exception or a Direct External exception, or if the state of the thread has been corrupted, the interrupt is not recoverable.

When the thread is in any power-saving level, a System Reset interrupt occurs when a System Reset exception exists. When the thread is in a power-saving level that was entered when $PSSCR_{EC}$=1, a System Reset interrupt also occurs when any of the following events occurs provided that the event is enabled to cause exit from power-saving mode (see Section 2.2). When the thread is in a power-saving level that allows the state of the LPCR to be lost, it is implementation-specific whether the following events, when enabled, cause exit, or whether only a system-reset exception causes exit.

- External
- Decrementer
- Directed Privileged Doorbell
- Directed Hypervisor Doorbell
- Hypervisor Maintenance

- Hypervisor Virtualization exception
- Implementation-specific

SRR1 indicates the exception that caused exit from power-saving mode as specified below.

The following registers are set:

SRR0 If the interrupt did not occur when the thread was in power-saving mode, set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present; if the interrupt occurred when the thread was in a power-saving mode that was entered with PSSCR bit ESL=0, and fields RL, MTL, and PSLL set to values that do not allow state loss, set to the effective address of the instruction following the stop instruction; otherwise, set to an undefined value.

If the interrupt occurred while the thread was in power-saving mode, set to the effective address of the instruction following the ***stop*** instruction when ***stop*** is executed with PSSCR bit ESL=0 and fields RL, MTL, and PSLL set to values that do not allow state loss; otherwise, set to an undefined value.

Programming Note

Whenever ***stop*** is executed in privileged non-hypervisor state, the hypervisor typically sets both $PSSCR_{ESL}$ and $PSSCR_{EC}$ to 0, and sets RL and MTL to values that do not cause state loss. If an interrupt causes exit to power-saving mode (either because the interrupt was a System Reset or Machine Check interrupt or MSR_{EE}=1), then SRR0 for that interrupt contains the effective address of the instruction immediately following ***stop***.

SRR1

33 Implementation-dependent.

34:36 Set to 0.

42:45 If the interrupt did not occur when the thread was in power-saving mode, set to an implementation-specific value. If the interrupt occurred when the thread was in power-saving mode, set to indicate the exception that caused exit from power-saving mode as shown below:

$SRR1_{42:45}$	Exception
0000	Reserved
0001	Reserved
0010	Implementation specific
0011	Directed Hypervisor Doorbell
0100	System Reset
0101	Directed Privlgd Doorbell
0110	Decrementer
0111	Reserved
1000	External
1001	Hypervisor Virtualization
1010	Hypervisor Maintenance
1011	Reserved
1100	Implementation specific
1101	Reserved
1110	Implementation specific
1111	Reserved

If multiple events that cause exit from power-saving mode exist, the event reported is the exception corresponding to the interrupt that would have occurred if the same conditions existed and the thread was not in power-saving mode.

46:47 Set to indicate whether the interrupt occurred when the thread was in power-saving mode and, if so, the extent to which resource state was maintained while the thread was in power-saving mode, as follows:

00 The interrupt did not occur when the thread was in power-saving mode.

01 The interrupt occurred when the thread was in power-saving mode. The state of all resources was maintained as if the thread was not in power-saving mode.

10 The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, but the state of all hypervisor resources, including the DEC, HDEC, TB, PURR, SPURR, and VTB, was maintained as if the thread was not in power-saving mode and the state of all other resources is such that the hypervisor can resume execution. (See Section 2.6 for the list of hypervisor resources.)

11 The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, and the state of some hypervisor resources was not maintained or the state of some resources is such that the hypervisor cannot resume execution.

Programming Note

Although the resources that are maintained in power-saving levels that allow loss of state are implementation-dependent, the hypervisor can avoid implementation-dependence in the portion of the System Reset and Machine Check interrupt handlers that recover from having been in power-saving mode by using the contents of $SRR1_{46:47}$, to determine what state to restore. (To avoid implementation-dependence, the hypervisor must assume that only the resources indicated in $SRR1_{46:47}$ have been preserved.

62 If the interrupt did not occur while the thread was in a power-saving level that was entered when $PSSCR_{EC}$=1, loaded from bit 62 of the MSR if the thread is in a recoverable state; otherwise set to 0. If the interrupt occurred while the thread was in a power-saving level that was entered when $PSSCR_{EC}$=1, set to 1 if the thread is in a recoverable state; otherwise set to 0.

Others Loaded from the MSR.

MSR See Figure 65 on page 1064.

In addition, if the interrupt occurs when the thread is in a power-saving level that was entered when $PSSCR_{EC}$=1 and is caused by an exception other than a System Reset exception, all other registers, except HSRR0 and HSRR1, that would be set by the corresponding interrupt if the exception occurred when the thread was not in power-saving mode are set by the System Reset interrupt, and are set to the values to which they would be set if the exception occurred when the thread was not in power-saving mode.

Execution resumes at effective address 0x0000_0000_0000_0100.

The means for software to distinguish between power-on Reset and other types of System Reset are implementation-dependent.

6.5.2 Machine Check Interrupt

The causes of Machine Check interrupts are implementation-dependent. For example, a Machine Check interrupt may be caused by a reference to a storage location that contains an uncorrectable error or does not exist (see Section 5.6), or by an error in the storage subsystem.

When the thread is not in power-saving mode, Machine Check interrupts are enabled when MSR_{ME}=1; if MSR_{ME}=0 and a Machine Check exception occurs, the thread enters the Checkstop state. When the thread is in a power-saving level that does not allow loss of hypervisor state, Machine Check interrupts are treated as enabled when $LPCR_{51}$=1 and cannot occur when $LPCR_{51}$=0. When the thread is in a power-saving level that allows loss of hypervisor state, it is implementation-specific whether Machine Check interrupts are treated as enabled $LPCR_{51}$=1 or if they cannot occur. If a Machine Check exception occurs while the thread is in power-saving mode and the Machine Check exception is not enabled to cause exit from power-saving mode, the result is implementation specific.

The Checkstop state may also be entered if an access is attempted to a storage location that does not exist (see Section 5.6), or if an implementation-dependent hardware error occurs that prevents continued operation.

Disabled Machine Check (Checkstop State)

When a thread is in Checkstop state, instruction processing is suspended and generally cannot be restarted without resetting the thread. Some implementations may preserve some or all of the internal state of the thread when entering Checkstop state, so that the state can be analyzed as an aid in problem determination.

Enabled Machine Check

If a Machine Check exception causes an interrupt that is not context synchronizing or causes the loss of a Direct External exception, or if the state of the thread has been corrupted, the interrupt is not recoverable.

The following registers are set:

SRR0 If the interrupt occurred when the thread was in a power-saving mode that was entered with PSSCR bit ESL=0, and fields RL, MTL, and PSLL set to values that do not allow state loss, set on a "best effort" basis to the effective address of some instruction that was executing or was about to be executed when the Machine Check exception occurred; otherwise set to an undefined value.

SRR1

46:47 Set to indicate whether the interrupt occurred when the thread was in power-saving mode and, if so, the extent to which resource state was maintained while the thread was in power-saving mode, as follows.

- 00 The interrupt did not occur when the thread was in power-saving mode.
- 01 The interrupt occurred when the thread was in power-saving mode. The state of all resources was maintained as if the thread was not in power-saving mode.
- 10 The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, but the state of all hypervisor resources, including the DEC, HDEC, TB, PURR, SPURR, and VTB, was maintained as if the thread was not in power-saving mode and the state of all other resources is such that the hypervisor can resume execution. (See Section 2.6 for the list of hypervisor resources.)
- 11 The interrupt occurred when the thread was in power-saving mode. The state of some resources was not maintained, and the state of some hypervisor resources was not maintained or the state of some resources is such that the hypervisor cannot resume execution.

Programming Note

Although the resources that are maintained in power-saving mode (except when all resources are maintained) are implementation-dependent, the hypervisor can avoid implementation-dependence in the portion of the System Reset and Machine Check interrupt handlers that recover from having been in power-saving mode by using the contents of $SRR1_{46:47}$, to determine what state to restore. (To avoid implementation-dependence in the portion of the hypervisor that enters power-saving mode, the hypervisor must use the specification of the four instructions to determine what state to save.)

62 If the interrupt did not occur while the thread was in a power-saving level that was entered when $PSSCR_{EC}$=1, loaded from bit 62 of the MSR if the thread is in a recoverable state; otherwise set to 0. If the interrupt occurred while the thread was in a power-saving level that was entered when $PSSCR_{EC}$=1, set to 1 if the thread is in a recoverable state; otherwise set to 0.

Others Set to an implementation-dependent value.

MSR See Figure 65.

DSISR Set to an implementation-dependent value.

DAR Set to an implementation-dependent value.

ASDR Set to an implementation-dependent value.

Execution resumes at effective address 0x0000_0000_0000_0200.

A Machine Check interrupt caused by the existence of multiple SLB entries or TLB entries (or similar entries in implementation-specific translation caches) which translate a given effective or virtual address (see Sections 5.7.8.2 and 5.7.9.2.) must occur while still in the context of the partition that caused it. The interrupt must be presented in a way that permits continuing execution, with damage limited to the causing partition. Treating the exception as instruction-caused will achieve these requirements.

Programming Note

If a Machine Check interrupt is caused by an error in the storage subsystem, the storage subsystem may return incorrect data, which may be placed into registers. This corruption of register contents may occur even if the interrupt is recoverable.

6.5.3 Data Storage Interrupt

A Data Storage interrupt occurs when no higher priority exception exists and either

(a) a copy-paste transfer other than from main storage to a properly initiated accelerator is attempted, or

(b) ($MSR_{HV\ PR}$=0b10) & (MSR_{DR}=0)) , or

(c) HPT translation is being performed, the value of the expression

(($MSR_{HV\ PR}$=0b10) | ((¬VPM|¬$PRTE_V$)& MSR_{DR}))

is 1, and a data access cannot be performed,

except for the case of $MSR_{HV\ PR}$≠0b10,

VPM=0, $LPCR_{KBV}$=1, and a Virtual Storage

Page Class Key Protection exception exists or

(d) Radix Tree translation is being performed, and

either a Data Address Watchpoint match occurs, an

attempt is made to execute an AMO with an invalid

function code, or process-scoped translation either

does not complete or prevents the data access from

being performed

for any of the following reasons that can occur in the respective translation state. (In the expression for (a) above, "¬$PRTE_V$" is shorthand representing the case of an invalid segment table descriptor stopping the translation process.)

- Data address translation is enabled (MSR_{DR}=1) and the effective or virtual address of any byte of the storage location specified by a *Load*, *Store*, ***icbi***, ***dcbz***, ***dcbst***, or ***dcbf***[***l***] instruction cannot be translated to a real address because no valid PTE was found for the process-scoped Radix Tree translation or HPT translation with VPM off.
- The address of the appropriate process table entry or segment table entry group cannot be translated when HR=0 and either VPM=0 or the process table entry is invalid (independent of VPM).
- The effective address specified by a ***lq***, ***stq***, ***lwat***, ***ldat***, ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, ***lqarx***, ***stwat***, ***stdat***, ***stbcx***., ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** instruction refers to storage that is Write Through Required or Caching Inhibited; or the effective address specified by a ***copy*** or ***paste.*** instruction refers to storage that is Caching Inhibited; or the effective address specified by a ***lwat***, ***ldat***, ***stwat***, or ***stdat*** instruction refers to storage that is Guarded.
- An accelerator is specified as the source of a ***copy*** instruction, normal memory is specified at the target of a ***paste.*** instruction, or an attempt is made to access an accelerator that is not properly configured for the software's use.
- The access violates Basic Storage Protection.
- The access violates Virtual Page Class Key Storage Protection and $LPCR_{KBV}$=0.
- The process- and partition-scoped page attributes conflict.
- An unsupported radix tree configuration is found in the process-scoped tables.
- A reference or change bit update cannot be performed in a process-scoped PTE.
- A Data Address Watchpoint match occurs.
- An attempt is made to execute a *Load Atomic* or *Store Atomic* instruction with an invalid function code.
- An attempt is made to execute a *Fixed-Point Load or Store Caching Inhibited* instruction with MSR_{DR}=1 or specifying a storage location that is specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded.

A Data Storage interrupt also occurs when no higher priority exception exists and an attempt is made to execute a *Load Atomic* or *Store Atomic* instruction specifying an invalid function code.

Programming Note

When an attempt to execute a *Load Atomic* or *Store Atomic* instruction containing an invalid function code (see Figures 3 and 4 in Book II) causes a DSI, the condition is very similar to an invalid form of an instruction. As a result, this instance of DSI occurs with a high prioirty that blocks the translation process and prevents Reference and Change bit updates.

If a ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** would not perform its store in the absence of a Data Storage interrupt, and either (a) the specified effective address refers to storage that is Write Through Required or Caching Inhibited, or (b) a non-conditional *Store* to the specified effective address would cause a Data Storage interrupt, it is implementation-dependent whether a Data Storage interrupt occurs.

If the XER specifies a length of zero for an indexed *Move Assist* instruction, a Data Storage interrupt does not occur.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33:36	Set to 0.
42:47	Set to 0.
Others	Loaded from the MSR.

MSR See Figure 65.

DSISR

32	Set to 0.

33 Set to 1 if MSR_{DR}=1 and the translation for an attempted access is not found in the Page Table; otherwise set to 0..

34 Set to 1 if the process- and partition-scoped page attributes conflict; otherwise set to 0.

35 Set to 0.

36 Set to 1 if the access is not permitted by Figure 44 46, or the privilege, read, or read/write bits in Figure 45 as appropriate; otherwise set to 0.

37 Set to 1 if the access is due to a ***lq***, ***stq***, ***lwat***, ***ldat***, ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, ***lqarx***, ***stwat***, ***stdat***, ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** instruction that addresses storage that is Write Through Required or Caching Inhibited; or if the access is due to a ***copy*** or ***paste.*** instruction that addresses storage that is Caching Inhibited; or if the access is due to a ***lwat***, ***ldat***, ***stwat***, or ***stdat*** instruction that addresses storage that is Guarded; otherwise set to 0.

38 Set to 1 for a *Store*, ***dcbz***, or *Load/Store Atomic* instruction; otherwise set to 0.

39:40 Set to 0.

41 Set to 1 if a Data Address Watchpoint match occurs; otherwise set to 0.

42 Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.

43 Set to 0.

44 Set to 1 if an unsupported radix tree configuration is found during the translation process; otherwise set to 0.

45 Set to 1 if an attempt to atomically set a reference or change bit fails; otherwise set to 0.

Programming Note

The number of attempts hardware makes to atomically set reference and change bits before triggering this exception is implementation dependent. The POWER9 processor makes no attempt. Software may still support the atomic update programming model to get performance benefits such as those described in Section 5.7.12.

46 Set to 1 if the address of the appropriate process table entry or segment table entry group cannot be translated when VPM=0 and HR=0, or the process table entry is invalid (independent of VPM) when HR=0.

47:59 Set to 0.

60 Set to 1 if an accelerator is specified as the source of a ***copy*** instruction, normal memory is specified as the target of a ***paste.*** instruction, or an attempt is made to access an accelerator that is not properly configured for the software's use; otherwise set to 0. These exceptions are presented differently from most instruction-caused exceptions. See Section 4.4, "Copy-Paste Facility", in Book II for details. Additional information may be retained by the platform if the accelerator is not properly configured.

61 Set to 1 if an attempt is made to execute a *Load Atomic* or *Store Atomic* instruction specifying an invalid function code; otherwise set to 0.

62 Set to 1 if an attempt is made to execute a *Fixed-Point Load or Store Caching Inhibited* instruction with MSR_{DR}=1 or specifying a storage location that is specified by the Hypervisor Real Mode Storage Control facility to be treated as non-Guarded.

63 Set to 0.

DAR Set to the effective address of a storage element as described in the following list. The list should be read from the top down; the DAR is set as described by the first item that corresponds to an exception that is reported in the DSISR. For example, if a *Load Word* instruction causes a storage protection violation and a Data Address Watchpoint match (and both are reported in the DSISR), the DAR is set to the effective address of a byte in the first aligned doubleword for which access was attempted in the page that caused the exception.

- undefined, for *Load Atomic* or *Store Atomic* instruction specifying an invalid function code
-
- undefined, when $DSISR_{60}$=1
- a Data Storage exception occurs for reasons other than a Data Address Watchpoint match
  - a byte in the block that caused the exception, for a *Cache Management* instruction
  - a byte in the first aligned quadword for which access was attempted in the page that caused the exception, for a quadword *Load* or *Store* instruction (i.e., a *Load* or *Store* instruction for which the storage operand is a quadword; "first" refers to address order: see Section 6.7)
  - a byte in the first aligned doubleword for which access was attempted in the page that caused the exception, for a non-quadword *Load* or *Store* instruction
- set as described in the previous major bullet, except that the low order 5 bits are undefined, for a Data Address Watchpoint match

For the cases in which the DAR is specified above to be set to a defined value, if the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

If multiple Data Storage exceptions occur for a given effective address, any one or more of the bits corresponding to these exceptions may be set to 1 in the DSISR. However, if one or more DSI-causing exceptions occur together with a Virtualized Page Class Key Storage Protection exception that occurs when $LPCR_{KBV}$=1 and Virtualized Partition Memory is disabled by VPM=0, an HDSI results, and all of the exceptions are reported in the HDSISR.

Execution resumes at effective address 0x0000_0000_0000_0300, possibly offset as specified in Figure 66.

6.5.4 Data Segment Interrupt

For Paravirtualized HPT Translation, a Data Segment interrupt occurs when no higher priority exception exists and a data access cannot be performed because data address translation is enabled and the effective address of any byte of the storage location specified by a *Load*, *Store*, ***icbi***, ***dcbz***, ***dcbst***, or ***dcbf***[***l***] instruction cannot be translated to a virtual address.

For Radix Tree Translation (in other than hypervisor real mode), a Data Segment interrupt occurs when no higher priority exception exists and a data access cannot be performed because for the effective address specified by a *Load*, *Store*, ***icbi***, ***dcbz***, ***dcbst***, or ***dcbf***[***l***] instruction, $EA_{0:1}$=0b01 or $EA_{0:1}$=0b10 when $MSR_{HV\ PR} \neq$ 0b10 and data address translation is enabled, or $EA_{2:63}$ is outside the range translated by the appropriate Radix Tree.

If a ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** would not perform its store in the absence of a Data Segment interrupt and a non-conditional *Store* to the specified effective address would cause a Data Segment interrupt, it is implementation-dependent whether a Data Segment interrupt occurs.

If the XER specifies a length of zero for an indexed *Move Assist* instruction, a Data Segment interrupt does not occur.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33:36	Set to 0.
42:47	Set to 0.
Others	Loaded from the MSR.

MSR See Figure 65.

DSISR Set to an undefined value.

DAR Set to the effective address of a storage element as described in the following list.

- a byte in the block that caused the exception, for a *Cache Management* instruction
- a byte in the first aligned quadword for which access was attempted in the segment that caused the exception, for a quadword *Load* or *Store* instruction (i.e., a *Load* or *Store* instruction for which the storage operand is a quadword; "first" refers to address order: see Section 6.7)
- a byte in the first aligned doubleword for which access was attempted in the segment that caused the exception, for a non-quadword *Load* or *Store* instruction

If the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

Execution resumes at effective address 0x0000_0000_0000_0380, possibly offset as specified in Figure 66.

Programming Note

A Data Segment interrupt occurs if MSR_{DR}=1 and the translation of the effective address of any byte of the specified storage location is not found in the SLB (or in any implementation-specific address translation lookaside information).

6.5.5 Instruction Storage Interrupt

An Instruction Storage interrupt occurs when no higher priority exception exists and either

(a) HPT Translation is being performed, the value of the expression

$((MSR_{HV\ PR}=0b10) \mid (\ (\neg VPM \mid \neg PRTE_V) \& MSR_{IR}))$

is 1, and the next instruction to be executed cannot be fetched, or

(b) Radix Tree translation is being performed and process-scoped translation prevents the next instruction to be executed from being fetched

for any of the following reasons. (In the expression for (a) above, "$\neg PRTE_V$" is shorthand representing the case of an invalid segment table descriptor stopping the translation process.)

- Instruction address translation is enabled and the effective or virtual address cannot be translated to a real address because no valid PTE was found for the process-scoped Radix Tree translation or HPT translation with VPM off.

- The address of the appropriate process table entry or segment table entry group cannot be translated when HR=0 and either VPM=0 or the process table entry is invalid (independent of VPM).
- The fetch access violates storage protection.
- The process- and partition-scoped page attributes conflict.
- An unsupported radix tree configuration is found in the process-scoped tables.
- A reference bit update cannot be performed in a process-scoped PTE.

The following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, SRR0 is set to the branch target address).

SRR1

33 Set to 1 if MSR_{IR}=1 and the translation for an attempted access is not found in the Page Table; otherwise set to 0.

34 Set to 1 if the process- and partition-scoped page attributes conflict; otherwise set to 0.

35 Set to 1 if the access is to No-execute (as indicated by the N bit in the segment table entry or the N bit in the HPT PTE or the Execute and Privilege bits in the EAA field of the Radix PTE and IAMR key 0) or Guarded storage; otherwise set to 0.

36 Set to 1 if the access is not permitted by Figure 44 or 46, as appropriate; otherwise set to 0.

42 Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.

43 Set to 0.

44 Set to 1 if an unsupported radix tree configuration is found during the translation process; otherwise set to 0.

45 Set to 1 if an attempt to atomically set a reference bit fails; otherwise set to 0.

Programming Note

The number of attempts hardware makes to atomically set reference and change bits before triggering this exception is implementation dependent. The POWER9 processor makes no attempt. Software may still support the atomic update programming model to get performance benefits such as those described in Section 5.7.12.

46 Set to 1 if the address of the appropriate process table entry or segment table entry group cannot be translated when VPM=0 and HR=0, or the process table entry is invalid (independent of VPM) when HR=0.

47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 65.

If multiple Instruction Storage exceptions occur due to attempting to fetch a single instruction, any one or more of the bits corresponding to these exceptions may be set to 1 in SRR1.

Execution resumes at effective address 0x0000_0000_0000_0400, possibly offset as specified in Figure 66.

6.5.6 Instruction Segment Interrupt

For Paravirtualized HPT Translation, an Instruction Segment interrupt occurs when no higher priority exception exists and the next instruction to be executed cannot be fetched because instruction address translation is enabled and the effective address cannot be translated to a virtual address.

For Radix Tree Translation (in other than hypervisor real mode), an Instruction Segment interrupt occurs when no higher priority exception exists and the next instruction to be executed cannot be fetched because $EA_{0:1}$=0b01 or $EA_{0:1}$=0b10 when $MSR_{HV\ PR} \neq$ 0b10 and instruction address translation is enabled, or $EA_{2:63}$ is outside the range translated by the appropriate Radix Tree.

The following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, SRR0 is set to the branch target address).

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0480, possibly offset as specified in Figure 66.

Programming Note

An Instruction Segment interrupt occurs if MSR_{IR}=1 and the translation of the effective address of the next instruction to be executed is not found in the SLB (or in any implementation-specific address translation lookaside information).

6.5.7 External Interrupt

An External interrupt is classified as being either a Direct External interrupt or a Mediated External interrupt. Throughout this Book, usage of the phrase “External interrupt’, without further classification, refers to both a Direct External interrupt and a Mediated External interrupt.

6.5.7.1 Direct External Interrupt

A Direct External interrupt occurs when no higher priority exception exists, a Direct External exception exists, and the value of the expression

MSR_{EE} & ¬(MSR_{HV} & ¬MSR_{PR} & $LPCR_{HEIC}$) | (¬(LPES) & (¬(MSR_{HV}) | MSR_{PR}))

is one. The occurrence of the interrupt does not cause the exception to cease to exist.

Programming Note

When HEIC=1, Direct External exceptions will not result in external interrupts when the processor is in hypervisor state even if MSR_{EE}=1. This enables the Hypervisor Interrupt Virtualization handler to prevent External interrupts from occurring during the Hypervisor Virtualization interrupt handler.

When LPES=0, the following registers are set:

HSRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

HSRR1

33:36	Set to 0.
42:47	Set to 0.
Others	Loaded from the MSR.

MSR See Figure 65 on page 1064.

When LPES=1, the following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

SRR1

33:36	Set to 0.
42:47	Set to 0.
Others	Loaded from the MSR.

MSR See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0500, possibly offset as specified in Figure 66.

Programming Note

Because the value of MSR_{EE} is always 1 when the thread is in problem state, the simpler expression

MSR_{EE} & ¬(MSR_{HV} & ¬MSR_{PR} & $LPCR_{HEIC}$) | ¬(LPES | MSR_{HV})

is equivalent to the expression given above.

Programming Note

The Direct External exception has the same meaning as the External exception in versions of the architecture prior to Version 2.05.

6.5.7.2 Mediated External Interrupt

A Mediated External interrupt occurs when no higher priority exception exists, a Mediated External exception exists (see the definition of $LPCR_{MER}$ in Section 2.2), and the value of the expression

$$MSR_{EE} \text{ \& } (\neg(MSR_{HV}) \mid MSR_{PR})$$

is one. The occurrence of the interrupt does not cause the exception to cease to exist.

When LPES=0, the following registers are set:

HSRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

HSRR1

33:36	Set to 0.
42	Set to 1.
43:47	Set to 0.
Others	Loaded from the MSR.

MSR See Figure 65 on page 1064.

When LPES=1, the following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

SRR1

33:36	Set to 0.
42:47	Set to 0.
Others	Loaded from the MSR.

MSR See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0500, possibly offset as specified in Figure 66.

6.5.8 Alignment Interrupt

Many causes of Alignment interrupt involve storage operand alignment. Storage operand alignment is defined in Section 1.11.1 of Book I.

An Alignment interrupt occurs when no higher priority exception exists and an attempt is made to execute an instruction in a manner that is required, by the instruction description, to cause an Alignment interrupt. These cases are as follows.

- A *Load/Store Multiple* instruction that is executed in Little-Endian mode
- A *Move Assist* instruction that is executed in Little-Endian mode, unless the string length is zero
- A ***copy***, ***paste.***, ***lwat***, ***ldat***, ***lharx***, ***lwarx***, ***ldarx***, ***lqarx***, ***stwat***, ***stdat***, ***sthcx***., ***stwcx.***, ***stdcx.***, or ***stqcx.*** instruction that has an unaligned storage operand, unless execution of the instruction yields boundedly undefined results
- The operand(s) of a *Load Atomic* or *Store Atomic* instruction cross(es) a 32-byte boundary.

An Alignment interrupt may occur when no higher priority exception exists and a data access cannot be performed for any of the following reasons.

- The storage operand of ***lfdp***, ***lfdpx***, ***stfdp***, ***stfdpx***, ***lxsihzx***, or ***stxsihx*** is unaligned.
- The storage operand of ***lq*** or ***stq*** is unaligned.
- The storage operand of a Floating-Point *Storage Access* or VSX *Storage Access* instruction other than ***lfdp***, ***lfdpx***, ***stfdp***, ***stfdpx, lxsihzx, lxsibzx, stxsihx,*** or ***stxsibx*** is not word-aligned.
- The storage operand of a *Load/Store Multiple Word* instruction is not word-aligned and the thread is in Big-Endian mode.
- The storage operand of a *Load/Store Multiple Doubleword* instruction is not doubleword-aligned and the thread is in Big-Endian mode.
- The storage operand of a *Load/Store Multiple, **lfdp***, ***lfdpx***, ***stfdp***, ***stfdpx***, or ***dcbz*** instruction is in storage that is Write Through Required or Caching Inhibited.
- The storage operand of a *Move Assist* instruction is in storage that is Write Through Required or Caching Inhibited and has length greater than zero.
- The storage operand of a *Load* or *Store* instruction is unaligned and is in storage that is Write Through Required or Caching Inhibited.
- The storage operand of a *Storage Access* instruction crosses a segment boundary, or crosses a boundary between virtual pages that have different storage control attributes.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33:36 Set to 0.
42:47 Set to 0.
Others Loaded from the MSR.

MSR See Figure 65.

DAR Set to the effective address computed by the instruction, except that if the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

Execution resumes at effective address 0x0000_0000_0000_0600, possibly offset as specified in Figure 66.

Programming Note

If an Alignment interrupt occurs for a case in the second bulleted list above, the Alignment interrupt handler should emulate the instruction. The emulation must satisfy the atomicity requirements described in Section 1.4 of Book II.

If an Alignment interrupt occurs for a case in the first bulleted list above, the Alignment interrupt handler must not attempt to emulate the instruction, but instead should treat the instruction as a programming error.

6.5.9 Program Interrupt

A Program interrupt occurs when no higher priority exception exists and one of the following exceptions arises during execution of an instruction:

Floating-Point Enabled Exception

A Floating-Point Enabled Exception type Program interrupt is generated when the value of the expression

$$(MSR_{FE0} \mid MSR_{FE1}) \ \& \ FPSCR_{FEX}$$

is 1. $FPSCR_{FEX}$ is set to 1 by the execution of a floating-point instruction that causes an enabled exception, including the case of a *Move To FPSCR* instruction that causes an exception bit and the corresponding enable bit both to be 1.

TM Bad Thing

A TM Bad Thing type Program interrupt is generated when any of the following occurs.

- An ***rfebb***, ***rfid***, ***rfscv***, ***hrfid***, or ***mtmsrd*** instruction attempts to cause an illegal transaction state transition (see Section 3.2.2).
- An ***rfid***, ***rfscv***, ***hrfid***, or ***mtmsrd*** instruction, executed when TM is made unavailable in problem state by the PCR ($PCR_{v2.06}$=1), attempts to cause a transition to problem state and also a transaction state transition that Table 3 on page 947 shows as legal and as resulting in the thread being in Transactional or Suspended state.

- An attempt is made to execute ***trechkpt.*** in Transactional or Suspended state or when $TEXASR_{FS}$=0.
- An attempt is made to execute ***tend.*** in Suspended state.
- An attempt is made to execute ***treclaim.*** in Non-transactional state.
- An attempt is made to execute an ***mtspr*** instruction targeting a TM register in other than Non-transactional state, with the exception of TFHAR in Suspended state.
- An attempt is made to execute a ***stop*** instruction in Suspended state.

Privileged Instruction

The following applies if the instruction is executed when MSR_{PR} = 1.

A Privileged Instruction type Program interrupt is generated when execution is attempted of a privileged instruction, or of an ***mtspr*** or ***mfspr*** instruction with an SPR field that contains a value having spr_0=1.

The following applies if the instruction is executed when $MSR_{HV\ PR}$ = 0b00 and $LPCR_{EVIRT}$=0.

A Privileged Instruction type Program interrupt is generated when execution is attempted of an ***mtspr*** or ***mfspr*** instruction with an SPR field that designates an SPR that is accessible by the instruction only when the thread is in hypervisor state, or when execution of a hypervisor-privileged instruction is attempted.

Programming Note

These are the only cases in which a Privileged Instruction type Program interrupt can be generated when MSR_{PR}=0. They can be distinguished from other causes of Privileged Instruction type Program interrupts by examining $SRR1_{49}$ (the bit in which MSR_{PR} was saved by the interrupt).

Trap

A Trap type Program interrupt is generated when any of the conditions specified in a *Trap* instruction is met.

The following registers are set:

SRR0 For all Program interrupts except a Floating-Point Enabled Exception type Program interrupt, set to the effective address of the instruction that caused the corresponding exception.

For a Floating-Point Enabled Exception type Program interrupt, set as described in the following list.

- If $MSR_{FE0\ FE1}$ = 0b00, $FPSCR_{FEX}$ = 1, and an instruction is executed that changes $MSR_{FE0\ FE1}$ to a nonzero value, set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

Programming Note

Recall that all instructions that can alter $MSR_{FE0\ FE1}$ are context synchronizing, and therefore are not initiated until all preceding instructions have reported all exceptions they will cause.

- If $MSR_{FE0\ FE}$ = 0b11, set to the effective address of the instruction that caused the Floating-Point Enabled Exception.
- If $MSR_{FE0\ FE}$ = 0b01 or 0b10, set to the effective address of the first instruction that caused a Floating-Point Enabled Exception since the most recent time $FPSCR_{FEX}$ was changed from 1 to 0 or of some subsequent instruction.

Programming Note

If SRR0 is set to the effective address of a subsequent instruction, that instruction will not be beyond the first such instruction at which synchronization of floating-point instructions occurs. (Recall that such synchronization is caused by *Floating-Point Status and Control Register* instructions, as well as by execution synchronizing instructions and events.)

SRR1

33:36 Set to 0.

42 Set to 1 for a TM Bad Thing type Program interrupt; otherwise set to 0.

43 Set to 1 for a Floating-Point Enabled Exception type Program interrupt; otherwise set to 0.

44 Set to 0.

45 Set to 1 for a Privileged Instruction type Program interrupt; otherwise set to 0.

46 Set to 1 for a Trap type Program interrupt; otherwise set to 0.

47 Set to 0 if SRR0 contains the address of the instruction causing the exception and there is only one such instruction; otherwise set to 1.

Programming Note

$SRR1_{47}$ can be set to 1 only if the exception is a Floating-Point Enabled Exception and either $MSR_{FE0\ FE1}$ = 0b01 or 0b10 or $MSR_{FE0\ FE1}$ has just been changed from 0b00 to a nonzero value. ($SRR1_{47}$ is always set to 1 in the last case.)

Others Loaded from the MSR.

Exactly one of bits 42, 43, 45, and 46 is set to 1.

MSR See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0700, possibly offset as specified in Figure 66.

Programming Note

In versions of the architecture that precede V. 2.05, the conditions that now cause a Hypervisor Emulation Assistance interrupt with $HSRR1_{45}$=0 instead caused an "Illegal Instruction type Program interrupt". This was a Program interrupt for which registers (SRR0, SRR1, and the MSR) were set as described above for the Privileged Instruction type Program interrupt, except that $SRR1_{44}$ was set to 1 and $SRR1_{45}$ was set to 0. Thus older operating systems have code to handle these conditions, at the Program interrupt vector location. For this reason, if a Hypervisor Emulation Assistance interrupt occurs with $HSRR1_{45}$=0 when the thread is not in hypervisor state, for an instruction that the hypervisor determines should be handled by the operating system, the hypervisor is expected to pass control to the operating system at the operating system's Program interrupt vector location, with all registers (SRR0, SRR1, MSR, GPRs, etc.) set as if the instruction had caused a Privileged Instruction type Program interrupt, except with $SRR1_{44:45}$ set to 0b10. (The Hypervisor Emulation Assistance interrupt was added to the architecture in V. 2.05, and the Illegal Instruction type Program interrupt was removed from the architecture in V. 2.06. In V. 2.05 the Hypervisor Emulation Assistance interrupt was optional: implementations that supported it generated it as described in V. 2.06, and never generated an Illegal Instruction type Program interrupt; implementations that did not support it generated an Illegal Instruction type Program interrupt as described above.)

Programming Note

When $LPCR_{EVIRT}$=1, some of the conditions that cause a Privileged Instruction type Program interrupt when $LPCR_{EVIRT}$=0 (attempted execution, in privileged but non-hypervisor state, of a hypervisor privileged instruction or of an ***mtspr*** or ***mfspr*** instruction specifying an SPR that is hypervisor privileged for the operation) instead cause a Hypervisor Emulation Assistance interrupt with $HSRR1_{45}$=1. Having these conditions cause a Hypervisor Emulation Assistance interrupt permits support of nested hypervisors through virtualization of hypervisor resources, and simplifies creation of a common kernel for the OS and the hypervisor. In versions of the architecture that precede V. 3.0, $LPCR_{EVIRT}$ did not exist and these conditions always caused a Privileged Instruction type Program interrupt. Thus older operating systems have code to handle these conditions, at the Program interrupt vector location. For this reason, if a Hypervisor Emulation Assistance interrupt occurs with $HSRR1_{45}$=1 for an instruction that the hypervisor determines should be handled by the operating system, the hypervisor is expected to pass control to the operating system at the operating system's Program interrupt vector location, with all registers (SRR0, SRR1, MSR, GPRs, etc.) set as if the instruction had caused a Privileged Instruction type Program interrupt.

6.5.10 Floating-Point Unavailable Interrupt

A Floating-Point Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point loads, stores, and moves), and MSR_{FP}=0.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0800, possibly offset as specified in Figure 66.

6.5.11 Decrementer Interrupt

A Decrementer interrupt occurs when no higher priority exception exists, a Decrementer exception exists, and MSR_{EE}=1.

The following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

SRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0900, possibly offset as specified in Figure 66.

6.5.12 Hypervisor Decrementer Interrupt

A Hypervisor Decrementer interrupt occurs when no higher priority exception exists, a Hypervisor Decrementer exception exists, and the value of the following expression is 1.

$(MSR_{EE} \mid \neg(MSR_{HV}) \mid MSR_{PR})$ & HDICE

The following registers are set:

HSRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

HSRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0980, possibly offset as specified in Figure 66.

Programming Note

Because the value of MSR_{EE} is always 1 when the thread is in problem state, the simpler expression

$(MSR_{EE} \mid \neg(MSR_{HV}))$ & HDICE

is equivalent to the expression given above.

6.5.13 Directed Privileged Doorbell Interrupt

A Directed Privileged Doorbell interrupt occurs when no higher priority exception exists, a Directed Privileged Doorbell exception is present, and MSR_{EE}=1. Directed Privileged Doorbell exceptions are generated when Directed Privileged Doorbell messages (see Chapter 10) are received and accepted by the thread.

The following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

SRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR **S**ee Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0A00, possibly offset as specified in Figure 66.

6.5.14 System Call Interrupt

A System Call interrupt occurs when a *System Call* instruction is executed.

The following registers are set:

SRR0 Set to the effective address of the instruction following the System Call instruction.

SRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0C00, possibly offset as specified in Figure 66.

Programming Note

An attempt to execute an ***sc*** instruction with LEV=1 in problem state should be treated as a programming error.

6.5.15 Trace Interrupt

A Trace interrupt occurs when no higher priority exception exists and any instruction except ***rfid***, ***hrfid, rfscv,*** or a *Power-Saving Mode* instruction is successfully completed, provided any of the following is true:

- the instruction is ***mtmsr[d]*** and MSR_{TE}=0b10 when the instruction was initiated,
- the instruction is not ***mtmsr[d]*** and MSR_{TE}=0b10,
- the instruction is a *Branch* instruction and MSR_{TE}=0b01, or
- a CIABR match occurs.

Successful completion for an instruction means that the instruction caused no other interrupt and, if the thread

is in Transactional state, did not cause the transaction to fail in such a way that the instruction did not complete (see Section 5.3.1 of Book II). Thus a Trace interrupt never occurs for a *System Call* or *System Call Vectored* instruction, or for a *Trap* instruction that traps, or for a ***dcbf*** that is executed in Transactional state. The instruction that causes a Trace interrupt is called the "traced instruction".

The following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

SRR1

33 Set to 1.

34 Set to 0.

35 Set to 1 if the the Trace interrupt is not the result of a CIABR match and the traced instruction is a *Load* instruction other than a *Load String* instruction with string length of 0 or is specified to be treated as a *Load* instruction; otherwise set to 0.

36 Set to 1 if the the Trace interrupt is not the result of a CIABR match and the traced instruction is a *Store* instruction other than a *Store String* instruction with string length of 0 or is specified to be treated as a *Store* instruction; otherwise set to 0.

43 Set to 1 if the traced instruction is the result of a CIABR match.

44:47 Set to 0.

Others Loaded from the MSR.

Programming Note

Bit 33 is set to 1 for historical reasons.

SIAR For all Trace interrupts other than those caused by a CIABR match, set to the effective address of the traced instruction; otherwise undefined.

SDAR For all Trace interrupts other than those caused by a CIABR match, set to the effective address of the storage operand (if any) of the traced instruction; otherwise undefined.

If the state of the Performance Monitor is such that the Performance Monitor may be altering the SIAR and SDAR (i.e., if $MMCR0_{PMAE}$=1), the contents of the SIAR and SDAR are undefined for the Trace interrupt and may change even when no Trace interrupt occurs.

MSR See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_00D0, possibly offset as specified in Figure 66. For a Trace interrupt resulting from execution of an instruction that modifies the value of MSR_{IR}, MSR_{DR}, MSR_{HV}, or $LPCR_{AIL}$, the Trace interrupt vector location is based on the modified values.

Programming Note

The following instructions are not traced.

- ***rfid***
- ***hrfid***
- ***rfscv***
- ***sc***, ***scv***, and *Trap* instructions that trap
- *Power-Saving Mode* instructions
- other instructions that cause interrupts (other than Trace interrupts)
- the first instructions of any interrupt handler
- instructions that are emulated by software
- instructions, executed in Transactional state, that are disallowed in Transactional state
- instructions, executed in Transactional state, that cause types of accesses that are disallowed in Transactional state
- ***mtspr***, executed in Transactional state, specifying an SPR that is not the GSR and is not part of the checkpointed registers
- ***tbegin.*** executed at maximum nesting depth

In general, interrupt handlers can achieve the effect of tracing these instructions.

6.5.16 Hypervisor Data Storage Interrupt

A Hypervisor Data Storage interrupt occurs when no higher priority exception exists, either the thread is not in hypervisor state or an unsupported MMU configuration has been found or the access has been prevented by a problem in partition-scoped Radix Tree translation, and either

(a) HPT translation is being performed, VPM=0,

$LPCR_{KBV}$=1, and a Virtual Storage Page Class Key

Protection exception exists or

(b) HPT translation is being performed, the value of the

expression $(\neg MSR_{DR})$ | (VPM & $PRTE_V$ & MSR_{DR})

is 1, and a data access cannot be performed, or

(c) Radix Tree translation is being performed and

partition-scoped translation either does not complete

or prevents an access from being performed

for any of the following reasons that can occur in the respective translation state. (In the expression for (b) above, "$PRTE_V$" is shorthand indicating that an invalid segment table descriptor did not stop the translation process. Note that an SLB hit may satisfy this condition even when the Process Table Entry is invalid.)

- HR=0, data address translation is enabled (MSR_{DR}=1) and the virtual address of any byte of

the storage location specified by a *Load*, *Store*, ***icbi***, ***dcbz***, ***dcbst***, or ***dcbf***[***l***] instruction cannot be translated to a real address because no valid PTE was found for the VPM translation.

- HR=1 and the guest real address of any byte of the storage location specified by a *Load*, *Store*, ***icbi***, ***dcbz***, ***dcbst***, or ***dcbf***[***l***] instruction cannot be translated to a host real address because no valid PTE was found in the partition-scoped page table.
- The guest real address of a page directory entry or process table entry could not be translated when HR=1; or the virtual address of a process table entry or segment table entry group could not be translated when VPM=1 and HR=0.
- An unsupported MMU configuration is found. In addition to an invalid radix tree configuration found in the partition-scoped tables, this type of exception will also be reported outside of hypervisor real mode for translation mode mismatches including UPRT=0 when HR=1, LPID=0 if MSR_{HV}=0 when HR=1, and HR=0 for LPID=0 when HR=1 for another partition ID.
- A reference or change bit update in a partition-scoped PTE cannot be performed (including for the process-scoped PDE or PTE or process table entry for a radix guest.

Programming Note

When reporting failure to set a reference or change bit for a table entry, whether the change bit must be set is inferred from whether the access is reported to be a store. (A load may report store if, when attempting to set the reference bit, the update of the change bit in the partition-scoped PTE mapping the process-scoped PTE fails.) Behavior is similar for access authority failures.

- HR=0, data address translation is disabled (MSR_{DR}=0), and the virtual address of any byte of the storage location specified by a *Load*, *Store*, ***icbi***, ***dcbz***, ***dcbst***, or ***dcbf***[***l***] instruction cannot be translated to a real address by means of the virtual real addressing mechanism.
- The effective address specified by a ***lq***, ***stq***, ***lwat***, ***ldat***, ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, ***lqarx***, ***stwat***, ***stdat***, ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** instruction refers to storage that is Write Through Required or Caching Inhibited; or the effective address specified by a ***copy*** or ***paste.*** instruction refers to storage that is Caching Inhibited; or the effective address specified by a ***lwat***, ***ldat***, ***stwat,*** or ***stdat*** instruction refers to storage that is Guarded.
- An accelerator is specified as the source of a ***copy*** instruction, normal memory is specified at the target of a ***paste.*** instruction, or an attempt is made to access an accelerator that is not properly configured for the software's use; HR=0 only.
- The access violates storage protection. In addition to the legacy VPM cases, this includes mismatches in access authority in which the process-scoped PTE permits the access but the partition-scoped PTE does not. It also includes lack of necessary authority for accesses to process-scoped tables, for example lack of write authority to set a reference bit in the process-scoped PTE. (In such a case, the "access" reported as failing would be the access to the process-scoped table. The HDAR would provide the guest real / (abbreviated) virtual address of the table entry.)
- A Data Address Watchpoint match occurs, HR=0 only.
- An attempt is made to execute a *Load Atomic* or *Store Atomic* instruction with an invalid function code, HR=0 only.

A Hypervisor Data Storage interrupt also occurs when no higher priority exception exists and an attempt is made to execute a *Load Atomic* or *Store Atomic* instruction specifying an invalid function code.

Programming Note

When an attempt to execute a *Load Atomic* or *Store Atomic* instruction containing an invalid function code (see Figures 3 and 4 in Book II) causes an HDSI, the condition is very similar to an invalid form of an instruction. As a result, this instance of HDSI occurs with a high prioirty that blocks the translation process and prevents Reference and Change bit updates.

If a ***stbcx***., ***sthcx.***, ***stwcx.***, ***stdcx.,*** or ***stqcx.*** would not perform its store in the absence of a Hypervisor Data Storage interrupt, and either (a) the specified effective address refers to storage that is Write Through Required or Caching Inhibited, or (b) a non-conditional *Store* to the specified effective address would cause a Hypervisor Data Storage interrupt, it is implementation-dependent whether a Hypervisor Data Storage interrupt occurs.

If the XER specifies a length of zero for an indexed *Move Assist* instruction, a Hypervisor Data Storage interrupt does not occur.

The following registers are set:

HSRR0 Set to the effective address of the instruction that caused the interrupt.

HSRR1

33:36	Set to 0.
42:47	Set to 0.
Others	Loaded from the MSR.

MSR See Figure 65.

HDSISR

32	Set to 0.

33 Set to 1 if the translation for an attempted access is not found in the Page Table; otherwise set to 0.

34:35 Set to 0.

36 Set to 1 if the access is not permitted by Figure 44 46, or the privilege, read, or read/write bits in Figure 45 as appropriate; otherwise set to 0.

37 Set to 1 if the access is due to a ***lq***, ***stq***, ***lwat***, ***ldat***, ***lbarx***, ***lharx***, ***lwarx***, ***ldarx***, ***lqarx***, ***stwat***, ***stdat***, ***stbcx.***, ***sthcx.***, ***stwcx.***, ***stdcx.***, or ***stqcx.*** instruction that addresses storage that is Write Through Required or Caching Inhibited; or if the access is due to a ***copy*** or ***paste.*** instruction that addresses storage that is caching inhibited; or if the access is due to a ***lwat***, ***ldat***, ***stwat,*** or ***stdat*** instruction that addresses storage that is Guarded; otherwise set to 0.

38 Set to 1 by an explicit access for a *Store,* ***dcbz***, or *Load/Store Atomic* instruction; set to 1 when a process-scoped PTE update fails due to a lack of write authority or the inability to set the change bit in the partition-scoped PTE; otherwise set to 0.

39:40 Set to 0.

41 Set to 1 if a Data Address Watchpoint match occurs; otherwise set to 0.

42 Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.

43 Set to 0.

44 Set to 1 if an unsupported MMU configuration is found during the translation process.

45 Set to 1 if an attempt to atomically set a reference or change bit fails; otherwise set to 0.

Programming Note

The number of attempts hardware makes to atomically set reference and change bits before triggering this exception is implementation dependent. The POWER9 processor makes no attempt. Software may still support the atomic update programming model to get performance benefits such as those described in Section 5.7.12.

46 Set to 1 if HR=1 and the virtual / guest real address of a page directory entry, page table entry, or process table entry could not be translated; or HR=0, VPM=1, and the virtual address of a process table entry or segment table entry group could not be translated; otherwise set to 0.

47:59 Set to 0.

60 Set to 1 if an accelerator is specified as the source of a ***copy*** instruction, normal memory is specified as the target of a ***paste.*** instruction, or an attempt is made to access an accelerator that is not properly configured for the software's use; otherwise set to 0. These exceptions are presented differently from most instruction-caused exceptions. See Section 4.4, "Copy-Paste Facility", in Book II for details. Additional information may be retained by the platform if the accelerator is not properly configured.

61 Set to 1 if an attempt is made to execute a *Load Atomic* or *Store Atomic* instruction specifying an invalid function code; otherwise set to 0.

62:63 Set to 0.

HDAR Set to the effective address or portion of the VPN of a storage element, or undefined, as described in the following list. The list should be read from the top down; the HDAR is set as described by the first item that corresponds to an exception that is reported in the HDSISR. For example, if a *Load Word* instruction causes a storage protection violation and a Data Address Watchpoint match (and both are reported in the HDSISR), the HDAR is set to the effective address of a byte in the first aligned doubleword for which access was attempted in the page that caused the exception.

- undefined, for *Load Atomic* or *Store Atomic* instruction specifying an invalid function code
-
- undefined, when $HDSISR_{60}$=1
- least significant 64 bits of the VA of the table entry or group when a process table entry or segment table entry group virtual address cannot be translated in Paravirtualized HPT mode with VPM=1.
- EA, when a Hypervisor Data Storage exception occurs for reasons other than a Data Address Watchpoint match
  - a byte in the block that caused the exception, for a *Cache Management* instruction
  - a byte in the first aligned quadword for which access was attempted in the page that caused the exception, for a quadword *Load* or *Store* instruction (i.e., a *Load* or *Store* instruction for which the storage operand is a quadword; "first" refers to address order: see Section 6.7)

- a byte in the first aligned doubleword for which access was attempted in the page that caused the exception, for a non-quadword *Load* or *Store* instruction

- set as described in the previous major bullet, except that the low order 5 bits are undefined, for a Data Address Watchpoint match

For the cases in which the HDAR is specified above to be set to an effective address, if the interrupt occurs in 32-bit mode the high-order 32 bits of the HDAR are set to 0.

Programming Note

Note that for HPT translation, the full EA is a superset of the bits required to construct the full VA, when also provided with the VSID in the ASDR.

ASDR When HR=0, loaded with VSID, B, Ks, Kp, N, C, L, and LP values from the segment descriptor that translated the access or indicated the base of the table, or undefined, as described in the following list. For a large segment the values of the bits below the VSID are undefined. When HR=1 (nested translaiton is taking place), loaded with the guest real address down to bit 51 of a storage element or table entry, or undefined, as described in the following list. The list should be read from the top down; the ASDR is set as described by the first item that corresponds to an exception that is reported in the HDSISR.

- undefined, for *Load Atomic* or *Store Atomic* instruction specifying an invalid function code
-
- undefined, when $HDSISR_{60}$=1
- the guest real page address of the table entry when a process table or process-scoped page directory or page table entry guest real address cannot be translated or the VSID of the table entry when a process or segment table entry virtual address cannot be translated (the rest of the segment descriptor is implied).
- the guest real address of the process-scoped PDE or PTE or process table entry when a reference or change bit in the partition-scoped PTE mapping the process-scoped PDE or PTE or process table entry cannot be set atomically
- the guest real address of the storage element when a reference or change bit in the partition-scoped PTE cannot be set atomically
- the guest real address of the storage element, process table entry, page directory entry, or page table entry (depending on which partition-scoped table has the flaw) for an unsupported radix tree configuration in the partition-scoped table (the effective address for other cases of the invalid MMU configuration exception is found in the HDAR)
- the guest real address of the process-scoped PTE when an attempt is made to set a reference or change bit without write authority in the partition-scoped PTE that maps it
- the guest real address or segment descriptor associated with the specified storage element when a Hypervisor Data Storage exception occurs for reasons other than a Data Address Watchpoint match
- undefined, for a Data Address Watchpoint match, unsupported MMU configuration, or accesses to storage that is Caching Inhibited or Write Through Required by the instructions that are prohibited from making such accesses.

If multiple Hypervisor Data Storage exceptions occur for a given effective address, any one or more of the bits corresponding to these exceptions may be set to 1 in the HDSISR. If the HDSISR reports other exceptions together with a Virtualized Page Class Key Storage Protection exception that occurs when $LPCR_{KBV}$=1 and Virtualized Partition Memory is disabled by VPM=0, the other exceptions are actually DSIs.

Programming Note

A Virtual Page Class Key Storage Protection exception that occurs with $LPCR_{KBV}$=1 and Virtualized Partition Memory disabled by VPM=0 identifies an access that must be emulated by the hypervisor. When it is reported together with other exceptions in the HDSISR, the hypervisor should service the Virtual Page Class Key Storage Protection exception first. This is in part because the operating system may be using some PTE fields for non-architected purposes, which could in turn cause spurious exceptions to be reported.

Execution resumes at effective address 0x0000_0000_0000_0E00, possibly offset as specified in Figure 66.

6.5.17 Hypervisor Instruction Storage Interrupt

A Hypervisor Instruction Storage interrupt occurs when either the thread is not in hypervisor state or an unsupported MMU configuration has been found or the access has been prevented by a problem in partition-scoped Radix Tree translation, no higher priority exception exists, and either

(a) HPT translation is being performed, the value of the expression

($\neg MSR_{IR}$) | (VPM & $PRTE_V$ & MSR_{IR}))

is 1, and the next instruction to be executed cannot be fetched for any of the following reasons, or

(b) Radix Tree translation is being performed and partition-scoped translation prevents the next instruction to be executed from being fetched for any of the following reasons.

(In the expression for (a) above, "$PRTE_V$" is shorthand indicating that an invalid segment table descriptor did not stop the translation process. Note that an SLB hit may satisfy this condition even when the Process Table Entry is invalid.)

A Hypervisor Instruction Storage interrupt also occurs when no higher priority exception exists, HR=0, and a reference or change bit update cannot be performed as described below.

- Instruction address translation is enabled (MSR_{IR}=1) and the virtual address cannot be translated to a real address because no valid PTE was found for the VPM translation.
- HR=1 and the guest real address of the instruction cannot be translated to a host real address because no valid PTE was found in the partition-scoped page table.
- The guest real address of a page directory entry or process table entry could not be translated when HR=1; or the virtual address of a process table entry or segment table entry group could not be translated when VPM=1 and HR=0.
- An unsupported MMU configuration is found. In addition to an invalid radix tree configuration found in the partition-scoped tables, this type of exception will also be reported outside of hypervisor real mode for translation mode mismatches including UPRT=0 when HR=1, LPID=0 if MSR_{HV}=0 when HR=1, and HR=0 for LPID=0 when HR=1 for another partition ID.
- A reference or change bit update in a partition-scoped PTE cannot be performed (including for the process-scoped PDE or PTE or process table entry for a radix guest.
- HR=0, instruction address translation is disabled (MSR_{IR}=0), and the virtual address cannot be translated to a real address by means of the virtual real addressing mechanism.
- The fetch violates storage protection. In addition to the legacy VPM cases, this includes mismatches in access authority in which the process-scoped PTE permits the access but the partition-scoped PTE does not. It also includes lack of necessary authority for accesses to process-scoped tables, for example lack of write authority to set a reference bit in the process-scoped PTE. (In such a case, the "access" reported as failing would be the access to the process-scoped table. The HDAR would provide the guest real / (abbreviated) virtual address of the table entry.)

The following registers are set:

HSRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, HSRR0 is set to the branch target address).

HSRR1

33 Set to 1 if the translation for an attempted access is not found in the Page Table; otherwise set to 0.

34 Set to 0.

35 Set to 1 if the access is to No-execute (as indicated by the N bit in the segment table entry and HPT PTE or the exec bit in the EAA field of the Radix PTE) or Guarded storage; otherwise set to 0.

36 Set to 1 if the access is not permitted by Figure 44 46, or the read or read/write bits in Figure 45 as appropriate; otherwise set to 0.

42 Set to 1 if the access is not permitted by virtual page class key protection; otherwise set to 0.

43 Set to 0.

44 Set to 1 if an unsupported MMU configuration is found during the translation process.

45 Set to 1 if an attempt to atomically set a reference or change bit fails; otherwise set to 0.

Programming Note

The number of attempts hardware makes to atomically set reference and change bits before triggering this exception is implementation dependent. The POWER9 processor makes no attempt. Software may still support the atomic update programming model to get performance benefits such as those described in Section 5.7.12.

46 Set to 1 if HR=1 and the guest real address of a page directory entry, page table entry, or process table entry could not be translated; or HR=0, VPM=1, and the virtual address of a process table entry or segment table entry group could not be translated; otherwise set to 0.

47 Set to 1 if the operation that caused the exception was attempting to update storage; otherwise set to 0. This bit may be set as a modifier to bit 45 to indicate that a change bit must be set. It may also be set as a modifier to bits 36 and 42, to indicate that write authority was required to complete the operation.

Others Loaded from the MSR.

HDAR Set to the least significant 64 bits of the VA of a table entry or group when HR=0 and a process table entry or segment table entry group virtual address cannot be translated and VPM=1. May be set spuriously in other cases.

ASDR When HR=0, loaded with VSID, B, Ks, Kp, N, C, L, and LP values from the segment descriptor that translated the access or indicated the base of the table, or undefined, as described in the following list. For a large segment the values of the bits below the VSID are undefined. When HR=1 (nested translaiton is taking place), set to the guest real address down to bit 51 of the instruction or table entry, or undefined, as described in the following list.

- the guest real address of the table entry when a process table or process-scoped page directory or page table entry guest real address cannot be translated or the VSID of the table entry when a process or segment table entry virtual address cannot be translated (the rest of the segment desrcriptor is implied).
- the guest real address of the process-scoped PDE or PTE or process table entry when a reference or change bit in the partition-scoped PTE mapping the process-scoped PDE or PTE or process table entry cannot be set atomically
- the guest real address of the instruction when a reference or change bit in the partition-scoped PTE cannot be set atomically
- the guest real address of the instruction, process table entry, page directory entry, or page table entry (depending on which partition-scoped table has the flaw) for an unsupported radix tree configuration in the partition-scoped table (the effective address for other cases of the invalid MMU configuration exception will be found in HSRR0)
- the guest real address of the process-scoped PTE when an attempt is made to set a reference bit without write authority in the partition-scoped PTE that maps it
- the guest real address or segment descriptor associated with the instruction that the thread would have attempted to execute next if no interrupt conditions were present (partition-scoped page fault or protection exception)
- undefined for unsupported MMU configuration

MSR See Figure 65.

If multiple Hypervisor Instruction Storage exceptions occur due to attempting to fetch a single instruction, any one or more of the bits corresponding to these exceptions may be set to 1 in HSRR1.

Execution resumes at effective address 0x0000_0000_0000_0E10, possibly offset as specified in Figure 66.

6.5.18 Hypervisor Emulation Assistance Interrupt

A Hypervisor Emulation Assistance interrupt is generated when execution is attempted of an illegal instruction, or of a reserved instruction or an instruction that is not provided by the implementation. It is also generated under the following conditions.

- When $MSR_{HV\ PR}$=0b00 and $LPCR_{EVIRT}$=1, execution is attempted of a hypervisor privileged instruction or of an ***mtspr*** or ***mfspr*** instruction that specifies an SPR that is hypervisor privileged for the operation.
- When MSR_{PR}=1, execution is attempted of an ***mtspr*** or ***mfspr*** instruction that specifies an SPR with spr_0=0 that is not provided by the implementation.
- When MSR_{PR}=0, execution is attempted of an ***mtspr*** or ***mfspr*** instruction that specifies SPR 0, 4, 5, or 6.
- When MSR_{PR}=0 and $LPCR_{EVIRT}$=1, execution is attempted of an ***mtspr*** or ***mfspr*** instruction that specifies an SPR other than 0, 4, 5, or 6 that is not provided by the implementation.

A Hypervisor Emulation Assistance interrupt may be generated when execution is attempted of an instruction that is in invalid form or that is treated as if the instruction form were invalid.

The following registers are set:

HSRR0 Set to the effective address of the instruction that caused the interrupt.

HSRR1

33:36 Set to 0.

42:44 Set to 0.

45 Set to 1 for an attempt, when $MSR_{HV\ PR}$ = 0b00 and $LPCR_{EVIRT}$=1, to execute a hypervisor privileged instruction or an ***mtspr*** or ***mfspr*** instruction that specifies an SPR that is hypervisor privileged for the operation; otherwise set to 0.

46:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 65 on page 1064.

HEIR Set to a copy of the instruction that caused the interrupt

If the interrupt is caused by an attempt to execute an invalid form of a hypervisor privileged instruction when $MSR_{HV\ PR}$ = 0b00 and $LPCR_{EVIRT}$=1, it is implementation dependent whether $HSRR1_{45}$ is set to 0 (reflecting the invalid instruction form) or to 1 (reflecting the privilege violation).

Execution resumes at effective address 0x0000_0000_0000_0E40, possibly offset as specified in Figure 66.

Programming Note

This Programming Note illustrates how Hypervisor Emulation Assistance interrupts should be handled by software, including in environments that support nested hypervisors.

In this Note, "the hypervisor" may be the hypervisor to which hardware passes control when a Hypervisor Emulation Assistance interrupt occurs or, in an environment that supports nested hypervisors, may be a nested hypervisor. The hypervisor to which hardware passes control when a Hypervisor Emulation Assistance interrupt occurs is here called the "level 0 hypervisor," and is the only level of hypervisor that runs with $MSR_{HV\ PR}$=0b10 and that can access hypervisor resources directly; nested hypervisors run with $MSR_{HV\ PR}$=0b00 and their attempts to access hypervisor resources are virtualized by a higher-level hypervisor as described below. In this Note, the hypervisor receiving the Hypervisor Emulation Assistance interrupt (which may have been passed from a higher-level hypervisor as described below) is called the "level N hypervisor." This Note assumes that $LPCR_{EVIRT}$=1 if nested hypervisors are used. (A Hypervisor Emulation Assistance interrupt can set $HSRR1_{45}$ to 1 only when $LPCR_{EVIRT}$=1.) Higher level numbers correspond to lower level hypervisors.

In the description immediately below, it is assumed that nested hypervisors (if any) are new versions of the existing hypervisor, and that the purpose of the nesting is to test the nested hypervisors before using them as level 0 hypervisors.

When a Hypervisor Emulation Assistance interrupt is received by the level N hypervisor, the cases and their suggested handling are as follows.

- The program that caused the interrupt is the level N hypervisor itself.
  - $HSRR1_{45}$=0: Emulate the instruction, recover from the error, or terminate this hypervisor, as appropriate.
  - $HSRR1_{45}$=1: Cannot occur for N=0; will not occur for N>0 if the hypervisor nesting software is written correctly.
- The program that caused the interrupt is not the level N hypervisor.
  - The program most recently dispatched by the level N hypervisor is a level N+1 hypervisor.
    - $HSRR1_{45}$=0: Pass control to the level N+1 hypervisor as if the instruction had caused a Hypervisor Emulation Assistance interrupt (with $HSRR1_{45}$=0) to that hypervisor.
    - $HSRR1_{45}$=1:
      - The program that caused the interrupt is the level N+1 hypervisor: Virtualize the instruction.
      - The program that caused the interrupt is not the level N+1 hypervisor: Pass control to the level N+1 hypervisor as if the instruction had caused a Hypervisor Emulation Assistance interrupt (with $HSRR1_{45}$=1) to that hypervisor.
  - The program most recently dispatched by the level N hypervisor is an operating system.
    - $HSRR1_{45}$=0: Emulate the instruction if appropriate (rather than pass control to the operating system to do the emulation); otherwise pass control to the operating system as if the instruction had caused an "Illegal Instruction type Program interrupt" as described in a Programming Note near the end of <xref to Section 6.5.9>.
    - $HSRR1_{45}$=1: Either terminate the operating system or pass control to the operating system as if the instruction had caused a Privileged Instruction type Program interrupt as described in a Programming Note near the end of <xref to Section 6.5.9>.
  - The program most recently dispatched by the level N hypervisor is an application program.
    - $HSRR1_{45}$=0: Emulate the instruction if appropriate; otherwise terminate the application program.
    - $HSRR1_{45}$=1: Cannot occur.

The preceding description implicitly assumes that any nested hypervisors being tested will, when run at level 0, be run on processors that support the same version of the architecture as the processor on which they are being tested. If instead they will be run on processors that support a newer version of the architecture, the level 0 hypervisor should behave as described above if the interrupt is caused by an instruction that is unchanged between the two architecture versions. However, if the interrupt is caused by an instruction that differs between the two architecture versions (e.g., an instruction that is added by the newer version of the architecture), the level 0 hypervisor should emulate the behavior of the newer processor, rather than, for example, passing the interrupt to a level 1 hypervisor.

Other uses of nested hypervisors are also possible. For example, software that is designed to interact, nearly simultaneously, with the hypervisor instance that is running on each of many processors could be tested on a single processor by running multiple level 1 hypervisors under a single level 0 hypervisor.

It is expected that in practice there will be at most two levels of nested hypervisor (i.e., $N \leq 2$). (For example, two levels are needed in the case described in detail above, to test the ability of the nested hypervisors at level 1 to support nested hypervisors.)

Programming Note

If a Hypervisor Emulation Assistance interrupt occurs with $HSRR1_{45}$=0 when the thread is not in hypervisor state, for an instruction that the hypervisor does not emulate, the hypervisor should pass control to the operating system as if the instruction had caused an "Illegal Instruction type Program interrupt", as described in a Programming Note near the end of Section 6.5.9, “Program Interrupt” on page 1074.

Similarly, if a Hypervisor Emulation Assistance interrupt occurs with $HSRR1_{45}$=1 when the thread is in privileged non-hypervisor state, for an instruction that the hypervisor does not virtualize, the hypervisor should pass control to the operating system as if the instruction had caused a Privileged Instruction type Program interrupt, as described in another Programming Note near the end of Section 6.5.9, “Program Interrupt” on page 1074.

Programming Note

In versions of the architecture that precede V. 3.0B, an attempt when MSR_{PR}=0 to execute an ***mtspr*** or ***mfspr*** instruction specifying an SPR that was not implemented (with the exception of SPR 0 for ***mtspr*** and SPRs 0, 4, 5, and 6 for ***mfspr***) was treated as a no-op. These former no-op cases now cause a Hypervisor Emulation Assistance interrupt (with $HSRR1_{45}$=0) when $LPCR_{EVIRT}$=1 to enable future functions to be emulated on older implementations. (An attempt when MSR_{PR}=0 to execute an ***mtspr*** instruction specifying SPRs 4, 5, and 6 now causes a Hypervisor Emulation Assistance interrupt regardless of the value of $LPCR_{EVIRT}$.) If there is no future function emulation to be performed, hypervisor software must choose a policy from the following.

- treat the instruction as an error
- emulate the legacy no-op behavior
- give control to the operating system

6.5.19 Hypervisor Maintenance Interrupt

A Hypervisor Maintenance interrupt occurs when no higher priority exception exists, a Hypervisor Maintenance exception exists (a bit in the HMER is set to one), the exception is enabled in the HMEER, and the value of the following expression is 1.

$(MSR_{EE} \mid \neg(MSR_{HV}) \mid MSR_{PR})$

The following registers are set:

HSRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

HSRR1

33:36 Set to 0.
42:47 Set to 0.
Others Loaded from the MSR.

MSR See Figure 65 on page 1064.

HMER See Section 6.2.9 on page 1051.

The exception bits in the HMER are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an ***mthmer*** instruction.

Execution resumes at effective address 0x0000_0000_0000_0E60.

Programming Note

Because the value of MSR_{EE} is always 1 when the thread is in problem state, the simpler expression

$(MSR_{EE} \mid \neg(MSR_{HV}))$

is equivalent to the expression given above.

Programming Note

If an implementation uses the HMER to record that a readable resource, such as the Time Base, has been corrupted, then, because the HMI is disabled in the hypervisor state, it is necessary for the hypervisor to check HMER after reading that resource to be sure an error has not occurred.

6.5.20 Directed Hypervisor Doorbell Interrupt

A Directed Hypervisor Doorbell interrupt occurs when no higher priority exception exists, a Directed Hypervisor Doorbell exception is present, and the value of the following expression is 1.

$(MSR_{EE} \mid \neg(MSR_{HV}) \mid MSR_{PR})$

Directed Hypervisor Doorbell exceptions are generated when Directed Hypervisor Doorbell messages (see Chapter 10) are received and accepted by the thread.

The following registers are set:

HSRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

HSRR1

33:36 Set to 0.
42:47 Set to 0.
Others Loaded from the MSR.

MSR See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0E80, possibly offset as specified in Figure 66.

Programming Note

Because the value of MSR_{EE} is always 1 when the thread is in problem state, the simpler expression

$(MSR_{EE} \mid \neg(MSR_{HV}))$

is equivalent to the expression given above.

6.5.21 Hypervisor Virtualization Interrupt

A Hypervisor Virtualization interrupt occurs when no higher priority exception exists, a Hypervisor Virtualization exception exists, and the value of the following equation is1.

$(MSR_{EE} \mid \neg(MSR_{HV}) \mid MSR_{PR})$ & HVICE

The occurrence of the interrupt does not cause the exception to cease to exist.

HSRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

HSRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0EA0, possibly offset as specified in Figure 66.

6.5.22 Performance Monitor Interrupt

A Performance Monitor interrupt occurs when no higher priority exception exists, a Performance Monitor exception exists, event-based branches are disabled ($MMCR0_{EBE}$=0), and MSR_{EE}=1, and either $HFSCR_{PM}$=1 or the thread is in hypervisor state.

If multiple Performance Monitor exceptions occur before the first causes a Performance Monitor interrupt, the interrupt reflects the most recent Performance Monitor exception and the preceding Performance Monitor exceptions are lost.

The following registers are set:

SRR0 Set to the effective address of the instruction that would have been attempted to be execute next if no interrupt conditions were present.

SRR1
- **33:36 and 42:47** Reserved.
- **Others** Loaded from the MSR.

MSR See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0F00, possibly offset as specified in Figure 66.

6.5.23 Vector Unavailable Interrupt

A Vector Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a Vector instruction (including Vector loads, stores, and moves), and MSR_{VEC}=0.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0F20, possibly offset as specified in Figure 66.

6.5.24 VSX Unavailable Interrupt

A VSX Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a VSX instruction (including VSX loads, stores, and moves), and MSR_{VSX}=0.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 65 on page 1064.

Execution resumes at effective address 0x0000_0000_0000_0F40, possibly offset as specified in Figure 66.

6.5.25 Facility Unavailable Interrupt

A Facility Unavailable interrupt occurs when no higher priority exception exists, and one of the following occurs.

- a facility is accessed in problem state when it has been made unavailable by the FSCR
- a Performance Monitor register is accessed or a ***clrbhrb*** or ***mfbhrbe*** instruction is executed in problem state when it has been made unavailable by MMCR0.
- the Transactional Memory Facility is accessed in any privilege state when it has been made unavailable by MSR_{TM}.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 65 on page 1064.

FSCR
- **0:7** See Section 6.2.11 on page 1051.
- **Others** Not changed.

Execution resumes at effective address 0x0000_0000_0000_0F60, possibly offset as specified in Figure 66.

Programming Note

For the case of an outer ***tbegin.***, the interrupt handler should either return to the ***tbegin.*** with MSR_{TM} = 1 (allowing the program to use transactions), or treat the attempt to initiate an outer transaction as a program error.

6.5.26 Hypervisor Facility Unavailable Interrupt

A Hypervisor Facility Unavailable interrupt occurs when no higher priority exception exists, and one of the following occurs.

- a facility is accessed in problem or privileged non-hypervisor states when it has been made unavailable by the HFSCR.
- The ***stop*** instruction is executed in privileged non hypervisor state when any of the following conditions exist.
  $PSSCR_{EC}$=1
  $PSSCR_{ESL}$=1
  $PSSCR_{MTL}$>$PSSCR_{PSLL}$
  $PSSCR_{RL}$>$PSSCR_{PSLL}$

The following registers are set:

HSRR0 Set to the effective address of the instruction that caused the interrupt.

HSRR1
- **33:36** Set to 0.
- **42:47** Set to 0.
- **Others** Loaded from the MSR.

MSR See Figure 65 on page 1064.

HFSCR
- **0:7** See Section 6.2.12 on page 1052.
- **Others** Not changed.

Execution resumes at effective address 0x0000_0000_0000_0F80, possibly offset as specified in Figure 66.

6.5.27 System Call Vectored Interrupt

A System Call Vectored interrupt occurs when a *System Call Vectored* instruction is executed.

The following registers are set:

LR Set to the effective address of the instruction following the System Call Vectored instruction.

CTR
- **33:36** undefined
- **42:47** undefined
- **Others** Loaded from corresponding bits of the MSR.

MSR See Figure 65 on page 1064.

Execution resumes at the effective address specified in Figure 66

Programming Note

When the System Call Vectored interrupt results in MSR_{IR} being 1 or MSR_{HV} being 0, the effective address described above is translated to a real address before being used to access storage. If the effective address cannot be translated, or if instructions cannot be fetched from the addressed storage location (e.g., the access would violate storage protection, or would be to No-execute storage), an [Hypervisor] Instruction Storage interrupt occurs before the first instruction at the effective address is executed.

Because the System Call Vectored interrupt uses save/restore registers that differ from those used by other interrupts, the System Call Vectored interrupt handler can run with address translation enabled and External interrupts enabled. Similarly, the Programming Note about managing MSR_{RI} at the end of Section 6.4.3 does not apply to the System Call Vectored interrupt handler (the System Call Vectored interrupt does not alter MSR_{RI}).

6.6 Partially Executed Instructions

If a Data Storage, Data Segment, Alignment, system-caused, or imprecise exception occurs while a *Load* or *Store* instruction is executing, the instruction may be aborted. In such cases the instruction is not completed, but may have been partially executed in the following respects.

- Some of the bytes of the storage operand may have been accessed, except that if access to a given byte of the storage operand would violate storage protection, that byte is neither copied to a register by a *Load* instruction nor modified by a *Store* instruction. Also, the rules for storage accesses given in Section 5.8.1, "Guarded Storage" and in Section 2.2 of Book II are obeyed.
- Some registers may have been altered as described in the Book II section cited above.
- Reference and Change bits may have been updated as described in Section 5.7.12.
- For a ***stbcx.***, ***sthcx.***, ***stwcx., stdcx.***, or ***stqcx.*** instruction that is executed in-order, CR0 may have been set to an undefined value and the reservation may have been cleared.

The architecture does not support continuation of an aborted instruction but intends that the aborted instruction be re-executed if appropriate.

Programming Note

An exception may result in the partial execution of a *Load* or *Store* instruction. For example, if the Page Table Entry that translates the address of the storage operand is altered, by a program running on another thread, such that the new contents of the Page Table Entry preclude performing the access, the alteration could cause the *Load* or *Store* instruction to be aborted after having been partially executed.

As stated in the Book II section cited above, if an instruction is partially executed the contents of registers are preserved to the extent that the instruction can be re-executed correctly. The consequent preservation is described in the following list. For any given instruction, zero, one, or two items in the list apply.

- For a fixed-point *Load* instruction that is not a multiple or string form, if RT=RA or RT=RB then the contents of register RT are not altered.
- For an ***lq*** instruction, if RT+1 = RA then the contents of register RT+1 are not altered.
- For an update form *Load* or *Store* instruction, the contents of register RA are not altered.

6.7 Exception Ordering

Since multiple exceptions can exist at the same time and the architecture does not provide for reporting more than one interrupt at a time, the generation of more than one interrupt is prohibited. Some exceptions, such as the Mediated External exception, persist and can be deferred. However, other exceptions would be lost if they were not recognized and handled when they occur. For example, if an External interrupt was generated when a Data Storage exception existed, the Data Storage exception would be lost. If the Data Storage exception was caused by a *Store Multiple* instruction for which the storage operand crosses a virtual page boundary and the exception was a result of attempting to access the second virtual page, the store could have modified locations in the first virtual page even though it appeared that the *Store Multiple* instruction was never executed.

For the above reasons, all exceptions are prioritized with respect to other exceptions that may exist at the same instant to prevent the loss of any exception that is not persistent. Some exceptions cannot exist at the same instant as some others.

Data Storage, Hypervisor Data Storage, Data Segment, and Alignment exceptions and transaction failure due to attempted access of a disallowed type while in Transactional state occur as if the storage operand were accessed one byte at a time in order of increasing effective address (with the obvious caveat if the operand includes both the maximum effective address and effective address 0). (The required ordering of exceptions on components of non-atomic accesses does not extend to the performing of the component accesses in the event of an exception. For example, if byte n causes a data storage exception, it is not necessarily true that the access to byte n-1 has been performed.)

6.7.1 Unordered Exceptions

With one exception, the exceptions listed here are unordered, meaning that they may occur at any time regardless of the state of the interrupt processing mechanism. These exceptions are recognized and processed when presented. The exception is that a Machine Check caused by an attempt to access an accelerator as other than an operand of ***copy*** or ***paste.*** is ordered similarly to a storage protection exception.

1. System Reset
2. Machine Check except for those caused by an invalid attempt to access an accelerator

6.7.2 Ordered Exceptions

The exceptions listed here are ordered with respect to the state of the interrupt processing mechanism. With one exception, in the following list the hypervisor forms of the Data Storage and Instruction Storage exceptions can be substituted for the non-hypervisor forms since the hypervisor forms cannot be caused by the same instruction and have the same ordering. The exception is that Virtual Page Class Key Storage Protection exceptions that occur when $LPCR_{KBV}$=1 and Virtualized Partition Memory is disabled by VPM=0 cause only a Hypervisor Data Storage exception (and never a Data Storage exception).

System-Caused or Imprecise

1. Program
   - Imprecise Mode Floating-Point Enabled Exception
2. Hypervisor Maintenance
3. Hypervisor Virtualization, External, [Hypervisor] Decrementer, Performance Monitor, Directed Privileged Doorbell, Directed Hypervisor Doorbell

Instruction-Caused and Precise

1. Instruction Segment
2. [Hypervisor] Instruction Storage or Machine Check for invalid accelerator access
3. Hypervisor Emulation Assistance or Program (Privileged Instruction)
4. Function-Dependent
   - 4.a Fixed-Point and Branch
     - 1 Hypervisor Facility Unavailable
     - 2 Facility Unavailable
     - 3a Program
       - Trap
       - TM Bad Thing
     - 3b System Call or System Call Vectored
     - 3c.1 Data Storage for the case of *Fixed-Point Load or Store Caching Inhibited* instructions with MSR_{DR}=1 or the case of an invalid function code for an Atomic Memory Operation
     - 3c.2 all other Data Storage, Hypervisor Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment
     - 4 Trace
   - 4.b Floating-Point
     - 1 Hypervisor Facility Unavailable
     - 2 Floating Point Unavailable
     - 3a Program
       - Precise Mode Floating-Pt Enabled Excep'n
     - 3b [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment
     - 4 Trace
   - 4.c Vector
     - 1 Hypervisor Facility Unavailable
     - 2 Vector Unavailable
     - 3a [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment
     - 4 Trace
   - 4.d VSX
     - 1 Hypervisor Facility Unavailable
     - 2 VSX Unavailable
     - 3a Program
       - Precise Mode Floating-Pt Enabled Excep'n
     - 3b [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment
     - 4 Trace
   - 4.e Other Instructions
     - 1 Hypervisor Facility Unavailable
     - 2 Facility Unavailable
     - 3a [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment
     - 4 Trace

For implementations that execute multiple instructions in parallel using pipeline or superscalar techniques, or combinations of these, it can be difficult to understand the ordering of exceptions.To understand this ordering it is useful to consider a model in which each instruction is fetched, then decoded, then executed, all before the next instruction is fetched. In this model, the exceptions a single instruction would generate are in the order shown in the list of instruction-caused exceptions. Exceptions with different numbers have different ordering. Exceptions with the same numbering but different lettering are mutually exclusive and cannot be caused by the same instruction. The Hypervisor Virtualization, External, [Hypervisor] Decrementer, Performance Monitor, Directed Privileged Doorbell, and Directed Hypervisor Doorbell interrupts have equal ordering. Similarly, where Data Storage, Data Segment, and Alignment exceptions are listed in the same item, and where Hypervisor Emulation Assistance and Privileged Instruction exceptions are listed in the same item, they have equal ordering.

Even on threads that are capable of executing several instructions simultaneously, or out of order, instruction-caused interrupts (precise and imprecise) occur in program order.

Programming Note

Despite that debug address matches are EA based, the exceptions they cause are not necessarily ordered before translation-caused exceptions. For example, it may be considered advantageous to take a page fault that would have prevented an access rather than a DAWR match exception

6.8 Event-Based Branch Exception Ordering

Event-based exceptions are not ordered because they can occur simultaneously. Whenever an event-based exception occurs and the exception is enabled, the corresponding "exception occurred" bit in the BESCR is set to 1. See Section 7.2.1 of Book II.

6.9 Interrupt Priorities

This section describes the relationship of nonmaskable, maskable, precise, and imprecise interrupts. In the following descriptions, the interrupt mechanism waiting for all possible exceptions to be reported includes only exceptions caused by previously initiated instructions (e.g., it does not include waiting for the

Decrementer to step through zero). The exceptions are listed in order of highest to lowest priority. The phrase "corresponding interrupt" means the interrupt having the same name as the exception unless the thread is in power-saving mode, in which case the phrase means the System Reset interrupt.

Unless otherwise stated or obvious from context, it is assumed below that one of the following conditions is satisfied.

- The thread is not in power-saving mode and the interrupt, unless it is the Machine Check interrupt, is not disabled. (For the Machine Check interrupt no assumption is made regarding enablement.)
- The thread is in power-saving mode and the exception is enabled to cause exit from the mode.

With one exception, in the following list the hypervisor forms of the Data Storage and Instruction Storage exceptions can be substituted for the non-hypervisor forms since the hypervisor forms cannot be caused by the same instruction and have the same priority. The exception is that exceptions caused by Virtual Page Class Key Storage Protection exceptions that occur when $LPCR_{KBV}$=1 and Virtualized Partition Memory is disabled by VPM=0 cause only a Hypervisor Data Storage exception (and never a Data Storage exception).

1. System Reset

   System Reset exception has the highest priority of all exceptions. If this exception exists, the interrupt mechanism ignores all other exceptions and generates a System Reset interrupt.

   Once the System Reset interrupt is generated, no nonmaskable interrupts are generated due to exceptions caused by instructions issued prior to the generation of this interrupt.

2. Machine Check

   With one exception, the Machine Check exception is the second highest priority exception. If this exception exists and a System Reset exception does not exist, the interrupt mechanism ignores all other exceptions and generates a Machine Check interrupt. The exception is that a Machine Check caused by an attempt to access an accelerator as other than an operand of ***copy*** or ***paste.*** is prioritized similarly to a storage protection exception.

   Once the Machine Check interrupt is generated, no nonmaskable interrupts are generated due to exceptions caused by instructions issued prior to the generation of this interrupt.

3. Instruction-Caused and Precise

   This exception is the third highest priority exception. When this exception is created, the interrupt mechanism waits for all possible Imprecise exceptions to be reported. It then generates the appropriate ordered interrupt if no higher priority exception exists when the interrupt is to be generated. Within this category a particular instruction may present more than a single exception. When this occurs, those exceptions are ordered in priority as indicated in the following lists. Where [Hypervisor] Data Storage, Data Segment, and Alignment exceptions are listed in the same item they have equal priority (i.e., the hardware may generate any one of the three interrupts for which an exception exists). For instructions that are disallowed in Transactional state, and for ***mtspr*** specifying an SPR that is not part of the checkpointed registers and is not the GSR or a Transactional Memory SPR, transaction failure takes priority over all interrupts except Privileged Instruction type Program interrupts, Hypervisor Emulation Assistance interrupts, and [Hypervisor] Facility Unavailable interrupts. For data accesses that are disallowed in Transactional state, transaction failure has the same priority as the group of “other” [Hypervisor] Data Storage, Data Segment, and Alignment exceptions. (See Section 5.3.1 of Book II.)

   A. Fixed-Point Loads and Stores
      a. These exceptions are mutually exclusive and have the same priority:
         - Hypervisor Emulation Assistance
         - Program - Privileged Instruction
      b. Hypervisor Facility Unavailable
      c. Facility Unavailable
      d. Data Storage for the case of *Fixed-Point Load or Store Caching Inhibited* instructions with MSR_{DR}=1 or the case of an invalid function code for an Atomic Memory Operation
      e. all other Data Storage, Hypervisor Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment
      f. Trace

   B. Floating-Point Loads and Stores
      a. Hypervisor Emulation Assistance
      b. Hypervisor Facility Unavailable
      c. Floating-Point Unavailable
      d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment
      e Trace

   C. Vector Loads and Stores
      a. Hypervisor Emulation Assistance
      b. Hypervisor Facility Unavailable
      c. Vector Unavailable
      d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment
      e. Trace

   D. VSX Loads and Stores

a. Hypervisor Emulation Assistance
b. Hypervisor Facility Unavailable
c. VSX Unavailable
d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, Machine Check for invalid accelerator access, or Alignment
e. Trace

E. Other Floating-Point Instructions
a. Hypervisor Emulation Assistance
b. Hypervisor Facility Unavailable
c. Floating-Point Unavailable
d. Program - Precise Mode Floating-Point Enabled Exception
e. Trace

F. Other Vector Instructions
a. Hypervisor Emulation Assistance
b. Hypervisor Facility Unavailable
c. Vector Unavailable
d. Trace

G. Other VSX Instructions
a. Hypervisor Emulation Assistance
b. Hypervisor Facility Unavailable
c. VSX Unavailable
d. Program - Precise Mode Floating-Point Enabled Exception
e. Trace

H. TM instruction, ***mt/fspr*** specifying TM SPR
a. Program - Privileged Instruction (only for ***treclaim.*** and ***trechkpt.***)
b. Hypervisor Facility Unavailable
c. Facility Unavailable
d. Program - TM Bad Thing (only for ***treclaim.***, ***trechkpt.***, and ***mtspr***)
e. Trace

I. ***rfid, hrfid, rfebb***, ***rfscv***, and ***mtmsr***[***d***]
a. These exceptions are mutually exclusive and have the same priority:
- Program - Privileged Instruction, for all except ***rfebb***
- Hypervisor Emulation Assistance, for ***hrfid*** only

b. Hypervisor Facility Unavailable (***rfebb*** only)
c. Facility Unavailable (***rfebb*** only)
d. Program - TM Bad Thing for all except ***mtmsr***
e. Program - Floating-Point Enabled Exception or all except ***rfebb***
f. Trace, for ***mtmsr***[***d***] and ***rfebb*** only

J. Other Instructions
a. These exceptions or groups of exceptions are mutually exclusive and have the same priority (the members of a group are not mutually exclusive, but have the same priority):
- Program - Trap
- System Call
- System Call Vectored
- Hypervisor Emulation Assistance or Program (Privileged Instruction)

b. Hypervisor Facility Unavailable
c. Facility Unavailable
d. Trace

K. [Hypervisor] Instruction Storage and Instruction Segment

These exceptions have the lowest priority in this category. They are recognized only when all instructions prior to the instruction causing one of these exceptions appear to have completed and that instruction is the next instruction to be executed. The two exceptions are mutually exclusive.

The priority of these exceptions is specified for completeness and to ensure that they are not given more favorable treatment. It is acceptable for an implementation to treat these exceptions as though they had a lower priority.

4. Program - Imprecise Mode Floating-Point Enabled Exception

This exception is the fourth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

5. Hypervisor Maintenance

This exception is the fifth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

If a Hypervisor Maintenance exception exists and each attempt to execute an instruction when the Hypervisor Maintenance interrupt is enabled causes an exception (see the Programming Note below), the Hypervisor Maintenance interrupt is not delayed indefinitely.

6. Hypervisor Virtualization, Direct External, Mediated External, and [Hypervisor] Decrementer, Performance Monitor, Directed Privileged Doorbell, Directed Hypervisor Doorbell

These exceptions are the lowest priority exceptions. All have equal priority (i.e., the hardware may generate any one of the corresponding interrupts for which an exception exists). When one of these exceptions is created, the interrupt processing mechanism waits for all other possible exceptions to be reported. It then generates the corresponding interrupt if no higher priority exception exists when the interrupt is to be generated.

If a Hypervisor Decrementer exception exists and each attempt to execute an instruction when the

Hypervisor Decrementer interrupt is enabled causes an exception (see the Programming Note below), the Hypervisor Decrementer interrupt is not delayed indefinitely.

If LPES=1 and a Direct External exception exists and each attempt to execute an instruction when this interrupt is enabled causes an exception (see the Programming Note below), the Direct External interrupt is not delayed indefinitely.

Programming Note

An incorrect or malicious operating system could corrupt the first instruction in the interrupt vector location for an instruction-caused interrupt such that the attempt to execute the instruction causes the same exception that caused the interrupt (a looping interrupt; e.g., *Trap* instruction and Program interrupt). Similarly, the first instruction of the interrupt vector for one instruction-caused interrupt could cause a different instruction-caused interrupt, and the first instruction of the interrupt vector for the second instruction-caused interrupt could cause the first instruction-caused interrupt (e.g., Program interrupt and Floating-Point Unavailable interrupt). The looping caused by these and similar cases is terminated by the occurrence of a System Reset or Hypervisor Decrementer interrupt.

6.10 Relationship of Event-Based Branches to Interrupts

6.10.1 EBB Exception Priority

Event-based branches have a priority lower than that of all interrupts. When an event-based exception is created, the Event-Based Branch facility waits for all possible exceptions that would cause interrupts to be reported. It then generates the event-based branch if no exception that would cause an interrupt exists when the event-based branch is to be generated.

6.10.2 EBB Synchronization

When an event-based branch occurs, EBBRR is set to point to an instruction such that all preceding instructions have completed execution, no subsequent instruction has begun execution, and the instruction addressed by EBBRR has not completed execution.

6.10.3 EBB Classes

Event-based branches are classified by whether they are directly caused by the execution of an instruction or are caused by some other system exception. Those that are “system-caused” are

- Performance Monitor
- External

7.

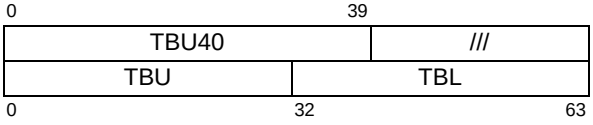
Chapter 7. Timer Facilities

7.1 Overview

The Time Base, Decrementer, Hypervisor Decrementer, Processor Utilization of Resources, and Scaled Processor Utilization of Resources registers provide timing functions for the system. The remainder of this section describes these registers and related facilities.

7.2 Time Base (TB)

The Time Base (TB) is a 64-bit register (see Figure 67) containing a 64-bit unsigned integer that is incremented periodically.



Field	Description
TBU40	Upper 40 bits of Time Base
TBU	Upper 32 bits of Time Base
TBL	Lower 32 bits of Time Base

Figure 67. Time Base

The Time Base is a hypervisor resource; see Chapter 2.

The SPRs TBU40, TBU, and TBL provide access to the fields of the Time Base shown in Figure 67. When a ***mtspr*** instruction is executed specifying one of these SPRs, the associated field of the Time Base is altered and the remaining bits of the Time Base are not affected.

See Chapter 6 of Book II for infromation about the update frequency of the Time Base.

The Time Base is implemented such that:

1. Loading a GPR from the Time Base has no effect on the accuracy of the Time Base.
2. Copying the contents of a GPR to the Time Base replaces the contents of the Time Base with the contents of the GPR.

The Power ISA does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock in a Power ISA system. The Time Base update frequency is not required to be constant. What *is* required, so that system software can keep time of day and operate interval timers, is one of the following.

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the Time Base changes, and a means to determine what the current update frequency is.
- The update frequency of the Time Base is under the control of the system software.

Implementations must provide a means for either preventing the Time Base from incrementing or preventing it from being read in problem state (MSR_{PR}=1). If the means is under software control, it must be accessible only in hypervisor state ($MSR_{HV\ PR}$ = 0b10). There must be a method for getting all Time Bases in the system to start incrementing with values that are identical or almost identical.

Programming Note

If software initializes the Time Base on power-on to some reasonable value and the update frequency of the Time Base is constant, the Time Base can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the Time Base are monotonically increasing (except when the Time Base wraps from 2^{64}-1 to 0). If a trace entry is recorded each time the update frequency changes, the sequence of Time Base values can be post-processed to become actual time values.

Successive readings of the Time Base may return identical values.

If Time Base bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0x0 only when bit 59 changes state regardless of whether or not they incremented to 0xF since they were previously set to 0x0.

See the description of the Time Base in Chapter 6 of Book II for ways to compute time of day in POSIX format from the Time Base.

7.2.1 Writing the Time Base

Writing the Time Base is privileged, and can be done only in hypervisor state. Reading the Time Base is not privileged; it is discussed in Chapter 6 of Book II.

It is not possible to write the entire 64-bit Time Base using a single instruction. The ***mttbl*** and ***mttbu*** extended mnemonics write the lower and upper halves of the Time Base (TBL and TBU), respectively, preserving the other half. These are extended mnemonics for the ***mtspr*** instruction; Figure 18.

The Time Base can be written by a sequence such as:

```
lwz    Rx,upper # load 64-bit value for
lwz    Ry,lower # TB into Rx and Ry
li     Rz,0
mttbl  Rz       # set TBL to 0
mttbu  Rx       # set TBU
mttbl  Ry       # set TBL
```

Provided that no interrupts occur while the last three instructions are being executed, loading 0 into TBL prevents the possibility of a carry from TBL to TBU while the Time Base is being initialized.

The preferred method of changing the Time Base utilizes the TBU40 facility. The following code sequence demonstrates the process. Assume the upper 40 bits of Rx contain the desired value upper 40 bits of the Time Base.

```
mftb    Ry       # Read 64-bit Time Base value
clrldi  Ry,Ry,40 # lower 24 bits of old TB
mttbu40 Rx       # write upper 40 bits of TB
mftb    Rz       # read TB value again
clrldi  Rz,Rz,40 # lower 24 bits of new TB
cmpld   Rz,Ry    # compare new and old lwr 24
bge     done     # no carry out of low 24 bits
addis   Rx,Rx,0x0100
                 #increment upper 40 bits
mttbu40 Rx       # update to adjust for carry
```

Programming Note

The instructions for writing the Time Base are mode-independent. Thus code written to set the Time Base will work correctly in either 64-bit or 32-bit mode.

7.3 Virtual Time Base

The Virtual Time Base (VTB) is a 64-bit incrementing counter.

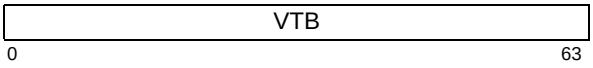


Figure 68. Virtual Time Base

Virtual Time Base increments at the same rate as the Time Base until its value becomes 0xFFFF_FFFF_FFFF_FFFF (2^{64} - 1); at the next increment its value becomes 0x0000_0000_0000_0000. There is no interrupt or other indication when this occurs.

The operation of the Virtual Time Base has the following additional properties.

1. Loading a GPR from the Virtual Time Base has no effect on the accuracy of the Virtual Time Base.
2. Copying the contents of a GPR to the Virtual Time Base replaces the contents of the Virtual Time Base with the contents of the GPR.

Programming Note

In systems that change the Time Base update frequency for purposes such as power management, the Virtual Time Base input frequency will also change. Software must be aware of this in order to set interval timers.

Programming Note

In configurations in which the hypervisor allows multiple partitions to time-share a processor, the Virtual Time Base can be managed by the hypervisor such that it appears to each partition as if it counts only during the times that the partition is executing.

In order to do this, the hypervisor saves the value of the Virtual Time Base as part of the program context when removing a partition from the processor, and restores it to its previous value when initiating the partition again on the same or another processor.

7.4 Decrementer

The Decrementer (DEC) is a decrementing counter that provides a mechanism for causing a Decrementer interrupt after a programmable delay.

The Decrementer is driven at the same frequency as the Time Base.

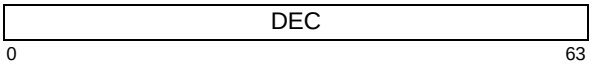


Figure 69. Decrementer

The LPCR is used to enable and disable Large Decrementer mode, as defined below. (See Section 2.2.)

When the Decrementer is not in Large Decrementer mode, it behaves as a 32-bit signed integer and operates as follows.

The Decrementer counts down until its value becomes 0x0000_0000_0000_0000; at the next decrement its value becomes 0x0000_0000_FFFF_FFFF. When reading the Decrementer using ***mfspr***, bits 0:31 always read back as 0s.

When the contents of DEC_{32} change from 0 to 1, a Decrementer exception will come into existence within a reasonable period of time. When the contents of DEC_{32} change from 1 to 0, the existing Decrementer exception, if any, will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event.

The preceding paragraph applies regardless of whether the change in the contents of DEC_{32} is the result of decrementation of the Decrementer by the hardware or of modification of the Decrementer caused by execution of an ***mtspr*** instruction.

When the Decrementer is in Large Decrementer mode, it behaves as a d-bit decrementing counter which is sign-extended to 64 bits. The value of d is implementation dependent but at least 32. When the Decrementer is written, bits 0:63-d are ignored by the hardware.

Programming Note

In Large Decrementer mode, the maximum positive value supported by the Decrementer is 2^{d-1}-1, represented with bits 0:64-d containing 0's and bits 65-d:63 containing 1's. The minimum value supported by the Decrementer is -2^{d-1}, represented as 0xFFFF_FFFF_FFFF_FFFF.

When in Large Decrementer mode, the Decrementer operates as follows.

The binary value of the Decrementer counts down until its value becomes 0x0000_0000_0000_0000; at the next decrement its value becomes the minimum value supported, which is represented as 0xFFFF_FFFF_FFFF_FFFF.

When the contents of the DEC_0 change from 0 to 1, a Decrementer exception will come into existence within a reasonable period of time. When the contents of DEC_0 change from 1 to 0, the existing Decrementer exception, if any, will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event.

The preceding paragraph applies regardless of whether the change in the contents of DEC_0 is the result of decrementation of the Decrementer by the hardware or of modification of the Decrementer caused by execution of an ***mtspr*** instruction.

The operation of the Decrementer has the following additional properties.

1. Loading a GPR from the Decrementer has no effect on the accuracy of the Time Base.
2. Copying the contents of a GPR to the Decrementer replaces the contents of the Decrementer with the contents of the GPR.

Programming Note

In systems that change the Time Base update frequency for purposes such as power management, the Decrementer input frequency will also change. Software must be aware of this in order to set interval timers.

If Decrementer bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0xF only when bit 59 changes state regardless of whether or not they decremented to 0x0 since they were previously set to 0xF.

7.4.1 Writing and Reading the Decrementer

The contents of the Decrementer can be read or written using the ***mfspr*** and ***mtspr*** instructions, both of which are privileged when they refer to the Decrementer. Using an extended mnemonic (Figure 18), the Decrementer can be written from GPR Rx using:

```
mtdec  Rx
```

The Decrementer can be read into GPR Rx using:

```
mfdec  Rx
```

Copying the Decrementer to a GPR has no effect on the Decrementer contents or on the interrupt mechanism.

7.5 Hypervisor Decrementer

The Hypervisor Decrementer is a h-bit decrementing counter that is sign-extended to 64 bits. The value of h is implementation dependent, however the number of bits supported by the Hypervisor Decrementer must be greater than or equal to the number of bits supported by the Decrementer. When the Decrementer is written, bits 0:63-h are ignored by the hardware.

Programming Note

The maximum positive value supported by the Hypervisor Decrementer is 2^{h-1}-1, represented with bits 0:64-h containing 0's and bits 65-h:63 containing 1's. The minimum value supported by the Hypervisor Decrementer is -2^{h-1}, represented as 0xFFFF_FFFF_FFFF_FFFF.

The binary value of the Hypervisor Decrementer counts down until its value becomes 0x0000_0000_0000_0000; at the next decrement its value becomes the minimum value supported, which is represented as 0xFFFF_FFFF_FFFF_FFFF.

When the contents of $HDEC_0$ change from 0 to 1 and the thread is not in a power-saving mode, a Hypervisor Decrementer exception will come into existence within a reasonable period of time. When a Hypervisor Decrementer interrupt occurs, the existing Hypervisor Decrementer exception will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event. Even if multiple $HDEC_0$ change transitions from 0 to 1 occur before a Hypervisor Decrementer interrupt occurs, at most one Hypervisor Decrementer exception exists.

The preceding paragraph applies regardless of whether the change in the contents of $HDEC_0$ is the result of decrementation of the Hypervisor Decrementer by the hardware or of modification of the Hypervisor Decrementer caused by execution of an ***mtspr*** instruction.

The operation of the Hypervisor Decrementer has the following additional properties.

1. Loading a GPR from the Hypervisor Decrementer has no effect on the accuracy of the Hypervisor Decrementer.
2. Copying the contents of a GPR to the Hypervisor Decrementer replaces the contents of the Hypervisor Decrementer with the contents of the GPR.

Programming Note

In systems that change the Time Base update frequency for purposes such as power management, the Hypervisor Decrementer update frequency will also change. Software must be aware of this in order to set interval timers.

If Hypervisor Decrementer bits 60:63 are used as part of a random number generator, software must account for the fact that these bits are set to 0xF only when bit 59 changes state regardless of whether or not they decremented to 0x0 since they were previously set to 0xF.

Programming Note

A Hypervisor Decrementer exception is not created if the thread is in a power-saving mode when $HDEC_0$ changes from 0 to 1 because having a Hypervisor Decrementer interrupt occur almost immediately after exiting the power-saving mode in this case is deemed unnecessary. The hypervisor already has control, and if a timed exit from the power-saving mode is necessary and possible, the hypervisor can use the Decrementer to exit the power-saving mode at the appropriate time. For some power-saving levels, the state of the Hypervisor Decrementer and Decrementer is not necessarily maintained and updated.

7.6 Processor Utilization of Resources Register (PURR)

The Processor Utilization of Resources Register (PURR) is a 64-bit counter, the contents of which provide an estimate of the resources used by the thread. The contents of the PURR are treated as a 64-bit unsigned integer.

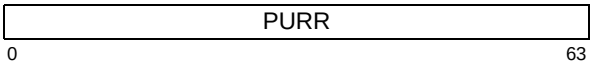


Figure 70. Processor Utilization of Resources Register

The PURR is a hypervisor resource; see Chapter 2.

The contents of the PURR increase monotonically, unless altered by software, until the sum of the contents plus the amount by which it is to be increased exceed 0xFFFF_FFFF_FFFF_FFFF (2^{64} - 1) at which point the contents are replaced by that sum modulo 2^{64}. There is no interrupt or other indication when this occurs.

The rate at which the value represented by the contents of the PURR increases is an estimate of the portion of resources used by the thread per unit time with respect to other threads that share those resources monitored by the PURR. When the thread is idle, the rate at which the PURR value increases is implementation dependent.

Let the difference between the value represented by the contents of the Time Base at times T_a and T_b be T_{ab}. Let the difference between the value represented by the contents of the PURR at time T_a and T_b be the value P_{ab}. The ratio of P_{ab}/T_{ab} is an estimate of the percentage of shared resources used by the thread during the interval T_{ab}. For the set {S} of threads that share the resources monitored by the PURR, the sum of the usage estimates for all the threads in the set is 1.0.

The definition of the set of threads S, the shared resources corresponding to the set S, and specifics of the algorithm for incrementing the PURR are implementation-specific.

The PURR is implemented such that:

1. Loading a GPR from the PURR has no effect on the accuracy of the PURR.
2. Copying the contents of a GPR to the PURR replaces the contents of the PURR with the contents of the GPR.

Programming Note

Estimates computed as described above may be useful for purposes related to resource utilization, including utilization-based system management and planning.

Because the rate at which the PURR accumulates resource usage estimates is dependent on the frequency at which the Time Base is incremented, and the frequency of the oscillator that drives instruction execution may vary independently from that of the Time Base, the interpretation of the contents of the PURR may be inaccurate as a measurement of capacity consumption for accounting purposes. The SPURR should be used for accounting purposes.

7.7 Scaled Processor Utilization of Resources Register (SPURR)

The Scaled Processor Utilization of Resources Register (SPURR) is a 64-bit counter, the contents of which provide an estimate of the resources used by the thread. The contents of the SPURR are treated as a 64-bit unsigned integer.

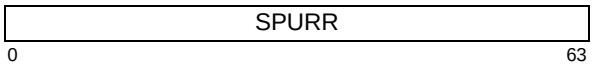


Figure 71. Scaled Processor Utilization of Resources Register

The SPURR is a hypervisor resource; see Section 2.6.

The contents of the SPURR increase monotonically, unless altered by software, until the sum of the contents plus the amount by which it is to be increased exceed 0xFFFF_FFFF_FFFF_FFFF (2^{64} - 1) at which point the contents are replaced by that sum modulo 2^{64}. There is no interrupt or other indication when this occurs.

The rate at which the value represented by the contents of the SPURR increases is an estimate of the portion of resources used by the thread with respect to other threads that share those resources monitored by the SPURR, and relative to the computational capacity provided by those resources. The computational capacity provided by the shared resources may vary as a function of the frequency of the oscillator which drives the resources or as a result of deliberate delays in processing that are created to reduce power consumption. When the thread is idle, the rate at which the SPURR value increases is implementation dependent.

Let the difference between the value represented by the contents of the Time Base at times T_a and T_b be T_{ab}. Let the ratio of the effective and nominal frequencies of the oscillator driving instruction execution f_e/f_n be f_r. Let the ratio of delay cycles created by power reduction circuitry and total cycles c_d/c_t be c_r. Let the difference between the value represented by the contents of the SPURR at time T_a and T_b be the value S_{ab}. The ratio of $S_{ab}/(T_{ab} \times f_r \times (1 - c_r))$ is an estimate of the percentage of shared resource capacity used by the thread during the interval T_{ab}. For the set {S} of threads that share the resources monitored by the SPURR, the sum of the usage estimates for all the threads in the set is 1.0.

The definition of the set of threads S, the shared resources corresponding to the set S, and specifics of the algorithm for incrementing the SPURR are implementation-specific.

The SPURR is implemented such that:

1. Loading a GPR from the SPURR has no effect on the accuracy of the SPURR.

2. Copying the contents of a GPR to the SPURR replaces the contents of the SPURR with the contents of the GPR.

Programming Note

Estimates computed as described above may be useful for purposes of resource use accounting, program dispatching, etc.

7.8 Instruction Counter

The Instruction Counter (IC) is a 64-bit incrementing counter that counts the number of instructions that the thread has completed (according to the sequential execution model; see Section 2.2 of Book I).

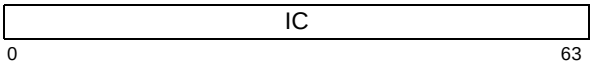


Figure 72. Instruction Counter

Chapter 8. Debug Facilities

8.1 Overview

Implementations provide debug facilities to enable hardware and software debug functions, such as control flow tracing, data address watchpoints, and program single-stepping. The debug facilities described in this section consist of the Come-From Address Register (see Section 8.2), Completed Instruction Address Breakpoint Register (see Section 8.3), and the Data Address Watchpoint Register (DAWRn) and Data Address Watchpoint Register Extension (DAWRXn) (see Section 8.4). The interrupt associated with the Data Address Breakpoint registers is described in Section 6.5.3. The interrupt associated with the Completed Instruction Address Breakpoint Register is described in Section 6.5.15. The Trace facility, which can be used for single-stepping as well as for control flow tracing, is described in Section 6.5.15.

The ***mfspr*** and ***mtspr*** instructions (see Section 4.4.4) provide access to the registers of the debug facilities.

In addition to the facilities mentioned above, implementations typically provide debug facilities, modes, and access mechanisms that are implementation-specific. For example, implementations typically provide facilities for instruction address tracing, and also access to certain debug facilities via a dedicated interface such as the IEEE 1149.1 Test Access Port (JTAG).

8.2 Come-From Address Register

The Come-From Address Register (CFAR) is a 64-bit register. When an ***rfebb***, ***rfid***, or ***rfscv*** instruction is executed, the register is set to the effective address of the instruction. When a *Branch* instruction is executed and the branch is taken, the register is set to the effective address of an instruction in the instruction cache block containing the *Branch* instruction, except that if the *Branch* instruction is a B-form *Branch* (i.e., ***bc***, ***bca***, ***bcl***, or ***bcla***) for which the target address is in the instruction cache block containing the *Branch* instruction or is in the previous or next cache block, the register is not necessarily set. For *Branch* instructions, the setting need not occur until a subsequent context synchronizing operation has occurred.

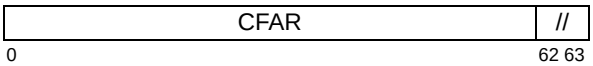


Figure 73. Come-From Address Register

The contents of the CFAR can be read and written using the ***mfspr*** and ***mtspr*** instructions. Acccess to the CFAR is privileged.

Programming Note

This register can be used for purposes of debugging software. For example, often a software bug results in the program executing a portion of the code that it should not have reached or causing an unexpected interrupt. In the former case, a breakpoint can be placed in the portion of the code that was erroneously reached and the program reexecuted. In either case, the interrupt handler can save the contents of the CFAR (before executing the first instruction that would modify the register), and then make the saved contents available for a debugger to use in determining the control flow path by which the exception was reached.

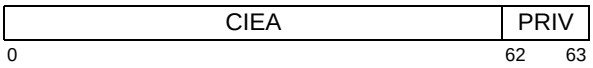
In order to preserve the CFAR's contents for each partition and to prevent it from being used to implement a "covert channel" between partitions, the hypervisor should initialize/save/restore the CFAR when switching partitions on a given thread.

8.3 Completed Instruction Address Breakpoint

The Completed Instruction Address Breakpoint mechanism provides a means of detecting an instruction completion at a specific instruction address. The address comparison is done on an effective address (EA).

The Completed Instruction Address Breakpoint mechanism is controlled by the Completed Instruction

Address Breakpoint Register (CIABR), shown in Figure 75.



Bit(s)	Name	Description
0:61	CIEA	Completed Instruction Effective Address
62:63	PRIV	Privilege 00: Disable matching 01: Match in problem state 10: Match in privileged (non-hypervisor) state 11: Match in hypervisor state

Figure 74. Completed Instruction Address Breakpoint Register

A Completed Instruction Address Breakpoint match occurs upon instruction completion if all of the following conditions are satisfied.

- the completed instruction address is equal to $CIEA_{0:61}$ || 0b00.
- the thread run level matches that specified in RLM.

In 32-bit mode the high-order 32 bits of the EA are treated as zeros for the purpose of detecting a match.

A Completed Instruction Address Breakpoint match causes a Trace exception provided that no higher priority interrupt occurs from the completion of the instruction (see Section 6.5.15).

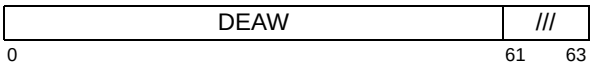
8.4 Data Address Watchpoint

The Data Address Watchpoint mechanism provides a means of detecting load and store accesses to a range of addresses starting at a designated doubleword. The address comparison is done on an effective address (EA).

Programming Note

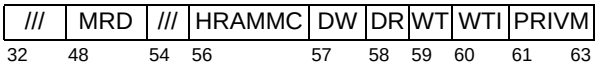
The Data Address Watchpoint mechanism employs a simple EA compare. It makes no attempt to take the radix table translation quadrants (keyed off $EA_{0:1}$) into account to enable a single setting to work in all privilege levels.

The Data Address Watchpoint mechanism is controlled by a single set of SPRs, numbered with n=0: the Data Address Watchpoint Register (DAWRn), shown in Figure 75, and the Data Address Watchpoint Register Extension (DAWRXn), shown in Figure 76.



Bit(s)	Name	Description
0:60	DEAW	Data Effective Address Watchpoint

Figure 75. Data Address Watchpoint Register



Bit(s)	Name	Description
48:53	MRD	Match Range in Doublewords biased by -1. (0b000000 = 1 DW, 0b111111 = 64 DW)
56	HRAMMC	Hypervisor Real Addressing Mode Match Control 0: $DEAW_0$ and EA_0 are used during matching in hypervisor real addressing mode 1: $DEAW_0$ and EA_0 are ignored during matching in hypervisor real addressing mode
57	DW	Data Write
58	DR	Data Read
59	WT	Watchpoint Translation
60	WTI	Watchpoint Translation Ignore
61:63	PRIVM	Privilege Mask
61	HYP	Hypervisor state
62	PNH	Privileged but Non-Hypervisor state
63	PRO	Problem state

All other fields are reserved.

Figure 76. Data Address Watchpoint Register Extension

The supported PRIVM values are 0b000, 0b001, 0b010, 0b011, 0b100, and 0b111. If the PRIVM field does not contain one of the supported values, then whether a match occurs for a given storage access is undefined. Elsewhere in this section it is assumed that the PRIVM field contains one of the supported values.

Programming Note

PRIVM value 0b000 causes matches not to occur regardless of the contents of other DAWRn and DAWRXn fields. PRIVM values 0b101 and 0b110 are not supported because a storage location that is shared between the hypervisor and non-hypervisor software is unlikely to be accessed using the same EA by both the hypervisor and the non-hypervisor software. (PRIVM value 0b111 is supported primarily for reasons of software compatibility with respect to emulation of the DABR facility as described in a subsequent Programming Note.)

A Data Address Watchpoint match occurs for a *Load* or *Store* instruction, or for an instruction that is treated as a *Load* or *Store*, if, for any byte accessed, all of the following conditions are satisfied.

- the access is
  - a quadword access and located in the range $(DEAW_{0:59} \,||\, 0b0) \le (EA_{0:59} \,||\, 0b0) \le ((DEAW_{0:59} \,||\, 0b0) + (^{55}0 \,||\, MRD_{0:4} \,||\, 0b0))$ such that $(EA_{0:60}$ AND $(^{55}1 \,||\, ^{6}0)) = (DEAW_{0:60}$ AND $(^{55}1 \,||\, ^{6}0))$.
  - not a quadword access and located in the range $DEAW_{0:60} \le EA_{0:60} \le (DEAW_{0:60} + (^{55}0 \,||\, MRD_{0:5}))$ such that $(EA_{0:60}$ AND $(^{55}1 \,||\, ^{6}0)) = (DEAW_{0:60}$ AND $(^{55}1 \,||\, ^{6}0))$.
- $(MSR_{DR} = DAWRXn_{WT}) \mid DAWRXn_{WTI}$
- the thread is in
  - hypervisor state and $DAWRXn_{HYP} = 1$, or
  - privileged but non-hypervisor state and $DAWRXn_{PNH} = 1$, or
  - problem state and $DAWRXn_{PR} = 1$
- the instruction is a *Store* or treated as a *Store* and $DAWRXn_{DW} = 1$, or the instruction is a *Load* or treated as a *Load* and $DAWRXn_{DR} = 1$.

In 32-bit mode the high-order 32 bits of the EA are treated as zeros for the purpose of detecting a match.

If the above conditions are satisfied, it is undefined whether a match occurs in the following cases.

- The instruction is *Store Conditional* but the store is not performed
- The instruction is ***dcbz***. (For the purpose of determining whether a match occurs, ***dcbz*** is treated as a *Store*.)

The *Cache Management* instructions other than ***dcbz*** never cause a match.

A Data Address Watchpoint match causes a Data Storage exception or a Hypervisor Data Storage exception (see Section 6.5.3, “Data Storage Interrupt” on page 1069 and Section 6.5.16, “Hypervisor Data Storage Interrupt” on page 1078). If a match occurs, some or all of the bytes of the storage operand may have been accessed; however, if a *Store* instruction causes the match, the storage operand is not modified if the instruction is one of the following:

- any *Store* instruction that causes an atomic access

Programming Note

The Data Address Watchpoint mechanism does not apply to instruction fetches.

Programming Note

Implementations that comply with versions of the architecture that precede Version 2.02 do not provide the DABRX (now replaced by DAWRXn). Forward compatibility for software that was written for such implementations (and uses the Data Address Breakpoint facility) can be obtained by setting $DAWRXn_{60:63}$ to 0b0111.

Chapter 9. Performance Monitor Facility

9.1 Overview

The Performance Monitor facility provides a means of collecting information about program and system performance.

9.2 Performance Monitor Operation

The Performance Monitor facility includes the following features.

- an MSR bit
  - PMM (Performance Monitor Mark), which can be used to select one or more programs for monitoring
- registers
  - PMC1 - PMC6 (Performance Monitor Counters 1 - 6), which count events
  - MMCR0, MMCR1, MMCR2, and MMCRA (Monitor Mode Control Registers 0, 1, 2, and A), which control the Performance Monitor facility
  - SIAR, SDAR, and SIER (Sampled Instruction Address Register, Sampled Data Address Register, and Sampled Instruction Event Register), which contain the address of the "sampled instruction" and of the "sampled data," and additional information about the "sampled instruction" (see Section 9.4.8 - Section 9.4.10).
- the Performance Monitor interrupt and Performance Monitor event-based branch, which can be caused by monitored conditions and events.

Many aspects of the operation of the Performance Monitor are summarized by the following hierarchy, which is described starting at the lowest level.

- A "counter negative condition" exists when the value in a PMC is negative (i.e., when bit 0 of the PMC is 1). A "Time Base transition event" occurs when a selected bit of the Time Base changes from 0 to 1 (the bit is selected by a field in MMCR0). The term "condition or event" is used as an abbreviation for "counter negative condition or Time Base transition event". A condition or event can be caused implicitly by the hardware (e.g., incrementing a PMC) or explicitly by software (***mtspr***).
- A condition or event is enabled if the corresponding "Enable" bit (i.e., PMC1CE, PMCjCE, or TBEE) in MMCR0 is 1. The occurrence of an enabled condition or event can have side effects within the Performance Monitor, such as causing the PMCs to cease counting.
- An enabled condition or event causes a Performance Monitor alert if Performance Monitor alerts are enabled by the corresponding "Enable" bit in MMCR0. Another cause of a Performance Monitor alert is the threshold event counter reaching its maximum value (see Section 9.4.3). A single Performance Monitor alert may reflect multiple enabled conditions and events.
- When a Performance Monitor alert occurs, $MMCR0_{PMAO}$ is set to 1 and the writing of BHRB entries, if in process, is suspended.

  When the contents of $MMCR0_{PMAO}$ change from 0 to 1, a Performance Monitor exception will come into existence within a reasonable period of time. When the contents of $MMCR0_{PMAO}$ change from 1 to 0, the existing Performance Monitor exception, if any, will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event.
- A Performance Monitor exception causes one of the following.
  - If MSR_{EE} = 1, $MMCR0_{EBE}$ = 0, and either $HFSCR_{PM}$=1 or the thread is in hypervisor state, an interrupt occurs.
  - If MSR_{PR} = 1, $MMCR0_{EBE}$ = 1, a Performance Monitor event-based exception occurs if $BESCR_{PME}$=1, provided that event-based exceptions are enabled by $FSCR_{EBB}$ and $HFSCR_{EBB}$. When a Performance Monitor

event-based exception occurs, an event-based branch is generated if BESCR$_{GE}$=1.

Programming Note

The Performance Monitor can be effectively disabled (i.e., put into a state in which Performance Monitor SPRs are not altered and Performance Monitor exceptions do not occur) by setting MMCR0 to 0x0000_0000_8000_0000.

The Performance Monitor also controls when BHRB entries are written, the instruction filters that are used when writing BHRB entries, and the availability of the BHRB in problem state. It also controls whether Performance Monitor exceptions cause Performance Monitor event-based exceptions or Performance Monitor interrupts. See Section 9.4.4.

9.3 No-op Instructions Reserved for the Performance Monitor

The following forms of the ***and*** *x,x,x* instruction are reserved for exclusive use by the Performance Monitor.

- ***and*** x,x,x, where x=0,1.

Programming Note

An example usage of a probe no-op by the Performance Monitor is to measure branch prediction effectiveness. In order to do this, one of probe no-ops is inserted in various sections of the code in which branch prediction efficiency is being studied. The Performance Monitor registers are then set up as follows.

MMCRA:
ES=010 (only probe no-ops eligible for sampling)
SM=00 (all eligible instructions)
SE=1 (enable random sampling).
Other fields in MMCRA are set as desired.

MMCR1:
PMC1SEL=E0 (count PMC1 on dispatch)
PMC4SEL=E0 (count PMC4 on completion)
Other counters initialized as desired.

MMCR2: Initialize as desired.

MMCR0:
FC is set to 0 to stop freezing the counters
PMAE is set to 1 to enable PMU alerts.
Other fields in MMCR0 are set as desired.

Subsequently, when a PMU alert occurs, PMCs 1 and 4 can be read. The difference between the two counter values provides an indication of branch prediction effectiveness in the areas of the code in which the probe no-op was inserted.

9.4 Performance Monitor Facility Registers

The Performance Monitor registers count events, control the operation of the Performance Monitor, and provide associated information.

The elapsed time between the execution of an instruction and the time at which events due to that instruction have been reflected in Performance Monitor registers is not defined. No means are provided by which software can ensure that all events due to preceding instructions have been reflected in Performance Monitor registers. Similarly, if the events being monitored may be caused by operations that are performed out-of-order, no means are provided by which software can prevent such events due to subsequent instructions from being reflected in Performance Monitor registers. Thus the contents obtained by reading a Performance Monitor register may not be precise: it may fail to reflect some events due to instructions that precede the ***mfspr*** and may reflect some events due to instructions that follow the ***mfspr***. This lack of precision applies regardless of whether the state of the thread is such that the register is subject to change by the hardware at the time the ***mfspr*** is executed. Similarly, if an ***mtspr*** instruction is executed that changes the contents of the Time Base, the change is not guaranteed to have taken effect with respect to causing Time Base transition events until after a subsequent context synchronizing instruction has been executed.

If an ***mtspr*** instruction is executed that changes the value of a Performance Monitor register other than SIAR, SDAR, and SIER, the change is not guaranteed to have taken effect until after a subsequent context synchronizing instruction has been executed (see Chapter 11. “Synchronization Requirements for Context Alterations” on page 1133).

Programming Note

Depending on the events being monitored, the contents of Performance Monitor registers may be affected by aspects of the runtime environment (e.g., cache contents) that are not directly attributable to the programs being monitored.

9.4.1 Performance Monitor SPR Numbers

The Performance Monitor registers have two sets of SPR numbers, one set that is non-privileged and another set that is privileged.

For the purpose of explanation elsewhere in the architecture, the non-privileged registers are divided into two groups as defined below.

- A: The non-privileged read/write Performance Monitor registers (i.e., the PMCs, MMCR0, MMCR2, and MMCRA at SPR numbers 771-776, 779, 769, and 770, respectively)
- B: The non-privileged read-only Performance Monitor registers (i.e., SIER, SIAR, SDAR, and MMCR1 at SPR numbers 768, 780, 781, and 782, respectively).

The SPRs in group B are treated as undefined registers for write (***mtspr***) operations. See the ***mtspr*** instruction description in Section 4.4.4 for additional information.

When the PCR makes a register in either group A or B unavailable in problem state, that SPR is not included in group A or B.

Programming Note

Older versions of Performance Monitor facilities used diffefrent sets of SPR numbers from those shown in Section 4.4.4. (All 32-bit PowerPC implementations used a different set.

9.4.2 Performance Monitor Counters

The six Performance Monitor Counters, PMC1 through PMC6, are 32-bit registers that count events.

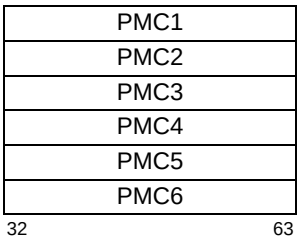


Figure 77. Performance Monitor Counter registers

PMC1 - PMC4 are referred to as “programmable” counters since the events that can be counted can be specified by the program. The events that are counted by each counter are specified in MMCR1.

PMC5 and PMC6 are not programmable and can be specified as being part of the Performance Monitor Facility or not part of it. PMC5 counts instructions completed, and PMC6 counts cycles. The PMCC field in MMCR0 controls whether or not PMCs 5-6 are part of the Performance Monitor Facility, and the result of accessing these counters when they are not part of the Performance Monitor Facility.

Programming Note

PMC5 and PMC6 are defined to facilitate calculating basic performance metrics such as cycles per instruction (CPI).

Programming Note

Software can use a PMC to “pace” the collection of Performance Monitor data. For example, if it is desired to collect event counts every n cycles, software can specify that a particular PMC count cycles, and set that PMC to 0x8000_0000 - n. The events of interest would be counted in other PMCs. The counter negative condition that will occur after n cycles can, with the appropriate setting of MMCR bits, cause counter values to become frozen, cause a Performance Monitor exception to occur, etc.

9.4.2.1 Event Counting and Sampling

The PMCs are enabled to count unless they are “frozen” by one or more of the “freeze counters” fields in MMCR0 or MMCR2.

Each of PMC’s 1-4 can be configured, using MMCR1, to count “continuous” events (events that can occur at any time), or to count “randomly sampled” events (or “sampled” events) that are associated with the execution of randomly sampled instructions.

Continuous events always cause the counters to count (unless counters are frozen). These events are specified for each counter by using encodes F0-FF in the PMCn Selector fields in MMCR1.

Randomly sampled events can cause the counters to count only when random sampling has been enabled by setting $MMCR0_{SE}$=1. The types of instructions that are sampled are specified in $MMCRA_{SM}$ and $MMCRA_{ES}$. Randomly sampled events are specified for each counter by using encodes E0-EF in the PMCn Selector fields in MMCR1.

Programming Note

A typical sequence of operations that enables use the PMCs is as follows.

- Freeze the counters by setting $MMCR0_{FC}$=1.
- Set control fields in MMCR0 and MMCR2 that control counting in various privilege states and other modes, and that enable counter negative conditions.
- Initialize the events to be counted by PMCs 1-4 using the PMCn Selector fields in MMCR1.
- Specify the BHRB filtering mode, threshold event Counter events, and whether or not random sampling is enabled in the corresponding fields in MMCRA.
- Initialize the PMCs to the values desired. For example, in order to configure a counter to cause a counter negative condition after n counts, that counter would be initialized to 2^{32}-n.
- Set $MMCR0_{FC}$ to 0 to disable freezing the counters, and set $MMCR0_{PMAE}$ to 1 if a Performance Monitor alert (and the corresponding Performance Monitor interrupt) is desired when an enabled condition or event occurs. (See Section 9.2 for the definition of enabled condition or event.)

When the Performance Monitor alert occurs, the program would typically read the values of the counters as well as the contents of SIAR, SDAR, SIER as needed in order to extract the information that was being monitored.

See Sections 9.4.4 - 9.4.10 for information regarding MMCRs, SIAR, SDAR, and SIER, and some additional usage examples.

9.4.3 Threshold Event Counter

The threshold event counter and associated controls are in MMCRA (see Section 9.4.7). When Performance Monitor alerts are enabled ($MMCR0_{PMAE}$=1), this counter begins incrementing from value 0 upon each occurrence of the event specified in the Threshold Event Counter Event (TECE) field after the event specified by the Threshold Start Event (TS) field occurs. The counter stops incrementing when the event specified in the Threshold End Event (TE) field occurs. The counter subsequently freezes until the event specified in the TS field is again recognized, at which point it restarts incrementing from value 0 as explained above. If the counter reaches its maximum value or a Performance Monitor alert occurs, incrementing stops. After the Performance Monitor alert occurs, the contents of the threshold event counter are not altered by the hardware until software sets $MMCR0_{PMAE}$ to 1.

Programming Note

Because hardware can modify the contents of the threshold event counter when random sampling is enabled ($MMCRA_{SE}$=1) and $MMCR0_{PMAE}$=1 at any time, any value written to the threshold event counter under this condition may be immediately overwritten by hardware.

The threshold event counter value is represented as a 3-bit integral power of 4, multiplied by a 7-bit integer. The exponent is contained in $MMCRA_{TECX}$, and the multiplier is contained in $MMCRA_{TECM}$. For a given counter exponent, e, and multiplier, m, the number represented is as follows:

$$N = 4^e \times m$$

This counter format allows the counter to represent a range of 0 through approximately 2 million counts with many fewer bits than would be required by a binary counter.

To represent a given counter value, hardware uses as e the smallest 3-bit integer for which a 7-bit integer exists such that the given counter value can be expressed using this format.

Programming Note

Software can obtain the number N from the contents of the threshold event counter by shifting the multiplier left twice times the value contained in the exponent.

The value in the counter is the exact number of events that occur for values from 0 through the maximum multiplier value (127), within 4 events of the exact value for values from 128 - 508 (or 127×4), within 16 events of the exact value for values from 512 - 2032 (or 127×4^2), and so on. This represents an event count accuracy of approximately 3%, which is expected to be sufficient for most situations in which a count of events between a start and end event is required.

Programming Note

When using the threshold event counter, software typically specifies a “threshold counter exceeded n” event in MMCR1. This enables a PMC to count the number of times the counter exceeded a specified threshold value during the time Performance Monitor alerts were enabled.

9.4.4 Monitor Mode Control Register 0

Monitor Mode Control Register 0 (MMCR0) is a 64-bit register as shown below.

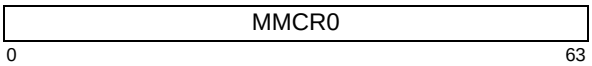


Figure 78. Monitor Mode Control Register 0

MMCR0 is used to control multiple functions of the Performance Monitor. Some fields of MMCR0 are altered by the hardware when various events occur.

The following notation is used in the definitions below. "PMCs" refers to PMCs 1 - n and "PMCj" refers to PMCj, where $2 \leq j \leq n$. n=4 when $MMCR0_{PMCC}$=0b11 and n=6 otherwise.

When $MMCR0_{PMCC}$ is set to 0b10 or 0b11, providing problem state programs read/write access to MMCR0, only FC, PMAE, PMAO can be accessed. All other bits are not changed when ***mtspr*** is executed in problem state, and all other bits return 0s when ***mfspr*** is executed in problem state.

Programming Note

When PMCC=0b10 or 0b11, problem state programs have write access to MMCR0 in order to enable event-based branch routines to reset the FC bit after it has been set to 1 as a result of an enabled condition or event (FCECE=1). During event processing, the event-based branch handler would write the desired initial values to the PMCs and reset the FC bit to 0. PMAO and PMAE can also be set to their appropriate values during the same write operation before returning.

The bit definitions of MMCR0 are as follows.

Bit(s) Description

0:31 Reserved

32 ***Freeze Counters*** (FC)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented.

The hardware sets this bit to 1 when an enabled condition or event occurs and $MMCR0_{FCECE}$=1.

33 ***Freeze Counters and BHRB in Privileged State*** (FCS)

0 The PMCs are incremented (if permitted by other MMCR bits), and entries are written into the BHRB (if permitted by the BHRB Instruction Filtering Mode field in MMCRA).
1 The PMCs are not incremented, and entries are not written into the BHRB, if $MSR_{HV\ PR}$=0b00.

34 ***Conditionally Freeze Counters and BHRB in Problem State*** (FCP)

If the value of bit 51 (FCPC) is 0, this field has the following meaning.

0 The PMCs are incremented (if permitted by other MMCR bits) and entries are written into the BHRB (if permitted by the BHRB Instruction Filtering Mode field in MMCRA).
1 The PMCs are not incremented, and entries are not written into the BHRB, if MSR_{PR}=1.

If the value of bit 51 (FCPC) is 1, this field has the following meaning.

0 The PMCs are not incremented, and entries are not written into the BHRB, if $MSR_{HV\ PR}$=0b01.
1 The PMCs are not incremented, and entries are not written into the BHRB, if $MSR_{HV\ PR}$=0b11.

Programming Note

In order to freeze counters in problem state regardless of MSR_{HV}, $MMCR0_{FCPC}$ must be set to 0 and $MMCR0_{FCP}$ must be set to 1.

35 ***Freeze Counters while Mark = 1*** (FCM1)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if MSR_{PMM}=1.

36 ***Freeze Counters while Mark = 0*** (FCM0)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if MSR_{PMM}=0.

37 ***Performance Monitor Alert Enable*** (PMAE)

0 Performance Monitor alerts are disabled and BHRB entries are not written.
1 Performance Monitor alerts are enabled, and BHRB entries are written (if enabled by other bits) until a Performance Monitor alert occurs, at which time:
- $MMCR0_{PMAE}$ is set to 0
- $MMCR0_{PMAO}$ is set to 1

Programming Note

Software can set this bit and $MMCR0_{PMAO}$ to 0 to prevent Performance Monitor exceptions.

Software can set this bit to 1 and then poll the bit to determine whether an enabled condition or event has occurred. This is especially useful for software that runs with MSR_{EE}=0.

In earlier versions of the architecture that lacked the concept of Performance Monitor alerts, this bit was called Performance Monitor Exception Enable (PMXE).

38 ***Freeze Counters on Enabled Condition or Event*** (FCECE)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are incremented (if permitted by other MMCR bits) until an enabled condition or event occurs when $MMCR0_{TRIGGER}$=0, at which time:
- $MMCR0_{FC}$ is set to 1

If the enabled condition or event occurs when $MMCR0_{TRIGGER}$=1, the FCECE bit is treated as if it were 0.

39:40 ***Time Base Selector*** (TBSEL)

This field selects the Time Base bit that can cause a Time Base transition event (the event occurs when the selected bit changes from 0 to 1).

00 Time Base bit 47 is selected.
01 Time Base bit 51 is selected.
10 Time Base bit 55 is selected.
11 Time Base bit 63 is selected.

Programming Note

Time Base transition events can be used to collect information about activity, as revealed by event counts in PMCs and by addresses in SIAR and SDAR, at periodic intervals.

In multi-threaded systems in which the Time Base registers are synchronized among the threads, Time Base transition events can be used to correlate the Performance Monitor data obtained by the several threads. For this use, software must specify the same TBSEL value for all the threads in the system.

Because the frequency of the Time Base is implementation-dependent, software should invoke a system service program to obtain the frequency before choosing a value for TBSEL.

41 ***Time Base Event Enable*** (TBEE)

0 Time Base transition events are disabled.
1 Time Base transition events are enabled.

Programming Note

When PMC3 is configured to count the occurrence of Time Base transition events, the events are counted regardless of the value of $MMCR0_{TBEE}$. (See Section 9.4.5.) The occurrence of a Time Base transition causes a Performance Monitor alert only if $MMCR0_{TBEE}$=1.

42 **BHRB Available (BHRBA)**
This field controls whether the BHRB instructions are available in problem state. If an attempt is made to execute a BHRB instruction in problem state when the BHRB instructions are not available, a Facility Unavailable interrupt will occur.

0 ***clrbhrb*** and ***mfbhrbe*** are not available in problem state.
1 ***clrbhrb*** and ***mfbhrbe*** are available in problem state unless they have been made unavailable by some other register.

43 **Performance Monitor Event-Based Branch Enable (EBE)**
This field controls whether Performance Monitor event-based branches and Performance Monitor event-based exceptions are enabled.

When Performance Monitor event-based branches and exceptions are disabled, no Performance Monitor event-based branches or exceptions occur regardless of the state of $BESCR_{PME}$.

0 Performance Monitor event-based branches and exceptions are disabled.
1 Performance Monitor event-based branches and exceptions are enabled.

Programming Note

In order to enable a problem state applications to use the event-based Branch facility for Performance Monitor events, privileged software initializes MMCR1 to specify the events to be counted, and sets MMCR2, and MMCRA to specify additional sampling controls. MMCR0 should be initialized with PMCC set to 0b10 or ob11 (to give problem state access to various Performance Monitor registers), PMAE and PMAO set to 0s (disabling Performance Monitor alerts), and EBE set to 1 (enabling Performance Monitor event-based branches and exceptions to occur). If the Event-Based Branch facility has not been enabled in the FSCR and HFSCR, it must be enabled in these registers as well.

The above operations by the operating system enable the application to control Performance Monitor event-based branching by means of $BESCR_{PME}$ (to enable or disable Performance Monitor event-based branching) and $MMCR0_{PMAE}$ (to enable or disable Performance Monitor alerts).

44:45 **PMC Control (PMCC)**

This field controls whether or not PMCs 5 - 6 are included in the Performance Monitor, and the accessibility of groups A and B (see Section 9.4.1) of non-privileged SPRs in problem state as described below.

Programming Note

The PMCC field does not affect the behavior of the privileged Performance Monitor registers (SPRs 784-792, 795-798); accesses to these SPRs in problem state result in Privileged Instruction type Program interrupts.

The PMCC field also does not affect the behavior of write operations to group B; write operations to SPRs in group B are treated as not supported regardless of privilege state. See the ***mtspr*** instruction description in Section 4.4.4 for additional information on accessing SPRs that are not supported.

Programming Note

When the PCR makes SPRs unavailable in problem state, they are treated as undefined, and they are not included in groups A or B regardless of the value of PMCC. Thus when the PCR indicates a version of the architecture prior to V. 2.07 (i.e., $PCR_{v2.06}$=1), the PMCC field does not affect SPRs MMCR2 or SIER, which are newly-defined in V. 2.07; these SPRs are treated as undefined registers. Accesses to them in problem state result in Hypervisor Emulation Assistance interrupts regardless of the value of PMCC, and Facility Unavailable interrupts do not occur for them. See Section 2.5 for additional information.

00 PMCs 5 - 6 are included in the Performance Monitor.
Groups A and B are read-only in problem state. If an attempt is made to write to an SPR in group A in problem state, a Hypervisor Emulation Assistance interrupt will occur.

01 PMCs 5 - 6 are included in the Performance Monitor.
Group A is not allowed to be read or written in problem state, and group B is not allowed to be read in problem state. If an attempt is made, in problem state, to read or write to an SPR in group A, or to read from an SPR in group B, a Facility Unavailable interrupt will occur.

10 PMCs 5 - 6 are included in the Performance Monitor.
Group A is allowed to be read and written in problem state, and group B except for MMCR1 (SPR 782) is allowed to be read in problem state. If an attempt is made to read MMCR1 in problem state, a Facility Unavailable interrupt will occur.

11 PMCs 5 - 6 are not included in the Performance Monitor. See Section 9.4.2 for details.
Group A except for PMCs 5-6 (SPRs 775,776) is allowed to be read and written in problem state, and group B except for MMCR1 (SPR 782) is allowed to be read in problem state.
If an attempt is made, in problem state, to read or write to PMCs 5-6 (SPRs 775,776), or to read from MMCR1, a Facility Unavailable interrupt will occur.

When an SPR is made available by the PMCC field, it is available only if it has not been made unavailable by the HFSCR (see Section 6.2.12).

Programming Note

In order to give problem state programs the same level of access to the Performance Monitor registers as was specified in Power ISA V 2.06, PMCC must be set to 0b00 (restricting access to read-only) and the PCR should indicate Version 2.06 (restricting access to the set of Performance Monitor SPRs and SPR bits that were defined in V 2.06).

When PMCC=0b00 and a write operation to a Performance Monitor register in group A or B is attempted in problem state, a Hypervisor Emulation Assistance interrupt occurs in order to maintain compatibility with V 2.06. For other values of PMCC, write or read operations to group A and read operations from group B that are not allowed result in Facility Unavailable interrupts. Facility Unavailable interrupts provide the operating system with more information about the type of disallowed access that was attempted than the Hypervisor Emulation Assistance interrupt provides. See Section 6.2.11 for additional information.

Programming Note

In order to prevent applications from accessing Performance Monitor registers, PMCC is set to 0b01.

In order to allow applications limited control over the Performance Monitor, PMCC is set to 0b10 or 0b11. These values are also used when Performance Monitor event-based branches are enabled.

46 ***Freeze Counters in Transactional State*** (FCTS)

0 PMCs are incremented (if permitted by other MMCR bits).
1 PMCs are not incremented when the thread is in Transactional state.

47 ***Freeze Counters in Non-Transactional State*** (FCNTS)

0 PMCs are incremented (if permitted by other MMCR bits).
1 PMCs are not incremented when the thread is in Non-transactional state.

48 ***PMC1 Condition Enable*** (PMC1CE)

This bit controls whether counter negative conditions due to a negative value in PMC1 are enabled.

0 Counter negative conditions for PMC1 are disabled.
1 Counter negative conditions for PMC1 are enabled.

49 ***PMCj Condition Enable*** (PMCjCE)

This bit controls whether counter negative conditions due to a negative value in any PMCj (i.e., in any PMC except PMC1) are enabled.

0 Counter negative conditions for all PMCjs are disabled.
1 Counter negative conditions for all PMCjs are enabled.

50 ***Trigger*** (TRIGGER)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 PMC1 is incremented (if permitted by other MMCR bits). The PMCjs are not incremented until PMC1 is negative or an enabled condition or event occurs, at which time:
- the PMCjs resume incrementing (if permitted by other MMCR bits)
- $MMCR0_{TRIGGER}$ is set to 0

See the description of the FCECE bit, above, regarding the interaction between TRIGGER and FCECE.

Programming Note

Uses of TRIGGER include the following.

- Resume counting in the PMCjs when PMC1 becomes negative, without causing a Performance Monitor interrupt. Then freeze all PMCs (and optionally cause a Performance Monitor interrupt) when a PMCj becomes negative. The PMCjs then reflect the events that occurred between the time PMC1 became negative and the time a PMCj becomes negative. This use requires the following MMCR0 bit settings.
  - TRIGGER=1
  - PMC1CE=0
  - PMCjCE=1
  - TBEE=0
  - FCECE=1
  - PMAE=1 (if a Performance Monitor interrupt is desired)
- Resume counting in the PMCjs when PMC1 becomes negative, and cause a Performance Monitor interrupt without freezing any PMCs. The PMCjs then reflect the events that occurred between the time PMC1 became negative and the time the interrupt handler reads them. This use requires the following MMCR0 bit settings.
  - TRIGGER=1
  - PMC1CE=1
  - TBEE=0
  - FCECE=0
  - PMAE=1

51 ***Freeze Counters and BHRB in Problem State Condition*** (FCPC)

This bit controls the meaning of bit 34 (FCP). See the definition of bit 34 for details.

Programming Note

In order to enable the FCP bit to freeze counters in problem state regardless of MSR_{HV}, $MMCR0_{FCPC}$ must be set to 0.

52 ***Performance Monitor Alert Qualifier*** (PMAQ)

This bit provides additional implementation-dependent information about the cause of the Performance Monitor alert. When a Performance Monitor alert occurs, this bit is set to 0 if no additional information is available.

53:54 Reserved

55 ***Control Counters 5 - 6 with Run Latch*** (CC5-6RUN)

When $MMCR0_{PMCC}$ = b11, the setting of this bit has no effect; otherwise it is defined as follows.

0 PMCs 5 and 6 are incremented if $CTRL_{RUN}$=1 (if permitted by other MMCR bits).
1 PMCs 5 and 6 are incremented regardless of the value of $CTRL_{RUN}$ (if permitted by other MMCR bits).

56 ***Performance Monitor Alert Occurred*** (PMAO)

0 A Performance Monitor alert has not occurred since the last time software set this bit to 0.
1 A Performance Monitor alert has occurred since the last time software set this bit to 0.

This bit is set to 1 by the hardware when a Performance Monitor alert occurs. This bit can be set to 0 only by the ***mtspr*** instruction.

Programming Note

Software can set this bit to 1 and set PMAE to 0 to simulate the occurrence of a Performance Monitor alert.

Software should set this bit to 0 after handling the Performance Monitor alert.

57 ***Freeze Counters in Suspended State*** (FCSS)

0 PMCs are incremented (if permitted by other MMCR bits).
1 PMCs are not incremented when the thread is in Suspended state.

58 ***Freeze Counters 1-4*** (FC1-4)

0 PMC1 - PMC4 are incremented (if permitted by other MMCR bits).
1 PMC1 - PMC4 are not incremented.

59 ***Freeze Counters 5-6*** (FC5-6)

0 PMC5 - PMC6 are incremented (if permitted by other MMCR bits).
1 PMC5 - PMC6 are not incremented.

60:61 Reserved

62 ***Freeze Counters 1-4 in Wait State*** (FC1-4WAIT)

0 PMCs 1-4 are incremented (if permitted by other MMCR bits).
1 PMCs 1-4, except for PMCs counting events that are not controlled by this bit, are not incremented if $CTRL_{RUN}$=0.

Programming Note

When PMC 1 is counting cycles, it is not controlled by this bit. See the description of the F0 event in Section 9.4.5.

63 ***Freeze Counters and BHRB in Hypervisor State*** (FCH)

0 The PMCs are incremented (if permitted by other MMCR bits) and BHRB entries are written (if permitted by the BHRB Instruction Filtering Mode field in MMCRA).
1 The PMCs are not incremented and BHRB entries are not written if $MSR_{HV\ PR}$=0b10.

9.4.5 Monitor Mode Control Register 1

Monitor Mode Control Register 1 (MMCR1) is a 64-bit register as shown below.

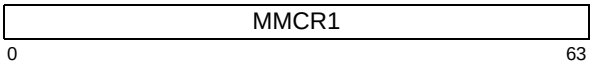


Figure 79. Monitor Mode Control Register 1

MMCR1 enables software to specify the events that are counted by the PMCs.

In the following descriptions, events due to randomly sampled instructions occur only if random sampling is enabled ($MMCRA_{SE}$=1); all other events occur whenever the event specification is met regardless of the value of $MMCRA_{SE}$.

Various events defined below refer to “threshold A” through “threshold H”. The table below specifies the number of threshold event counter events corresponding to each of these thresholds.

Threshold	Events
A	4096
B	32
C	64
D	128
E	256
F	512
G	1024
H	2048

Table 5: Event Counts for thesholds A-H

The bit definitions of MMCR1 are as follows. Implementation-dependent MMCR1 bits that are not supported are treated as reserved.

Bit(s) Description

0:31 Problem state access (SPR 782)
Reserved

Privileged access (SPR 782 or 798)
Implementation-dependent

32:39 ***PMC1 Selector*** (PMC1SEL)
The value of PMC1SEL specifies the event to be counted by PMC1 as defined below.
All values in the range of E0 - FF that are not specified below are reserved.

Hex Event

00 Disable events. (No events occur.)
01-BF Implementation-dependent
C0-DF Reserved

The following events can occur only when random sampling is enabled ($MMCRA_{SE}$=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in $MMCRA_{SM}$.)

E0 The thread has dispatched a randomly sampled instruction. (RIS)
E2 The thread has completed a randomly sampled *Branch* instruction for which the branch was taken. (RIS, RBS)
E4 The thread has failed to locate a randomly sampled instruction in the primary instruction cache. (RIS)
E6 The threshold event counter has exceeded the number of events corresponding to threshold A (see Table 5). (RIS, RLS, RBS)
E8 The threshold event counter has exceeded the number of events corresponding to threshold E (see Table 5). (RIS, RLS, RBS)
EA The thread filled a block in a data cache with data that were accessed by a randomly sampled *Load* instruction. (RIS, RLS)
EC The threshold event counter has reached its maximum value. (RIS, RLS, RBS)

The following events can occur regardless of whether random sampling is enabled.

F0 A cycle has occurred. This event is not controlled by $MMCR0_{FC1\text{-}4WAIT}$.
F2 A cycle has occurred in which the thread completed one or more instructions.
F4 The thread has completed a *Floating-Point, Vector Floating-Point,* or *VSX Floating-Point* instruction other than a

Load or *Store* instruction to the point at which it has reported all exceptions it will cause.

F6 The thread has failed to locate an ERAT entry during instruction address translation.

F8 A cycle has occurred during which all previously initiated instructions have completed and no instructions are available for initiation.

FA A cycle has occurred during which the RUN bit of the CTRL register for one or more threads of the multi-threaded processor was set to 1.

FC A load type instruction finished. If the instruction caused more than one reference, only one will be counted.

FE The thread has completed an instruction.

40:47 ***PMC2 Selector*** (PMC2SEL)

The value of PMC2SEL specifies the event to be counted by PMC2 as defined below.

All values in the range of E0 - FF that are not specified below are reserved.

Hex	Event
00	Disable events. (No events occur.)
01-BF	Implementation-dependent
C0-DF	Reserved

The following events can occur only when random sampling is enabled ($MMCRA_{SE}$=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in $MMCRA_{SM}$.)

E0 The thread has obtained the data for a randomly sampled *Load* instruction from storage that did not reside in any cache. (RIS, RLS)

E2 The thread has failed to locate the data for a randomly sampled *Load* instruction in the primary data cache. (RIS, RLS)

E4 The thread filled a block in the primary data cache with data that were accessed by a randomly sampled *Load* instruction and obtained from a location other than the secondary or tertiary cache. (RIS, RLS)

E6 The threshold event counter has exceeded the number of events corresponding to threshold B (see Table 5). (RIS, RLS, RBS)

E6 The threshold event counter has exceeded the number of events corresponding to threshold F (see Table 5). (RIS, RLS, RBS)

The following events can occur regardless of whether random sampling is enabled.

F0 The thread has completed a *Store* instruction to the point at which it has reported all the exceptions it will cause.

F2 The thread has dispatched an instruction.

F4 A cycle has occurred during which the RUN bit of the thread's CTRL register contained 1.

F6 The thread has failed to locate an ERAT entry during data address translation, and a new ERAT entry corresponding to the data effective address has been written.

F8 An external interrupt for the thread has occurred.

FA The thread has completed a *Branch* instruction for which the branch was taken.

FC The thread has failed to locate an instruction in the primary cache.

FE The thread has filled a block in the primary data cache with data that were accessed by a *Load* instruction and obtained from a location other than the secondary cache.

48:55 ***PMC3Selector*** (PMC3SEL)

The value of PMC3SEL specifies the event to be counted by PMC3 as defined below.

All values in the range of E0 - FF that are not specified below are reserved.

Hex	Event
00	Disable events. (No events occur.)
01-BF	Implementation-dependent
C0-DF	Reserved

The following events can occur only when random sampling is enabled ($MMCRA_{SE}$=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in $MMCRA_{SM}$.)

E2 The thread has completed a randomly sampled *Store* instruction to the point at which it has reported all exceptions it will cause. (RIS,RLS)

E4 The thread has mispredicted either whether or not the branch would be taken, or if taken, the target address of a randomly sampled *Branch* instruction. (RIS, RBS)

E6 The thread has failed to locate an ERAT entry during data address translation for a randomly sampled instruction. (RIS,RLS)

E8 The threshold event counter has exceeded the number of events corresponding to threshold C (see Table 5). (RIS, RLS, RBS)

EA The threshold event counter has exceeded the number of events corresponding to threshold G (see Table 5). (RIS, RLS, RBS)

The following events can occur regardless of whether random sampling is enabled.

F0 The thread has attempted to store data in the primary data cache but no block corresponding to the real address existed.
F2 The thread has dispatched an instruction.
F4 The thread has completed an instruction when the RUN bit of the CTRL register for all threads on the multi-threaded processor contained 1.
F6 The thread has filled a block in the primary data cache with data that were accessed by a *Load* instruction.
F8 A Time Base transition event has occurred for the thread. This event is counted regardless of whether or not Time Base transition events are enabled by $MMCR0_{TBEE}$.
FA The thread has loaded an instruction from a higher level cache than the tertiary cache.
FC The thread was unable to translate a data virtual address using the TLB.
FE The thread has filled a block in the primary data cache with data that were accessed by a *Load* instruction and obtained from a location other than the secondary or tertiary cache.

56:63 ***PMC4 Selector*** (PMC4SEL)
The value of PMC4SEL specifies the event to be counted by PMC4 as defined below.
All values in the range of E0 - FF that are not specified below are reserved.

Hex	Event
00	Disable events. (No events occur.)
01-BF	Implementation-dependent
C0-DF	Reserved

The following events can occur only when random sampling is enabled ($MMCRA_{SE}$=1). The sampling modes corresponding to each event are listed in parentheses. (The sampling mode is specified in $MMCRA_{SM}$.)

E0 The thread has completed a randomly sampled instruction. (RIS, RLS, RBS)
E4 The thread was unable to translate a data virtual address using the TLB for a randomly sampled instruction. (RIS,RLS)
E6 The thread has loaded a randomly sampled instruction from a higher level cache than the tertiary cache. (RIS)
E8 The thread has filled a block in the primary data cache with data that were accessed by a randomly sampled *Load* instruction and obtained from a location other than the secondary cache. (RIS, RLS)
EA The threshold event counter has exceeded the number of events corresponding to threshold D (see Table 5). (RIS, RLS, RBS)
EC The threshold event counter has exceeded the number of events corresponding to threshold H (see Table 5). (RIS, RLS, RBS)

The following events can occur regardless of whether random sampling is enabled.

F0 The thread has attempted to load data from the primary data cache but no block corresponding to the real address existed.
F2 A cycle has occurred during which the thread has dispatched one or more instructions.
F4 A cycle has occurred during which the PURR was incremented when the RUN bit of the thread's CTRL register contained 1.
F6 The thread has mispredicted either whether or not the branch would be taken, or if taken, the target address of a *Branch* instruction.
F8 The thread has discarded prefetched instructions.
FA The thread has completed an instruction when the RUN bit of the thread's CTRL register contained 1.
FC The thread was unable to translate an instruction virtual address using the TLB, and a new TLB entry corresponding to the instruction virtual address has been written.
FE The thread has obtained the data for a *Load* instruction from storage that did not reside in any cache.

Compatibility Note

In versions of the architecture that precede Version 2.02 the PMC Selector Fields were six bits long, and were split between MMCR0 and MMCR1. PMC1-8 were all programmable.

If more programmable PMCs are implemented in the future, additional MMCRs may be defined to cover the additional selectors.

9.4.6 Monitor Mode Control Register 2

Monitor Mode Control Register 2 (MMCR2) is a 64-bit register that contains 9-bit control fields for controlling the operation of PMC1 - PMC6 as shown below.

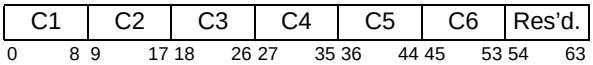


Figure 80. Monitor Mode Control Register 2

When $MMCR0_{PMCC}$ = 0b11, fields C1 - C4 control the operation of PMC1 - PMC4, respectively and fields C5 and C6 are ignored by the hardware; otherwise, fields C1 - C6 control the operation of PMC1 - PMC6, respectively. The bit definitions of each Cn field are as follows, where n = 1,...6.

When $MMCR0_{PMCC}$ is set to 0b10 or 0b11, providing problem state programs read/write access to MMCR2, only the FCnP0 bits can be accessed. All other bits are not changed when ***mtspr*** is executed in problem state, and all other bits return 0s when ***mfspr*** is executed in problem state.

Bit **Description**

0 ***Freeze Counter n in Privileged State*** (FCnS)

- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if $MSR_{HV\ PR}$=0b00.

1 ***Freeze Counter n in Problem State if MSR_{HV}=0*** (FCnP0)

- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if $MSR_{HV\ PR}$=0b01.

Programming Note

Problem state programs need access to this field in order to enable them to individually enable counters when analyzing sections of code. All the other fields will typically be initialized by the operating system.

2 ***Freeze Counter n in Problem State if MSR_{HV}=1*** (FCnP1)

- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if $MSR_{HV\ PR}$=0b11.

3 ***Freeze Counter n while Mark = 1*** (FCnM1)

- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if MSR_{PMM}=1.

4 ***Freeze Counter n while Mark = 0*** (FCnM0)

- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if MSR_{PMM}=0.

5 ***Freeze Counter n in Wait State*** (FCnWAIT)

- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if $CTRL_{RUN}$=0.

Programming Note

The operating system is expected to set $CTRL_{RUN}$ to 0 when the thread is in a “wait state”, i.e., when there is no process ready to run.

6 ***Freeze Counter n in Hypervisor State*** (FCnH)

- 0 PMCn is incremented (if permitted by other MMCR bits).
- 1 PMCn is not incremented if $MSR_{HV\ PR}$=0b10.

Bits 54:63 of MMCR2 are reserved.

9.4.7 Monitor Mode Control Register A

Monitor Mode Control Register A (MMCRA) is a 64-bit register as shown below.

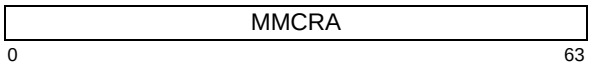


Figure 81. Monitor Mode Control Register A

MMCRA gives privileged programs the ability to control the sampling process, BHRB filtering, and threshold events.

When $MMCR0_{PMCC}$ is set to 0b10 or 0b11, providing problem state programs read/write access to MMCRA, the Threshold Event Counter Exponent (TECX) and Threshold Event Counter Multiplier (TECM) fields are read-only, and all other fields return 0s, when ***mfspr*** is executed in problem state; all fields are not changed when ***mtspr*** is executed in problem state.

Programming Note

Read/write access is provided to MMCRA in problem state (SPR 770) when $MMCR0_{PMCC}$ = 0b10 or 0b11 even though no fields can be modified by ***mtspr*** because future versions of the architecture may allow various fields of MMCRA to be modified in problem state.

The bit definitions of MMCRA are as follows.

Bit(s) **Description**

0:31 Problem state access (SPR 770)
Reserved

Privileged access (SPR 770 or 786)
Implementation-dependent

32:33 ***BHRB Instruction Filtering Mode*** (IFM)

This field controls the filter criterion used by the hardware when recording *Branch* instructions into the BHRB. See Section 9.5.

00 All taken Branch instructions are entered into the BHRB unless prevented by other filtering fields.

01 Do not record any *Branch* instructions in which the LK field is set to 0.

10 Do not record *I-Form* instructions. For *B-Form* and *XL-Form* instructions for which the BO field indicates "Branch always," do not record the instruction if it is *B-Form* and do not record the instruction address but record only the branch target address if it is XL-Form.

11 Filter and enter BHRB entries as for mode 10, but for *B-Form* and *XL-Form* instructions for which BO_0=1 or for which the "a" bit in the BO field is set to 1, do not record the instruction if it is *B-Form* and do not record the instruction address but record only the branch target address if it is XL-Form.

Programming Note

Filtering mode 10 provides additional filtering for unconditional *Branch* instructions, and for indirect *Branch* instructions only the target address is recorded.

Filtering mode 11 provides additional filtering for instructions that provide a hint or for which the outcome does not depend on the value of the Condition Register.

34:36 ***Threshold Event Counter Exponent*** (TECX)

This field species the exponent of the threshold event counter value. See Section 9.4.3 for additional information. The maximum exponent supported is at least 5.

37 Reserved

38:44 ***Threshold Event Counter Multiplier*** (TECM)

This field species the multiplier of the threshold event counter value. See Section 9.4.3 for additional information.

Programming Note

When $MMCR0_{PMCC}$ = 0b10 or 0b11, providing problem-state programs read-write access to MMCRA, problem state programs are able to read only the TECX and TECM fields (and are not able to write any fields). The values of these fields are needed during the processing of an event-based branch that occurs due to a counter negative condition for a PMC that was counting "threshold counter exceeded n" events (e.g. $MMCR1_{PMC1SEL}$ = 0xE8). Reading these fields enables the application to determine the amount by which the threshold was exceeded. Applications are not given access to other fields, and these other fields must initialized by the operating system.

45:47 ***Threshold Event Counter Event*** (TECE)

This field specifies the event, if any, that is counted by the threshold event counter. The values and meanings are follows.

Value	Event
000	Disable counting.
001	A cycle has occurred.
010	An instruction has completed.
011	Reserved

All other values are implementation-dependent.

48:51 ***Threshold Start Event*** (TS)

This field specifies the event that causes the threshold event counter to start counting occurrences of the event specified in the Threshold Event Counter Event (TECE) field. The events only occur if $MMCRA_{SE}$=1 (random sampling enabled) and one of the sampling modes listed in parenthesis is in effect. (The sampling mode that is currently in effect is specified in $MMCRA_{SM}$.)

0000 Reserved.

0001 The thread has randomly sampled an instruction while it is being decoded. (RIS)

0010 The thread has dispatched a randomly sampled instruction. (RIS)

0011 A randomly sampled instruction has been sent to a facility (e.g. *Branch, Fixed Point*, etc.) (RIS, RLS, RBS)

0100 The thread has completed a randomly sampled instruction to the point at which it has reported all exceptions it will cause. (RIS, RLS, RBS)

0101 The thread has completed a randomly sampled instruction. (RIS, RLS, RBS)

0110 The thread has failed to locate data for a randomly sampled *Load* instruction in the primary data cache. (RIS, RLS)
0111 The thread has filled a block in the primary data cache with data that were accessed by a randomly sampled *Load* instruction. (RIS, RLS)

The definition of the following values depends on whether the access to MMCRA is in problem state or in privileged state.

Problem state access (SPR 770)
1000 - 1111 - Reserved

Privileged access (SPR 770 or 786)
1000 - 1111 - Implementation-dependent

52:55 ***Threshold End Event*** (TE)

This field specifies the event that causes the threshold event counter to stop counting occurrences of the event specified in the Threshold Event Counter Event (TECE) field. The events only occur if $MMCRA_{SE}$=1 (random sampling enabled) and one of the sampling modes listed in parenthesis is in effect. (The sampling mode that is currently in effect is specified in $MMCRA_{SM}$.)
0000 Reserved
0001 The thread has randomly sampled an instruction while it is being decoded. (RIS)
0010 The thread has dispatched a randomly sampled instruction. (RIS)
0011 A randomly sampled instruction has been sent to a facility (e.g. *Branch, Fixed Point*, etc.) (RIS, RLS, RBS)
0100 The thread has completed a randomly sampled instruction to the point at which it has reported all exceptions that it will cause. (RIS, RLS, RBS)
0101 The thread has completed a randomly sampled instruction. (RIS, RLS, RBS)
0110 The thread has failed to locate data for a randomly sampled *Load* instruction in the primary data cache. (RIS, RLS)
0111 The thread has filled a block in the primary data cache with data that were accessed by a randomly sampled *Load* instruction. (RIS, RLS)

The definition of the following values depends on whether the access to MMCRA is in problem state or in privileged state.

Problem state access (SPR 770)
1000 - 1111 - Reserved

Privileged access (SPR 770 or 786)
1000 - 1111 - Implementation-dependent

56 Reserved

57:59 ***Eligibility for Random Sampling*** (ES)
When random sampling is enabled ($MMCRA_{SE}$=1) and the SM field indicates random instruction sampling (RIS), the encodings of this field specify the instructions that are eligible to be sampled as follows.

000 All instructions
001 All *Load* and *Store* instructions
010 All probe no-op instructions
011 Reserved

The definition of the following values depends on whether the access to MMCRA is in problem state or in privileged state.

Problem state access (SPR 770)
100 - 111 - Reserved

Privileged access (SPR 770 or 786)
100 - 111 - Implementation-dependent

When random sampling is enabled ($MMCRA_{SE}$=1) and the SM field indicates random Load/Store Facility sampling (RLS), the encodings of this field specify the instructions that are eligible to be sampled as follows.
000 Instructions for which the thread has attempted to load data from the data cache but no block corresponding to the real address existed.
001 Reserved
010 Reserved
011 Reserved

The definition of the following values depends on whether the access to MMCRA is in problem state or in privileged state.

Problem state access (SPR 770)
100 - 111 - Reserved

Privileged access (SPR 770 or 786)
100 - 111 - Implementation-dependent

When random sampling is enabled ($MMCRA_{SE}$=1) and the SM field indicates random Branch Facility sampling (RBS), the encodings of this field specify the instructions that are eligible to be sampled as follows.
000 Instructions for which the thread has either mispredicted whether or not the branch would be taken, or if taken, the target address of a *Branch* instruction.
001 Instructions for which the thread has mispredicted whether or not the branch of a *Branch* instruction would be taken because the contents of the Condition Register differed from the predicted contents.
010 Instructions for which the thread has mispredicted the target address of a *Branch* instruction.

011 All *Branch* instructions for which the branch was taken.

The definition of the following values depends on whether the access to MMCRA is in problem state or in privileged state.

Problem state access (SPR 770)
100 - 111 - Reserved

Privileged access (SPR 770 or 786)
100 - 111 - Implementation-dependent

60 Reserved

61:62 ***Random Sampling Mode*** (SM)

00 **Random Instruction Sampling** (RIS) - Instructions that meet the criterion specified in the ES field for random instruction sampling are eligible to be sampled.
01 **Random Load/Store Facility Sampling** (RLS) - Instructions that meet the criterion specified in the ES field for random Load/Store Facility sampling are eligible for sampling.
10 **Random Branch Facility Sampling** (RBS) - Instructions that meet the criterion specified in the ES field for random Branch Facility sampling are eligible for sampling.
11 Reserved

63 ***Random Sampling Enable*** (SE)

0 Random sampling is disabled.
1 Random sampling is enabled.

See Section 9.4.2.1 for information about random sampling.

9.4.8 Sampled Instruction Address Register

The Sampled Instruction Address Register (SIAR) is a 64-bit register.

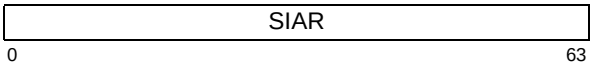


Figure 82. Sampled Instruction Address Register

When a Performance Monitor alert occurs because of an event caused by execution of a randomly sampled instruction, the SIAR contains the effective address of the instruction if $SIER_{SIARV}$ = 1 and contains an undefined value if $SIER_{SIARV}$ = 0.

When a Performance Monitor alert occurs because of an event other than an event caused by execution of a randomly sampled instruction, the SIAR contains the effective address of an instruction that was being executed, possibly out-of-order, at or around the time that the Performance Monitor alert occurred.

The instruction located at the effective address contained in the SIAR is called the “sampled instruction”.

The contents of SIAR may be altered by the hardware if and only if $MMCR0_{PMAE}$=1. Thus after the Performance Monitor alert occurs, the contents of SIAR are not altered by the hardware until software sets $MMCR0_{PMAE}$ to 1. After software sets $MMCR0_{PMAE}$ to 1, the contents of SIAR are undefined until the next Performance Monitor alert occurs.

Programming Note

When the Performance Monitor alert occurs, $SIER_{AMPPR\ SAMPHV}$ indicates the value of $MSR_{HV\ PR}$ that was in effect when the sampled instruction was being executed. (The contents of these SIER bits are visible only in privileged state.)

9.4.9 Sampled Data Address Register

The Sampled Data Address Register (SDAR) is a 64-bit register.

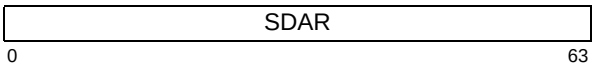


Figure 83. Sampled Data Address Register

When a Performance Monitor alert occurs because of an event caused by execution of a randomly sampled instruction, the SDAR contains the effective address of the storage operand of the instruction if $SIER_{SDARV}$ = 1 and contains an undefined value if $SIER_{SDARV}$ = 0.

When a Performance Monitor alert occurs because of an event other than an event caused by execution of a randomly sampled instruction, the SDAR contains the effective address of the storage operand of an instruction that was being executed, possibly out-of-order, at or around the time that the Performance Monitor alert occurred. This storage operand may or may not be the storage operand (if any) of the sampled instruction.

The data located at the effective address contained in the SDAR are called the “sampled data.”

The contents of SDAR may be altered by the hardware if and only if $MMCR0_{PMAE}$=1. Thus after the Performance Monitor alert occurs, the contents of SDAR are not altered by the hardware until software sets $MMCR0_{PMAE}$ to 1. After software sets $MMCR0_{PMAE}$ to 1, the contents of SDAR are undefined until the next Performance Monitor alert occurs.

9.4.10 Sampled Instruction Event Register

The Sampled Instruction Event Register (SIER) is a 64-bit register.

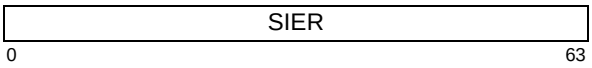


Figure 84. Sampled Instruction Event Register

When random sampling is enabled and a Performance Monitor alert occurs because of an event caused by execution of a randomly sampled instruction, the SIER contains information about the sampled instruction. The contents of all fields are valid unless otherwise indicated.

Programming Note

A Performance Monitor alert occurs because of an event caused by execution of a randomly sampled instruction if random sampling Is enabled and a counter negative condition exists in a PMC that was counting events based on randomly sampled instructions.

When random sampling is disabled or when a Performance Monitor alert occurs because of an event that was not caused by execution of a randomly sampled instruction, the contents of the SIER are undefined.

The contents of SIER may be altered by the hardware if and only if $MMCR0_{PMAE}$=1. Thus after the Performance Monitor alert occurs, the contents of SIER are not altered by the hardware until software sets $MMCR0_{PMAE}$ to 1. After software sets $MMCR0_{PMAE}$ to 1, the contents of SIER are undefined until the next Performance Monitor alert occurs.

The bit definitions of the SIER are as follows.

0:37 The definition of these bits depends on whether the access to SIER is in problem state or in privileged state.

Problem state access (SPR 768)
Reserved

Privileged access (SPR 768 or 784)
Implementation-dependent

38:40 The definition of these bits depends on whether the access to SIER is in problem state or in privileged state.

Problem state access (SPR 768)
Reserved

Privileged access (SPR 768 or 784)

38 ***Sampled MSR_{PR}*** (SAMPPR)
Value of MSR_{PR} when the Performance Monitor alert occurred.

39 ***Sampled MSR_{HV}*** (SAMPHV)
Value of MSR_{HV} when the Performance Monitor alert occurred.

40 Reserved

41 ***SIAR Valid*** (SIARV)

Set to 1 when the contents of the SIAR are valid (i.e., they contain the effective address of the sampled instruction); otherwise set to 0.

42 ***SDAR Valid*** (SDARV)

Set to 1 when the contents of the SDAR are valid (i.e., they contain the effective address of the sampled instruction); otherwise set to 0.

43 **T*hreshold Exceeded*** (TE)

Set to 1 by the hardware if the contents of the threshold event counter exceeded the maximum value when the Performance Monitor alert occurred; otherwise set to 0 by the hardware.

44 **Slew Down**

Set to 1 by the hardware if the processor clock was lower than nominal when the Performance Monitor alert occurred; otherwise set to 0 by the hardware.

45 **Slew Up**

Set to 1 by the hardware if the processor clock was higher than nominal when the Performance Monitor alert occurred; otherwise set to 0 by the hardware.

46:48 ***Sampled Instruction Type*** (SITYPE)
This field indicates the sampled instruction type. The values and their meanings are as follows.

000 The hardware is unable to indicate the sampled instruction type
001 *Load* Instruction
010 *Store* instruction
011 *Branch* Instruction
100 *Floating-Point* Instruction other than a *Load* or *Store* instruction
101 *Fixed-Point* Instruction other than a *Load* or *Store* instruction
110 *Condition Register* or *System Call* instruction
111 Reserved

49:51 ***Sampled Instruction Cache Information*** (SICACHE)

This field provides cache-related information about the sampled instruction.

000 The hardware is unable to provide any cache-related information for the sampled insttuction.
001 The thread obtained the instruction in the primary instruction cache.

010 The thread obtained the instruction in the secondary cache.
011 The thread obtained the instruction in the tertiary cache.
100 The thread failed to obtain the instruction in the primary, secondary, or tertiary cache
101 Reserved
110 Reserved
111 Reserved

52 ***Sampled Instruction Taken Branch*** (SITAKBR)

Set to 1 if the SITYPE field indicates a *Branch* instruction and the branch was taken; otherwise set to 0.

53 ***Sampled Instruction Mispredicted Branch*** (SIMISPRED)

Set to 1 if the SITYPE field indicates a *Branch* instruction and the thread has mispredicted either whether or not the branch would be taken, or if taken, the target address; otherwise set to 0.

54:55 ***Sampled Branch Instruction Misprediction Information*** (SIMISPREDI)

If SIMISPRED=1, this field indicates how the thread mispredicted the outcome of a *Branch* instruction; otherwise this field is set to 0s.
00 The instruction was not a mispredicted *Branch* instruction.
01 The thread mispredicted whether or not the branch would be taken because the contents of the Condition Register differed from the predicted contents.
10 The thread mispredicted the target address of the instruction.
11 Reserved

56 ***Sampled Instruction Data ERAT Miss (SIDERAT)***

When the SITYPE field indicates a *Load* or *Store* instruction, this field is set to 1 if the thread has failed to locate an ERAT entry during data address translation for the sampled instruction and otherwise is set to 0.

When the SITYPE field does not indicate a *Load* or *Store* instruction, the contents of this field are undefined.

57:59 ***Sampled Instruction Data Address Translation Information*** (SIDAXLATE)

This field contains information about data address translation for the sampled instruction. If multiple data address translations were performed, the information pertains to the last translation. The values and their meanings are as follows.
000 The instruction did not require data address translation.
001 The thread translated the data virtual address using the TLB.
010 A PTEG required for data address translation for the instruction was obtained from the secondary cache.
011 A PTEG required for data address translation for the instruction was obtained from the tertiary cache.
100 A PTEG required for data address translation for the instruction was obtained from storage that did not reside in any cache.
101 A PTEG required for data address translation for the instruction was obtained from a cache on a different multi-threaded processor that resides on the same chip as the thread.
110 A PTEG required for data address translation for the instruction was obtained from a cache on a different chip from the thread.
111 Reserved

60:62 ***Sampled Instruction Data Storage Access Information*** (SIDSAI)

This field contains information about data storage accesses made by the sampled instruction. The values and their meanings are as follows.
000 The instruction did not require data address translation.
001 The instruction was a *Read* for which the thread obtained the referenced data from the primary data cache.
010 The instruction was a *Read* for which the thread obtained the referenced data from the secondary cache.
011 The instruction was a *Read* for which the thread obtained the referenced data-from the tertiary cache.
100 The instruction was a *Read* for which the thread obtained the referenced data-from storage that did not reside in any cache.
101 The instruction was a *Read* for which the thread obtained the referenced data from a cache on a different multi-threaded processor that resides on the same chip as the thread.
110 The instruction was a *Read* for which the thread obtained the referenced data from a cache on a different chip from the thread.
111 The instruction was a *Store* for which the data were placed into a location other than the primary data cache.

63 ***Sampled Instruction Completed*** (SICMPL)

Set to 1 if the sampled instruction has completed; otherwise set to 0.

9.5 Branch History Rolling Buffer

The Branch History Rolling Buffer (BHRB) is described in Chapter 8 of Book II but only at the level required by application programmers. Additional aspects of the BHRB are described here.

In order to enable problem state programs to use the BHRB, $MMCR0_{BHRBA}$ must be set to 1 to enable execution of ***clrbhrb*** and ***mfbhrbe*** instructions in problem state. Additionally, $MMCR0_{PMCC}$ must be set to 0b10 or 0b11 to allow problem state programs to read and write the necessary Performance Monitor registers. (See Section 9.4.4.)

If Performance Monitor event-based branching is desired, $MMCR0_{EBE}$ must also be set to 1 to enable Performance Monitor event-based branches.

Programming Note

Enabling Performance Monitor event-based branching eliminates the need for the problem state program to poll $MMCR0_{PMAO}$ in order to determine when a Performance Monitor alert occurs.

The BHRB is written by the hardware if and only if Performance Monitor alerts are enabled by setting $MMCR0_{PMAE}$ to 1. After $MMCR0_{PMAE}$ has been set to 1 and a Performance Monitor alert occurs, $MMCR0_{PMAE}$ is set to 0 and the BHRB is not altered by hardware until software sets $MMCR0_{PMAE}$ to 1 again.

When $MMCR0_{PMAE}$=1, ***mfbhrbe*** instructions return 0s to the target register.

Programming Note

mfbhrbe instructions return 0s when $MMCR0_{PMAE}$=1 in order to prevent software from reading the BHRB while it is being written by hardware.

BHRB Filtering

When the BHRB is written by hardware, only those *Branch* instructions that meet the filtering criterion specified in $MMCRA_{IFM}$ and for which the branch was taken are included.

9.6 Interaction With Other Facilities

If tracing is active (MSR_{SE}=1 or MSR_{BE}=1), the contents of SIAR and SDAR as used by the Performance Monitor facility are undefined and may change even when $MMCR0_{PMAE}$=0.

Programming Note

A potential combined use of the Trace and Performance Monitor facilities is to trace the control flow of a program and simultaneously count events for that program.

Chapter 10. Processor Control

10.1 Overview

The Processor Control facility provides a mechanism for the hypervisor to send messages to other threads in the system. Privileged non-hypervisor programs are able to send messages to other threads on the same multi-threaded processor; however if the processor is configured into sub-processors, privileged non-hypervisor programs can only send messages to other threads on the same sub-processor.

10.2 Programming Model

Both hypervisor-level and privileged-level messages can be sent. Hypervisor-level messages are sent using the ***msgsnd*** instruction and cause hypervisor-level exceptions when received. Privileged-level messages are sent using the ***msgsndp*** instruction and cause privileged-level exceptions when received. For both instructions, the message type and destination threads are specified in a General Purpose Register.

If a message is received by a thread, the exception corresponding to the message type is generated. When the exception is generated, the corresponding interrupt occurs when no higher priority exception exists and the interrupt is enabled (MSR_{EE}=1 for the Directed Privileged Doorbell interrupt and MSR_{EE}=1 or MSR_{HV}=0 for the Directed Hypervisor Doorbell interrupt).

A Directed Privileged Doorbell exception remains until the corresponding interrupt occurs, or the exception is cleared by execution of a ***mtspr***(DPDES) or ***msgclrp*** instruction.

A Directed Hypervisor Doorbell exception remains until the corresponding interrupt occurs, or the exception is cleared by execution of a ***msgclr*** instruction.

If a doorbell exception is present and the corresponding interrupt is pended because MSR_{EE}=0, additional doorbell exceptions are ignored until the exception is cleared.

10.3 Processor Control Registers

10.3.1 Directed Privileged Doorbell Exception State

The layout of the Directed Privileged Doorbell Exception State (DPDES) register is shown in Figure 85.

Figure 85. Directed Privileged Doorbell Exception State Register

The DPDES register is a 64-bit register. For t < T, where T is the number of threads on the sub-processor (or on the multi-threaded processor if sub-processors are not supported), bit 63-t corresponds to the thread with privileged thread number t.

The value of bit t indicates the presence of a Directed Privileged Doorbell exception on the thread with privileged thread number t. Bit t is cleared when a Directed Privileged Doorbell interrupt occurs on thread t.

When the contents of $DPDES_{63-t}$ change from 0 to 1, a Directed Privileged Doorbell exception will come into existence on privileged thread number t within a reasonable period of time. When the contents of $DPDES_{63}$-t change from 1 to 0, the existing Directed Privileged Doorbell exception, if any, on privileged thread number t, will cease to exist within a reasonable period of time, but not later than the completion of the next context synchronizing instruction or event on privileged thread number t.

The preceding paragraph applies regardless of whether the change in the contents of $DPDES_{63-t}$ is the result a ***msgsndp*** or ***msgclrp*** instruction or of modification of the DPDES register caused by execution of an ***mtspr*** (DPDES) instruction.

Bits 0:63-T of the DPDES are reserved.

Programming Note

The primary use of the DPDES is to provide the means for the hypervisor to save a [sub-]processor's Directed Privileged Doorbell exception state when the set of programs running on the [sub-]processor is swapped out or moved from one [sub-]processor to another. Since there is no such need for a similar function for the hypervisor, there is no similar register for the hypervisor. Privileged programs are able to read the DPDES in order to poll for Directed Privileged Doorbell exceptions when the corresponding interrupt is disabled (MSR_{EE}=1).

10.4 Processor Control Instructions

msgsnd, msgsndp, msgclr, and ***msgclrp*** instructions are provided for sending and clearing messages. ***msgsync*** is provided to enable the thread that is target of a ***msgsnd*** instruction to ensure that stores performed by the message-sending thread before it executed ***msgsnd*** have been performed with respect to the target thread. ***msgsndp*** and ***msgclrp*** are privileged instructions, ***msgsnd***, ***msgclr***, and ***msgsync*** are hypervisor privileged instructions.

Message Send *X-form*

msgsnd RB

31	///	///	RB	206	/
0	6	11	16	21	31

```
msgtype ← GPR(RB)32:36
payload ← GPR(RB)37:63
If(msgtype = 0x05)then
   send_msg(msgtype, payload)
```

msgsnd sends a message to other threads in the system. The message type and destination thread(s) are specified in RB.

RB

<-Message Payload->

/ / /	TYPE	B	/ / /	PROCIDTAG
0	32	37	39	44 63

Figure 86. RB Contents for *msgsnd*

The contents of RB are defined below. Bits 37:63 are referred to as the message payload.

Field **Description**

0:31 Reserved

32:36 **Type**

If Type=0x05, then a Directed Hypervisor Doorbell message is to be sent to the thread(s) specified in the Message Payload field.

All other values of the Type field are reserved; if the instruction is executed with this field set to a reserved value, the instruction is treated as a no-op.

37:38 **Broadcast** (B)

00 The message is sent to the thread for which $PIR_{44:63}$ is equal to the value of the PROCIDTAG field in the message payload.

01 The message is sent to all threads on the same sub-processor as the thread for which $PIR_{44:63}$ is equal to the value of the PROCIDTAG field in the message payload.

10 The message is sent to all threads on the same multi-threaded processor as the thread for which $PIR_{44:63}$ is equal to the value of the PROCIDTAG field in the message payload.

11 Reserved

39:43 Reserved

44:63 **PROCIDTAG**

This field indicates the recipient thread(s) as specified in the B field. If this field set to a value that is not the same as bits $PIR_{44:63}$ of any thread in the system, then the instruction behaves as if it were a no-op.

The actions taken on receipt of a message are defined in Section 10.2.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Programming Note

If ***msgsnd*** is used to notify the receiver that updates have been made to storage, an ***[lw]sync*** should be placed between the stores and the ***msgsnd***. See Section 5.9.2.

Message Clear *X-form*

msgclr RB

31	///	///	RB	238	/
0	6	11	16	21	31

```
t ← hypervisor thread number of executing thread
If(msgtype = 0x05) then
   clear any Directed Hypervisor Doorbell exception
   for thread t.
```

msgclr clears a message previously accepted by the thread executing the ***msgclr***.

Let msgtype be $(RB)_{32:36}$, and let t be the hypervisor thread number of the thread executing the ***msgclr*** instruction.

If msgtype = 0x05, then clear any Directed Hypervisor Doorbell exception that exists on thread t; otherwise, this instruction is treated as a no-op.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Programming Note

msgclr is typically issued only when MSR_{EE}=0. If ***msgclr*** is executed when MSR_{EE}=1 when a Directed Hypervisor Doorbell interrupt is about to occur, the corresponding interrupt may or may not occur.

Message Send Privileged *X-form*

msgsndp RB

31	///	///	RB	142	/
0	6	11	16	21	31

```
msgtype ← (RB)32:36
payload ← (RB)37:63
t ← (RB)57:63
if msgtype = 5 and
   t ≤ maximum privileged thread number
       on processor or sub-processor
  then
    DPDES63-t ← 1
    send_msg(msgtype, payload, t)
```

msgsndp sends a message to other threads that are on the same multi-threaded processor (if the processor is not in sub-processor mode) or to other threads that are on the same sub-processor (if the processor is in sub-processor mode). The message type and destination thread(s) are specified in RB.

RB

Message Payload

/ / /	TYPE	/ / /	TIRTAG
0	32	37 39	57 63

Figure 87. RB Contents for *msgsndp*

The contents of RB are defined below. Bits 37:63 are referred to as the message payload.

Bits	Description
37:56	Reserved
57:63	**TIRTAG** This message is sent to the thread for which the privileged thread number is equal to contents of the TIRTAG field of the message payload, and one of the following conditions applies. – for processors that are not partitioned into sub-processors, the thread is sent to the thread on the same multi-threaded processor for which the privileged thread number is equal to the contents of the TIRTAG field of the message payload. – for processors that are partitioned into sub-processors, the thread is sent to the thread on the same sub-processor for which the privileged thread number is equal to the contents of the TIRTAG field of the message payload. If ***msgsndp*** is executed with TIRTAG set to a value greater than the highest privileged thread number on the sub-processor (or on the multi-threaded processor if sub-processors are not supported), then this instruction behaves as a no-op

The actions taken on receipt of a message are defined in Section 10.2.

This instruction is privileged.

Special Registers Altered:
DPDES

Programming Note

If ***msgsndp*** is used to notify the receiver that updates have been made to storage, a ***lwsync*** or ***sync*** should be placed between the stores and the ***msgsndp***. See Section 5.9.2.

Message Clear Privileged *X-form*

msgclrp RB

31	///	///	RB	174	/
0	6	11	16	21	31

```
msgtype ← (RB)32:36
t ← privileged thread number of executing thread
IF(msgtype = 0x05)
  then
          DPDES63-t ← 0
```

msgclrp clears a message previously accepted by the thread executing the ***msgclrp***.

Let msgtype be $(RB)_{32:36}$, and let t be the privileged thread number of the thread executing the ***msgclrp***.

If msgtype = 0x05, then clear any Directed Privileged Doorbell exception that exists on thread t by setting $DPDES_{63-t}$ to 0; otherwise, this instruction is treated as a no-op.

This instruction is privileged.

Special Registers Altered:

DPDES

Programming Note

msgclrp is typically issued only when MSR_{EE}=0. If ***msgclrp*** is executed when MSR_{EE}=1 when a Directed Hypervisor Doorbell interrupt is about to occur, the corresponding interrupt may or may not occur.

Message Synchronize *X-form*

msgsync

31	///	///	///	886	/
0	6	11	16	21	31

In conjunction with the *Synchronize* and ***msgsnd*** instructions, the ***msgsync*** instruction provides an ordering function for stores that have been performed with respect to the thread executing the *Synchronize* and ***msgsnd*** instructions, relative to data accesses by other threads that are performed after a Directed Hypervisor Doorbell interrupt has occurred, as described in the *Synchronize* instruction description on p. 1021.

This instruction is hypervisor privileged.

Special Registers Altered:

None

Programming Note

When used in conjunction with ***msgsnd***, *Synchronize* with L = 0 or 2 is executed on the thread that will execute the ***msgsnd***, and ***msgsync*** is executed on another thread -- typically the thread that is the target of the ***msgsnd***, but possibly any other thread (partly because the software that services the Directed Hypervisor Doorbell interrupt may ultimately run on a thread other than that which received the exception). The *Synchronize* precedes the ***msgsnd***; the ***msgsync*** is executed after the Directed Hypervisor Doorbell interrupt occurs, and precedes all instructions that need to "see" the values stored by the stores that are in set A of the memory barrier created by the *Synchronize*; see Section 5.9.2, “Synchronize Instruction”.

Chapter 11. Synchronization Requirements for Context Alterations

Changing the contents of certain System Registers, the contents of SLB entries, or the contents of other system resources that control the context in which a program executes can have the side effect of altering the context in which data addresses and instruction addresses are interpreted, and in which instructions are executed and data accesses are performed. For example, changing MSR_{IR} from 0 to 1 has the side effect of enabling translation of instruction addresses. These side effects need not occur in program order, and therefore may require explicit synchronization by software. (Program order is defined in Book II.)

An instruction that alters the context in which data addresses or instruction addresses are interpreted, or in which instructions are executed or data accesses are performed, is called a *context-altering instruction*. This chapter covers all the context-altering instructions. The software synchronization required for them is shown in Table 6 (for data access) and Table 7 (for instruction fetch and execution).

The notation "CSI" in the tables means any context synchronizing instruction (e.g., ***sc***, ***isync***, or ***rfid***). A context synchronizing interrupt (i.e., any interrupt except non-recoverable System Reset or non-recoverable Machine Check) can be used instead of a context synchronizing instruction. If it is, phrases like "the synchronizing instruction", below, should be interpreted as meaning the instruction at which the interrupt occurs. If no software synchronization is required before (after) a context-altering instruction, "the synchronizing instruction before (after) the context-altering instruction" should be interpreted as meaning the context-altering instruction itself.

The synchronizing instruction before the context-altering instruction ensures that all instructions up to and including that synchronizing instruction are fetched and executed in the context that existed before the alteration. The synchronizing instruction after the context-altering instruction ensures that all instructions after that synchronizing instruction are fetched and executed in the context established by the alteration. Instructions after the first synchronizing instruction, up to and including the second synchronizing instruction, may be fetched or executed in either context.

If a sequence of instructions contains context-altering instructions and contains no instructions that are affected by any of the context alterations, no software synchronization is required within the sequence.

Programming Note

Sometimes advantage can be taken of the fact that certain events, such as interrupts, and certain instructions that occur naturally in the program, such as the ***rfid*** that returns from an interrupt handler, provide the required synchronization.

No software synchronization is required before or after a context-altering instruction that is also context synchronizing or when altering the MSR in most cases (see the tables). No software synchronization is required before most of the other alterations shown in Table 7, because all instructions preceding the context-altering instruction are fetched and decoded before the context-altering instruction is executed (the hardware must determine whether any of these preceding instructions are context synchronizing).

In situations such as context switch in which multiple SPRs are loaded in sequence, it is often the case that the composition of the implicit (implementation-specific, nonarchitectural) synchronizations performed for each individual ***mtspr*** will be excessive for the purpose. Software may identify such sequences by placing a ***mtgsr*** before the sequence. Hardware may respond to this identification by removing redundant synchronization so that the net synchronization effect approaches that of a single context synchronization at the end of the sequence. A potential side effect of the optimization is that the SPRs specified by the sequence may be loaded in an order other than that specified by the program with the result that if an exception interrupts the sequence, ***mtspr*** instructions past the point of interruption may have loaded their SPRs. When control returns to the interrupted sequence, any such ***mtspr*** instructions are re-executed. The programmer must ensure that this side effect will not affect the outcome of the sequence. The degree of optimization is implementation-specific. Transaction failure may compromise optimization.

Programming Note

Because the individual ***mtspr*** instructions in an optimized sequence may be executed in any order, a single sequence should not contain multiple loads of the same SPR, and should not contain any set of SPRs for which the relative order of execution of the ***mtspr*** instructions targeting SPRs in the set matters.

Unless otherwise stated, the material in this chapter assumes a single-threaded environment.

Instruction or Event	Required Before	Required After	Notes
event-based branch and ***rfebb***	none	none	21
interrupt	none	none	
rfid	none	none	
hrfid	none	none	
rfscv	none	none	
sc	none	none	
scv	none	none	
Trap	none	none	
mtspr (AMR)	CSI	CSI	13
mtspr (PIDR)	CSI	CSI	6
mtspr (DAWRn)	CSI	CSI	
mtspr (DAWRXn)	CSI	CSI	
mtspr (HRMOR)	CSI	CSI	11,17
mtspr (LPCR)	CSI	CSI	11, 14
mtspr (PTCR)	***ptesync***	CSI	3
mtmsrd (SF)	none	none	
mtmsrd (TS)	none	none	
mtmsrd (TM)	none	none	
mtmsr[***d***] (PR)	none	none	
mtmsr[***d***] (DR)	none	none	
mtspr (LPIDR)	CSI	CSI	6
slbie	CSI	CSI	4
slbieg	CSI	CSI	4,6
slbia	CSI	CSI	4
slbmte	CSI	CSI	4,10
tlbie	CSI	CSI	4,6
tlbiel	CSI	***ptesync***	4
Store(PTE)	none	{***ptesync***, CSI}	5,6
Store(STE)	none	{***ptesync***, CSI}	5,6
Store(PRTE)	none	{***ptesync***, CSI}	5,6
Store(PATE)	none	{***ptesync***, CSI}	5,6
transaction failure and all TM instructions except ***tcheck***	none	none	19

Table 6: Synchronization requirements for data access

Instruction or Event	Required Before	Required After	Notes
event-based branch and ***rfebb***	none	none	21
interrupt	none	none	
rfid	none	none	
hrfid	none	none	
rfscv	none	none	
sc	none	none	
scv	none	none	
Trap	none	none	
mtmsrd (SF)	none	none	7
mtmsrd (TS)	none	none	
mtmsrd (TM)	none	none	
mtmsr[***d***] (EE)	none	none	1
mtmsr[***d***] (PR)	none	none	8
mtmsr[***d***] (FP)	none	none	
mtmsr[***d***](FE0,FE1)	none	none	
mtmsr[***d***] (TE)	none	none	
mtmsr[***d***] (IR)	none	none	8
mtmsr[***d***] (RI)	none	none	
mtspr (DEC)	none	none	9
mtspr (PIDR)	CSI	CSI	6
mtspr (IAMR)	none	CSI	
mtspr (TFHAR)	none	none	
mtspr (TEXASR)	none	none	
mtspr (CTRL)	none	none	
mtspr (FSCR)	none	CSI	
mtspr (DPDES)	none	CSI	17
mtspr (CIABR)	none	CSI	
mtspr (HFSCR)	none	CSI	
mtspr (HDEC)	none	none	9
mtspr (HRMOR)	none	CSI	8,11,17
mtspr (LPCR)	none	CSI	11, 12, 14
mtspr (LPIDR)	CSI	CSI	6,14,17
mtspr (PCR)	none	CSI	17
mtspr (PTCR)	***ptesync***	CSI	3,17
mtspr (Perf. Mon.)	none	CSI	15,18
mtspr (BESCR)	none	CSI	16,18
slbie	none	CSI	4
slbieg	none	CSI	4,6
slbia	none	CSI	4
slbmte	none	CSI	4,8,10
tlbie	none	CSI	4,6
tlbiel	none	CSI	4
Store(PTE)	none	{***ptesync***, CSI}	5,6,8
Store(STE)	none	{***ptesync***, CSI}	5,6,8
Store(PRTE)	none	{***ptesync***, CSI}	5,6,8
Store(PATE)	none	{***ptesync***, CSI}	5,6,8
transaction failure and all TM instructions except ***tcheck***	none	none	19

Table 7: Synchronization requirements for instruction fetch and/or execution

Table 7: Synchronization requirements for instruction fetch and/or execution

Notes:

1. The effect of changing the EE bit is immediate, even if the ***mtmsr***[***d***] instruction is not context synchronizing (i.e., even if L=1).
   - If an ***mtmsr***[***d***] instruction sets the EE bit to 0, neither an External interrupt, a Decrementer interrupt nor a Performance Monitor interrupt occurs after the ***mtmsr***[***d***] is executed.
   - If an ***mtmsr***[***d***] instruction changes the EE bit from 0 to 1 when an External, Decrementer, Performance Monitor or higher priority exception exists, the corresponding interrupt occurs immediately after the ***mtmsr***[***d***] is executed, and before the next instruction is executed in the program that set EE to 1.
   - If a hypervisor executes the ***mtmsr***[***d***] instruction that sets the EE bit to 0, a Hypervisor Decrementer interrupt does not occur after ***mtmsr***[***d***] is executed as long as the thread remains in hypervisor state.
   - If the hypervisor executes an ***mtmsr***[***d***] instruction that changes the EE bit from 0 to 1 when a Hypervisor Decrementer or higher priority exception exists, the corresponding interrupt occurs immediately after the ***mtmsr***[***d***] instruction is executed, and before the next instruction is executed, provided HDICE is 1.
2. Synchronization requirements for this instruction are implementation-dependent.
3. The PTCR controls all implicit and explicit storage accesses performed by all threads on the processor when the thread is not in hypervisor real addressing mode. Modifying the PTCR requires that the following conditions be achieved on all threads on the processor.
   - the thread is in hypervisor real addressing mode
   - all previous accesses (implicit and explicit) initiated when the thread was not in hypervisor real addressing mode have been performed with respect to all threads
   - no subsequent accesses which require translation have been initiated
4. For data accesses, the context synchronizing instruction before the ***slbie***, ***slbieg***, ***slbia***, ***slbmte***, ***tlbie***, or ***tlbiel*** instruction ensures that all preceding instructions that access data storage have completed to a point at which they have reported all exceptions they will cause.

   The context synchronizing instruction after the ***slbie***, ***slbieg***, ***slbia***, ***slbmte***, ***tlbie*** or ***tlbiel*** instruction ensures that storage accesses associated with instructions following the context synchronizing instruction will not use the TLB entry(s) being invalidated.

   (For ***tlbie*** and ***tlbiel***, if it is necessary to order storage accesses associated with preceding instructions, or Reference and Change bit updates associated with preceding address translations, with respect to subsequent data accesses, a ***ptesync*** instruction must also be used, either before or after the ***tlbie*** or ***tlbiel*** instruction. These effects of the ***ptesync*** instruction are described in the last paragraph of Note 5.)
5. The notation "{***ptesync***,CSI}" denotes an instruction sequence. Other instructions may be interleaved with this sequence, but these instructions must appear in the order shown.

   No software synchronization is required before the *Store* instruction because (a) stores are not performed out-of-order and (b) address translations associated with instructions preceding the *Store* instruction are not performed again after the store has been performed (see Section 5.5). These properties ensure that all address translations associated with instructions preceding the *Store* instruction will be performed using the old contents of the PTE.

   The ***ptesync*** instruction after the *Store* instruction ensures that all searches of the Page Table that are performed after the ***ptesync*** instruction completes will use the value stored (or a value stored subsequently). The context synchronizing instruction after the ***ptesync*** instruction ensures that any address translations associated with instructions following the context synchronizing instruction that were performed using the old contents of the PTE will be discarded, with the result that these address translations will be performed again and, if there is no corresponding entry in any TLB, SLB, page walk cache, cache of Partition or Process Table entries, or implementation-specific address translation lookaside information, will use the value stored (or a value stored subsequently).

   The ***ptesync*** instruction also ensures that all storage accesses associated with instructions preceding the ***ptesync*** instruction, and all Reference and Change bit updates associated with additional address translations that were performed, by the thread executing the ***ptesync*** instruction, before the ***ptesync*** instruction is executed, will be performed with respect to any thread or mechanism, to the extent required by the associated Memory Coherence Required attributes, before any data accesses caused by instructions following the ***ptesync*** instruction are performed with respect to that thread or mechanism.
6. There are additional software synchronization requirements for this instruction in multi-threaded environments (e.g., it may be necessary to invalidate one or more TLB entries on all threads in the system and to be able to determine that the invalidations have completed and that all side effects of the invalidations have taken effect).

Section 5.10 gives examples of using ***tlbie***, *Store*, and related instructions to maintain the Page Table, in both multi-threaded environments and environments consisting of only a single-threaded processor.

Programming Note

In a multi-threaded system, if software locking is used to help ensure that the requirements described in Section 5.10 are satisfied, the ***lwsync*** instruction near the end of the lock acquisition sequence (see Section B.2.1.1 of Book II) may naturally provide the context synchronization that is required before the alteration.

7. The alteration must not cause an implicit branch in effective address space. Thus, when changing MSR_{SF} from 1 to 0, the ***mtmsrd*** instruction must have an effective address that is less than 2^{32} - 4. Furthermore, when changing MSR_{SF} from 0 to 1, the ***mtmsrd*** instruction must not be at effective address 2^{32} - 4 (see Section 5.3.2 on page 981).

8. The alteration must not cause an implicit branch in real address space. Thus the real address of the context-altering instruction and of each subsequent instruction, up to and including the next context synchronizing instruction, must be independent of whether the alteration has taken effect.

Programming Note

If it is desired to set MSR_{IR} to 1 early in an operating system interrupt handler, advantage can sometimes be taken of the fact that $EA_{0:3}$ are ignored when forming the real address when address translation is disabled and MSR_{HV} = 0. For example, if address translation resources are set such that effective address 0xn000_0000_0000_0000 maps to real address 0x000_0000_0000_0000 when address translation is enabled, where n is an arbitrary 4-bit value, the following code sequence, in real page 0, can be used early in the interrupt handler.

```
       la      rx,target
       li      ry,0xn000
       sldi    ry,ry,48
       or      rx,rx,ry  # set high-order
                           nibble of target
                           addr to 0xn
       mtctr   rx
       bcctr             # branch to targ

targ:  mfmsr   rx
       orir    x,rx,0x0020
       mtmsrd  rx        # set MSR_IR to 1
```

The ***mtmsrd*** does not cause an implicit branch in real address space because the real address of the next sequential instruction is independent of MSR_{IR}. Using ***mtmsrd***, rather than ***rfid*** (or similar context synchronizing instruction that alters the control flow), may yield better performance on some implementations.

(Variations on the technique are possible. For example, the target instruction of the ***bcctr*** can be in arbitrary real page P, where P is a 48-bit value, provided that effective address 0xn || P || 0x000 maps to real address P || 0x000 when address translation is enabled.)

9. The elapsed time between the contents of the Decrementer or Hypervisor Decrementer becoming negative and the signaling of the corresponding exception is not defined.

10. If an ***slbmte*** instruction alters the mapping, or associated attributes, of a currently mapped ESID, the ***slbmte*** must be preceded by an ***slbie*** (or ***slbia***) instruction that invalidates the existing translation. This applies even if the corresponding entry is no longer in the SLB (the translation may still be in implementation-specific address translation lookaside information). No software synchronization is needed between the ***slbie*** and the ***slbmte***, regardless of whether the index of the SLB entry (if any) containing the current translation is the same as the SLB index specified by the ***slbmte***.

No ***slbie*** (or ***slbia***) is needed if the ***slbmte*** instruction replaces a valid SLB entry with a mapping of a different ESID (e.g., to satisfy an SLB miss). However, the ***slbie*** is needed later if and when the translation that was contained in the replaced SLB entry is to be invalidated.

11. When the HRMOR or the VC field of the LPCR is modified, software must invalidate all implementation-specific lookaside information used in address translation that depends on the old contents of the register or field (i.e., the contents immediately before the modification). The ***slbia*** instruction can be used to invalidate all such implementation-specific lookaside information.

12. A context synchronizing instruction or event that is executed or occurs when $LPCR_{MER}$ = 1 does not necessarily ensure that the exception effects of $LPCR_{MER}$ are consistent with the contents of $LPCR_{MER}$. See Section 2.2.

13. This line applies regardless of which SPR number (13 or 29) is used for the AMR.

14. LPIDR when using HPT translation and $LPCR_{HR}$ must not be altered when MSR_{DR}=1 or MSR_{IR}=1; if they are, the results are undefined.

Programming Note

The prohibitions above are because of the difficulty of avoiding an implicit branch relative to the value of enabling software to avoid using hypervisor real addressing mode for the operation. (The tables used for translation are determined by the partition ID and $LPCR_{HR}$ is used as a shortcut. See Section 5.7.6 for details.)

15. This line applies to the following Performance Monitor SPRs: PMC1-6, MMCR0, MMCR1, MMCR2, and MMCRA.

16. This line applies to all SPR numbers that access the BESCR (800-803, 806).

17. There are additional software synchronization requirements when an ***mtspr*** instruction modifies this SPR in a multi-threaded environment. See Section 2.7.

18. As an alternative to a CSI, the execution of an ***rfebb*** instruction or the occurrence of an event-based branch is sufficient to provide the necessary synchronization.

19. These instructions and events, with the exception of nested ***tbegin.*** nested ***tend.***, TM instructions that except or are described to be treated as no-ops, *Transaction Abort Conditional* instructions that do not abort, and events and ***rfebb*** instructions for which the event did not take place in Transactional state, will change MSR_{TS}. No software synchronization is required.

Power ISA Book I-III Appendices

Appendix A. Illegal Instructions

With the exception of the instruction consisting entirely of binary 0s, the instructions in this class are available for future extensions of the Power ISA; that is, some future version of the Power ISA may define any of these instructions to perform new functions.

The following primary opcodes are illegal.

1, 5, 6

The following primary opcodes have unused extended opcodes. Their unused extended opcodes can be determined from the opcode maps in Appendix C of Book Appendices. All unused extended opcodes are illegal.

4, 19, 30, 31, 56, 5 , 58, 59, 60, 62, 63

The following primary+extended opcodes have unused expanded opcodes. Their unused expanded opcodes can be determined from the opcode maps in Appendix C of Book Appendices. All unused expanded opcodes are illegal.

primary / extended opcode
4 / 0b10110_000001
4 / 0b11110_000001
4 / 0b11000_000010
60 / 0b01011_01000.
60 / 0b10101_1011..
60 / 0b11101_1011..
63 / 0b11001_00100.
63 / 0b11010_00100.
63 / 0b10010_00111.

An instruction consisting entirely of binary 0s is illegal, and is guaranteed to be illegal in all future versions of this architecture.

Appendix B. Reserved Instructions

The instructions in this class are allocated to specific purposes that are outside the scope of the Power ISA.

The following types of instruction are included in this class.

1. The instruction having primary opcode 0, except the instruction consisting entirely of binary 0s (which is an illegal instruction; see Section 1.8.2, “Illegal Instruction Class” on page 22) and the extended opcode shown below.

   256 Service Processor “Attention”

2. Instructions for the POWER Architecture that have not been included in the Power ISA.
3. Implementation-specific instructions used to conform to the Power ISA specification.
4. Any other implementation-dependent instructions that are not defined in the Power ISA.

Appendix C. Opcode Maps

This appendix contains opcode maps showing the primary opcodes, extended opcodes, and expanded opcodes.

Table 8 describes the conventions used in the opcode maps.

The instruction consisting entirely of binary 0s causes the system illegal instruction error handler to be invoked for all members of the POWER family, and this is likely to remain true in future models (it is guaranteed in the Power ISA). An instruction having primary opcode 0 but not consisting entirely of binary 0s is reserved except for the following extended opcode (instruction bits 21:30).

256 Service Processor “Attention”

Table 8: Opcode Maps Legend

po

primary opcode (decimal format)

xop

extended or expanded opcode image (binary format)

- 0 instruction bit corresponding to an extended/expanded opcode bit having value of 0
- 1 instruction bit corresponding to an extended/expanded opcode bit having value of 1
- / reserved instruction bit, must have value of 0, otherwise invalid form
- . instruction bit corresponding to an operand or control bit, can have a value of either 0 or 1

book

Book instruction defined

version

ISA version instruction introduced

privilege

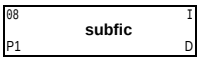
- P privileged instruction
- H hypervisor-privileged instruction

format

instruction format

Illegal opcode

Opcode having no previous or current assignment, available for future use

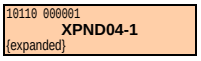


Defined opcode (primary, extended, or expanded)

Opcode assigned to a defined instruction

Primary opcode having an extended opcode field

Opcode having extended opcode field used to identify multiple instructions



Extended opcode having an expanded opcode field

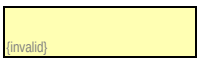
Opcode having expanded opcode field used to identify multiple instructions

Reserved opcode (primary, extended, or expanded)

Opcode is not available for future use without careful consideration

1. Opcode corresponds to an instruction defined in a previous version of the architecture that has been subsequently removed from the architecture. The opcode is treated as an illegal opcode.
2. Or, opcode is reserved for implementation-dependent use.

These opcodes will not be assigned a meaning in the Power ISA except after careful consideration of the effect of such assignment on existing implementations.



Invalid form opcode

Opcode corresponding to a defined instruction encoding with one or more reserved opcode bits having a value of 1

Table 9: Primary Opcode Map (opcode bits 0:5)

	000	001	010	011	100	101	110	111	
000	0	1	2 I **tdi** PPC D	3 I **twi** P1 D	4 **EXT04** {extended}	5	6	7 I **mulli** P1 D	000
001	8 I **subfic** P1 D	9 {reserved}	10 I **cmpli** P1 D	11 I **cmpi** P1 D	12 I **addic** P1 D	13 I **addic.** P1 D	14 I **addi** P1 D	15 I **addis** P1 D	001
010	16 I **bc[l][a]** P1 B	17 **EXT17** {extended}	18 I **b[l][a]** P1 I	19 **EXT19** {extended}	20 I **rlwimi[.]** P1 M	21 I **rlwinm[.]** P1 M	22 {reserved}	23 I **rlwnm[.]** P1 M	010
011	24 I **ori** P1 D	25 I **oris** P1 D	26 I **xori** P1 D	27 I **xoris** P1 D	28 I **andi.** P1 D	29 I **andis.** P1 D	30 **EXT30** {extended}	31 **EXT31** {extended}	011
100	32 I **lwz** P1 D	33 I **lwzu** P1 D	34 I **lbz** P1 D	35 I **lbzu** P1 D	36 I **stw** P1 D	37 I **stwu** P1 D	38 I **stb** P1 D	39 I **stbu** P1 D	100
101	40 I **lhz** P1 D	41 I **lhzu** P1 D	42 I **lha** P1 D	43 I **lhau** P1 D	44 I **sth** P1 D	45 I **sthu** P1 D	46 I **lmw** P1 D	47 I **stmw** P1 D	101
110	48 I **lfs** P1 D	49 I **lfsu** P1 D	50 I **lfd** P1 D	51 I **lfdu** P1 D	52 I **stfs** P1 D	53 I **stfsu** P1 D	54 I **stfd** P1 D	55 I **stfdu** P1 D	110
111	56 I **lq** v2.03 DQ	57 **EXT57** {extended}	58 **EXT58** {extended}	59 **EXT59** {extended}	60 **EXT60** {extended}	61 **EXT61** {extended}	62 **EXT62** {extended}	63 **EXT63** {extended}	111
	000	001	010	011	100	101	110	111	

Table 10: EXT17: Extended Opcode Map for Primary Opcode 17 (opcode bits 30:31)

00	01	10	11
	01 I **scv** v3.0 SC	1/ I **sc** PPC SC	1/ **sc** {invalid}
00	01	10	11

Table 11: EXT30: Extended Opcode Map for Primary Opcode 30 (opcode bits 27:30)

	000	001	010	011	100	101	110	111	
0	000- I **rldicl[.]** PPC MD	000- I **rldicl[.]** PPC MD	001- I **rldicr[.]** PPC MD	001- I **rldicr[.]** PPC MD	010- I **rldic[.]** PPC MD	010- I **rldic[.]** PPC MD	011- I **rldimi[.]** PPC MD	011- I **rldimi[.]** PPC MD	0
1	1000 I **rldcl[.]** PPC MDS	1001 I **rldcr[.]** PPC MDS			{reserved}	{reserved}	{reserved}	{reserved}	1
	000	001	010	011	100	101	110	111	

Table 12: EXT57: Extended Opcode Map for Primary Opcode 57 (opcode bits 30:31)

00	01	10	11
00 I **lfdp** v2.05 DS	{reserved}	10 I **lxsd** v3.0 DS	11 I **lxssp** v3.0 DS
00	01	10	11

Table 13: EXT58: Extended Opcode Map for Primary Opcode 58 (opcode bits 30:31)

00	01	10	11
00 I **ld** PPC DS	01 I **ldu** PPC DS	10 I **lwa** PPC DS	{reserved}
00	01	10	11

Table 14: EXT61: Extended Opcode Map for Primary Opcode 61 (opcode bits 21:30)

000	001	010	011	100	101	110	111
-00 I **stfdp** v2.05 DS	001 I **lxv** v3.0 DQ	-10 I **stxsd** v3.0 DS	-11 I **stxssp** v3.0 DS	-00 I **stfdp** v2.05 DS	101 I **stxv** v3.0 DQ	-10 I **stxsd** v3.0 DS	-11 I **stxssp** v3.0 DS
000	001	010	011	100	101	110	111

Table 15: EXT62: Extended Opcode Map for Primary Opcode 62 (opcode bits 21:30)

00	01	10	11
00 I **std** PPC DS	01 I **stdu** PPC DS	10 I **stq** v2.03 DS	{reserved}
00	01	10	11

Table 16: EXT04: Extended Opcode Map for Primary Opcode 4 (opcode bits 0:5) (Sheet 1 of 8)

	000000	000001	000010	000011	000100	000101	000110	000111	
00000	00000 000000 I **vaddubm** v2.03 VX	00000 000001 I **vmul10cuq** v3.0 VX	00000 000010 I **vmaxub** v2.03 VX		00000 000100 I **vrlb** v2.03 VX		00000 000110 I **vcmpequb** v2.03 VC	00000 000111 I **vcmpneb** v3.0 VC	00000
00001	00001 000000 I **vadduhm** v2.03 VX	00001 000001 I **vmul10ecuq** v3.0 VX	00001 000010 I **vmaxuh** v2.03 VX		00001 000100 I **vrlh** v2.03 VX		00001 000110 I **vcmpequh** v2.03 VC	00001 000111 I **vcmpneh** v3.0 VC	00001
00010	00010 000000 I **vadduwm** v2.03 VX		00010 000010 I **vmaxuw** v2.03 VX		00010 000100 I **vrlw** v2.03 VX	00010 000101 I **vrlwmi** v3.0 VX	00010 000110 I **vcmpequw** v2.03 VC	00010 000111 I **vcmpnew** v3.0 VC	00010
00011	00011 000000 I **vaddudm** v2.07 VX		00011 000010 I **vmaxud** v2.07 VX		00011 000100 I **vrld** v2.07 VX	00011 000101 I **vrldmi** v3.0 VX	00011 000110 I **vcmpeqfp** v2.03 VC	00011 000111 I **vcmpequd** v2.07 VC	00011
00100	00100 000000 I **vadduqm** v2.07 VX		00100 000010 I **vmaxsb** v2.03 VX		00100 000100 I **vslb** v2.03 VX			00100 000111 I **vcmpnezb** v3.0 VC	00100
00101	00101 000000 I **vaddcuq** v2.07 VX		00101 000010 I **vmaxsh** v2.03 VX		00101 000100 I **vslh** v2.03 VX			00101 000111 I **vcmpnezh** v3.0 VC	00101
00110	00110 000000 I **vaddcuw** v2.03 VX		00110 000010 I **vmaxsw** v2.03 VX		00110 000100 I **vslw** v2.03 VX	00110 000101 I **vrlwnm** v3.0 VX		00110 000111 I **vcmpnezw** v3.0 VC	00110
00111			00111 000010 I **vmaxsd** v2.07 VX		00111 000100 I **vsl** v2.03 VX	00111 000101 I **vrldnm** v3.0 VX	00111 000110 I **vcmpgefp** v2.03 VC		00111
01000	01000 000000 I **vaddubs** v2.03 VX	01000 000001 I **vmul10uq** v3.0 VX	01000 000010 I **vminub** v2.03 VX		01000 000100 I **vsrb** v2.03 VX		01000 000110 I **vcmpgtub** v2.03 VC		01000
01001	01001 000000 I **vadduhs** v2.03 VX	01001 000001 I **vmul10euq** v3.0 VX	01001 000010 I **vminuh** v2.03 VX		01001 000100 I **vsrh** v2.03 VX		01001 000110 I **vcmpgtuh** v2.03 VC		01001
01010	01010 000000 I **vadduws** v2.03 VX		01010 000010 I **vminuw** v2.03 VX		01010 000100 I **vsrw** v2.03 VX		01010 000110 I **vcmpgtuw** v2.03 VC		01010
01011			01011 000010 I **vminud** v2.07 VX		01011 000100 I **vsr** v2.03 VX		01011 000110 I **vcmpgtfp** v2.03 VC	01011 000111 I **vcmpgtud** v2.07 VC	01011
01100	01100 000000 I **vaddsbs** v2.03 VX		01100 000010 I **vminsb** v2.03 VX		01100 000100 I **vsrab** v2.03 VX		01100 000110 I **vcmpgtsb** v2.03 VC		01100
01101	01101 000000 I **vaddshs** v2.03 VX	01101 000001 I **bcdcpsgn.** v3.0 VX	01101 000010 I **vminsh** v2.03 VX		01101 000100 I **vsrah** v2.03 VX		01101 000110 I **vcmpgtsh** v2.03 VC		01101
01110	01110 000000 I **vaddsws** v2.03 VX		01110 000010 I **vminsw** v2.03 VX		01110 000100 I **vsraw** v2.03 VX		01110 000110 I **vcmpgtsw** v2.03 VC		01110
01111			01111 000010 I **vminsd** v2.07 VX		01111 000100 I **vsrad** v2.07 VX		01111 000110 I **vcmpbfp** v2.03 VC	01111 000111 I **vcmpgtsd** v2.07 VC	01111
10000	10000 000000 I **vsububm** v2.03 VX	1-000 000001 I **bcdadd.** v2.07 VX	10000 000010 I **vavgub** v2.03 VX	10000 000011 I **vabsdub** v3.0 VX	10000 000100 I **vand** v2.03 VX		10000 000110 I **vcmpequb.** v2.03 VC	10000 000111 I **vcmpneb.** v3.0 VC	10000
10001	10001 000000 I **vsubuhm** v2.03 VX	1-001 000001 I **bcdsub.** v2.07 VX	10001 000010 I **vavguh** v2.03 VX	10001 000011 I **vabsduh** v3.0 VX	10001 000100 I **vandc** v2.03 VX		10001 000110 I **vcmpequh.** v2.03 VC	10001 000111 I **vcmpneh.** v3.0 VC	10001
10010	10010 000000 I **vsubuwm** v2.03 VX	1/010 000001 I **bcdus.** v3.0 VX	10010 000010 I **vavguw** v2.03 VX	10010 000011 I **vabsduw** v3.0 VX	10010 000100 I **vor** v2.03 VX		10010 000110 I **vcmpequw.** v2.03 VC	10010 000111 I **vcmpnew.** v3.0 VC	10010
10011	10011 000000 I **vsubudm** v2.07 VX	1-011 000001 I **bcds.** v3.0 VX			10011 000100 I **vxor** v2.03 VX		10011 000110 I **vcmpeqfp.** v2.03 VC	10011 000111 I **vcmpequd.** v2.07 VC	10011
10100	10100 000000 I **vsubuqm** v2.07 VX	1-100 000001 I **bcdtrunc.** v3.0 VX	10100 000010 I **vavgsb** v2.03 VX		10100 000100 I **vnor** v2.03 VX			10100 000111 I **vcmpnezb.** v3.0 VC	10100
10101	10101 000000 I **vsubcuq** v2.07 VX	1/101 000001 I **bcdutrunc.** v3.0 VX	10101 000010 I **vavgsh** v2.03 VX		10101 000100 I **vorc** v2.07 VX			10101 000111 I **vcmpnezh.** v3.0 VC	10101
10110	10110 000000 I **vsubcuw** v2.03 VX	10110 000001 **XPND04-1A** {expanded}	10110 000010 I **vavgsw** v2.03 VX		10110 000100 I **vnand** v2.07 VX			10110 000111 I **vcmpnezw.** v3.0 VC	10110
10111		1-111 000001 I **bcdsr.** v3.0 VX			10111 000100 I **vsld** v2.07 VX		10111 000110 I **vcmpgefp.** v2.03 VC		10111
11000	11000 000000 I **vsububs** v2.03 VX	1-000 000001 I **bcdadd.** v2.07 VX	11000 000010 **XPND04-2** {expanded}		11000 000100 I **mfvscr** v2.03 VX		11000 000110 I **vcmpgtub.** v2.03 VC		11000
11001	11001 000000 I **vsubuhs** v2.03 VX	1-001 000001 I **bcdsub.** v2.07 VX			11001 000100 I **mtvscr** v2.03 VX		11001 000110 I **vcmpgtuh.** v2.03 VC		11001
11010	11010 000000 I **vsubuws** v2.03 VX	1/010 000001 **bcdus.** {invalid}	11010 000010 I **vshasigmaw** v2.07 VX		11010 000100 I **veqv** v2.07 VX		11010 000110 I **vcmpgtuw.** v2.03 VC		11010
11011		1-011 000001 I **bcds.** v3.0 VX	11011 000010 I **vshasigmad** v2.07 VX		11011 000100 I **vsrd** v2.07 VX		11011 000110 I **vcmpgtfp.** v2.03 VC	11011 000111 I **vcmpgtud.** v2.07 VC	11011
11100	11100 000000 I **vsubsbs** v2.03 VX	1-100 000001 I **bcdtrunc.** v3.0 VX	11100 000010 I **vclzb** v2.07 VX	11100 000011 I **vpopcntb** v2.07 VX	11100 000100 I **vsrv** v3.0 VX		11100 000110 I **vcmpgtsb.** v2.03 VC		11100
11101	11101 000000 I **vsubshs** v2.03 VX	1/101 000001 **bcdutrunc.** {invalid}	11101 000010 I **vclzh** v2.07 VX	11101 000011 I **vpopcnth** v2.07 VX	11101 000100 I **vslv** v3.0 VX		11101 000110 I **vcmpgtsh.** v2.03 VC		11101
11110	11110 000000 I **vsubsws** v2.03 VX	11110 000001 **XPND04-1B** {expanded}	11110 000010 I **vclzw** v2.07 VX	11110 000011 I **vpopcntw** v2.07 VX			11110 000110 I **vcmpgtsw.** v2.03 VC		11110
11111		1-111 000001 I **bcdsr.** v3.0 VX	11111 000010 I **vclzd** v2.07 VX	11111 000011 I **vpopcntd** v2.07 VX			11111 000110 I **vcmpbfp.** v2.03 VC	11111 000111 I **vcmpgtsd.** v2.07 VC	11111
	000000	000001	000010	000011	000100	000101	000110	000111	

Table 16: EXT04: Extended Opcode Map for Primary Opcode 4 (opcode bits 0:5) (Sheet 2 of 8)

	001000	001001	001010	001011	001100	001101	001110	001111	
00000	00000 001000 I **vmuloub** v2.03 VX		00000 001010 I **vaddfp** v2.03 VX		00000 001100 I **vmrghb** v2.03 VX		00000 001110 I **vpkuhum** v2.03 VX		00000
00001	00001 001000 I **vmulouh** v2.03 VX		00001 001010 I **vsubfp** v2.03 VX		00001 001100 I **vmrghh** v2.03 VX		00001 001110 I **vpkuwum** v2.03 VX		00001
00010	00010 001000 I **vmulouw** v2.07 VX	00010 001001 I **vmuluwm** v2.07 VX			00010 001100 I **vmrghw** v2.03 VX		00010 001110 I **vpkuhus** v2.03 VX		00010
00011							00011 001110 I **vpkuwus** v2.03 VX		00011
00100	00100 001000 I **vmulosb** v2.03 VX		00100 001010 I **vrefp** v2.03 VX		00100 001100 I **vmrglb** v2.03 VX		00100 001110 I **vpkshus** v2.03 VX		00100
00101	00101 001000 I **vmulosh** v2.03 VX		00101 001010 I **vrsqrtefp** v2.03 VX		00101 001100 I **vmrglh** v2.03 VX		00101 001110 I **vpkswus** v2.03 VX		00101
00110	00110 001000 I **vmulosw** v2.07 VX		00110 001010 I **vexptefp** v2.03 VX		00110 001100 I **vmrglw** v2.03 VX		00110 001110 I **vpkshss** v2.03 VX		00110
00111			00111 001010 I **vlogefp** v2.03 VX				00111 001110 I **vpkswss** v2.03 VX		00111
01000	01000 001000 I **vmuleub** v2.03 VX		01000 001010 I **vrfin** v2.03 VX		01000 001100 I **vspltb** v2.03 VX	01000 001101 I **vextractub** v3.0 VX	01000 001110 I **vupkhsb** v2.03 VX		01000
01001	01001 001000 I **vmuleuh** v2.03 VX		01001 001010 I **vrfiz** v2.03 VX		01001 001100 I **vsplth** v2.03 VX	01001 001101 I **vextractuh** v3.0 VX	01001 001110 I **vupkhsh** v2.03 VX		01001
01010	01010 001000 I **vmuleuw** v2.07 VX		01010 001010 I **vrfip** v2.03 VX		01010 001100 I **vspltw** v2.03 VX	01010 001101 I **vextractuw** v3.0 VX	01010 001110 I **vupklsb** v2.03 VX		01010
01011			01011 001010 I **vrfim** v2.03 VX			01011 001101 I **vextractd** v3.0 VX	01011 001110 I **vupklsh** v2.03 VX		01011
01100	01100 001000 I **vmulesb** v2.03 VX		01100 001010 I **vcfux** v2.03 VX		01100 001100 I **vspltisb** v2.03 VX	01100 001101 I **vinsertb** v3.0 VX	01100 001110 I **vpkpx** v2.03 VX		01100
01101	01101 001000 I **vmulesh** v2.03 VX		01101 001010 I **vcfsx** v2.03 VX		01101 001100 I **vspltish** v2.03 VX	01101 001101 I **vinserth** v3.0 VX	01101 001110 I **vupkhpx** v2.03 VX		01101
01110	01110 001000 I **vmulesw** v2.07 VX		01110 001010 I **vctuxs** v2.03 VX		01110 001100 I **vspltisw** v2.03 VX	01110 001101 I **vinsertw** v3.0 VX			01110
01111			01111 001010 I **vctsxs** v2.03 VX			01111 001101 I **vinsertd** v3.0 VX	01111 001110 I **vupklpx** v2.03 VX		01111
10000	10000 001000 I **vpmsumb** v2.07 VX		10000 001010 I **vmaxfp** v2.03 VX		10000 001100 I **vslo** v2.03 VX				10000
10001	10001 001000 I **vpmsumh** v2.07 VX		10001 001010 I **vminfp** v2.03 VX		10001 001100 I **vsro** v2.03 VX		10001 001110 I **vpkudum** v2.07 VX		10001
10010	10010 001000 I **vpmsumw** v2.07 VX								10010
10011	10011 001000 I **vpmsumd** v2.07 VX						10011 001110 I **vpkudus** v2.07 VX		10011
10100	10100 001000 I **vcipher** v2.07 VX	10100 001001 I **vcipherlast** v2.07 VX			10100 001100 I **vgbbd** v2.07 VX				10100
10101	10101 001000 I **vncipher** v2.07 VX	10101 001001 I **vncipherlast** v2.07 VX			10101 001100 I **vbpermq** v2.07 VX		10101 001110 I **vpksdus** v2.07 VX		10101
10110									10110
10111	10111 001000 I **vsbox** v2.07 VX				10111 001100 I **vbpermd** v3.0 VX		10111 001110 I **vpksdss** v2.07 VX		10111
11000	11000 001000 I **vsum4ubs** v2.03 VX					11000 001101 I **vextublx** v3.0 VX			11000
11001	11001 001000 I **vsum4shs** v2.03 VX					11001 001101 I **vextuhlx** v3.0 VX	11001 001110 I **vupkhsw** v2.07 VX		11001
11010	11010 001000 I **vsum2sws** v2.03 VX				11010 001100 I **vmrgow** v2.07 VX	11010 001101 I **vextuwlx** v3.0 VX			11010
11011							11011 001110 I **vupklsw** v2.07 VX		11011
11100	11100 001000 I **vsum4sbs** v2.03 VX					11100 001101 I **vextubrx** v3.0 VX			11100
11101						11101 001101 I **vextuhrx** v3.0 VX			11101
11110	11110 001000 I **vsumsws** v2.03 VX				11110 001100 I **vmrgew** v2.07 VX	11110 001101 I **vextuwrx** v3.0 VX			11110
11111									11111
	001000	001001	001010	001011	001100	001101	001110	001111	

Table 16: EXT04: Extended Opcode Map for Primary Opcode 4 (opcode bits 0:5) (Sheet 3 of 8)

	010000	010001	010010	010011	010100	010101	010110	010111	
00000									00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	010000	010001	010010	010011	010100	010101	010110	010111	

Table 16: EXT04: Extended Opcode Map for Primary Opcode 4 (opcode bits 0:5) (Sheet 4 of 8)

	011000	011001	011010	011011	011100	011101	011110	011111	
00000									00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	011000	011001	011010	011011	011100	011101	011110	011111	

Table 16: EXT04: Extended Opcode Map for Primary Opcode 4 (opcode bits 0:5) (Sheet 5 of 8)

	100000	100001	100010	100011	100100	100101	100110	100111	
00000	----- 100000 I **vmhaddshs** v2.03 VA	----- 100001 I **vmhraddshs** v2.03 VA	----- 100010 I **vmladduhm** v2.03 VA	----- 100011 I **vmsumudm** v3.0B VA	----- 100100 I **vmsumubm** v2.03 VA	----- 100101 I **vmsummbm** v2.03 VA	----- 100110 I **vmsumuhm** v2.03 VA	----- 100111 I **vmsumuhs** v2.03 VA	00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	100000	100001	100010	100011	100100	100101	100110	100111	

Table 16: EXT04: Extended Opcode Map for Primary Opcode 4 (opcode bits 0:5) (Sheet 6 of 8)

	101000	101001	101010	101011	101100	101101	101110	101111	
00000	----- 101000 I **vmsumshm** v2.03 VA	----- 101001 I **vmsumshs** v2.03 VA	----- 101010 I **vsel** v2.03 VA	----- 101011 I **vperm** v2.03 VA	/---- 101100 I **vsldoi** v2.03 VA	----- 101101 I **vpermxor** v2.07 VA	----- 101110 I **vmaddfp** v2.03 VA	----- 101111 I **vnmsubfp** v2.03 VA	00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000					/---- 101100 **vsldoi** {invalid}				10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	101000	101001	101010	101011	101100	101101	101110	101111	

Table 16: EXT04: Extended Opcode Map for Primary Opcode 4 (opcode bits 0:5) (Sheet 7 of 8)

	110000	110001	110010	110011	110100	110101	110110	110111	
00000	----- 110000 I **maddhd** v3.0 VA	----- 110001 I **maddhdu** v3.0 VA		----- 110011 I **maddld** v3.0 VA					00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	110000	110001	110010	110011	110100	110101	110110	110111	

Table 16: EXT04: Extended Opcode Map for Primary Opcode 4 (opcode bits 0:5) (Sheet 8 of 8)

	111000	111001	111010	111011	111100	111101	111110	111111	
00000				111011 I **vpermr** v3.0 VA	111100 I **vaddeuqm** v2.07 VA	111101 I **vaddecuq** v2.07 VA	111110 I **vsubeuqm** v2.07 VA	111111 I **vsubecuq** v2.07 VA	00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	111000	111001	111010	111011	111100	111101	111110	111111	

Table 17: XPND04-1A: Extended Opcode Map for PO=4 XO=0b10110_000001 (opcode bits 11:15)

	000	001	010	011	100	101	110	111	
00	00 000 I **bcdctsq.** v3.0 VX		00 010 I **bcdcfsq.** v3.0 VX		00 100 I **bcdctz.** v3.0 VX	00 101 I **bcdctn.** v3.0 VX	00 110 I **bcdcfz.** v3.0 VX	00 111 I **bcdcfn.** v3.0 VX	00
01									01
10									10
11								11111 I **bcdsetsgn.** v3.0 VX	11
	000	001	010	011	100	101	110	111	

Table 18: XPND04-1B: Extended Opcode Map for PO=4 XO=0b11110_000001 (opcode bits 11:15)

	000	001	010	011	100	101	110	111	
00	00 000 I **bcdctsq.** {invalid} VX		00 010 I **bcdcfsq.** v3.0 VX		00 100 I **bcdctz.** v3.0 VX	00101 1/110 000001 I **bcdctn.** {invalid} VX	00 110 I **bcdcfz.** v3.0 VX	00 111 I **bcdcfn.** v3.0 VX	00
01									01
10									10
11								11 111 I **bcdsetsgn.** v3.0 VX	11
	000	001	010	011	100	101	110	111	

Table 19: XPND04-2: Extended Opcode Map for PO=4 XO=0b11000 000010 (opcode bits 11:15)

	000	001	010	011	100	101	110	111	
00	00 000 I **vclzlsbb** v3.0 VX	00 001 I **vctzlsbb** v3.0 VX					00 110 I **vnegw** v3.0 VX	00 111 I **vnegd** v3.0 VX	00
01	01 000 I **vprtybw** v3.0 VX	01 001 I **vprtybd** v3.0 VX	01 010 I **vprtybq** v3.0 VX						01
10	10 000 I **vextsb2w** v3.0 VX	10 001 I **vextsh2w** v3.0 VX							10
11	11 000 I **vextsb2d** v3.0 VX	11 001 I **vextsh2d** v3.0 VX	11 010 I **vextsw2d** v3.0 VX		11 100 I **vctzb** v3.0 VX	11 101 I **vctzh** v3.0 VX	11 110 I **vctzw** v3.0 VX	11 111 I **vctzd** v3.0 VX	11
	000	001	010	011	100	101	110	111	

Table 20: EXT19: Extended Opcode Map for Primary Opcode 19 (opcode bits 21:30) (Sheet 1 of 4)

	00000	00001	00010	00011	00100	00101	00110	00111	
00000	00000 00000 I **mcrf** P1 XL		----- 00010 I **addpcis** v3.0 DX						00000
00001		00001 00001 I **crnor** P1 XL							00001
00010									00010
00011									00011
00100		00100 00001 I **crandc** P1 XL							00100
00101									00101
00110		00110 00001 I **crxor** P1 XL							00110
00111		00111 00001 I **crnand** P1 XL							00111
01000		01000 00001 I **crand** P1 XL							01000
01001		01001 00001 I **creqv** P1 XL							01001
01010									01010
01011									01011
01100									01100
01101		01101 00001 I **crorc** P1 XL							01101
01110		01110 00001 I **cror** P1 XL							01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	00000	00001	00010	00011	00100	00101	00110	00111	

Table 20: EXT19: Extended Opcode Map for Primary Opcode 19 (opcode bits 21:30) (Sheet 2 of 4)

	01000	01001	01010	01011	01100	01101	01110	01111	
00000									00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	01000	01001	01010	01011	01100	01101	01110	01111	

Table 20: EXT19: Extended Opcode Map for Primary Opcode 19 (opcode bits 21:30) (Sheet 3 of 4)

	10000	10001	10010	10011	10100	10101	10110	10111	
00000	00000 10000 I **bclr[l]** P1 XL		00000 10010 III **rfid** PPC P XL						00000
00001			{reserved}	{reserved}					00001
00010			00010 10010 III **rfscv** v3.0 P XL						00010
00011									00011
00100			00100 10010 I **rfebb** v2.07 XL				00100 10110 II **isync** P1 XL		00100
00101									00101
00110									00110
00111									00111
01000			01000 10010 III **hrfid** v2.02 HV XL						01000
01001									01001
01010									01010
01011			01011 10010 III **stop** v3.0 P XL						01011
01100			{reserved}						01100
01101			{reserved}						01101
01110			{reserved}						01110
01111			{reserved}						01111
10000	10000 10000 I **bcctr[l]** P1 XL								10000
10001	10001 10000 I **bctar[l]** v2.07 XL								10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	10000	10001	10010	10011	10100	10101	10110	10111	

Table 20: EXT19: Extended Opcode Map for Primary Opcode 19 (opcode bits 21:30) (Sheet 4 of 4)

	11000	11001	11010	11011	11100	11101	11110	11111	
00000									00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	11000	11001	11010	11011	11100	11101	11110	11111	

Table 21: EXT31: Extended Opcode Map for Primary Opcode 31 (opcode bits 21:30) (Sheet 1 of 4)

	00000	00001	00010	00011	00100	00101	00110	00111	
00000	00000 00000 I **cmp** P1 X				00000 00100 I **tw** P1 X		00000 00110 I **lvsl** v2.03 X	00000 00111 I **lvebx** v2.03 X	00000
00001	00001 00000 I **cmpl** P1 X				{reserved}		00001 00110 I **lvsr** v2.03 X	00001 00111 I **lvehx** v2.03 X	00001
00010	{reserved}				00010 00100 I **td** PPC X			00010 00111 I **lvewx** v2.03 X	00010
00011					{reserved}			00011 00111 I **lvx** v2.03 X	00011
00100	00100 00000 I **setb** v3.0 VX			{reserved}				00100 00111 I **stvebx** v2.03 X	00100
00101				{reserved}	{reserved}			00101 00111 I **stvehx** v2.03 X	00101
00110	00110 00000 I **cmprb** v3.0 X							00110 00111 I **stvewx** v2.03 X	00110
00111	00111 00000 I **cmpeqb** v3.0 X				{reserved}			00111 00111 I **stvx** v2.03 X	00111
01000							{reserved}		01000
01001					{reserved}				01001
01010				{reserved}					01010
01011					{reserved}			01011 00111 I **lvxl** v2.03 X	01011
01100									01100
01101					{reserved}				01101
01110				{reserved}			{reserved}		01110
01111					{reserved}		{reserved}	01111 00111 I **stvxl** v2.03 X	01111
10000	{reserved}							{reserved}	10000
10001	{reserved}				{reserved}			{reserved}	10001
10010	10010 00000 I **mcrxrx** v3.0 X						10010 00110 II **lwat** v3.0 X		10010
10011					{reserved}		10011 00110 II **ldat** v3.0 X		10011
10100								{reserved}	10100
10101					{reserved}			{reserved}	10101
10110							10110 00110 II **stwat** v3.0 X		10110
10111					{reserved}		10111 00110 II **stdat** v3.0 X		10111
11000							11000 00110 II **copy** v3.0 X	/V{reserved}	11000
11001					{reserved}			{reserved}	11001
11010							11010 00110 II **cpabort** v3.0 X		11010
11011					{reserved}				11011
11100							11100 00110 II **paste[.]** v3.0 X	{reserved}	11100
11101					{reserved}			{reserved}	11101
11110							{reserved}		11110
11111					{reserved}		{reserved}		11111
	00000	00001	00010	00011	00100	00101	00110	00111	

Table 21: EXT31: Extended Opcode Map for Primary Opcode 31 (opcode bits 21:30) (Sheet 2 of 4)

	01000	01001	01010	01011	01100	01101	01110	01111	
00000	00000 01000 I **subfc[.]** P1 XO	/0000 01001 I **mulhdu[.]** PPC XO	00000 01010 I **addc[.]** P1 XO	/0000 01011 I **mulhwu[.]** PPC XO	00000 01100 I **lxsiwzx** v2.07 X			----- 01111 I **isel** v2.03 A	00000
00001	00001 01000 I **subf[.]** PPC XO								00001
00010		/0010 01001 I **mulhd[.]** PPC XO	/0010 01010 I **addg6s** v2.06 XO	/0010 01011 I **mulhw[.]** PPC XO	00010 01100 I **lxsiwax** v2.07 X				00010
00011	00011 01000 I **neg[.]** P1 XO			{reserved}					00011
00100	00100 01000 I **subfe[.]** P1 XO		00100 01010 I **adde[.]** P1 XO		00100 01100 I **stxsiwx** v2.07 X		00100 01110 III **msgsndp** v2.07 P X		00100
00101			--101 01010 I **addex** v3.0B X				00101 01110 III **msgclrp** v2.07 P X		00101
00110	00110 01000 I **subfze[.]** P1 XO		00110 01010 I **addze[.]** P1 XO				00110 01110 III **msgsnd** v2.07 HV X		00110
00111	00111 01000 I **subfme[.]** P1 XO	00111 01001 I **mulld[.]** PPC XO	00111 01010 I **addme[.]** P1 XO	00111 01011 I **mullw[.]** P1 XO			00111 01110 III **msgclr** v2.07 HV X		00111
01000	{reserved}	01000 01001 I **modud** v3.0 X	01000 01010 I **add[.]** P1 XO	01000 01011 I **moduw** v3.0 X	01000 01100 I **lxvx** v3.0 X	01000 01101 I **lxvl** v3.0 X			01000
01001						01001 01101 I **lxvll** v3.0 X	01001 01110 I **mfbhrbe** v2.07 X		01001
01010				{reserved}	01010 01100 I **lxvdsx** v2.06 X				01010
01011	{reserved}			{reserved}	01011 01100 I **lxvwsx** v3.0 X				01011
01100		01100 01001 I **divdeu[.]** v2.06 XO		01100 01011 I **divweu[.]** v2.06 XO	01100 01100 I **stxvx** v3.0 X	01100 01101 I **stxvl** v3.0 X			01100
01101		01101 01001 I **divde[.]** v2.06 XO	--101 01010 I **addex** v3.0B X	01101 01011 I **divwe[.]** v2.06 XO		01101 01101 I **stxvll** v3.0 X	01101 01110 I **clrbhrb** v2.07 X		01101
01110		01110 01001 I **divdu[.]** PPC XO		01110 01011 I **divwu[.]** PPC XO					01110
01111	{reserved}	01111 01001 I **divd[.]** PPC XO		01111 01011 I **divw[.]** PPC XO					01111
10000	10000 01000 I **subfco[.]** P1 XO	/0000 01001 **mulhdu[.]** {invalid}	10000 01010 I **addco[.]** P1 XO	/0000 01011 **mulhwu[.]** {invalid}	10000 01100 I **lxsspx** v2.07 X				10000
10001	10001 01000 I **subfo[.]** PPC XO								10001
10010		/0010 01001 **mulhd[.]** {invalid}	/0010 01010 **addg6s** {invalid}	/0010 01011 **mulhw[.]** {invalid}	10010 01100 I **lxsdx** v2.06 X				10010
10011	10011 01000 I **nego[.]** P1 XO			{reserved}					10011
10100	10100 01000 I **subfeo[.]** P1 XO		10100 01010 I **addeo[.]** P1 XO		10100 01100 I **stxsspx** v2.07 X		10100 01110 II **tbegin.** v2.07 X		10100
10101			--101 01010 I **addex** v3.0B X				10101 01110 II **tend.** v2.07 X		10101
10110	10110 01000 I **subfzeo[.]** P1 XO		10110 01010 I **addzeo[.]** P1 XO		10110 01100 I **stxsdx** v2.06 X		10110 01110 II **tcheck** v2.07 X		10110
10111	10111 01000 I **subfmeo[.]** P1 XO	10111 01001 I **mulldo[.]** PPC XO	10111 01010 I **addmeo[.]** P1 XO	10111 01011 I **mullwo[.]** P1 XO			10111 01110 II **tsr.** v2.07 X		10111
11000	{reserved}	11000 01001 I **modsd** v3.0 X	11000 01010 I **addo[.]** P1 XO	11000 01011 I **modsw** v3.0 X	11000 01100 I **lxvw4x** v2.06 X	11000 01101 I **lxsibzx** v3.0 X	11000 01110 II **tabortwc.** v2.07 X		11000
11001					11001 01100 I **lxvh8x** v3.0 X	11001 01101 I **lxsihzx** v3.0 X	11001 01110 II **tabortdc.** v2.07 X		11001
11010				{reserved}	11010 01100 I **lxvd2x** v2.06 X		11010 01110 II **tabortwci.** v2.07 X		11010
11011	{reserved}			{reserved}	11011 01100 I **lxvb16x** v3.0 X		11011 01110 II **tabortdci.** v2.07 X		11011
11100		11100 01001 I **divdeuo[.]** v2.06 XO		11100 01011 I **divweuo[.]** v2.06 XO	11100 01100 I **stxvw4x** v2.06 X	11100 01101 I **stxsibx** v3.0 X	11100 01110 II **tabort.** v2.07 X		11100
11101		11101 01001 I **divdeo[.]** v2.06 XO	--101 01010 I **addex** v3.0B X	11101 01011 I **divweo[.]** v2.06 XO	11101 01100 I **stxvh8x** v3.0 X	11101 01101 I **stxsihx** v3.0 X	11101 01110 II **treclaim.** v2.07 X		11101
11110		11110 01001 I **divduo[.]** PPC XO		11110 01011 I **divwuo[.]** PPC XO	11110 01100 I **stxvd2x** v2.06 X				11110
11111	{reserved}	11111 01001 I **divdo[.]** PPC XO		11111 01011 I **divwo[.]** PPC XO	11111 01100 I **stxvb16x** v3.0 X		11111 01110 II **trechkpt.** v2.07 X		11111
	01000	01001	01010	01011	01100	01101	01110	01111	

Table 21: EXT31: Extended Opcode Map for Primary Opcode 31 (opcode bits 21:30) (Sheet 3 of 4)

	10000	10001	10010	10011	10100	10101	10110	10111	
00000				00000 10011 I **mfcr/mfocrf** P1/v2.01 XFX	00000 10100 II **lwarx** PPC X	00000 10101 I **ldx** PPC X	00000 10110 II **icbt** v2.07 X	00000 10111 I **lwzx** P1 X	00000
00001				00001 10011 I **mfvsrd** v2.07 XX1	00001 10100 II **lbarx** v2.06 X	00001 10101 I **ldux** PPC X	00001 10110 II **dcbst** PPC X	00001 10111 I **lwzux** P1 X	00001
00010			{reserved}	00010 10011 III **mfmsr** P1 P X	00010 10100 II **ldarx** PPC X		00010 10110 II **dcbf** PPC X	00010 10111 I **lbzx** P1 X	00010
00011			{reserved}	00011 10011 I **mfvsrwz** v2.07 XX1	00011 10100 II **lharx** v2.06 X		{reserved}	00011 10111 I **lbzux** P1 X	00011
00100	00100 10000 I **mtcrf/mtocrf** P1/v2.01 XFX		00100 10010 III **mtmsr** P1 P X	{reserved}		00100 10101 I **stdx** PPC X	00100 10110 II **stwcx.** PPC X	00100 10111 I **stwx** P1 X	00100
00101			00101 10010 III **mtmsrd** PPC P X	00101 10011 I **mtvsrd** v2.07 XX1		00101 10101 I **stdux** PPC X	00101 10110 I **stqcx.** v2.07 X	00101 10111 I **stwux** P1 X	00101
00110			{reserved}	00110 10011 I **mtvsrwa** v2.07 XX1			00110 10110 II **stdcx.** PPC X	00110 10111 I **stbx** P1 X	00110
00111			{reserved} X	00111 10011 I **mtvsrwz** v2.07 XX1			00111 10110 II **dcbtst** PPC X	00111 10111 I **stbux** P1 X	00111
01000			01000 10010 III **tlbiel** v2.03 P X		01000 10100 I **lqarx** v2.07 X	{reserved}	01000 10110 II **dcbt** PPC X	01000 10111 I **lhzx** P1 X	01000
01001			01001 10010 III **tlbie** P1 HV X	01001 10011 I **mfvsrld** v3.0 XX1	{reserved}		{reserved}	01001 10111 I **lhzux** P1 X	01001
01010			01010 10010 III **slbsync** v3.0 P X	01010 10011 X **mfspr** P1 O X		01010 10101 I **lwax** PPC X	{reserved}	01010 10111 I **lhax** P1 X	01010
01011			{reserved}	01011 10011 II **mftb** PPC X		01011 10101 I **lwaux** PPC X	{reserved}	01011 10111 I **lhaux** P1 X	01011
01100			01100 10010 III **slbmte** v2.00 P X	01100 10011 I **mtvsrws** v3.0 XX1				01100 10111 I **sthx** P1 X	01100
01101			01101 10010 III **slbie** PPC P X	01101 10011 I **mtvsrdd** v3.0 XX1			{reserved}	01101 10111 I **sthux** P1 X	01101
01110			01110 10010 III **slbieg** v3.0 P X	01110 10011 X **mtspr** P1 O X			{reserved}		01110
01111			01111 10010 III **slbia** PPC P X	{reserved}			{reserved}		01111
10000			10000 10010 I **nop** v2.05 X	{reserved}	10000 10100 I **ldbrx** v2.06 X	10000 10101 I **lswx** P1 X	10000 10110 I **lwbrx** P1 X	10000 10111 I **lfsx** P1 X	10000
10001			10001 10010 I **nop** v2.05 X			{reserved}	10001 10110 III **tlbsync** PPC HV/P X	10001 10111 I **lfsux** P1 X	10001
10010			10010 10010 I **nop** v2.05 X	{reserved}		10010 10101 I **lswi** P1 X	10010 10110 II **sync** P1 X	10010 10111 I **lfdx** P1 X	10010
10011			10011 10010 I **nop** v2.05 X	{reserved}		{reserved}	{reserved}	10011 10111 I **lfdux** P1 X	10011
10100			10100 10010 I **nop** v2.05 X	{reserved}	10100 10100 I **stdbrx** v2.06 X	10100 10101 I **stswx** P1 X	10100 10110 I **stwbrx** P1 X	10100 10111 I **stfsx** P1 X	10100
10101			10101 10010 I **nop** v2.05 X			{reserved}	10101 10110 II **stbcx.** v2.06 X	10101 10111 I **stfsux** P1 X	10101
10110			10110 10010 I **nop** v2.05 X			10110 10101 I **stswi** P1 X	10110 10110 II **sthcx.** v2.06 X	10110 10111 I **stfdx** P1 X	10110
10111			10111 10010 I **nop** v2.05 X	10111 10011 I **darn** v3.0 X		{reserved}	{reserved}	10111 10111 I **stfdux** P1 X	10111
11000						11000 10101 III **lwzcix** v2.05 HV X	11000 10110 I **lhbrx** P1 X	11000 10111 I **lfdpx** v2.05 X	11000
11001			{reserved}			11001 10101 III **lhzcix** v2.05 HV X	{reserved} X	{reserved}	11001
11010			11010 10010 III **slbiag** v3.0B P X	11010 10011 III **slbmfev** v2.00 P X		11010 10101 III **lbzcix** v2.05 HV X	11010 10110 II **eieio** PPC X	11010 10111 I **lfiwax** v2.05 X	11010
11011						11011 10101 III **ldcix** v2.05 HV X	11011 10110 III **msgsync** v3.0 HV X	11011 10111 I **lfiwzx** v2.06 X	11011
11100			{reserved}	11100 10011 III **slbmfee** v2.00 P X		11100 10101 III **stwcix** v2.05 HV X	11100 10110 I **sthbrx** P1 X	11100 10111 I **stfdpx** v2.05 X	11100
11101			{reserved}			11101 10101 III **sthcix** v2.05 HV X		{reserved}	11101
11110			{reserved}	11110 10011 III **slbfee.** v2.05 P X		11110 10101 III **stbcix** v2.05 HV X	11110 10110 II **icbi** PPC X	11110 10111 I **stfiwx** PPC X	11110
11111			{reserved}			11111 10101 III **stdcix** v2.05 HV X	11111 10110 II **dcbz** P1 X		11111
	10000	10001	10010	10011	10100	10101	10110	10111	

Table 21: EXT31: Extended Opcode Map for Primary Opcode 31 (opcode bits 21:30) (Sheet 4 of 4)

	11000	11001	11010	11011	11100	11101	11110	11111	
00000	00000 11000 I **slw[.]** P1 X		00000 11010 I **cntlzw[.]** P1 X	00000 11011 I **sld[.]** PPC X	00000 11100 I **and[.]** P1 X	{reserved}	00000 11110 II **wait** v3.0 X		00000
00001			00001 11010 I **cntlzd[.]** PPC X		00001 11100 I **andc[.]** P1 X	{reserved}			00001
00010						{reserved}			00010
00011			00011 11010 I **popcntb** v2.02 X		00011 11100 I **nor[.]** P1 X				00011
00100	{reserved}	P1{reserved}	00100 11010 I **prtyw** v2.05 X						00100
00101	{reserved}		00101 11010 I **prtyd** v2.05 X						00101
00110	{reserved}	P1{reserved}							00110
00111	{reserved}				00111 11100 I **bpermd** v2.06 X				00111
01000			01000 11010 I **cdtbcd** v2.06 X		01000 11100 I **eqv[.]** P1 X				01000
01001			01001 11010 I **cbcdtd** v2.06 X		01001 11100 I **xor[.]** P1 X				01001
01010									01010
01011			01011 11010 I **popcntw** v2.06 X						01011
01100					01100 11100 I **orc[.]** P1 X				01100
01101					01101 11100 I **or[.]** P1 X				01101
01110					01110 11100 I **nand[.]** P1 X				01110
01111			01111 11010 I **popcntd** v2.06 X		01111 11100 I **cmpb** v2.05 X				01111
10000	10000 11000 I **srw[.]** P1 X	{reserved}	10000 11010 I **cnttzw[.]** v3.0 X	10000 11011 I **srd[.]** PPC X		{reserved}			10000
10001			10001 11010 I **cnttzd[.]** v3.0 X						10001
10010									10010
10011									10011
10100	{reserved}	{reserved}							10100
10101	{reserved}								10101
10110	{reserved}	{reserved}							10110
10111	{reserved}								10111
11000	11000 11000 I **sraw[.]** P1 X		11000 11010 I **srad[.]** PPC X						11000
11001	11001 11000 I **srawi[.]** P1 X		11001 1101- I **sradi[.]** PPC XS						11001
11010									11010
11011			11011 1101- I **extswsli[.]** v3.0 XS						11011
11100	{reserved}	{reserved}	11100 11010 I **extsh[.]** P1 X						11100
11101	{reserved}		11101 11010 I **extsb[.]** PPC X						11101
11110			11110 11010 I **extsw[.]** PPC X						11110
11111									11111
	11000	11001	11010	11011	11100	11101	11110	11111	

Table 22: EXT59: Extended Opcode Map for Primary Opcode 59 (opcode bits 21:30) (Sheet 1 of 4)

	00000	00001	00010	00011	00100	00101	00110	00111	
00000			00000 00010 I **dadd[.]** v2.05 X	--000 00011 I **dqua[.]** v2.05 Z23					00000
00001			00001 00010 I **dmul[.]** v2.05 X	--001 00011 I **drrnd[.]** v2.05 Z23					00001
00010			-0010 00010 I **dscli[.]** v2.05 Z22	--010 00011 I **dquai[.]** v2.05 Z23					00010
00011			-0011 00010 I **dscri[.]** v2.05 Z22	--011 00011 I **drintx[.]** v2.05 Z23					00011
00100			00100 00010 I **dcmpo** v2.05 X						00100
00101			00101 00010 I **dtstex** v2.05 X						00101
00110			-0110 00010 I **dtstdc** v2.05 Z22						00110
00111			-0111 00010 I **dtstdg** v2.05 Z22	--111 00011 I **drintn[.]** v2.05 Z23					00111
01000			01000 00010 I **dctdp[.]** v2.05 X	--000 00011 I **dqua[.]** v2.05 Z23					01000
01001			01001 00010 I **dctfix[.]** v2.05 X	--001 00011 I **drrnd[.]** v2.05 Z23					01001
01010			01010 00010 I **ddedpd[.]** v2.05 X	--010 00011 I **dquai[.]** v2.05 Z23					01010
01011			01011 00010 I **dxex[.]** v2.05 X	--011 00011 I **drintx[.]** v2.05 Z23					01011
01100									01100
01101									01101
01110									01110
01111				--111 00011 I **drintn[.]** v2.05 Z23					01111
10000			10000 00010 I **dsub[.]** v2.05 X	--000 00011 I **dqua[.]** v2.05 Z23					10000
10001			10001 00010 I **ddiv[.]** v2.05 X	--001 00011 I **drrnd[.]** v2.05 Z23					10001
10010			-0010 00010 I **dscli[.]** v2.05 Z22	--010 00011 I **dquai[.]** v2.05 Z23					10010
10011			-0011 00010 I **dscri[.]** v2.05 Z22	--011 00011 I **drintx[.]** v2.05 Z23					10011
10100			10100 00010 I **dcmpu** v2.05 X						10100
10101			10101 00010 I **dtstsf** v2.05 X	10101 00011 I **dtstsfi** v3.0 X					10101
10110			-0110 00010 I **dtstdc** v2.05 Z22						10110
10111			-0111 00010 I **dtstdg** v2.05 Z22	--111 00011 I **drintn[.]** v2.05 Z23					10111
11000			11000 00010 I **drsp[.]** v2.05 X	--000 00011 I **dqua[.]** v2.05 Z23					11000
11001			11001 00010 I **dcffix[.]** v2.06 X	--001 00011 I **drrnd[.]** v2.05 Z23					11001
11010			11010 00010 I **denbcd[.]** v2.05 X	--010 00011 I **dquai[.]** v2.05 Z23					11010
11011			11011 00010 I **diex[.]** v2.05 X	--011 00011 I **drintx[.]** v2.05 Z23					11011
11100									11100
11101									11101
11110									11110
11111				--111 00011 I **drintn[.]** v2.05 Z23					11111
	00000	00001	00010	00011	00100	00101	00110	00111	

Table 22: EXT59: Extended Opcode Map for Primary Opcode 59 (opcode bits 21:30) (Sheet 2 of 4)

	01000	01001	01010	01011	01100	01101	01110	01111	
00000									00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010							11010 01110 I **fcfids[.]** v2.06 X		11010
11011									11011
11100									11100
11101									11101
11110							11110 01110 I **fcfidus[.]** v2.06 X		11110
11111									11111
	01000	01001	01010	01011	01100	01101	01110	01111	

Table 22: EXT59: Extended Opcode Map for Primary Opcode 59 (opcode bits 21:30) (Sheet 3 of 4)

	10000	10001	10010	10011	10100	10101	10110	10111	
00000			///// 10010 I **fdivs[.]** PPC A		///// 10100 I **fsubs[.]** PPC A	///// 10101 I **fadds[.]** PPC A	///// 10110 I **fsqrts[.]** PPC A		00000
00001			///// 10010 **fdivs[.]** {invalid}		///// 10100 **fsubs[.]** {invalid}	///// 10101 **fadds[.]** {invalid}	///// 10110 **fsqrts[.]** {invalid}		00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	10000	10001	10010	10011	10100	10101	10110	10111	

Table 22: EXT59: Extended Opcode Map for Primary Opcode 59 (opcode bits 21:30) (Sheet 4 of 4)

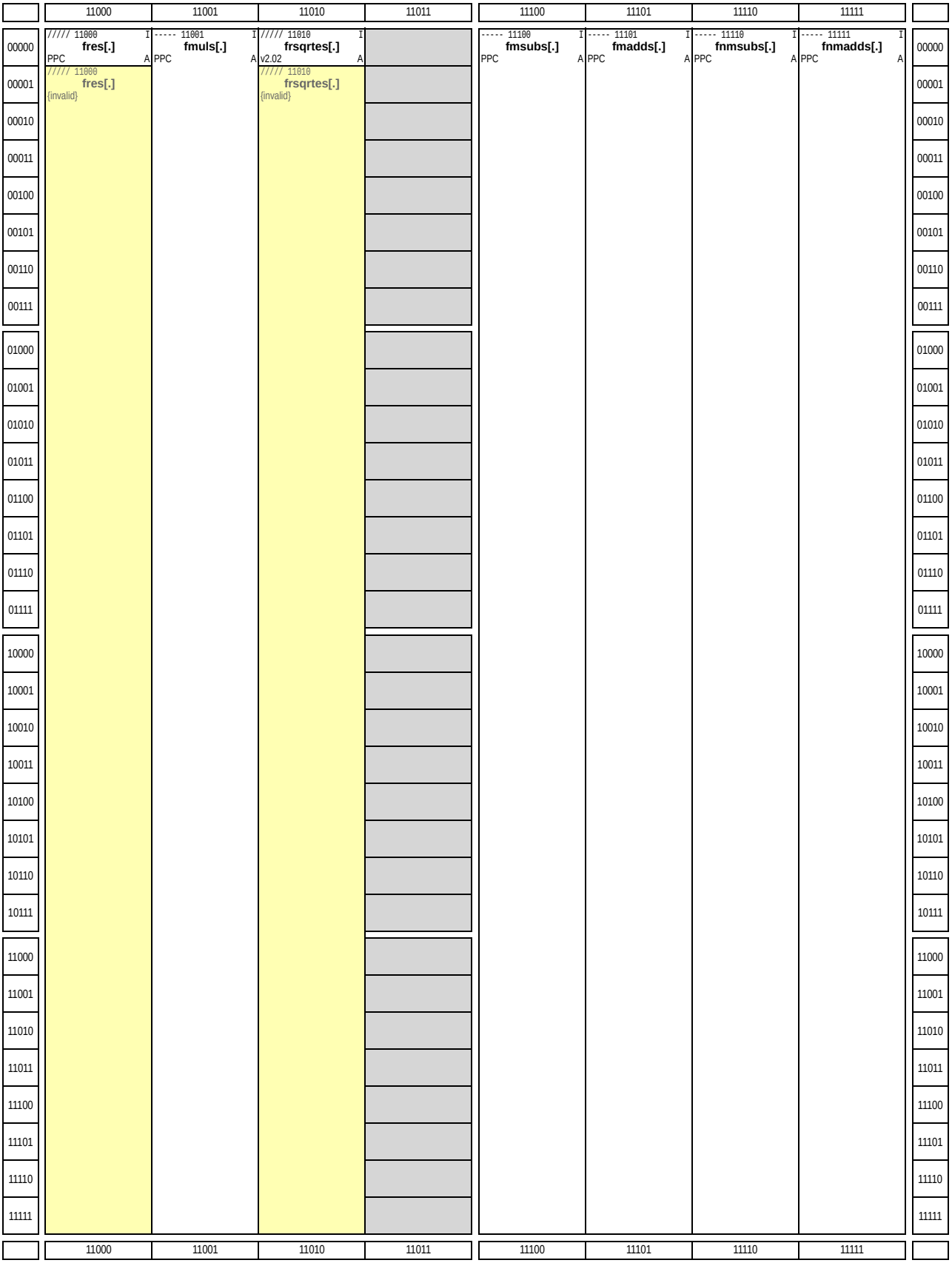


Table 23: EXT60: Extended Opcode Map for Primary Opcode 60 (opcode bits 21:30) (Sheet 1 of 4)

	00000	00001	00010	00011	00100	00101	00110	00111	
00000	00000 000-- I **xsaddsp** v2.07 XX3				00000 001-- I **xsmaddasp** v2.07 XX3				00000
00001	00001 000-- I **xssubsp** v2.07 XX3				00001 001-- I **xsmaddmsp** v2.07 XX3				00001
00010	00010 000-- I **xsmulsp** v2.07 XX3				00010 001-- I **xsmsubasp** v2.07 XX3				00010
00011	00011 000-- I **xsdivsp** v2.07 XX3				00011 001-- I **xsmsubmsp** v2.07 XX3				00011
00100	00100 000-- I **xsadddp** v2.06 XX3				00100 001-- I **xsmaddadp** v2.06 XX3				00100
00101	00101 000-- I **xssubdp** v2.06 XX3				00101 001-- I **xsmaddmdp** v2.06 XX3				00101
00110	00110 000-- I **xsmuldp** v2.06 XX3				00110 001-- I **xsmsubadp** v2.06 XX3				00110
00111	00111 000-- I **xsdivdp** v2.06 XX3				00111 001-- I **xsmsubmdp** v2.06 XX3				00111
01000	01000 000-- I **xvaddsp** v2.06 XX3				01000 001-- I **xvmaddasp** v2.06 XX3				01000
01001	01001 000-- I **xvsubsp** v2.06 XX3				01001 001-- I **xvmaddmsp** v2.06 XX3				01001
01010	01010 000-- I **xvmulsp** v2.06 XX3				01010 001-- I **xvmsubasp** v2.06 XX3				01010
01011	01011 000-- I **xvdivsp** v2.06 XX3				01011 001-- I **xvmsubmsp** v2.06 XX3				01011
01100	01100 000-- I **xvadddp** v2.06 XX3				01100 001-- I **xvmaddadp** v2.06 XX3				01100
01101	01101 000-- I **xvsubdp** v2.06 XX3				01101 001-- I **xvmaddmdp** v2.06 XX3				01101
01110	01110 000-- I **xvmuldp** v2.06 XX3				01110 001-- I **xvmsubadp** v2.06 XX3				01110
01111	01111 000-- I **xvdivdp** v2.06 XX3				01111 001-- I **xvmsubmdp** v2.06 XX3				01111
10000	10000 000-- I **xsmaxcdp** v3.0 XX3				10000 001-- I **xsnmaddasp** v2.07 XX3				10000
10001	10001 000-- I **xsmincdp** v3.0 XX3				10001 001-- I **xsnmaddmsp** v2.07 XX3				10001
10010	10010 000-- I **xsmaxjdp** v3.0 XX3				10010 001-- I **xsnmsubasp** v2.07 XX3				10010
10011	10011 000-- I **xsminjdp** v3.0 XX3				10011 001-- I **xsnmsubmsp** v2.07 XX3				10011
10100	10100 000-- I **xsmaxdp** v2.06 XX3				10100 001-- I **xsnmaddadp** v2.06 XX3				10100
10101	10101 000-- I **xsmindp** v2.06 XX3				10101 001-- I **xsnmaddmdp** v2.06 XX3				10101
10110	10110 000-- I **xscpsgndp** v2.06 XX3				10110 001-- I **xsnmsubadp** v2.06 XX3				10110
10111					10111 001-- I **xsnmsubmdp** v2.06 XX3				10111
11000	11000 000-- I **xvmaxsp** v2.06 XX3				11000 001-- I **xvnmaddasp** v2.06 XX3				11000
11001	11001 000-- I **xvminsp** v2.06 XX3				11001 001-- I **xvnmaddmsp** v2.06 XX3				11001
11010	11010 000-- I **xvcpsgnsp** v2.06 XX3				11010 001-- I **xvnmsubasp** v2.06 XX3				11010
11011	11011 000-- I **xviexpsp** v3.0 XX3				11011 001-- I **xvnmsubmsp** v2.06 XX3				11011
11100	11100 000-- I **xvmaxdp** v2.06 XX3				11100 001-- I **xvnmaddadp** v2.06 XX3				11100
11101	11101 000-- I **xvmindp** v2.06 XX3				11101 001-- I **xvnmaddmdp** v2.06 XX3				11101
11110	11110 000-- I **xvcpsgndp** v2.06 XX3				11110 001-- I **xvnmsubadp** v2.06 XX3				11110
11111	11111 000-- I **xviexpdp** v3.0 XX3				11111 001-- I **xvnmsubmdp** v2.06 XX3				11111
	00000	00001	00010	00011	00100	00101	00110	00111	

Table 23: EXT60: Extended Opcode Map for Primary Opcode 60 (opcode bits 21:30) (Sheet 2 of 4)

	01000	01001	01010	01011	01100	01101	01110	01111	
00000	0--00 010-- I **xxsldwi** v2.06 XX3				00000 011-- I **xscmpeqdp** v3.0 XX3				00000
00001	0--01 010-- I **xxpermdi** v2.06 XX3				00001 011-- I **xscmpgtdp** v3.0 XX3				00001
00010	00010 010-- I **xxmrghw** v2.06 XX3				00010 011-- I **xscmpgedp** v3.0 XX3				00010
00011	00011 010-- I **xxperm** v3.0 XX3								00011
00100	0--00 010-- I **xxsldwi** v2.06 XX3				00100 011-- I **xscmpudp** v2.06 XX3				00100
00101	0--01 010-- I **xxpermdi** v2.06 XX3				00101 011-- I **xscmpodp** v2.06 XX3				00101
00110	00110 010-- I **xxmrglw** v2.06 XX3								00110
00111	00111 010-- I **xxpermr** v3.0 XX3				00111 011-- I **xscmpexpdp** v3.0 XX3				00111
01000	0--00 010-- I **xxsldwi** v2.06 XX3				01000 011-- I **xvcmpeqsp** v2.06 XX3				01000
01001	0--01 010-- I **xxpermdi** v2.06 XX3				01001 011-- I **xvcmpgtsp** v2.06 XX3				01001
01010	01010 0100- I **xxspltw** v2.06 XX2		01010 0101- I **xxextractuw** v3.0 XX2		01010 011-- I **xvcmpgesp** v2.06 XX3				01010
01011	01011 01000 **XPND60-1** {expanded}		01011 0101- I **xxinsertw** v3.0 XX2						01011
01100	0--00 010-- I **xxsldwi** v2.06 XX3				01100 011-- I **xvcmpeqdp** v2.06 XX3				01100
01101	0--01 010-- I **xxpermdi** v2.06 XX3				01101 011-- I **xvcmpgtdp** v2.06 XX3				01101
01110					01110 011-- I **xvcmpgedp** v2.06 XX3				01110
01111									01111
10000	10000 010-- I **xxland** v2.06 XX3								10000
10001	10001 010-- I **xxlandc** v2.06 XX3								10001
10010	10010 010-- I **xxlor** v2.06 XX3								10010
10011	10011 010-- I **xxlxor** v2.06 XX3								10011
10100	10100 010-- I **xxlnor** v2.06 XX3								10100
10101	10101 010-- I **xxlorc** v2.07 XX3								10101
10110	10110 010-- I **xxlnand** v2.07 XX3								10110
10111	10111 010-- I **xxleqv** v2.07 XX3								10111
11000					11000 011-- I **xvcmpeqsp.** v2.06 XX3				11000
11001					11001 011-- I **xvcmpgtsp.** v2.06 XX3				11001
11010					11010 011-- I **xvcmpgesp.** v2.06 XX3				11010
11011									11011
11100					11100 011-- I **xvcmpeqdp.** v2.06 XX3				11100
11101					11101 011-- I **xvcmpgtdp.** v2.06 XX3				11101
11110					11110 011-- I **xvcmpgedp.** v2.06 XX3				11110
11111									11111
	01000	01001	01010	01011	01100	01101	01110	01111	

Table 23: EXT60: Extended Opcode Map for Primary Opcode 60 (opcode bits 21:30) (Sheet 3 of 4)

	10000	10001	10010	10011	10100	10101	10110	10111	
00000					00000 1010- I **xsrsqrtesp** v2.07 XX2		00000 1011- I **xssqrtsp** v2.07 XX2		00000
00001					00001 1010- I **xsresp** v2.07 XX2				00001
00010									00010
00011									00011
00100	00100 1000- I **xscvdpuxws** v2.06 XX2		00100 1001- I **xsrdpi** v2.06 XX2		00100 1010- I **xsrsqrtedp** v2.06 XX2		00100 1011- I **xssqrtdp** v2.06 XX2		00100
00101	00101 1000- I **xscvdpsxws** v2.06 XX2		00101 1001- I **xsrdpiz** v2.06 XX2		00101 1010- I **xsredp** v2.06 XX2				00101
00110			00110 1001- I **xsrdpip** v2.06 XX2		00110 1010- I **xstsqrtdp** v2.06 XX2		00110 1011- I **xsrdpic** v2.06 XX2		00110
00111			00111 1001- I **xsrdpim** v2.06 XX2		00111 101-- I **xstdivdp** v2.06 XX3				00111
01000	01000 1000- I **xvcvspuxws** v2.06 XX2		01000 1001- I **xvrspi** v2.06 XX2		01000 1010- I **xvrsqrtesp** v2.06 XX2		01000 1011- I **xvsqrtsp** v2.06 XX2		01000
01001	01001 1000- I **xvcvspsxws** v2.06 XX2		01001 1001- I **xvrspiz** v2.06 XX2		01001 1010- I **xvresp** v2.06 XX2				01001
01010	01010 1000- I **xvcvuxwsp** v2.06 XX2		01010 1001- I **xvrspip** v2.06 XX2		01010 1010- I **xvtsqrtsp** v2.06 XX2		01010 1011- I **xvrspic** v2.06 XX2		01010
01011	01011 1000- I **xvcvsxwsp** v2.06 XX2		01011 1001- I **xvrspim** v2.06 XX2		01011 101-- I **xvtdivsp** v2.06 XX3				01011
01100	01100 1000- I **xvcvdpuxws** v2.06 XX2		01100 1001- I **xvrdpi** v2.06 XX2		01100 1010- I **xvrsqrtedp** v2.06 XX2		01100 1011- I **xvsqrtdp** v2.06 XX2		01100
01101	01101 1000- I **xvcvdpsxws** v2.06 XX2		01101 1001- I **xvrdpiz** v2.06 XX2		01101 1010- I **xvredp** v2.06 XX2				01101
01110	01110 1000- I **xvcvuxwdp** v2.06 XX2		01110 1001- I **xvrdpip** v2.06 XX2		01110 1010- I **xvtsqrtdp** v2.06 XX2		01110 1011- I **xvrdpic** v2.06 XX2		01110
01111	01111 1000- I **xvcvsxwdp** v2.06 XX2		01111 1001- I **xvrdpim** v2.06 XX2		01111 101-- I **xvtdivdp** v2.06 XX3				01111
10000			10000 1001- I **xscvdpsp** v2.06 XX2				10000 1011- I **xscvdpspn** v2.07 XX2		10000
10001			10001 1001- I **xsrsp** v2.07 XX2						10001
10010	10010 1000- I **xscvuxdsp** v2.07 XX2				10010 1010- I **xststdcsp** v3.0 XX2				10010
10011	10011 1000- I **xscvsxdsp** v2.07 XX2								10011
10100	10100 1000- I **xscvdpuxds** v2.06 XX2		10100 1001- I **xscvspdp** v2.06 XX2				10100 1011- I **xscvspdpn** v2.07 XX2		10100
10101	10101 1000- I **xscvdpsxds** v2.06 XX2		10101 1001- I **xsabsdp** v2.06 XX2				10101 1011- **XPND60-2** {expanded}		10101
10110	10110 1000- I **xscvuxddp** v2.06 XX2		10110 1001- I **xsnabsdp** v2.06 XX2		10110 1010- I **xststdcdp** v3.0 XX2				10110
10111	10111 1000- I **xscvsxddp** v2.06 XX2		10111 1001- I **xsnegdp** v2.06 XX2						10111
11000	11000 1000- I **xvcvspuxds** v2.06 XX2		11000 1001- I **xvcvdpsp** v2.06 XX2						11000
11001	11001 1000- I **xvcvspsxds** v2.06 XX2		11001 1001- I **xvabssp** v2.06 XX2						11001
11010	11010 1000- I **xvcvuxdsp** v2.06 XX2		11010 1001- I **xvnabssp** v2.06 XX2		1101- 101-- I **xvtstdcsp** v3.0 XX2				11010
11011	11011 1000- I **xvcvsxdsp** v2.06 XX2		11011 1001- I **xvnegsp** v2.06 XX2		1101- 101-- I **xvtstdcsp** v3.0 XX2				11011
11100	11100 1000- I **xvcvdpuxds** v2.06 XX2		11100 1001- I **xvcvspdp** v2.06 XX2				11100 10110 I **xsiexpdp** v3.0 XX1		11100
11101	11101 1000- I **xvcvdpsxds** v2.06 XX2		11101 1001- I **xvabsdp** v2.06 XX2				11101 1011- **XPND60-3** {expanded}		11101
11110	11110 1000- I **xvcvuxddp** v2.06 XX2		11110 1001- I **xvnabsdp** v2.06 XX2		1111- 101-- I **xvtstdcdp** v3.0 XX2				11110
11111	11111 1000- I **xvcvsxddp** v2.06 XX2		11111 1001- I **xvnegdp** v2.06 XX2		1111- 101-- I **xvtstdcdp** v3.0 XX2				11111
	10000	10001	10010	10011	10100	10101	10110	10111	

Table 23: EXT60: Extended Opcode Map for Primary Opcode 60 (opcode bits 21:30) (Sheet 4 of 4)

	11000	11001	11010	11011	11100	11101	11110	11111	
00000	----- 11--- **xxsel** v2.06	I XX4							00000
00001									00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	11000	11001	11010	11011	11100	11101	11110	11111	

Table 24: XPND60-1: Extended Opcode Map for PO=60 XO=0b01011_01000 (opcode bits 11:15)

	000	001	010	011	100	101	110	111	
00	00 --- I **xxspltib** v3.0 XX1								00
01									01
10									10
11									11
	000	001	010	011	100	101	110	111	

Table 25: XPND60-2: Extended Opcode Map for PO=60 XO=0b10101_1011- (opcode bits 11:15)

	000	001	010	011	100	101	110	111	
00	00 000 I **xsxexpdp** v3.0 XX2	00 001 I **xsxsigdp** v3.0 XX2							00
01									01
10	10 000 I **xscvhpdp** v3.0 XX2	10 001 I **xscvdphp** v3.0 XX2							10
11									11
	000	001	010	011	100	101	110	111	

Table 26: XPND60-3: Extended Opcode Map for PO=60 XO=0b11101_1011- (opcode bits 11:15)

	000	001	010	011	100	101	110	111	
00	00 000 I **xvxexpdp** v3.0 XX2	00 001 I **xvxsigdp** v3.0 XX2						00 111 I **xxbrh** v3.0 XX2	00
01	01 000 I **xvxexpsp** v3.0 XX2	01 001 I **xvxsigsp** v3.0 XX2						01 111 I **xxbrw** v3.0 XX2	01
10								10 111 I **xxbrd** v3.0 XX2	10
11	11 000 I **xvcvhpsp** v3.0 XX2	11 001 I **xvcvsphp** v3.0 XX2						11 111 I **xxbrq** v3.0 XX2	11
	000	001	010	011	100	101	110	111	

Table 27: EXT63: Extended Opcode Map for Primary Opcode 63 (opcode bits 21:30) (Sheet 1 of 4)

	00000	00001	00010	00011	00100	00101	00110	00111	
00000	00000 00000 I **fcmpu** P1 X		00000 00010I **daddq[.]** v2.05 X	--000 00011I **dquaq[.]** v2.05 Z23	00000 00100I **xsaddqp[o]** v3.0 X	--000 00101I **xsrqpi[x]** v3.0 X			00000
00001	00001 00000 I **fcmpo** P1 X		00001 00010I **dmulq[.]** v2.05 X	--001 00011I **drrndq[.]** v2.05 Z23	00001 00100I **xsmulqp[o]** v3.0 X	--001 00101I **xsrqpxp** v3.0 X	00001 00110I **mtfsb1[.]** P1 X		00001
00010	00010 00000 I **mcrfs** P1 X		-0010 00010I **dscliq[.]** v2.05 Z22	--010 00011I **dquaiq[.]** v2.05 Z23			00010 00110I **mtfsb0[.]** P1 X		00010
00011			-0011 00010I **dscriq[.]** v2.05 Z22	--011 00011I **drintxq[.]** v2.05 Z23	00011 00100I **xscpsgnqp** v3.0 X				00011
00100	00100 00000 I **ftdiv** v2.06 X		00100 00010I **dcmpoq** v2.05 X		00100 00100I **xscmpoqp** v3.0 X		00100 00110I **mtfsfi[.]** P1 X		00100
00101	00101 00000 I **ftsqrt** v2.06 X		00101 00010I **dtstexq** v2.05 X		00101 00100I **xscmpexpqp** v3.0 X				00101
00110			-0110 00010I **dtstdcq** v2.05 Z22						00110
00111			-0111 00010I **dtstdgq** v2.05 Z22	--111 00011I **drintnq[.]** v2.05 Z23					00111
01000			01000 00010I **dctqpq[.]** v2.05 X	--000 00011I **dquaq[.]** v2.05 Z23		--000 00101I **xsrqpi[x]** v3.0 X			01000
01001			01001 00010I **dctfixq[.]** v2.05 X	--001 00011I **drrndq[.]** v2.05 Z23		--001 00101I **xsrqpxp** v3.0 X			01001
01010			01010 00010I **ddedpdq[.]** v2.05 X	--010 00011I **dquaiq[.]** v2.05 Z23					01010
01011			01011 00010I **dxexq[.]** v2.05 X	--011 00011I **drintxq[.]** v2.05 Z23					01011
01100					01100 00100I **xsmaddqp[o]** v3.0 X				01100
01101					01101 00100I **xsmsubqp[o]** v3.0 X				01101
01110					01110 00100I **xsnmaddqp[o]** v3.0 X				01110
01111				--111 00011I **drintnq[.]** v2.05 Z23	01111 00100I **xsnmsubqp[o]** v3.0 X				01111
10000			10000 00010I **dsubq[.]** v2.05 X	--000 00011I **dquaq[.]** v2.05 Z23	10000 00100I **xssubqp[o]** v3.0 X	--000 00101I **xsrqpi[x]** v3.0 X			10000
10001			10001 00010I **ddivq[.]** v2.05 X	--001 00011I **drrndq[.]** v2.05 Z23	10001 00100I **xsdivqp[o]** v3.0 X	--001 00101I **xsrqpxp** v3.0 X			10001
10010			-0010 00010I **dscliq[.]** v2.05 Z22	--010 00011I **dquaiq[.]** v2.05 Z23				10010 00111 **XPND63-1** {expanded}	10010
10011			-0011 00010I **dscriq[.]** v2.05 Z22	--011 00011I **drintxq[.]** v2.05 Z23					10011
10100			10100 00010I **dcmpuq** v2.05 X		10100 00100I **xscmpuqp** v3.0 X				10100
10101			10101 00010I **dtstsfq** v2.05 X	10101 00011I **dtstsfiq** v3.0 X					10101
10110			-0110 00010I **dtstdcq** v2.05 Z22		10110 00100I **xststdcqp** v3.0 X			10110 00111 I **mtfsf[.]** P1 XFL	10110
10111			-0111 00010I **dtstdgq** v2.05 Z22	--111 00011I **drintnq[.]** v2.05 Z23					10111
11000			11000 00010I **drdpq[.]** v2.05 X	--000 00011I **dquaq[.]** v2.05 Z23		--000 00101I **xsrqpi[x]** v3.0 X			11000
11001			11001 00010I **dcffixq[.]** v2.05 X	--001 00011I **drrndq[.]** v2.05 Z23	11001 00100 **XPND63-2** {expanded}	--001 00101I **xsrqpxp** v3.0 X			11001
11010			11010 00010I **denbcdq[.]** v2.05 X	--010 00011I **dquaiq[.]** v2.05 Z23	11010 00100 **XPND63-3** {expanded}		11010 00110I **fmrgow** v2.07 X		11010
11011			11011 00010I **diexq[.]** v2.05 X	--011 00011I **drintxq[.]** v2.05 Z23	11011 00100I **xsiexpqp** v3.0 X				11011
11100									11100
11101									11101
11110							11110 00110I **fmrgew** v2.07 X		11110
11111				--111 00011I **drintnq[.]** v2.05 Z23					11111
	00000	00001	00010	00011	00100	00101	00110	00111	

Table 27: EXT63: Extended Opcode Map for Primary Opcode 63 (opcode bits 21:30) (Sheet 2 of 4)

	01000	01001	01010	01011	01100	01101	01110	01111	
00000	00000 01000 I **fcpsgn[.]** v2.05 X				00000 01100I **frsp[.]** P1 X		00000 01110I **fctiw[.]** P2 X	00000 01111 I **fctiwz[.]** P2 X	00000
00001	00001 01000 I **fneg[.]** P1 X								00001
00010	00010 01000 I **fmr[.]** P1 X								00010
00011									00011
00100	00100 01000 I **fnabs[.]** P1 X						00100 01110I **fctiwu[.]** v2.06 X	00100 01111 I **fctiwuz[.]** v2.06 X	00100
00101									00101
00110									00110
00111									00111
01000	01000 01000 I **fabs[.]** P1 X								01000
01001									01001
01010									01010
01011									01011
01100	01100 01000 I **frin[.]** v2.02 X								01100
01101	01101 01000 I **friz[.]** v2.02 X								01101
01110	01110 01000 I **frip[.]** v2.02 X								01110
01111	01111 01000 I **frim[.]** v2.02 X								01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001							11001 01110I **fctid[.]** PPC X	11001 01111 I **fctidz[.]** PPC X	11001
11010							11010 01110I **fcfid[.]** PPC X		11010
11011									11011
11100									11100
11101							11101 01110I **fctidu[.]** v2.06 X	11101 01111 I **fctiduz[.]** v2.06 X	11101
11110							11110 01110I **fcfidu[.]** v2.06 X		11110
11111									11111
	01000	01001	01010	01011	01100	01101	01110	01111	

Table 27: EXT63: Extended Opcode Map for Primary Opcode 63 (opcode bits 21:30) (Sheet 3 of 4)

	10000	10001	10010	10011	10100	10101	10110	10111	
00000			///// 10010I **fdiv[.]** P1 A		///// 10100I **fsub[.]** P1 A	///// 10101I **fadd[.]** P1 A	///// 10110I **fsqrt[.]** P2 A	----- 10111 I **fsel[.]** PPC A	00000
00001			///// 10010 **fdiv[.]** {invalid}		///// 10100 **fsub[.]** {invalid}	///// 10101 **fadd[.]** {invalid}	///// 10110 **fsqrt[.]** {invalid}		00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	10000	10001	10010	10011	10100	10101	10110	10111	

Table 27: EXT63: Extended Opcode Map for Primary Opcode 63 (opcode bits 21:30) (Sheet 4 of 4)

	11000	11001	11010	11011	11100	11101	11110	11111	
00000	///// 11000 I **fre[.]** v2.02 A	----- 11001 I **fmul[.]** P1 A	///// 11010I **frsqrte[.]** PPC A		----- 11100I **fmsub[.]** P1 A	----- 11101I **fmadd[.]** P1 A	----- 11110I **fnmsub[.]** P1 A	----- 11111 I **fnmadd[.]** P1 A	00000
00001	///// 11000 **fre[.]** {invalid}		///// 11010 **frsqrte[.]** {invalid}						00001
00010									00010
00011									00011
00100									00100
00101									00101
00110									00110
00111									00111
01000									01000
01001									01001
01010									01010
01011									01011
01100									01100
01101									01101
01110									01110
01111									01111
10000									10000
10001									10001
10010									10010
10011									10011
10100									10100
10101									10101
10110									10110
10111									10111
11000									11000
11001									11001
11010									11010
11011									11011
11100									11100
11101									11101
11110									11110
11111									11111
	11000	11001	11010	11011	11100	11101	11110	11111	

Table 28: XPND63-1: Extended Opcode Map for PO=63 XO=0b10010_00111 (opcode bits 11:15)

	000	001	010	011	100	101	110	111	
00	00 000 I **mffs[.]** P1 X	00 001 I **mffsce** v3.0B X							00
01									01
10					10 100 I **mffscdrn** v3.0B X	10 101 I **mffscdrni** v3.0B X	10 110 I **mffscrn** v3.0B X	10 111 I **mffscrni** v3.0B X	10
11	11 000 I **mffsl** v3.0B X								11
	000	001	010	011	100	101	110	111	

Table 29: XPND63-2: Extended Opcode Map for PO=63 XO=0b11001_00100 (opcode bits 11:15)

	000	001	010	011	100	101	110	111	
00	00 000 I **xsabsqp** v3.0 X		00 010 I **xsxexpqp** v3.0 X						00
01	01 000 I **xsnabsqp** v3.0 X								01
10	10 000 I **xsnegqp** v3.0 X		10 010 I **xsxsigqp** v3.0 X						10
11				11 011 I **xssqrtqp[o]** v3.0 X					11
	000	001	010	011	100	101	110	111	

Table 30: XPND63-3: Extended Opcode Map for PO=63 XO=0b11010_00100 (opcode bits 11:15)

	000	001	010	011	100	101	110	111	
00		00 001 I **xscvqpuwz** v3.0 X	00 010 I **xscvudqp** v3.0 X						00
01		01 001 I **xscvqpswz** v3.0 X	01 010 I **xscvsdqp** v3.0 X						01
10		10 001 I **xscvqpudz** v3.0 X			10 100 I **xscvqpdp[o]** v3.0 X		10 110 I **xscvdpqp** v3.0 X		10
11		11 001 I **xscvqpsdz** v3.0 X							11
	000	001	010	011	100	101	110	111	

Appendix D. Power ISA Instruction Set Sorted by Opcode

This appendix lists all the instructions in the Power ISA, sorted by primary opcode, then by extended opcode bits 26:31 (if any), then by opcode bits 21:25 (if any), then by expanded opcode bits 11:15 (if any).

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
000010	D	I	91	tdi	PPC			Trap Doubleword Immediate
000011	D	I	90	twi	P1			Trap Word Immediate
000100 .... 00000 000000	VX	I	270	vaddubm	v2.03			Vector Add Unsigned Byte Modulo
000100 .... 00001 000000	VX	I	271	vadduhm	v2.03			Vector Add Unsigned Halfword Modulo
000100 .... 00010 000000	VX	I	271	vadduwm	v2.03			Vector Add Unsigned Word Modulo
000100 .... 00011 000000	VX	I	270	vaddudm	v2.07			Vector Add Unsigned Doubleword Modulo
000100 .... 00100 000000	VX	I	270	vadduqm	v2.07			Vector Add Unsigned Quadword Modulo
000100 .... 00101 000000	VX	I	273	vaddcuq	v2.07			Vector Add & write Carry Unsigned Quadword
000100 .... 00110 000000	VX	I	269	vaddcuw	v2.03			Vector Add & Write Carry-Out Unsigned Word
000100 .... 01000 000000	VX	I	272	vaddubs	v2.03			Vector Add Unsigned Byte Saturate
000100 .... 01001 000000	VX	I	272	vadduhs	v2.03			Vector Add Unsigned Halfword Saturate
000100 .... 01010 000000	VX	I	272	vadduws	v2.03			Vector Add Unsigned Word Saturate
000100 .... 01100 000000	VX	I	269	vaddsbs	v2.03			Vector Add Signed Byte Saturate
000100 .... 01101 000000	VX	I	269	vaddshs	v2.03			Vector Add Signed Halfword Saturate
000100 .... 01110 000000	VX	I	270	vaddsws	v2.03			Vector Add Signed Word Saturate
000100 .... 10000 000000	VX	I	277	vsububm	v2.03			Vector Subtract Unsigned Byte Modulo
000100 .... 10001 000000	VX	I	277	vsubuhm	v2.03			Vector Subtract Unsigned Halfword Modulo
000100 .... 10010 000000	VX	I	277	vsubuwm	v2.03			Vector Subtract Unsigned Word Modulo
000100 .... 10011 000000	VX	I	277	vsubudm	v2.07			Vector Subtract Unsigned Doubleword Modulo
000100 .... 10100 000000	VX	I	279	vsubuqm	v2.07			Vector Subtract Unsigned Quadword Modulo
000100 .... 10101 000000	VX	I	279	vsubcuq	v2.07			Vector Subtract & write Carry Unsigned Quadword
000100 .... 10110 000000	VX	I	275	vsubcuw	v2.03			Vector Subtract & Write Carry-Out Unsigned Word
000100 .... 11000 000000	VX	I	278	vsububs	v2.03			Vector Subtract Unsigned Byte Saturate
000100 .... 11001 000000	VX	I	278	vsubuhs	v2.03			Vector Subtract Unsigned Halfword Saturate
000100 .... 11010 000000	VX	I	278	vsubuws	v2.03			Vector Subtract Unsigned Word Saturate
000100 .... 11100 000000	VX	I	275	vsubsbs	v2.03			Vector Subtract Signed Byte Saturate
000100 .... 11101 000000	VX	I	275	vsubshs	v2.03			Vector Subtract Signed Halfword Saturate
000100 .... 11110 000000	VX	I	276	vsubsws	v2.03			Vector Subtract Signed Word Saturate
000100 .... ///// 00000 000001	VX	I	355	vmul10cuq	v3.0			Vector Multiply-by-10 & write Carry Unsigned Quadword
000100 .... 00001 000001	VX	I	355	vmul10ecuq	v3.0			Vector Multiply-by-10 Extended & write Carry Unsigned Quadword
000100 .... ///// 01000 000001	VX	I	355	vmul10uq	v3.0			Vector Multiply-by-10 Unsigned Quadword
000100 .... 01001 000001	VX	I	355	vmul10euq	v3.0			Vector Multiply-by-10 Extended Unsigned Quadword
000100 .... 01101 000001	VX	I	356	bcdcpsgn.	v3.0			Decimal CopySign & record
000100 .... 1.000 000001	VX	I	348	bcdadd.	v2.07			Decimal Add Modulo & record
000100 .... 1.001 000001	VX	I	348	bcdsub.	v2.07			Decimal Subtract Modulo & record
000100 .... 1/010 000001	VX	I	358	bcdus.	v3.0			Decimal Unsigned Shift & record
000100 .... 1.011 000001	VX	I	357	bcds.	v3.0			Decimal Shift & record
000100 .... 1.100 000001	VX	I	360	bcdtrunc.	v3.0			Decimal Truncate & record
000100 .... 1/101 000001	VX	I	361	bcdutrunc.	v3.0			Decimal Unsigned Truncate & record

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 1 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
000100 ..... 00000 ..... 1/110 000001	VX	I	354	bcdctsq.	v3.0			Decimal Convert To Signed Quadword & record
000100 ..... 00010 ..... 1.110 000001	VX	I	354	bcdcfsq.	v3.0			Decimal Convert From Signed Quadword & record
000100 ..... 00100 ..... 1.110 000001	VX	I	353	bcdctz.	v3.0			Decimal Convert To Zoned & record
000100 ..... 00101 ..... 1/110 000001	VX	I	352	bcdctn.	v3.0			Decimal Convert To National & record
000100 ..... 00110 ..... 1.110 000001	VX	I	351	bcdcfz.	v3.0			Decimal Convert From Zoned & record
000100 ..... 00111 ..... 1.110 000001	VX	I	350	bcdcfn.	v3.0			Decimal Convert From National & record
000100 ..... 11111 ..... 1.110 000001	VX	I	356	bcdsetsgn.	v3.0			Decimal Set Sign & record
000100 .... 1.111 000001	VX	I	359	bcdsr.	v3.0			Decimal Shift & Round & record
000100 .... 00000 000010	VX	I	299	vmaxub	v2.03			Vector Maximum Unsigned Byte
000100 .... 00001 000010	VX	I	300	vmaxuh	v2.03			Vector Maximum Unsigned Halfword
000100 .... 00010 000010	VX	I	300	vmaxuw	v2.03			Vector Maximum Unsigned Word
000100 .... 00011 000010	VX	I	299	vmaxud	v2.07			Vector Maximum Unsigned Doubleword
000100 .... 00100 000010	VX	I	299	vmaxsb	v2.03			Vector Maximum Signed Byte
000100 .... 00101 000010	VX	I	300	vmaxsh	v2.03			Vector Maximum Signed Halfword
000100 .... 00110 000010	VX	I	300	vmaxsw	v2.03			Vector Maximum Signed Word
000100 .... 00111 000010	VX	I	299	vmaxsd	v2.07			Vector Maximum Signed Doubleword
000100 .... 01000 000010	VX	I	301	vminub	v2.03			Vector Minimum Unsigned Byte
000100 .... 01001 000010	VX	I	302	vminuh	v2.03			Vector Minimum Unsigned Halfword
000100 .... 01010 000010	VX	I	302	vminuw	v2.03			Vector Minimum Unsigned Word
000100 .... 01011 000010	VX	I	301	vminud	v2.07			Vector Minimum Unsigned Doubleword
000100 .... 01100 000010	VX	I	301	vminsb	v2.03			Vector Minimum Signed Byte
000100 .... 01101 000010	VX	I	302	vminsh	v2.03			Vector Minimum Signed Halfword
000100 .... 01110 000010	VX	I	302	vminsw	v2.03			Vector Minimum Signed Word
000100 .... 01111 000010	VX	I	301	vminsd	v2.07			Vector Minimum Signed Doubleword
000100 .... 10000 000010	VX	I	296	vavgub	v2.03			Vector Average Unsigned Byte
000100 .... 10001 000010	VX	I	296	vavguh	v2.03			Vector Average Unsigned Halfword
000100 .... 10010 000010	VX	I	296	vavguw	v2.03			Vector Average Unsigned Word
000100 .... 10100 000010	VX	I	295	vavgsb	v2.03			Vector Average Signed Byte
000100 .... 10101 000010	VX	I	295	vavgsh	v2.03			Vector Average Signed Halfword
000100 .... 10110 000010	VX	I	295	vavgsw	v2.03			Vector Average Signed Word
000100 ..... 00000 ..... 11000 000010	VX	I	342	vclzlsbb	v3.0			Vector Count Leading Zero Least-Significant Bits Byte
000100 ..... 00001 ..... 11000 000010	VX	I	342	vctzlsbb	v3.0			Vector Count Trailing Zero Least-Significant Bits Byte
000100 ..... 00110 ..... 11000 000010	VX	I	293	vnegw	v3.0			Vector Negate Word
000100 ..... 00111 ..... 11000 000010	VX	I	293	vnegd	v3.0			Vector Negate Doubleword
000100 ..... 01000 ..... 11000 000010	VX	I	314	vprtybw	v3.0			Vector Parity Byte Word
000100 ..... 01001 ..... 11000 000010	VX	I	314	vprtybd	v3.0			Vector Parity Byte Doubleword
000100 ..... 01010 ..... 11000 000010	VX	I	314	vprtybq	v3.0			Vector Parity Byte Quadword
000100 ..... 10000 ..... 11000 000010	VX	I	294	vextsb2w	v3.0			Vector Extend Sign Byte to Word
000100 ..... 10001 ..... 11000 000010	VX	I	294	vextsh2w	v3.0			Vector Extend Sign Halfword to Word
000100 ..... 11000 ..... 11000 000010	VX	I	294	vextsb2d	v3.0			Vector Extend Sign Byte to Doubleword
000100 ..... 11001 ..... 11000 000010	VX	I	294	vextsh2d	v3.0			Vector Extend Sign Halfword to Doubleword
000100 ..... 11010 ..... 11000 000010	VX	I	294	vextsw2d	v3.0			Vector Extend Sign Word to Doubleword
000100 ..... 11100 ..... 11000 000010	VX	I	341	vctzb	v3.0			Vector Count Trailing Zeros Byte
000100 ..... 11101 ..... 11000 000010	VX	I	341	vctzh	v3.0			Vector Count Trailing Zeros Halfword
000100 ..... 11110 ..... 11000 000010	VX	I	341	vctzw	v3.0			Vector Count Trailing Zeros Word
000100 ..... 11111 ..... 11000 000010	VX	I	341	vctzd	v3.0			Vector Count Trailing Zeros Doubleword
000100 .... 11010 000010	VX	I	335	vshasigmaw	v2.07			Vector SHA-256 Sigma Word
000100 .... 11011 000010	VX	I	335	vshasigmad	v2.07			Vector SHA-512 Sigma Doubleword
000100 ..... ///// 11100 000010	VX	I	340	vclzb	v2.07			Vector Count Leading Zeros Byte
000100 ..... ///// 11101 000010	VX	I	340	vclzh	v2.07			Vector Count Leading Zeros Halfword
000100 ..... ///// 11110 000010	VX	I	340	vclzw	v2.07			Vector Count Leading Zeros Word
000100 ..... ///// 11111 000010	VX	I	340	vclzd	v2.07			Vector Count Leading Zeros Doubleword
000100 .... 10000 000011	VX	I	297	vabsdub	v3.0			Vector Absolute Difference Unsigned Byte
000100 .... 10001 000011	VX	I	297	vabsduh	v3.0			Vector Absolute Difference Unsigned Halfword
000100 .... 10010 000011	VX	I	298	vabsduw	v3.0			Vector Absolute Difference Unsigned Word
000100 ..... ///// 11100 000011	VX	I	345	vpopcntb	v2.07			Vector Population Count Byte

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 2 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
000100 ..... ///// 11101 000011	VX	I	345	vpopcnth	v2.07			Vector Population Count Halfword
000100 ..... ///// 11110 000011	VX	I	345	vpopcntw	v2.07			Vector Population Count Word
000100 ..... ///// 11111 000011	VX	I	345	vpopcntd	v2.07			Vector Population Count Doubleword
000100 .... 00000 000100	VX	I	315	vrlb	v2.03			Vector Rotate Left Byte
000100 .... 00001 000100	VX	I	315	vrlh	v2.03			Vector Rotate Left Halfword
000100 .... 00010 000100	VX	I	315	vrlw	v2.03			Vector Rotate Left Word
000100 .... 00011 000100	VX	I	315	vrld	v2.07			Vector Rotate Left Doubleword
000100 .... 00100 000100	VX	I	316	vslb	v2.03			Vector Shift Left Byte
000100 .... 00101 000100	VX	I	316	vslh	v2.03			Vector Shift Left Halfword
000100 .... 00110 000100	VX	I	316	vslw	v2.03			Vector Shift Left Word
000100 .... 00111 000100	VX	I	264	vsl	v2.03			Vector Shift Left
000100 .... 01000 000100	VX	I	317	vsrb	v2.03			Vector Shift Right Byte
000100 .... 01001 000100	VX	I	317	vsrh	v2.03			Vector Shift Right Halfword
000100 .... 01010 000100	VX	I	317	vsrw	v2.03			Vector Shift Right Word
000100 .... 01011 000100	VX	I	264	vsr	v2.03			Vector Shift Right
000100 .... 01100 000100	VX	I	318	vsrab	v2.03			Vector Shift Right Algebraic Byte
000100 .... 01101 000100	VX	I	318	vsrah	v2.03			Vector Shift Right Algebraic Halfword
000100 .... 01110 000100	VX	I	318	vsraw	v2.03			Vector Shift Right Algebraic Word
000100 .... 01111 000100	VX	I	318	vsrad	v2.07			Vector Shift Right Algebraic Doubleword
000100 .... 10000 000100	VX	I	312	vand	v2.03			Vector Logical AND
000100 .... 10001 000100	VX	I	312	vandc	v2.03			Vector Logical AND with Complement
000100 .... 10010 000100	VX	I	313	vor	v2.03			Vector Logical OR
000100 .... 10011 000100	VX	I	313	vxor	v2.03			Vector Logical XOR
000100 .... 10100 000100	VX	I	313	vnor	v2.03			Vector Logical NOR
000100 .... 10101 000100	VX	I	313	vorc	v2.07			Vector OR with Complement
000100 .... 10110 000100	VX	I	312	vnand	v2.07			Vector NAND
000100 .... 10111 000100	VX	I	316	vsld	v2.07			Vector Shift Left Doubleword
000100 ..... ///// ///// 11000 000100	VX	I	362	mfvscr	v2.03			Move From VSCR
000100 ///// ///// 11001 000100	VX	I	362	mtvscr	v2.03			Move To VSCR
000100 .... 11010 000100	VX	I	312	veqv	v2.07			Vector Equivalence
000100 .... 11011 000100	VX	I	317	vsrd	v2.07			Vector Shift Right Doubleword
000100 .... 11100 000100	VX	I	265	vsrv	v3.0			Vector Shift Right Variable
000100 .... 11101 000100	VX	I	265	vslv	v3.0			Vector Shift Left Variable
000100 .... 00010 000101	VX	I	319	vrlwmi	v3.0			Vector Rotate Left Word then Mask Insert
000100 .... 00011 000101	VX	I	320	vrldmi	v3.0			Vector Rotate Left Doubleword then Mask Insert
000100 .... 00110 000101	VX	I	319	vrlwnm	v3.0			Vector Rotate Left Word then AND with Mask
000100 .... 00111 000101	VX	I	320	vrldnm	v3.0			Vector Rotate Left Doubleword then AND with Mask
000100 .... .0000 000110	VC	I	303	vcmpequb[.]	v2.03			Vector Compare Equal To Unsigned Byte
000100 .... .0001 000110	VC	I	303	vcmpequh[.]	v2.03			Vector Compare Equal To Unsigned Halfword
000100 .... .0010 000110	VC	I	304	vcmpequw[.]	v2.03			Vector Compare Equal To Unsigned Word
000100 .... .0011 000110	VC	I	329	vcmpeqfp[.]	v2.03			Vector Compare Equal To Floating-Point
000100 .... .0111 000110	VC	I	329	vcmpgefp[.]	v2.03			Vector Compare Greater Than or Equal To Floating-Point
000100 .... .1000 000110	VC	I	307	vcmpgtub[.]	v2.03			Vector Compare Greater Than Unsigned Byte
000100 .... .1001 000110	VC	I	308	vcmpgtuh[.]	v2.03			Vector Compare Greater Than Unsigned Halfword
000100 .... .1010 000110	VC	I	308	vcmpgtuw[.]	v2.03			Vector Compare Greater Than Unsigned Word
000100 .... .1011 000110	VC	I	330	vcmpgtfp[.]	v2.03			Vector Compare Greater Than Floating-Point
000100 .... .1100 000110	VC	I	305	vcmpgtsb[.]	v2.03			Vector Compare Greater Than Signed Byte
000100 .... .1101 000110	VC	I	306	vcmpgtsh[.]	v2.03			Vector Compare Greater Than Signed Halfword
000100 .... .1110 000110	VC	I	306	vcmpgtsw[.]	v2.03			Vector Compare Greater Than Signed Word
000100 .... .1111 000110	VC	I	328	vcmpbfp[.]	v2.03			Vector Compare Bounds Floating-Point
000100 .... .0000 000111	VC	I	309	vcmpneb[.]	v3.0			Vector Compare Not Equal Byte
000100 .... .0001 000111	VC	I	310	vcmpneh[.]	v3.0			Vector Compare Not Equal Halfword
000100 .... .0010 000111	VC	I	311	vcmpnew[.]	v3.0			Vector Compare Not Equal Word
000100 .... .0011 000111	VC	I	304	vcmpequd[.]	v2.07			Vector Compare Equal To Unsigned Doubleword
000100 .... .0100 000111	VC	I	309	vcmpnezb[.]	v3.0			Vector Compare Not Equal or Zero Byte
000100 .... .0101 000111	VC	I	310	vcmpnezh[.]	v3.0			Vector Compare Not Equal or Zero Halfword

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 3 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
000100 .... .0110 000111	VC	I	311	vcmpnezw[.]	v3.0			Vector Compare Not Equal or Zero Word
000100 .... .1011 000111	VC	I	307	vcmpgtud[.]	v2.07			Vector Compare Greater Than Unsigned Doubleword
000100 .... .1111 000111	VC	I	305	vcmpgtsd[.]	v2.07			Vector Compare Greater Than Signed Doubleword
000100 .... 00000 001000	VX	I	281	vmuloub	v2.03			Vector Multiply Odd Unsigned Byte
000100 .... 00001 001000	VX	I	282	vmulouh	v2.03			Vector Multiply Odd Unsigned Halfword
000100 .... 00010 001000	VX	I	283	vmulouw	v2.07			Vector Multiply Odd Unsigned Word
000100 .... 00100 001000	VX	I	281	vmulosb	v2.03			Vector Multiply Odd Signed Byte
000100 .... 00101 001000	VX	I	282	vmulosh	v2.03			Vector Multiply Odd Signed Halfword
000100 .... 00110 001000	VX	I	283	vmulosw	v2.07			Vector Multiply Odd Signed Word
000100 .... 01000 001000	VX	I	281	vmuleub	v2.03			Vector Multiply Even Unsigned Byte
000100 .... 01001 001000	VX	I	282	vmuleuh	v2.03			Vector Multiply Even Unsigned Halfword
000100 .... 01010 001000	VX	I	283	vmuleuw	v2.07			Vector Multiply Even Unsigned Word
000100 .... 01100 001000	VX	I	281	vmulesb	v2.03			Vector Multiply Even Signed Byte
000100 .... 01101 001000	VX	I	282	vmulesh	v2.03			Vector Multiply Even Signed Halfword
000100 .... 01110 001000	VX	I	283	vmulesw	v2.07			Vector Multiply Even Signed Word
000100 .... 10000 001000	VX	I	336	vpmsumb	v2.07			Vector Polynomial Multiply-Sum Byte
000100 .... 10001 001000	VX	I	337	vpmsumh	v2.07			Vector Polynomial Multiply-Sum Halfword
000100 .... 10010 001000	VX	I	337	vpmsumw	v2.07			Vector Polynomial Multiply-Sum Word
000100 .... 10011 001000	VX	I	336	vpmsumd	v2.07			Vector Polynomial Multiply-Sum Doubleword
000100 .... 10100 001000	VX	I	333	vcipher	v2.07			Vector AES Cipher
000100 .... 10101 001000	VX	I	334	vncipher	v2.07			Vector AES Inverse Cipher
000100 .... ///// 10111 001000	VX	I	334	vsbox	v2.07			Vector AES S-Box
000100 .... 11000 001000	VX	I	292	vsum4ubs	v2.03			Vector Sum across Quarter Unsigned Byte Saturate
000100 .... 11001 001000	VX	I	291	vsum4shs	v2.03			Vector Sum across Quarter Signed Halfword Saturate
000100 .... 11010 001000	VX	I	290	vsum2sws	v2.03			Vector Sum across Half Signed Word Saturate
000100 .... 11100 001000	VX	I	291	vsum4sbs	v2.03			Vector Sum across Quarter Signed Byte Saturate
000100 .... 11110 001000	VX	I	290	vsumsws	v2.03			Vector Sum across Signed Word Saturate
000100 .... 00010 001001	VX	I	284	vmuluwm	v2.07			Vector Multiply Unsigned Word Modulo
000100 .... 10100 001001	VX	I	333	vcipherlast	v2.07			Vector AES Cipher Last
000100 .... 10101 001001	VX	I	334	vncipherlast	v2.07			Vector AES Inverse Cipher Last
000100 .... 00000 001010	VX	I	321	vaddfp	v2.03			Vector Add Floating-Point
000100 .... 00001 001010	VX	I	321	vsubfp	v2.03			Vector Subtract Floating-Point
000100 ..... ///// 00100 001010	VX	I	332	vrefp	v2.03			Vector Reciprocal Estimate Floating-Point
000100 ..... ///// 00101 001010	VX	I	332	vrsqrtefp	v2.03			Vector Reciprocal Square Root Estimate Floating-Point
000100 ..... ///// 00110 001010	VX	I	331	vexptefp	v2.03			Vector 2 Raised to the Exponent Estimate Floating-Point
000100 .... 00111 001010	VX	I	331	vlogefp	v2.03			Vector Log Base 2 Estimate Floating-Point
000100 ..... ///// 01000 001010	VX	I	326	vrfin	v2.03			Vector Round to Floating-Point Integral Nearest
000100 ..... ///// 01001 001010	VX	I	327	vrfiz	v2.03			Vector Round to Floating-Point Integral toward Zero
000100 ..... ///// 01010 001010	VX	I	326	vrfip	v2.03			Vector Round to Floating-Point Integral toward +Infinity
000100 ..... ///// 01011 001010	VX	I	326	vrfim	v2.03			Vector Round to Floating-Point Integral toward -Infinity
000100 .... 01100 001010	VX	I	325	vcfux	v2.03			Vector Convert with round to nearest Unsigned Word format to FP
000100 .... 01101 001010	VX	I	325	vcfsx	v2.03			Vector Convert with round to nearest Signed Word format to FP
000100 .... 01110 001010	VX	I	324	vctuxs	v2.03			Vector Convert with round to zero FP To Unsigned Word format Saturate
000100 .... 01111 001010	VX	I	324	vctsxs	v2.03			Vector Convert with round to zero FP To Signed Word format Saturate
000100 .... 10000 001010	VX	I	323	vmaxfp	v2.03			Vector Maximum Floating-Point
000100 .... 10001 001010	VX	I	323	vminfp	v2.03			Vector Minimum Floating-Point
000100 .... 00000 001100	VX	I	255	vmrghb	v2.03			Vector Merge High Byte
000100 .... 00001 001100	VX	I	255	vmrghh	v2.03			Vector Merge High Halfword
000100 .... 00010 001100	VX	I	256	vmrghw	v2.03			Vector Merge High Word
000100 .... 00100 001100	VX	I	255	vmrglb	v2.03			Vector Merge Low Byte
000100 .... 00101 001100	VX	I	255	vmrglh	v2.03			Vector Merge Low Halfword
000100 .... 00110 001100	VX	I	256	vmrglw	v2.03			Vector Merge Low Word
000100 ..... /.... ..... 01000 001100	VX	I	258	vspltb	v2.03			Vector Splat Byte
000100 ..... //... ..... 01001 001100	VX	I	258	vsplth	v2.03			Vector Splat Halfword

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 4 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
000100 ..... ///.. ..... 01010 001100	VX	I	258	vspltw	v2.03			Vector Splat Word
000100 .... ///// 01100 001100	VX	I	259	vspltisb	v2.03			Vector Splat Immediate Signed Byte
000100 .... ///// 01101 001100	VX	I	259	vspltish	v2.03			Vector Splat Immediate Signed Halfword
000100 .... ///// 01110 001100	VX	I	259	vspltisw	v2.03			Vector Splat Immediate Signed Word
000100 .... 10000 001100	VX	I	264	vslo	v2.03			Vector Shift Left by Octet
000100 .... 10001 001100	VX	I	264	vsro	v2.03			Vector Shift Right by Octet
000100 ..... ///// 10100 001100	VX	I	339	vgbbd	v2.07			Vector Gather Bits by Byte by Doubleword
000100 .... 10101 001100	VX	I	346	vbpermq	v2.07			Vector Bit Permute Quadword
000100 .... 10111 001100	VX	I	346	vbpermd	v3.0			Vector Bit Permute Doubleword
000100 .... 11010 001100	VX	I	257	vmrgow	v2.07			Vector Merge Odd Word
000100 .... 11110 001100	VX	I	257	vmrgew	v2.07			Vector Merge Even Word
000100 ..... /.... ..... 01000 001101	VX	I	267	vextractub	v3.0			Vector Extract Unsigned Byte
000100 ..... /.... ..... 01001 001101	VX	I	267	vextractuh	v3.0			Vector Extract Unsigned Halfword
000100 ..... /.... ..... 01010 001101	VX	I	267	vextractuw	v3.0			Vector Extract Unsigned Word
000100 ..... /.... ..... 01011 001101	VX	I	267	vextractd	v3.0			Vector Extract Doubleword
000100 ..... /.... ..... 01100 001101	VX	I	268	vinsertb	v3.0			Vector Insert Byte
000100 ..... /.... ..... 01101 001101	VX	I	268	vinserth	v3.0			Vector Insert Halfword
000100 ..... /.... ..... 01110 001101	VX	I	268	vinsertw	v3.0			Vector Insert Word
000100 ..... /.... ..... 01111 001101	VX	I	268	vinsertd	v3.0			Vector Insert Doubleword
000100 .... 11000 001101	VX	I	343	vextublx	v3.0			Vector Extract Unsigned Byte Left-Indexed
000100 .... 11001 001101	VX	I	343	vextuhlx	v3.0			Vector Extract Unsigned Halfword Left-Indexed
000100 .... 11010 001101	VX	I	344	vextuwlx	v3.0			Vector Extract Unsigned Word Left-Indexed
000100 .... 11100 001101	VX	I	343	vextubrx	v3.0			Vector Extract Unsigned Byte Right-Indexed
000100 .... 11101 001101	VX	I	343	vextuhrx	v3.0			Vector Extract Unsigned Halfword Right-Indexed
000100 .... 11110 001101	VX	I	344	vextuwrx	v3.0			Vector Extract Unsigned Word Right-Indexed
000100 .... 00000 001110	VX	I	251	vpkuhum	v2.03			Vector Pack Unsigned Halfword Unsigned Modulo
000100 .... 00001 001110	VX	I	252	vpkuwum	v2.03			Vector Pack Unsigned Word Unsigned Modulo
000100 .... 00010 001110	VX	I	252	vpkuhus	v2.03			Vector Pack Unsigned Halfword Unsigned Saturate
000100 .... 00011 001110	VX	I	252	vpkuwus	v2.03			Vector Pack Unsigned Word Unsigned Saturate
000100 .... 00100 001110	VX	I	250	vpkshus	v2.03			Vector Pack Signed Halfword Unsigned Saturate
000100 .... 00101 001110	VX	I	251	vpkswus	v2.03			Vector Pack Signed Word Unsigned Saturate
000100 .... 00110 001110	VX	I	249	vpkshss	v2.03			Vector Pack Signed Halfword Signed Saturate
000100 .... 00111 001110	VX	I	250	vpkswss	v2.03			Vector Pack Signed Word Signed Saturate
000100 ..... ///// 01000 001110	VX	I	254	vupkhsb	v2.03			Vector Unpack High Signed Byte
000100 ..... ///// 01001 001110	VX	I	254	vupkhsh	v2.03			Vector Unpack High Signed Halfword
000100 ..... ///// 01010 001110	VX	I	254	vupklsb	v2.03			Vector Unpack Low Signed Byte
000100 ..... ///// 01011 001110	VX	I	254	vupklsh	v2.03			Vector Unpack Low Signed Halfword
000100 .... 01100 001110	VX	I	248	vpkpx	v2.03			Vector Pack Pixel
000100 ..... ///// 01101 001110	VX	I	253	vupkhpx	v2.03			Vector Unpack High Pixel
000100 ..... ///// 01111 001110	VX	I	253	vupklpx	v2.03			Vector Unpack Low Pixel
000100 .... 10001 001110	VX	I	251	vpkudum	v2.07			Vector Pack Unsigned Doubleword Unsigned Modulo
000100 .... 10011 001110	VX	I	251	vpkudus	v2.07			Vector Pack Unsigned Doubleword Unsigned Saturate
000100 .... 10101 001110	VX	I	249	vpksdus	v2.07			Vector Pack Signed Doubleword Unsigned Saturate
000100 .... 10111 001110	VX	I	248	vpksdss	v2.07			Vector Pack Signed Doubleword Signed Saturate
000100 ..... ///// 11001 001110	VX	I	254	vupkhsw	v2.07			Vector Unpack High Signed Word
000100 ..... ///// 11011 001110	VX	I	254	vupklsw	v2.07			Vector Unpack Low Signed Word
000100 100000	VA	I	285	vmhaddshs	v2.03			Vector Multiply-High-Add Signed Halfword Saturate
000100 100001	VA	I	285	vmhraddshs	v2.03			Vector Multiply-High-Round-Add Signed Halfword Saturate
000100 100010	VA	I	286	vmladduhm	v2.03			Vector Multiply-Low-Add Unsigned Halfword Modulo
000100 100011	VA	I	289	vmsumudm	v3.0B			Vector Multiply-Sum Unsigned Doubleword Modulo
000100 100100	VA	I	286	vmsumubm	v2.03			Vector Multiply-Sum Unsigned Byte Modulo
000100 100101	VA	I	287	vmsummbm	v2.03			Vector Multiply-Sum Mixed Byte Modulo
000100 100110	VA	I	288	vmsumuhm	v2.03			Vector Multiply-Sum Unsigned Halfword Modulo
000100 100111	VA	I	289	vmsumuhs	v2.03			Vector Multiply-Sum Unsigned Halfword Saturate
000100 101000	VA	I	287	vmsumshm	v2.03			Vector Multiply-Sum Signed Halfword Modulo
000100 101001	VA	I	288	vmsumshs	v2.03			Vector Multiply-Sum Signed Halfword Saturate

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 5 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
000100 101010	VA	I	261	vsel	v2.03			Vector Select
000100 101011	VA	I	260	vperm	v2.03			Vector Permute
000100 .... /.... 101100	VA	I	263	vsldoi	v2.03			Vector Shift Left Double by Octet Immediate
000100 101101	VA	I	338	vpermxor	v2.07			Vector Permute & Exclusive-OR
000100 101110	VA	I	322	vmaddfp	v2.03			Vector Multiply-Add Floating-Point
000100 101111	VA	I	322	vnmsubfp	v2.03			Vector Negative Multiply-Subtract Floating-Point
000100 110000	VA	I	80	maddhd	v3.0			Multiply-Add High Doubleword
000100 110001	VA	I	80	maddhdu	v3.0			Multiply-Add High Doubleword Unsigned
000100 110011	VA	I	80	maddld	v3.0			Multiply-Add Low Doubleword
000100 111011	VA	I	260	vpermr	v3.0			Vector Permute Right-indexed
000100 111100	VA	I	273	vaddeuqm	v2.07			Vector Add Extended Unsigned Quadword Modulo
000100 111101	VA	I	273	vaddecuq	v2.07			Vector Add Extended & write Carry Unsigned Quadword
000100 111110	VA	I	279	vsubeuqm	v2.07			Vector Subtract Extended Unsigned Quadword Modulo
000100 111111	VA	I	279	vsubecuq	v2.07			Vector Subtract Extended & write Carry Unsigned Quadword
000111	D	I	73	mulli	P1			Multiply Low Immediate
001000	D	I	70	subfic	P1		SR	Subtract From Immediate Carrying
001010 .../.	D	I	86	cmpli	P1			Compare Logical Immediate
001011 .../.	D	I	85	cmpi	P1			Compare Immediate
001100	D	I	69	addic	P1		SR	Add Immediate Carrying
001101	D	I	69	addic.	P1		SR	Add Immediate Carrying & record
001110	D	I	67	addi	P1			Add Immediate
001111	D	I	67	addis	P1			Add Immediate Shifted
010000	B	I	37	bc[l][a]	P1		CT	Branch Conditional [& Link] [Absolute]
010001 ///// ///// ////. ..... .///01	SC	I	42	scv	v3.0			System Call Vectored
010001 ///// ///// ////. ..... .///1/	SC	I	42	sc	PPC			System Call
010010	I	I	37	b[l][a]	P1			Branch [& Link] [Absolute]
010011 .../// .../// ///// 00000 00000/	XL	I	41	mcrf	P1			Move CR Field
010011 .... 00001 00001/	XL	I	41	crnor	P1			CR NOR
010011 .... 00100 00001/	XL	I	41	crandc	P1			CR AND with Complement
010011 .... 00110 00001/	XL	I	40	crxor	P1			CR XOR
010011 .... 00111 00001/	XL	I	40	crnand	P1			CR NAND
010011 .... 01000 00001/	XL	I	40	crand	P1			CR AND
010011 .... 01001 00001/	XL	I	41	creqv	P1			CR Equivalent
010011 .... 01101 00001/	XL	I	41	crorc	P1			CR OR with Complement
010011 .... 01110 00001/	XL	I	40	cror	P1			CR OR
010011 .... 00010.	DX	I	68	addpcis	v3.0			Add PC Immediate Shifted
010011 .... ///.. 00000 10000.	XL	I	38	bclr[l]	P1		CT	Branch Conditional to LR [& Link]
010011 .... ///.. 10000 10000.	XL	I	38	bcctr[l]	P1		CT	Branch Conditional to CTR [& Link]
010011 .... ///.. 10001 10000.	XL	I	39	bctar[l]	v2.07			Branch Conditional to BTAR [& Link]
010011 ///// ///// ///// 00000 10010/	XL	III	955	rfid	PPC	P		Return from Interrupt Doubleword
010011 ///// ///// ///// 00010 10010/	XL	III	953	rfscv	v3.0	P		Return From System Call Vectored
010011 ///// ///// ////. 00100 10010/	XL	I	905	rfebb	v2.07			Return from Event Based Branch
010011 ///// ///// ///// 01000 10010/	XL	III	956	hrfid	v2.02	HV		Return From Interrupt Doubleword Hypervisor
010011 ///// ///// ///// 01011 10010/	XL	III	958	stop	v3.0	P		Stop
010011 ///// ///// ///// 00100 10110/	XL	II	863	isync	P1			Instruction Synchronize
010100	M	I	103	rlwimi[.]	P1		SR	Rotate Left Word Immediate then Mask Insert
010101	M	I	102	rlwinm[.]	P1		SR	Rotate Left Word Immediate then AND with Mask
010111	M	I	103	rlwnm[.]	P1		SR	Rotate Left Word then AND with Mask
011000	D	I	92	ori	P1			OR Immediate
011001	D	I	93	oris	P1			OR Immediate Shifted
011010 00000 00000 00000 00000 000000	D	I	93	xnop	v2.05			Executed No Operation
011010	D	I	93	xori	P1			XOR Immediate
011011	D	I	93	xoris	P1			XOR Immediate Shifted
011100	D	I	92	andi.	P1		SR	AND Immediate & record
011101	D	I	92	andis.	P1		SR	AND Immediate Shifted & record
011110 .... .000..	MD	I	105	rldicl[.]	PPC		SR	Rotate Left Doubleword Immediate then Clear Left

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 6 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
011110 .... .001..	MD	I	106	rldicr[.]	PPC		SR	Rotate Left Doubleword Immediate then Clear Right
011110 .... .010..	MD	I	105	rldic[.]	PPC		SR	Rotate Left Doubleword Immediate then Clear
011110 .... .011..	MD	I	106	rldimi[.]	PPC		SR	Rotate Left Doubleword Immediate then Mask Insert
011110 .... .1000.	MDS	I	104	rldcl[.]	PPC		SR	Rotate Left Doubleword then Clear Left
011110 .... .1001.	MDS	I	104	rldcr[.]	PPC		SR	Rotate Left Doubleword then Clear Right
011111 .../. 00000 00000/	X	I	85	cmp	P1			Compare
011111 .../. 00001 00000/	X	I	86	cmpl	P1			Compare Logical
011111 ..../// ///// 00100 000000	VX	I	122	setb	v3.0			Set Boolean
011111 .../. 00110 00000/	X	I	87	cmprb	v3.0			Compare Ranged Byte
011111 .../// 00111 00000/	X	I	88	cmpeqb	v3.0			Compare Equal Byte
011111 .../// ///// ///// 10010 00000/	X	I	120	mcrxrx	v3.0			Move XER to CR Extended
011111 .... 00000 00100/	X	I	90	tw	P1			Trap Word
011111 .... 00010 00100/	X	I	91	td	PPC			Trap Doubleword
011111 .... 00000 00110/	X	I	247	lvsl	v2.03			Load Vector for Shift Left
011111 .... 00001 00110/	X	I	247	lvsr	v2.03			Load Vector for Shift Right
011111 .... 10010 00110/	X	II	860	lwat	v3.0			Load Word ATomic
011111 .... 10011 00110/	X	II	860	ldat	v3.0			Load Doubleword ATomic
011111 .... 10110 00110/	X	II	862	stwat	v3.0			Store Word ATomic
011111 .... 10111 00110/	X	II	862	stdat	v3.0			Store Doubleword ATomic
011111 ////. 11000 00110/	X	II	855	copy	v3.0			Copy
011111 ///// ///// ///// 11010 00110/	X	II	856	cp_abort	v3.0			CP_Abort
011111 ////. .... 11100 00110.	X	II	855	paste[.]	v3.0			Paste
011111 .... 00000 00111/	X	I	242	lvebx	v2.03			Load Vector Element Byte Indexed
011111 .... 00001 00111/	X	I	242	lvehx	v2.03			Load Vector Element Halfword Indexed
011111 .... 00010 00111/	X	I	243	lvewx	v2.03			Load Vector Element Word Indexed
011111 .... 00011 00111/	X	I	243	lvx	v2.03			Load Vector Indexed
011111 .... 00100 00111/	X	I	245	stvebx	v2.03			Store Vector Element Byte Indexed
011111 .... 00101 00111/	X	I	245	stvehx	v2.03			Store Vector Element Halfword Indexed
011111 .... 00110 00111/	X	I	246	stvewx	v2.03			Store Vector Element Word Indexed
011111 .... 00111 00111/	X	I	246	stvx	v2.03			Store Vector Indexed
011111 .... 01011 00111/	X	I	243	lvxl	v2.03			Load Vector Indexed Last
011111 .... 01111 00111/	X	I	246	stvxl	v2.03			Store Vector Indexed Last
011111 .... .0000 01000.	XO	I	70	subfc[o][.]	P1		SR	Subtract From Carrying
011111 .... .0001 01000.	XO	I	69	subf[o][.]	PPC		SR	Subtract From
011111 .... ///// .0011 01000.	XO	I	72	neg[o][.]	P1		SR	Negate
011111 .... .0100 01000.	XO	I	71	subfe[o][.]	P1		SR	Subtract From Extended
011111 .... ///// .0110 01000.	XO	I	72	subfze[o][.]	P1		SR	Subtract From Zero Extended
011111 .... ///// .0111 01000.	XO	I	71	subfme[o][.]	P1		SR	Subtract From Minus One Extended
011111 .... /0000 01001.	XO	I	79	mulhdu[.]	PPC		SR	Multiply High Doubleword Unsigned
011111 .... /0010 01001.	XO	I	79	mulhd[.]	PPC		SR	Multiply High Doubleword
011111 .... .0111 01001.	XO	I	79	mulld[o][.]	PPC		SR	Multiply Low Doubleword
011111 .... 01000 01001/	X	I	83	modud	v3.0			Modulo Unsigned Doubleword
011111 .... .1100 01001.	XO	I	82	divdeu[o][.]	v2.06		SR	Divide Doubleword Extended Unsigned
011111 .... .1101 01001.	XO	I	82	divde[o][.]	v2.06		SR	Divide Doubleword Extended
011111 .... .1110 01001.	XO	I	81	divdu[o][.]	PPC		SR	Divide Doubleword Unsigned
011111 .... .1111 01001.	XO	I	81	divd[o][.]	PPC		SR	Divide Doubleword
011111 .... 11000 01001/	X	I	83	modsd	v3.0			Modulo Signed Doubleword
011111 .... .0000 01010.	XO	I	70	addc[o][.]	P1		SR	Add Carrying
011111 .... /0010 01010/	XO	I	111	addg6s	v2.06			Add & Generate Sixes
011111 .... .0100 01010.	XO	I	71	adde[o][.]	P1		SR	Add Extended
011111 ....101 01010/	X	I	72	addex	v3.0B			Add Extended using alternate carry
011111 .... ///// .0110 01010.	XO	I	72	addze[o][.]	P1		SR	Add to Zero Extended
011111 .... ///// .0111 01010.	XO	I	71	addme[o][.]	P1		SR	Add to Minus One Extended
011111 .... .1000 01010.	XO	I	69	add[o][.]	P1		SR	Add
011111 .... /0000 01011.	XO	I	73	mulhwu[.]	PPC		SR	Multiply High Word Unsigned
011111 .... /0010 01011.	XO	I	73	mulhw[.]	PPC		SR	Multiply High Word

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 7 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
011111 .... .0111 01011.	XO	I	73	mullw[o][.]	P1		SR	Multiply Low Word
011111 .... 01000 01011/	X	I	77	moduw	v3.0			Modulo Unsigned Word
011111 .... .1100 01011.	XO	I	75	divweu[o][.]	v2.06		SR	Divide Word Extended Unsigned
011111 .... .1101 01011.	XO	I	75	divwe[o][.]	v2.06		SR	Divide Word Extended
011111 .... .1110 01011.	XO	I	74	divwu[o][.]	PPC		SR	Divide Word Unsigned
011111 .... .1111 01011.	XO	I	74	divw[o][.]	PPC		SR	Divide Word
011111 .... 11000 01011/	X	I	77	modsw	v3.0			Modulo Signed Word
011111 .... 00000 01100.	X	I	484	lxsiwzx	v2.07			Load VSX Scalar as Integer Word & Zero Indexed
011111 .... 00010 01100.	X	I	483	lxsiwax	v2.07			Load VSX Scalar as Integer Word Algebraic Indexed
011111 .... 00100 01100.	X	I	500	stxsiwx	v2.07			Store VSX Scalar as Integer Word Indexed
011111 .... 01000 01100.	X	I	492	lxvx	v3.0			Load VSX Vector Indexed
011111 .... 01010 01100.	X	I	494	lxvdsx	v2.06			Load VSX Vector Doubleword & Splat Indexed
011111 .... 01011 01100.	X	I	497	lxvwsx	v3.0			Load VSX Vector Word & Splat Indexed
011111 .... 01100 01100.	X	I	510	stxvx	v3.0			Store VSX Vector Indexed
011111 .... 10000 01100.	X	I	485	lxsspx	v2.07			Load VSX Scalar Single-Precision Indexed
011111 .... 10010 01100.	X	I	480	lxsdx	v2.06			Load VSX Scalar Doubleword Indexed
011111 .... 10100 01100.	X	I	502	stxsspx	v2.07			Store VSX Scalar Single-Precision Indexed
011111 .... 10110 01100.	X	I	498	stxsdx	v2.06			Store VSX Scalar Doubleword Indexed
011111 .... 11000 01100.	X	I	496	lxvw4x	v2.06			Load VSX Vector Word*4 Indexed
011111 .... 11001 01100.	X	I	495	lxvh8x	v3.0			Load VSX Vector Halfword*8 Indexed
011111 .... 11010 01100.	X	I	488	lxvd2x	v2.06			Load VSX Vector Doubleword*2 Indexed
011111 .... 11011 01100.	X	I	487	lxvb16x	v3.0			Load VSX Vector Byte*16 Indexed
011111 .... 11100 01100.	X	I	506	stxvw4x	v2.06			Store VSX Vector Word*4 Indexed
011111 .... 11101 01100.	X	I	505	stxvh8x	v3.0			Store VSX Vector Halfword*8 Indexed
011111 .... 11110 01100.	X	I	504	stxvd2x	v2.06			Store VSX Vector Doubleword*2 Indexed
011111 .... 11111 01100.	X	I	503	stxvb16x	v3.0			Store VSX Vector Byte*16 Indexed
011111 .... 01000 01101.	X	I	489	lxvl	v3.0			Load VSX Vector with Length
011111 .... 01001 01101.	X	I	491	lxvll	v3.0			Load VSX Vector Left-justified with Length
011111 .... 01100 01101.	X	I	507	stxvl	v3.0			Store VSX Vector with Length
011111 .... 01101 01101.	X	I	509	stxvll	v3.0			Store VSX Vector Left-justified with Length
011111 .... 11000 01101.	X	I	482	lxsibzx	v3.0			Load VSX Scalar as Integer Byte & Zero Indexed
011111 .... 11001 01101.	X	I	482	lxsihzx	v3.0			Load VSX Scalar as Integer Halfword & Zero Indexed
011111 .... 11100 01101.	X	I	499	stxsibx	v3.0			Store VSX Scalar as Integer Byte Indexed
011111 .... 11101 01101.	X	I	499	stxsihx	v3.0			Store VSX Scalar as Integer Halfword Indexed
011111 ///// ///// 00100 01110/	X	III	1131	msgsndp	v2.07	P		Message Send Privileged
011111 ///// ///// 00101 01110/	X	III	1132	msgclrp	v2.07	P		Message Clear Privileged
011111 ///// ///// 00110 01110/	X	III	1129	msgsnd	v2.07	HV		Message Send
011111 ///// ///// 00111 01110/	X	III	1130	msgclr	v2.07	HV		Message Clear
011111 .... 01001 01110/	X	I	909	mfbhrbe	v2.07			Move From BHRB
011111 ///// ///// ///// 01101 01110/	X	I	909	clrbhrb	v2.07			Clear BHRB
011111 .//// ///// ///// 10101 01110/	X	II	891	tend.	v2.07			Transaction End & record
011111 ...// ///// ///// 10110 01110/	X	II	895	tcheck	v2.07			Transaction Check & record
011111 ////. ///// ///// 10111 01110/	X	II	895	tsr.	v2.07			Transaction Suspend or Resume & record
011111 .///. ///// ///// 10100 011101	X	II	890	tbegin.	v2.07			Transaction Begin & record
011111 .... 11000 011101	X	II	893	tabortwc.	v2.07			Transaction Abort Word Conditional & record
011111 .... 11001 011101	X	II	894	tabortdc.	v2.07			Transaction Abort Doubleword Conditional & record
011111 .... 11010 011101	X	II	893	tabortwci.	v2.07			Transaction Abort Word Conditional Immediate & record
011111 .... 11011 011101	X	II	894	tabortdci.	v2.07			Transaction Abort Doubleword Conditional Immediate & record
011111 ///// ..... ///// 11100 011101	X	II	892	tabort.	v2.07			Transaction Abort & record
011111 ///// ..... ///// 11101 011101	X	II	969	treclaim.	v2.07			Transaction Reclaim & record
011111 ///// ///// ///// 11111 011101	X	II	970	trechkpt.	v2.07			Transaction Recheckpoint & record
011111 .... 01111/	A	I	91	isel	v2.03			Integer Select
011111 ..... 0.... ..../ 00100 10000/	XFX	I	121	mtcrf	P1			Move To CR Fields
011111 ..... 1.... ..../ 00100 10000/	XFX	I	121	mtocrf	v2.01			Move To One CR Field
011111 ..... ////. ///// 00100 10010/	X	III	977	mtmsr	P1	P		Move To MSR
011111 ..... ////. ///// 00101 10010/	X	III	978	mtmsrd	PPC	P		Move To MSR Doubleword

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 8 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
011111 ..... /.... ..... 01000 10010/	X	III	1038	tlbiel	v2.03	P	64	TLB Invalidate Entry Local
011111 ..... /.... ..... 01001 10010/	X	III	1034	tlbie	P1	HV	64	TLB Invalidate Entry
011111 ///// ///// ///// 01010 10010/	X	III	1032	slbsync	v3.0	P		SLB Synchronize
011111 ..... ///// 01100 10010/	X	III	1029	slbmte	v2.00	P		SLB Move To Entry
011111 ///// ///// 01101 10010/	X	III	1024	slbie	PPC	P		SLB Invalidate Entry
011111 ..... ///// 01110 10010/	X	III	1025	slbieg	v3.0	P		SLB Invalidate Entry Global
011111 //... ///// ///// 01111 10010/	X	III	1026	slbia	PPC	P		SLB Invalidate All
011111 ..... ///// 11010 10010/	X	III	1028	slbiag	v3.0B	P		SLB Invalidate All Global
011111 ..... 0//// ///// 00000 10011/	XFX	I	122	mfcr	P1			Move From CR
011111 ..... 1.... ..../ 00000 10011/	XFX	I	122	mfocrf	v2.01			Move From One CR Field
011111 .... ///// 00001 10011.	XX1	I	112	mfvsrd	v2.07			Move From VSR Doubleword
011111 ..... ///// ///// 00010 10011/	X	III	979	mfmsr	P1	P		Move From MSR
011111 .... ///// 00011 10011.	XX1	I	113	mfvsrwz	v2.07			Move From VSR Word & Zero
011111 .... ///// 00101 10011.	XX1	I	114	mtvsrd	v2.07			Move To VSR Doubleword
011111 .... ///// 00110 10011.	XX1	I	114	mtvsrwa	v2.07			Move To VSR Word Algebraic
011111 .... ///// 00111 10011.	XX1	I	115	mtvsrwz	v2.07			Move To VSR Word & Zero
011111 .... ///// 01001 10011.	XX1	I	112	mfvsrld	v3.0			Move From VSR Lower Doubleword
011111 .... 01010 10011/	X	X	119 975	mfspr	P1	O		Move From SPR
011111 .... 01011 10011/	X	II	898	mftb	PPC			Move From Time Base
011111 .... ///// 01100 10011.	XX1	I	116	mtvsrws	v3.0			Move To VSR Word & Splat
011111 .... 01101 10011.	XX1	I	115	mtvsrdd	v3.0			Move To VSR Double Doubleword
011111 .... 01110 10011/	X	X	117 974	mtspr	P1	O		Move To SPR
011111 ..... ///.. ///// 10111 10011/	X	I	78	darn	v3.0			Deliver A Random Number
011111 ..... ///// 11010 10011/	X	III	1030	slbmfev	v2.00	P		SLB Move From Entry VSID
011111 ..... ///// 11100 10011/	X	III	1031	slbmfee	v2.00	P		SLB Move From Entry ESID
011111 ..... ///// 11110 100111	X	III	1031	slbfee.	v2.05	P	SR	SLB Find Entry ESID & record
011111 .... 00000 10100/	X	II	865	lwarx	PPC			Load Word & Reserve Indexed
011111 .... 00001 10100.	X	II	864	lbarx	v2.06			Load Byte And Reserve Indexed
011111 .... 00010 10100/	X	II	869	ldarx	PPC			Load Doubleword And Reserve Indexed
011111 .... 00011 10100.	X	II	865	lharx	v2.06			Load Halfword And Reserve Indexed Xform
011111 .... 01000 10100.	X	I	871	lqarx	v2.07			Load Quadword And Reserve Indexed
011111 .... 10000 10100/	X	I	61	ldbrx	v2.06			Load Doubleword Byte-Reverse Indexed
011111 .... 10100 10100/	X	I	61	stdbrx	v2.06			Store Doubleword Byte-Reverse Indexed
011111 .... 00000 10101/	X	I	53	ldx	PPC			Load Doubleword Indexed
011111 .... 00001 10101/	X	I	53	ldux	PPC			Load Doubleword with Update Indexed
011111 .... 00100 10101/	X	I	57	stdx	PPC			Store Doubleword Indexed
011111 .... 00101 10101/	X	I	57	stdux	PPC			Store Doubleword with Update Indexed
011111 .... 01010 10101/	X	I	52	lwax	PPC			Load Word Algebraic Indexed
011111 .... 01011 10101/	X	I	52	lwaux	PPC			Load Word Algebraic with Update Indexed
011111 .... 10000 10101/	X	I	64	lswx	P1			Load String Word Indexed
011111 .... 10010 10101/	X	I	64	lswi	P1			Load String Word Immediate
011111 .... 10100 10101/	X	I	65	stswx	P1			Store String Word Indexed
011111 .... 10110 10101/	X	I	65	stswi	P1			Store String Word Immediate
011111 .... 11000 10101/	X	III	966	lwzcix	v2.05	HV		Load Word & Zero Caching Inhibited Indexed
011111 .... 11001 10101/	X	III	966	lhzcix	v2.05	HV		Load Halfword & Zero Caching Inhibited Indexed
011111 .... 11010 10101/	X	III	966	lbzcix	v2.05	HV		Load Byte & Zero Caching Inhibited Indexed
011111 .... 11011 10101/	X	III	966	ldcix	v2.05	HV		Load Doubleword Caching Inhibited Indexed
011111 .... 11100 10101/	X	III	967	stwcix	v2.05	HV		Store Word Caching Inhibited Indexed
011111 .... 11101 10101/	X	III	967	sthcix	v2.05	HV		Store Halfword Caching Inhibited Indexed
011111 .... 11110 10101/	X	III	967	stbcix	v2.05	HV		Store Byte Caching Inhibited Indexed
011111 .... 11111 10101/	X	III	967	stdcix	v2.05	HV		Store Doubleword Caching Inhibited Indexed
011111 /.... .... 00000 10110/	X	II	840	icbt	v2.07			Instruction Cache Block Touch
011111 ///// 00001 10110/	X	II	851	dcbst	PPC			Data Cache Block Store
011111 ///.. .... 00010 10110/	X	II	852	dcbf	PPC			Data Cache Block Flush
011111 .... 00111 10110/	X	II	850	dcbtst	PPC			Data Cache Block Touch for Store

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 9 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
011111 .... 01000 10110/	X	II	849	dcbt	PPC			Data Cache Block Touch
011111 .... 10000 10110/	X	I	60	lwbrx	P1			Load Word Byte-Reverse Indexed
011111 ///// ///// ///// 10001 10110/	X	III	1042	tlbsync	PPC	HV/P		TLB Synchronize
011111 ///.. ///// ///// 10010 10110/	X	II	873	sync	P1			Synchronize
011111 .... 10100 10110/	X	I	60	stwbrx	P1			Store Word Byte-Reverse Indexed
011111 .... 11000 10110/	X	I	60	lhbrx	P1			Load Halfword Byte-Reverse Indexed
011111 ///// ///// ///// 11010 10110/	X	II	875	eieio	PPC			Enforce In-order Execution of I/O
011111 ///// ///// ///// 11011 10110/	X	III	1132	msgsync	v3.0	HV		Message Synchronize
011111 .... 11100 10110/	X	I	60	sthbrx	P1			Store Halfword Byte-Reverse Indexed
011111 ///// 11110 10110/	X	II	840	icbi	PPC			Instruction Cache Block Invalidate
011111 ///// 11111 10110/	X	II	851	dcbz	P1			Data Cache Block Zero
011111 .... 00100 101101	X	II	868	stwcx.	PPC			Store Word Conditional Indexed & record
011111 .... 00101 101101	X	I	872	stqcx.	v2.07			Store Quadword Conditional Indexed & record
011111 .... 00110 101101	X	II	869	stdcx.	PPC			Store Doubleword Conditional Indexed & record
011111 .... 10101 101101	X	II	866	stbcx.	v2.06			Store Byte Conditional Indexed & record
011111 .... 10110 101101	X	II	867	sthcx.	v2.06			Store Halfword Conditional Indexed & record
011111 .... 00000 10111/	X	I	51	lwzx	P1			Load Word & Zero Indexed
011111 .... 00001 10111/	X	I	51	lwzux	P1			Load Word & Zero with Update Indexed
011111 .... 00010 10111/	X	I	48	lbzx	P1			Load Byte & Zero Indexed
011111 .... 00011 10111/	X	I	48	lbzux	P1			Load Byte & Zero with Update Indexed
011111 .... 00100 10111/	X	I	56	stwx	P1			Store Word Indexed
011111 .... 00101 10111/	X	I	56	stwux	P1			Store Word with Update Indexed
011111 .... 00110 10111/	X	I	54	stbx	P1			Store Byte Indexed
011111 .... 00111 10111/	X	I	54	stbux	P1			Store Byte with Update Indexed
011111 .... 01000 10111/	X	I	49	lhzx	P1			Load Halfword & Zero Indexed
011111 .... 01001 10111/	X	I	49	lhzux	P1			Load Halfword & Zero with Update Indexed
011111 .... 01010 10111/	X	I	50	lhax	P1			Load Halfword Algebraic Indexed
011111 .... 01011 10111/	X	I	50	lhaux	P1			Load Halfword Algebraic with Update Indexed
011111 .... 01100 10111/	X	I	55	sthx	P1			Store Halfword Indexed
011111 .... 01101 10111/	X	I	55	sthux	P1			Store Halfword with Update Indexed
011111 .... 10000 10111/	X	I	141	lfsx	P1			Load Floating Single Indexed
011111 .... 10001 10111/	X	I	142	lfsux	P1			Load Floating Single with Update Indexed
011111 .... 10010 10111/	X	I	142	lfdx	P1			Load Floating Double Indexed
011111 .... 10011 10111/	X	I	143	lfdux	P1			Load Floating Double with Update Indexed
011111 .... 10100 10111/	X	I	145	stfsx	P1			Store Floating Single Indexed
011111 .... 10101 10111/	X	I	145	stfsux	P1			Store Floating Single with Update Indexed
011111 .... 10110 10111/	X	I	146	stfdx	P1			Store Floating Double Indexed
011111 .... 10111 10111/	X	I	146	stfdux	P1			Store Floating Double with Update Indexed
011111 .... 11000 10111/	X	I	149	lfdpx	v2.05			Load Floating Double Pair Indexed
011111 .... 11010 10111/	X	I	143	lfiwax	v2.05			Load Floating as Integer Word Algebraic Indexed
011111 .... 11011 10111/	X	I	143	lfiwzx	v2.06			Load Floating as Integer Word & Zero Indexed
011111 .... 11100 10111/	X	I	149	stfdpx	v2.05			Store Floating Double Pair Indexed
011111 .... 11110 10111/	X	I	147	stfiwx	PPC			Store Floating as Integer Word Indexed
011111 .... 00000 11000.	X	I	107	slw[.]	P1		SR	Shift Left Word
011111 .... 10000 11000.	X	I	107	srw[.]	P1		SR	Shift Right Word
011111 .... 11000 11000.	X	I	108	sraw[.]	P1		SR	Shift Right Algebraic Word
011111 .... 11001 11000.	X	I	108	srawi[.]	P1		SR	Shift Right Algebraic Word Immediate
011111 .... ///// 00000 11010.	X	I	96	cntlzw[.]	P1		SR	Count Leading Zeros Word
011111 .... ///// 00001 11010.	X	I	99	cntlzd[.]	PPC		SR	Count Leading Zeros Doubleword
011111 .... ///// 00011 11010/	X	I	97	popcntb	v2.02			Population Count Byte
011111 .... ///// 00100 11010/	X	I	98	prtyw	v2.05			Parity Word
011111 .... ///// 00101 11010/	X	I	98	prtyd	v2.05			Parity Doubleword
011111 .... ///// 01000 11010/	X	I	111	cdtbcd	v2.06			Convert Declets To Binary Coded Decimal
011111 .... ///// 01001 11010/	X	I	111	cbcdtd	v2.06			Convert Binary Coded Decimal To Declets
011111 .... ///// 01011 11010/	X	I	97	popcntw	v2.06			Population Count Words
011111 .... ///// 01111 11010/	X	I	99	popcntd	v2.06			Population Count Doubleword

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 10 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
011111 .... ///// 10000 11010.	X	I	96	cnttzw[.]	v3.0			Count Trailing Zeros Word
011111 .... ///// 10001 11010.	X	I	99	cnttzd[.]	v3.0			Count Trailing Zeros Doubleword
011111 .... 11000 11010.	X	I	110	srad[.]	PPC		SR	Shift Right Algebraic Doubleword
011111 .... 11001 1101..	XS	I	110	sradi[.]	PPC		SR	Shift Right Algebraic Doubleword Immediate
011111 .... 11011 1101..	XS	I	110	extswsli[.]	v3.0			Extend Sign Word & Shift Left Immediate
011111 .... ///// 11100 11010.	X	I	96	extsh[.]	P1		SR	Extend Sign Halfword
011111 .... ///// 11101 11010.	X	I	96	extsb[.]	PPC		SR	Extend Sign Byte
011111 .... ///// 11110 11010.	X	I	99	extsw[.]	PPC		SR	Extend Sign Word
011111 .... 00000 11011.	X	I	109	sld[.]	PPC		SR	Shift Left Doubleword
011111 .... 10000 11011.	X	I	109	srd[.]	PPC		SR	Shift Right Doubleword
011111 .... 00000 11100.	X	I	94	and[.]	P1		SR	AND
011111 .... 00001 11100.	X	I	95	andc[.]	P1		SR	AND with Complement
011111 .... 00011 11100.	X	I	95	nor[.]	P1		SR	NOR
011111 .... 00111 11100/	X	I	100	bpermd	v2.06			Bit Permute Doubleword
011111 .... 01000 11100.	X	I	95	eqv[.]	P1		SR	Equivalent
011111 .... 01001 11100.	X	I	94	xor[.]	P1		SR	XOR
011111 .... 01100 11100.	X	I	95	orc[.]	P1		SR	OR with Complement
011111 .... 01101 11100.	X	I	94	or[.]	P1		SR	OR
011111 .... 01110 11100.	X	I	94	nand[.]	P1		SR	NAND
011111 .... 01111 11100/	X	I	97	cmpb	v2.05			Compare Byte
011111 ///.. ///// ///// 00000 11110/	X	II	876	wait	v3.0			Wait for Interrupt
100000	D	I	51	lwz	P1			Load Word & Zero
100001	D	I	51	lwzu	P1			Load Word & Zero with Update
100010	D	I	48	lbz	P1			Load Byte & Zero
100011	D	I	48	lbzu	P1			Load Byte & Zero with Update
100100	D	I	56	stw	P1			Store Word
100101	D	I	56	stwu	P1			Store Word with Update
100110	D	I	54	stb	P1			Store Byte
100111	D	I	54	stbu	P1			Store Byte with Update
101000	D	I	49	lhz	P1			Load Halfword & Zero
101001	D	I	49	lhzu	P1			Load Halfword & Zero with Update
101010	D	I	50	lha	P1			Load Halfword Algebraic
101011	D	I	50	lhau	P1			Load Halfword Algebraic with Update
101100	D	I	55	sth	P1			Store Halfword
101101	D	I	55	sthu	P1			Store Halfword with Update
101110	D	I	62	lmw	P1			Load Multiple Word
101111	D	I	62	stmw	P1			Store Multiple Word
110000	D	I	140	lfs	P1			Load Floating Single
110001	D	I	141	lfsu	P1			Load Floating Single with Update
110010	D	I	142	lfd	P1			Load Floating Double
110011	D	I	142	lfdu	P1			Load Floating Double with Update
110100	D	I	145	stfs	P1			Store Floating Single
110101	D	I	145	stfsu	P1			Store Floating Single with Update
110110	D	I	146	stfd	P1			Store Floating Double
110111	D	I	146	stfdu	P1			Store Floating Double with Update
111000	DQ	I	58	lq	v2.03			Load Quadword
111001 00	DS	I	149	lfdp	v2.05			Load Floating Double Pair
111001 10	DS	I	480	lxsd	v3.0			Load VSX Scalar Doubleword
111001 11	DS	I	485	lxssp	v3.0			Load VSX Scalar Single
111010 00	DS	I	53	ld	PPC			Load Doubleword
111010 01	DS	I	53	ldu	PPC			Load Doubleword with Update
111010 10	DS	I	52	lwa	PPC			Load Word Algebraic
111011 .... 00000 00010.	X	I	193	dadd[.]	v2.05			DFP Add
111011 .... 00001 00010.	X	I	195	dmul[.]	v2.05			DFP Multiply
111011 .... .0010 00010.	Z22	I	220	dscli[.]	v2.05			DFP Shift Significand Left Immediate
111011 .... .0011 00010.	Z22	I	220	dscri[.]	v2.05			DFP Shift Significand Right Immediate

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 11 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111011 ...// 00100 00010/	X	I	199	dcmpo	v2.05			DFP Compare Ordered
111011 ...// 00101 00010/	X	I	201	dtstex	v2.05			DFP Test Exponent
111011 ...// .... .0110 00010/	Z22	I	200	dtstdc	v2.05			DFP Test Data Class
111011 ...// .... .0111 00010/	Z22	I	200	dtstdg	v2.05			DFP Test Data Group
111011 ..... ///// ..... 01000 00010.	X	I	213	dctdp[.]	v2.05			DFP Convert To DFP Long
111011 ..... ///// ..... 01001 00010.	X	I	215	dctfix[.]	v2.05			DFP Convert To Fixed
111011 ....// ..... 01010 00010.	X	I	217	ddedpd[.]	v2.05			DFP Decode DPD To BCD
111011 ..... ///// ..... 01011 00010.	X	I	218	dxex[.]	v2.05			DFP Extract Exponent
111011 .... 10000 00010.	X	I	193	dsub[.]	v2.05			DFP Subtract
111011 .... 10001 00010.	X	I	196	ddiv[.]	v2.05			DFP Divide
111011 ...// 10100 00010/	X	I	198	dcmpu	v2.05			DFP Compare Unordered
111011 ...// 10101 00010/	X	I	202	dtstsf	v2.05			DFP Test Significance
111011 ..... ///// ..... 11000 00010.	X	I	214	drsp[.]	v2.05			DFP Round To DFP Short
111011 ..... ///// ..... 11001 00010.	X	I	215	dcffix[.]	v2.06			DFP Convert From Fixed
111011 ..... .//// ..... 11010 00010.	X	I	217	denbcd[.]	v2.05			DFP Encode BCD To DPD
111011 .... 11011 00010.	X	I	218	diex[.]	v2.05			DFP Insert Exponent
111011 ....000 00011.	Z23	I	204	dqua[.]	v2.05			DFP Quantize
111011 ....001 00011.	Z23	I	206	drrnd[.]	v2.05			DFP Reround
111011 ....010 00011.	Z23	I	203	dquai[.]	v2.05			DFP Quantize Immediate
111011 ..... ////. ....011 00011.	Z23	I	209	drintx[.]	v2.05			DFP Round To FP Integer With Inexact
111011 ..... ////. ....111 00011.	Z23	I	211	drintn[.]	v2.05			DFP Round To FP Integer Without Inexact
111011 ...// 10101 00011/	X	I	202	dtstsfi	v3.0			DFP Test Significance Immediate
111011 ..... ///// ..... 11010 01110.	X	I	164	fcfids[.]	v2.06			Floating Convert with round Signed Doubleword to Single-Precision format
111011 ..... ///// ..... 11110 01110.	X	I	165	fcfidus[.]	v2.06			Floating Convert with round Unsigned Doubleword to Single-Precision format
111011 .... ///// 10010.	A	I	153	fdivs[.]	PPC			Floating Divide Single
111011 .... ///// 10100.	A	I	152	fsubs[.]	PPC			Floating Subtract Single
111011 .... ///// 10101.	A	I	152	fadds[.]	PPC			Floating Add Single
111011 ..... ///// ..... ///// 10110.	A	I	154	fsqrts[.]	PPC			Floating Square Root Single
111011 ..... ///// ..... ///// 11000.	A	I	154	fres[.]	PPC			Floating Reciprocal Estimate Single
111011 .... ///// ..... 11001.	A	I	153	fmuls[.]	PPC			Floating Multiply Single
111011 ..... ///// ..... ///// 11010.	A	I	155	frsqrtes[.]	v2.02			Floating Reciprocal Square Root Estimate Single
111011 .... 11100.	A	I	158	fmsubs[.]	PPC			Floating Multiply-Subtract Single
111011 .... 11101.	A	I	157	fmadds[.]	PPC			Floating Multiply-Add Single
111011 .... 11110.	A	I	158	fnmsubs[.]	PPC			Floating Negative Multiply-Subtract Single
111011 .... 11111.	A	I	158	fnmadds[.]	PPC			Floating Negative Multiply-Add Single
111100 .... 00000 000...	XX3	I	518	xsaddsp	v2.07			VSX Scalar Add Single-Precision
111100 .... 00001 000...	XX3	I	649	xssubsp	v2.07			VSX Scalar Subtract Single-Precision
111100 .... 00010 000...	XX3	I	604	xsmulsp	v2.07			VSX Scalar Multiply Single-Precision
111100 .... 00011 000...	XX3	I	566	xsdivsp	v2.07			VSX Scalar Divide Single-Precision
111100 .... 00100 000...	XX3	I	513	xsadddp	v2.06			VSX Scalar Add Double-Precision
111100 .... 00101 000...	XX3	I	645	xssubdp	v2.06			VSX Scalar Subtract Double-Precision
111100 .... 00110 000...	XX3	I	600	xsmuldp	v2.06			VSX Scalar Multiply Double-Precision
111100 .... 00111 000...	XX3	I	562	xsdivdp	v2.06			VSX Scalar Divide Double-Precision
111100 .... 01000 000...	XX3	I	663	xvaddsp	v2.06			VSX Vector Add Single-Precision
111100 .... 01001 000...	XX3	I	755	xvsubsp	v2.06			VSX Vector Subtract Single-Precision
111100 .... 01010 000...	XX3	I	723	xvmulsp	v2.06			VSX Vector Multiply Single-Precision
111100 .... 01011 000...	XX3	I	698	xvdivsp	v2.06			VSX Vector Divide Single-Precision
111100 .... 01100 000...	XX3	I	659	xvadddp	v2.06			VSX Vector Add Double-Precision
111100 .... 01101 000...	XX3	I	753	xvsubdp	v2.06			VSX Vector Subtract Double-Precision
111100 .... 01110 000...	XX3	I	721	xvmuldp	v2.06			VSX Vector Multiply Double-Precision
111100 .... 01111 000...	XX3	I	696	xvdivdp	v2.06			VSX Vector Divide Double-Precision
111100 .... 10000 000...	XX3	I	581	xsmaxcdp	v3.0			VSX Scalar Maximum Type-C Double-Precision
111100 .... 10001 000...	XX3	I	587	xsmincdp	v3.0			VSX Scalar Minimum Type-C Double-Precision
111100 .... 10010 000...	XX3	I	583	xsmaxjdp	v3.0			VSX Scalar Maximum Type-J Double-Precision
111100 .... 10011 000...	XX3	I	589	xsminjdp	v3.0			VSX Scalar Minimum Type-J Double-Precision

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 12 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111100 .... 10100 000...	XX3	I	579	xsmaxdp	v2.06			VSX Scalar Maximum Double-Precision
111100 .... 10101 000...	XX3	I	585	xsmindp	v2.06			VSX Scalar Minimum Double-Precision
111100 .... 10110 000...	XX3	I	533	xscpsgndp	v2.06			VSX Scalar Copy Sign Double-Precision
111100 .... 11000 000...	XX3	I	709	xvmaxsp	v2.06			VSX Vector Maximum Single-Precision
111100 .... 11001 000...	XX3	I	713	xvminsp	v2.06			VSX Vector Minimum Single-Precision
111100 .... 11010 000...	XX3	I	671	xvcpsgnsp	v2.06			VSX Vector Copy Sign Single-Precision
111100 .... 11011 000...	XX3	I	700	xviexpsp	v3.0			VSX Vector Insert Exponent Single-Precision
111100 .... 11100 000...	XX3	I	707	xvmaxdp	v2.06			VSX Vector Maximum Double-Precision
111100 .... 11101 000...	XX3	I	711	xvmindp	v2.06			VSX Vector Minimum Double-Precision
111100 .... 11110 000...	XX3	I	671	xvcpsgndp	v2.06			VSX Vector Copy Sign Double-Precision
111100 .... 11111 000...	XX3	I	700	xviexpdp	v3.0			VSX Vector Insert Exponent Double-Precision
111100 .... 00000 001...	XX3	I	573	xsmaddasp	v2.07			VSX Scalar Multiply-Add Type-A Single-Precision
111100 .... 00001 001...	XX3	I	573	xsmaddmsp	v2.07			VSX Scalar Multiply-Add Type-M Single-Precision
111100 .... 00010 001...	XX3	I	594	xsmsubasp	v2.07			VSX Scalar Multiply-Subtract Type-A Single-Precision
111100 .... 00011 001...	XX3	I	594	xsmsubmsp	v2.07			VSX Scalar Multiply-Subtract Type-M Single-Precision
111100 .... 00100 001...	XX3	I	570	xsmaddadp	v2.06			VSX Scalar Multiply-Add Type-A Double-Precision
111100 .... 00101 001...	XX3	I	570	xsmaddmdp	v2.06			VSX Scalar Multiply-Add Type-M Double-Precision
111100 .... 00110 001...	XX3	I	591	xsmsubadp	v2.06			VSX Scalar Multiply-Subtract Type-A Double-Precision
111100 .... 00111 001...	XX3	I	591	xsmsubmdp	v2.06			VSX Scalar Multiply-Subtract Type-M Double-Precision
111100 .... 01000 001...	XX3	I	704	xvmaddasp	v2.06			VSX Vector Multiply-Add Type-A Single-Precision
111100 .... 01001 001...	XX3	I	704	xvmaddmsp	v2.06			VSX Vector Multiply-Add Type-M Single-Precision
111100 .... 01010 001...	XX3	I	718	xvmsubasp	v2.06			VSX Vector Multiply-Subtract Type-A Single-Precision
111100 .... 01011 001...	XX3	I	718	xvmsubmsp	v2.06			VSX Vector Multiply-Subtract Type-M Single-Precision
111100 .... 01100 001...	XX3	I	701	xvmaddadp	v2.06			VSX Vector Multiply-Add Type-A Double-Precision
111100 .... 01101 001...	XX3	I	701	xvmaddmdp	v2.06			VSX Vector Multiply-Add Type-M Double-Precision
111100 .... 01110 001...	XX3	I	715	xvmsubadp	v2.06			VSX Vector Multiply-Subtract Type-A Double-Precision
111100 .... 01111 001...	XX3	I	715	xvmsubmdp	v2.06			VSX Vector Multiply-Subtract Type-M Double-Precision
111100 .... 10000 001...	XX3	I	613	xsnmaddasp	v2.07			VSX Scalar Negative Multiply-Add Type-A Single-Precision
111100 .... 10001 001...	XX3	I	613	xsnmaddmsp	v2.07			VSX Scalar Negative Multiply-Add Type-M Single-Precision
111100 .... 10010 001...	XX3	I	622	xsnmsubasp	v2.07			VSX Scalar Negative Multiply-Subtract Type-A Single-Precision
111100 .... 10011 001...	XX3	I	622	xsnmsubmsp	v2.07			VSX Scalar Negative Multiply-Subtract Type-M Single-Precision
111100 .... 10100 001...	XX3	I	608	xsnmaddadp	v2.06			VSX Scalar Negative Multiply-Add Type-A Double-Precision
111100 .... 10101 001...	XX3	I	608	xsnmaddmdp	v2.06			VSX Scalar Negative Multiply-Add Type-M Double-Precision
111100 .... 10110 001...	XX3	I	619	xsnmsubadp	v2.06			VSX Scalar Negative Multiply-Subtract Type-A Double-Precision
111100 .... 10111 001...	XX3	I	619	xsnmsubmdp	v2.06			VSX Scalar Negative Multiply-Subtract Type-M Double-Precision
111100 .... 11000 001...	XX3	I	732	xvnmaddasp	v2.06			VSX Vector Negative Multiply-Add Type-A Single-Precision
111100 .... 11001 001...	XX3	I	732	xvnmaddmsp	v2.06			VSX Vector Negative Multiply-Add Type-M Single-Precision
111100 .... 11010 001...	XX3	I	738	xvnmsubasp	v2.06			VSX Vector Negative Multiply-Subtract Type-A Single-Precision
111100 .... 11011 001...	XX3	I	738	xvnmsubmsp	v2.06			VSX Vector Negative Multiply-Subtract Type-M Single-Precision
111100 .... 11100 001...	XX3	I	727	xvnmaddadp	v2.06			VSX Vector Negative Multiply-Add Type-A Double-Precision
111100 .... 11101 001...	XX3	I	727	xvnmaddmdp	v2.06			VSX Vector Negative Multiply-Add Type-M Double-Precision
111100 .... 11110 001...	XX3	I	735	xvnmsubadp	v2.06			VSX Vector Negative Multiply-Subtract Type-A Double-Precision
111100 .... 11111 001...	XX3	I	735	xvnmsubmdp	v2.06			VSX Vector Negative Multiply-Subtract Type-M Double-Precision
111100 .... 0..00 010...	XX3	I	774	xxsldwi	v2.06			VSX Vector Shift Left Double by Word Immediate
111100 .... 0..01 010...	XX3	I	773	xxpermdi	v2.06			VSX Vector Doubleword Permute Immediate
111100 .... 00010 010...	XX3	I	771	xxmrghw	v2.06			VSX Vector Merge Word High
111100 .... 00011 010...	XX3	I	772	xxperm	v3.0			VSX Vector Permute
111100 .... 00110 010...	XX3	I	771	xxmrglw	v2.06			VSX Vector Merge Word Low
111100 .... 00111 010...	XX3	I	772	xxpermr	v3.0			VSX Vector Permute Right-indexed
111100 ..... ///.. ..... 01010 0100..	XX2	I	774	xxspltw	v2.06			VSX Vector Splat Word
111100 ..... 00... ..... 01011 01000.	XX1	I	774	xxspltib	v3.0			VSX Vector Splat Immediate Byte
111100 .... 10000 010...	XX3	I	767	xxland	v2.06			VSX Vector Logical AND
111100 .... 10001 010...	XX3	I	767	xxlandc	v2.06			VSX Vector Logical AND with Complement
111100 .... 10010 010...	XX3	I	770	xxlor	v2.06			VSX Vector Logical OR
111100 .... 10011 010...	XX3	I	770	xxlxor	v2.06			VSX Vector Logical XOR
111100 .... 10100 010...	XX3	I	769	xxlnor	v2.06			VSX Vector Logical NOR

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 13 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111100 .... 10101 010...	XX3	I	769	xxlorc	v2.07			VSX Vector Logical OR with Complement
111100 .... 10110 010...	XX3	I	768	xxlnand	v2.07			VSX Vector Logical NAND
111100 .... 10111 010...	XX3	I	768	xxleqv	v2.07			VSX Vector Logical Equivalence
111100 ..... /.... ..... 01010 0101..	XX2	I	766	xxextractuw	v3.0			VSX Vector Extract Unsigned Word
111100 ..... /.... ..... 01011 0101..	XX2	I	766	xxinsertw	v3.0			VSX Vector Insert Word
111100 .... 00000 011...	XX3	I	524	xscmpeqdp	v3.0			VSX Scalar Compare Equal Double-Precision
111100 .... 00001 011...	XX3	I	526	xscmpgtdp	v3.0			VSX Scalar Compare Greater Than Double-Precision
111100 .... 00010 011...	XX3	I	525	xscmpgedp	v3.0			VSX Scalar Compare Greater Than or Equal Double-Precision
111100 .../// .... 00100 011../	XX3	I	530	xscmpudp	v2.06			VSX Scalar Compare Unordered Double-Precision
111100 .../// .... 00101 011../	XX3	I	527	xscmpodp	v2.06			VSX Scalar Compare Ordered Double-Precision
111100 .../// .... 00111 011../	XX3	I	522	xscmpexpdp	v3.0			VSX Scalar Compare Exponents Double-Precision
111100 .... .1000 011...	XX3	I	666	xvcmpeqsp[.]	v2.06			VSX Vector Compare Equal Single-Precision
111100 .... .1001 011...	XX3	I	670	xvcmpgtsp[.]	v2.06			VSX Vector Compare Greater Than Single-Precision
111100 .... .1010 011...	XX3	I	668	xvcmpgesp[.]	v2.06			VSX Vector Compare Greater Than or Equal Single-Precision
111100 .... .1100 011...	XX3	I	665	xvcmpeqdp[.]	v2.06			VSX Vector Compare Equal Double-Precision
111100 .... .1101 011...	XX3	I	669	xvcmpgtdp[.]	v2.06			VSX Vector Compare Greater Than Double-Precision
111100 .... .1110 011...	XX3	I	667	xvcmpgedp[.]	v2.06			VSX Vector Compare Greater Than or Equal Double-Precision
111100 ..... ///// ..... 00100 1000..	XX2	I	544	xscvdpuxws	v2.06			VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format
111100 ..... ///// ..... 00101 1000..	XX2	I	540	xscvdpsxws	v2.06			VSX Scalar Convert with round to zero Double-Precision to Signed Word format
111100 ..... ///// ..... 01000 1000..	XX2	I	690	xvcvspuxws	v2.06			VSX Vector Convert with round to zero Single-Precision to Unsigned Word format
111100 ..... ///// ..... 01001 1000..	XX2	I	686	xvcvspsxws	v2.06			VSX Vector Convert with round to zero Single-Precision to Signed Word format
111100 ..... ///// ..... 01010 1000..	XX2	I	695	xvcvuxwsp	v2.06			VSX Vector Convert with round Unsigned Word to Single-Precision format
111100 ..... ///// ..... 01011 1000..	XX2	I	693	xvcvsxwsp	v2.06			VSX Vector Convert with round Signed Word to Single-Precision format
111100 ..... ///// ..... 01100 1000..	XX2	I	679	xvcvdpuxws	v2.06			VSX Vector Convert with round to zero Double-Precision to Unsigned Word format
111100 ..... ///// ..... 01101 1000..	XX2	I	675	xvcvdpsxws	v2.06			VSX Vector Convert with round to zero Double-Precision to Signed Word format
111100 ..... ///// ..... 01110 1000..	XX2	I	695	xvcvuxwdp	v2.06			VSX Vector Convert Unsigned Word to Double-Precision format
111100 ..... ///// ..... 01111 1000..	XX2	I	693	xvcvsxwdp	v2.06			VSX Vector Convert Signed Word to Double-Precision format
111100 ..... ///// ..... 10010 1000..	XX2	I	561	xscvuxdsp	v2.07			VSX Scalar Convert with round Unsigned Doubleword to Single-Precision format
111100 ..... ///// ..... 10011 1000..	XX2	I	559	xscvsxdsp	v2.07			VSX Scalar Convert with round Signed Doubleword to Single-Precision format
111100 ..... ///// ..... 10100 1000..	XX2	I	542	xscvdpuxds	v2.06			VSX Scalar Convert with round to zero Double-Precision to Unsigned Doubleword format
111100 ..... ///// ..... 10101 1000..	XX2	I	537	xscvdpsxds	v2.06			VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format
111100 ..... ///// ..... 10110 1000..	XX2	I	561	xscvuxddp	v2.06			VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format
111100 ..... ///// ..... 10111 1000..	XX2	I	559	xscvsxddp	v2.06			VSX Scalar Convert with round Signed Doubleword to Double-Precision format
111100 ..... ///// ..... 11000 1000..	XX2	I	688	xvcvspuxds	v2.06			VSX Vector Convert with round to zero Single-Precision to Unsigned Doubleword format
111100 ..... ///// ..... 11001 1000..	XX2	I	684	xvcvspsxds	v2.06			VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format
111100 ..... ///// ..... 11010 1000..	XX2	I	694	xvcvuxdsp	v2.06			VSX Vector Convert with round Unsigned Doubleword to Single-Precision format
111100 ..... ///// ..... 11011 1000..	XX2	I	692	xvcvsxdsp	v2.06			VSX Vector Convert with round Signed Doubleword to Single-Precision format
111100 ..... ///// ..... 11100 1000..	XX2	I	677	xvcvdpuxds	v2.06			VSX Vector Convert with round to zero Double-Precision to Unsigned Doubleword format
111100 ..... ///// ..... 11101 1000..	XX2	I	673	xvcvdpsxds	v2.06			VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format
111100 ..... ///// ..... 11110 1000..	XX2	I	694	xvcvuxddp	v2.06			VSX Vector Convert with round Unsigned Doubleword to Double-Precision format

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 14 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111100 ..... ///// ..... 11111 1000..	XX2	I	692	xvcvsxddp	v2.06			VSX Vector Convert with round Signed Doubleword to Double-Precision format
111100 ..... ///// ..... 00100 1001..	XX2	I	628	xsrdpi	v2.06			VSX Scalar Round Double-Precision to Integral
111100 ..... ///// ..... 00101 1001..	XX2	I	631	xsrdpiz	v2.06			VSX Scalar Round Double-Precision to Integral toward Zero
111100 ..... ///// ..... 00110 1001..	XX2	I	630	xsrdpip	v2.06			VSX Scalar Round Double-Precision to Integral toward +Infinity
111100 ..... ///// ..... 00111 1001..	XX2	I	630	xsrdpim	v2.06			VSX Scalar Round Double-Precision to Integral toward -Infinity
111100 ..... ///// ..... 01000 1001..	XX2	I	746	xvrspi	v2.06			VSX Vector Round Single-Precision to Integral
111100 ..... ///// ..... 01001 1001..	XX2	I	748	xvrspiz	v2.06			VSX Vector Round Single-Precision to Integral toward Zero
111100 ..... ///// ..... 01010 1001..	XX2	I	747	xvrspip	v2.06			VSX Vector Round Single-Precision to Integral toward +Infinity
111100 ..... ///// ..... 01011 1001..	XX2	I	747	xvrspim	v2.06			VSX Vector Round Single-Precision to Integral toward -Infinity
111100 ..... ///// ..... 01100 1001..	XX2	I	741	xvrdpi	v2.06			VSX Vector Round Double-Precision to Integral
111100 ..... ///// ..... 01101 1001..	XX2	I	743	xvrdpiz	v2.06			VSX Vector Round Double-Precision to Integral toward Zero
111100 ..... ///// ..... 01110 1001..	XX2	I	742	xvrdpip	v2.06			VSX Vector Round Double-Precision to Integral toward +Infinity
111100 ..... ///// ..... 01111 1001..	XX2	I	742	xvrdpim	v2.06			VSX Vector Round Double-Precision to Integral toward -Infinity
111100 ..... ///// ..... 10000 1001..	XX2	I	536	xscvdpsp	v2.06			VSX Scalar Convert with round Double-Precision to Single-Precision format
111100 ..... ///// ..... 10001 1001..	XX2	I	638	xsrsp	v2.07			VSX Scalar Round Double-Precision to Single-Precision
111100 ..... ///// ..... 10100 1001..	XX2	I	557	xscvspdp	v2.06			VSX Scalar Convert Single-Precision to Double-Precision format
111100 ..... ///// ..... 10101 1001..	XX2	I	512	xsabsdp	v2.06			VSX Scalar Absolute Double-Precision
111100 ..... ///// ..... 10110 1001..	XX2	I	606	xsnabsdp	v2.06			VSX Scalar Negative Absolute Double-Precision
111100 ..... ///// ..... 10111 1001..	XX2	I	607	xsnegdp	v2.06			VSX Scalar Negate Double-Precision
111100 ..... ///// ..... 11000 1001..	XX2	I	672	xvcvdpsp	v2.06			VSX Vector Convert with round Double-Precision to Single-Precision format
111100 ..... ///// ..... 11001 1001..	XX2	I	658	xvabssp	v2.06			VSX Vector Absolute Single-Precision
111100 ..... ///// ..... 11010 1001..	XX2	I	725	xvnabssp	v2.06			VSX Vector Negative Absolute Single-Precision
111100 ..... ///// ..... 11011 1001..	XX2	I	726	xvnegsp	v2.06			VSX Vector Negate Single-Precision
111100 ..... ///// ..... 11100 1001..	XX2	I	682	xvcvspdp	v2.06			VSX Vector Convert Single-Precision to Double-Precision format
111100 ..... ///// ..... 11101 1001..	XX2	I	658	xvabsdp	v2.06			VSX Vector Absolute Double-Precision
111100 ..... ///// ..... 11110 1001..	XX2	I	725	xvnabsdp	v2.06			VSX Vector Negative Absolute Double-Precision
111100 ..... ///// ..... 11111 1001..	XX2	I	726	xvnegdp	v2.06			VSX Vector Negate Double-Precision
111100 ..... ///// ..... 00000 1010..	XX2	I	640	xsrsqrtesp	v2.07			VSX Scalar Reciprocal Square Root Estimate Single-Precision
111100 ..... ///// ..... 00001 1010..	XX2	I	633	xsresp	v2.07			VSX Scalar Reciprocal Estimate Single-Precision
111100 ..... ///// ..... 00100 1010..	XX2	I	639	xsrsqrtedp	v2.06			VSX Scalar Reciprocal Square Root Estimate Double-Precision
111100 ..... ///// ..... 00101 1010..	XX2	I	632	xsredp	v2.06			VSX Scalar Reciprocal Estimate Double-Precision
111100 ...// ///// 00110 1010./	XX2	I	652	xstsqrtdp	v2.06			VSX Scalar Test for software Square Root Double-Precision
111100 ...// .... 00111 101../	XX3	I	651	xstdivdp	v2.06			VSX Scalar Test for software Divide Double-Precision
111100 ..... ///// ..... 01000 1010..	XX2	I	750	xvrsqrtesp	v2.06			VSX Vector Reciprocal Square Root Estimate Single-Precision
111100 ..... ///// ..... 01001 1010..	XX2	I	745	xvresp	v2.06			VSX Vector Reciprocal Estimate Single-Precision
111100 ...// ///// 01010 1010./	XX2	I	759	xvtsqrtsp	v2.06			VSX Vector Test for software Square Root Single-Precision
111100 ...// .... 01011 101../	XX3	I	758	xvtdivsp	v2.06			VSX Vector Test for software Divide Single-Precision
111100 ..... ///// ..... 01100 1010..	XX2	I	748	xvrsqrtedp	v2.06			VSX Vector Reciprocal Square Root Estimate Double-Precision
111100 ..... ///// ..... 01101 1010..	XX2	I	744	xvredp	v2.06			VSX Vector Reciprocal Estimate Double-Precision
111100 ...// ///// 01110 1010./	XX2	I	759	xvtsqrtdp	v2.06			VSX Vector Test for software Square Root Double-Precision
111100 ...// .... 01111 101../	XX3	I	757	xvtdivdp	v2.06			VSX Vector Test for software Divide Double-Precision
111100 .... 10010 1010./	XX2	I	655	xststdcsp	v3.0			VSX Scalar Test Data Class Single-Precision
111100 .... 10110 1010./	XX2	I	653	xststdcdp	v3.0			VSX Scalar Test Data Class Double-Precision
111100 .... 1101. 101...	XX2	I	761	xvtstdcsp	v3.0			VSX Vector Test Data Class Single-Precision
111100 .... 1111. 101...	XX2	I	760	xvtstdcdp	v3.0			VSX Vector Test Data Class Double-Precision
111100 ..... ///// ..... 00000 1011..	XX2	I	644	xssqrtsp	v2.07			VSX Scalar Square Root Single-Precision
111100 ..... ///// ..... 00100 1011..	XX2	I	641	xssqrtdp	v2.06			VSX Scalar Square Root Double-Precision
111100 ..... ///// ..... 00110 1011..	XX2	I	629	xsrdpic	v2.06			VSX Scalar Round Double-Precision to Integral using Current rounding mode
111100 ..... ///// ..... 01000 1011..	XX2	I	752	xvsqrtsp	v2.06			VSX Vector Square Root Single-Precision
111100 ..... ///// ..... 01010 1011..	XX2	I	746	xvrspic	v2.06			VSX Vector Round Single-Precision to Integral using Current rounding mode
111100 ..... ///// ..... 01100 1011..	XX2	I	751	xvsqrtdp	v2.06			VSX Vector Square Root Double-Precision
111100 ..... ///// ..... 01110 1011..	XX2	I	741	xvrdpic	v2.06			VSX Vector Round Double-Precision to Integral using Current rounding mode

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 15 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111100 ..... ///// ..... 10000 1011..	XX2	I	537	xscvdpspn	v2.07			VSX Scalar Convert Double-Precision to Single-Precision Non-signalling format
111100 ..... ///// ..... 10100 1011..	XX2	I	558	xscvspdpn	v2.07			VSX Scalar Convert Single-Precision to Double-Precision Non-signalling format
111100 ..... 00000 ..... 10101 1011./	XX2	I	656	xsxexpdp	v3.0			VSX Scalar Extract Exponent Double-Precision
111100 ..... 00001 ..... 10101 1011./	XX2	I	657	xsxsigdp	v3.0			VSX Scalar Extract Significand Double-Precision
111100 ..... 10000 ..... 10101 1011..	XX2	I	546	xscvhpdp	v3.0			VSX Scalar Convert Half-Precision to Double-Precision format
111100 ..... 10001 ..... 10101 1011..	XX2	I	534	xscvdphp	v3.0			VSX Scalar Convert with round Double-Precision to Half-Precision format
111100 .... 11100 10110.	XX1	I	568	xsiexpdp	v3.0			VSX Scalar Insert Exponent Double-Precision
111100 ..... 00000 ..... 11101 1011..	XX2	I	762	xvxexpdp	v3.0			VSX Vector Extract Exponent Double-Precision
111100 ..... 00001 ..... 11101 1011..	XX2	I	763	xvxsigdp	v3.0			VSX Vector Extract Significand Double-Precision
111100 ..... 00111 ..... 11101 1011..	XX2	I	764	xxbrh	v3.0			VSX Vector Byte-Reverse Halfword
111100 ..... 01000 ..... 11101 1011..	XX2	I	762	xvxexpsp	v3.0			VSX Vector Extract Exponent Single-Precision
111100 ..... 01001 ..... 11101 1011..	XX2	I	763	xvxsigsp	v3.0			VSX Vector Extract Significand Single-Precision
111100 ..... 01111 ..... 11101 1011..	XX2	I	765	xxbrw	v3.0			VSX Vector Byte-Reverse Word
111100 ..... 10111 ..... 11101 1011..	XX2	I	764	xxbrd	v3.0			VSX Vector Byte-Reverse Doubleword
111100 ..... 11000 ..... 11101 1011..	XX2	I	681	xvcvhpsp	v3.0			VSX Vector Convert Half-Precision to Single-Precision format
111100 ..... 11001 ..... 11101 1011..	XX2	I	683	xvcvsphp	v3.0			VSX Vector Convert with round Single-Precision to Half-Precision format
111100 ..... 11111 ..... 11101 1011..	XX2	I	765	xxbrq	v3.0			VSX Vector Byte-Reverse Quadword
111100 .... 11....	XX4	I	773	xxsel	v2.06			VSX Vector Select
111101 00	DS	I	149	stfdp	v2.05			Store Floating Double Pair
111101 001	DQ	I	492	lxv	v3.0			Load VSX Vector
111101 10	DS	I	498	stxsd	v3.0			Store VSX Scalar Doubleword
111101 11	DS	I	501	stxssp	v3.0			Store VSX Scalar Single-Precision
111101 101	DQ	I	507	stxv	v3.0			Store VSX Vector
111110 00	DS	I	57	std	PPC			Store Doubleword
111110 01	DS	I	57	stdu	PPC			Store Doubleword with Update
111110 10	DS	I	59	stq	v2.03			Store Quadword
111111 .../// 00000 00000/	X	I	167	fcmpu	P1			Floating Compare Unordered
111111 .../// 00001 00000/	X	I	167	fcmpo	P1			Floating Compare Ordered
111111 .../// .../// ///// 00010 00000/	X	I	171	mcrfs	P1			Move To CR from FPSCR
111111 .../// 00100 00000/	X	I	156	ftdiv	v2.06			Floating Test for software Divide
111111 .../// ///// 00101 00000/	X	I	156	ftsqrt	v2.06			Floating Test for software Square Root
111111 .... 00000 00010.	X	I	193	daddq[.]	v2.05			DFP Add Quad
111111 .... 00001 00010.	X	I	195	dmulq[.]	v2.05			DFP Multiply Quad
111111 .... .0010 00010.	Z22	I	220	dscliq[.]	v2.05			DFP Shift Significand Left Immediate Quad
111111 .... .0011 00010.	Z22	I	220	dscriq[.]	v2.05			DFP Shift Significand Right Immediate Quad
111111 .../// 00100 00010/	X	I	199	dcmpoq	v2.05			DFP Compare Ordered Quad
111111 .../// 00101 00010/	X	I	201	dtstexq	v2.05			DFP Test Exponent Quad
111111 .../// .... .0110 00010/	Z22	I	200	dtstdcq	v2.05			DFP Test Data Class Quad
111111 .../// .... .0111 00010/	Z22	I	200	dtstdgq	v2.05			DFP Test Data Group Quad
111111 ..... ///// ..... 01000 00010.	X	I	213	dctqpq[.]	v2.05			DFP Convert To DFP Extended
111111 ..... ///// ..... 01001 00010.	X	I	215	dctfixq[.]	v2.05			DFP Convert To Fixed Quad
111111 ..../// ..... 01010 00010.	X	I	217	ddedpdq[.]	v2.05			DFP Decode DPD To BCD Quad
111111 ..... ///// ..... 01011 00010.	X	I	218	dxexq[.]	v2.05			DFP Extract Exponent Quad
111111 .... 10000 00010.	X	I	193	dsubq[.]	v2.05			DFP Subtract Quad
111111 .... 10001 00010.	X	I	196	ddivq[.]	v2.05			DFP Divide Quad
111111 .../// 10100 00010/	X	I	198	dcmpuq	v2.05			DFP Compare Unordered Quad
111111 .../// 10101 00010/	X	I	202	dtstsfq	v2.05			DFP Test Significance Quad
111111 ..... ///// ..... 11000 00010.	X	I	214	drdpq[.]	v2.05			DFP Round To DFP Long
111111 ..... ///// ..... 11001 00010.	X	I	215	dcffixq[.]	v2.05			DFP Convert From Fixed Quad
111111 ..... .//// ..... 11010 00010.	X	I	217	denbcdq[.]	v2.05			DFP Encode BCD To DPD Quad
111111 .... 11011 00010.	X	I	218	diexq[.]	v2.05			DFP Insert Exponent Quad
111111 ....000 00011.	Z23	I	204	dquaq[.]	v2.05			DFP Quantize Quad
111111 ....001 00011.	Z23	I	206	drrndq[.]	v2.05			DFP Reround Quad

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 16 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111111 ....010 00011.	Z23	I	203	dquaiq[.]	v2.05			DFP Quantize Immediate Quad
111111 ..... ////. ....011 00011.	Z23	I	209	drintxq[.]	v2.05			DFP Round To FP Integer With Inexact Quad
111111 ..... ////. ....111 00011.	Z23	I	211	drintnq[.]	v2.05			DFP Round To FP Integer Without Inexact Quad
111111 ...// 10101 00011/	X	I	202	dtstsfiq	v3.0			DFP Test Significance Immediate Quad
111111 .... 00000 00100.	X	I	520	xsaddqp[o]	v3.0			VSX Scalar Add Quad-Precision [with round to Odd]
111111 .... 00001 00100.	X	I	602	xsmulqp[o]	v3.0			VSX Scalar Multiply Quad-Precision [with round to Odd]
111111 .... 00011 00100/	X	I	533	xscpsgnqp	v3.0			VSX Scalar Copy Sign Quad-Precision
111111 ...// 00100 00100/	X	I	529	xscmpoqp	v3.0			VSX Scalar Compare Ordered Quad-Precision
111111 ...// 00101 00100/	X	I	523	xscmpexpqp	v3.0			VSX Scalar Compare Exponents Quad-Precision
111111 .... 01100 00100.	X	I	576	xsmaddqp[o]	v3.0			VSX Scalar Multiply-Add Quad-Precision [with round to Odd]
111111 .... 01101 00100.	X	I	597	xsmsubqp[o]	v3.0			VSX Scalar Multiply-Subtract Quad-Precision [with round to Odd]
111111 .... 01110 00100.	X	I	616	xsnmaddqp[o]	v3.0			VSX Scalar Negative Multiply-Add Quad-Precision [with round to Odd]
111111 .... 01111 00100.	X	I	625	xsnmsubqp[o]	v3.0			VSX Scalar Negative Multiply-Subtract Quad-Precision [with round to Odd]
111111 .... 10000 00100.	X	I	647	xssubqp[o]	v3.0			VSX Scalar Subtract Quad-Precision [with round to Odd]
111111 .... 10001 00100.	X	I	564	xsdivqp[o]	v3.0			VSX Scalar Divide Quad-Precision [with round to Odd]
111111 ...// 10100 00100/	X	I	532	xscmpuqp	v3.0			VSX Scalar Compare Unordered Quad-Precision
111111 .... 10110 00100/	X	I	654	xststdcqp	v3.0			VSX Scalar Test Data Class Quad-Precision
111111 ..... 00000 ..... 11001 00100/	X	I	512	xsabsqp	v3.0			VSX Scalar Absolute Quad-Precision
111111 ..... 00010 ..... 11001 00100/	X	I	656	xsxexpqp	v3.0			VSX Scalar Extract Exponent Quad-Precision
111111 ..... 01000 ..... 11001 00100/	X	I	606	xsnabsqp	v3.0			VSX Scalar Negative Absolute Quad-Precision
111111 ..... 10000 ..... 11001 00100/	X	I	607	xsnegqp	v3.0			VSX Scalar Negate Quad-Precision
111111 ..... 10010 ..... 11001 00100/	X	I	657	xsxsigqp	v3.0			VSX Scalar Extract Significand Quad-Precision
111111 ..... 11011 ..... 11001 00100.	X	I	642	xssqrtqp[o]	v3.0			VSX Scalar Square Root Quad-Precision [with round to Odd]
111111 ..... 00001 ..... 11010 00100/	X	I	554	xscvqpuwz	v3.0			VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format
111111 ..... 00010 ..... 11010 00100/	X	I	560	xscvudqp	v3.0			VSX Scalar Convert Unsigned Doubleword to Quad-Precision format
111111 ..... 01001 ..... 11010 00100/	X	I	550	xscvqpswz	v3.0			VSX Scalar Convert with round to zero Quad-Precision to Signed Word format
111111 ..... 01010 ..... 11010 00100/	X	I	556	xscvsdqp	v3.0			VSX Scalar Convert Signed Doubleword to Quad-Precision format
111111 ..... 10001 ..... 11010 00100/	X	I	552	xscvqpudz	v3.0			VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format
111111 ..... 10100 ..... 11010 00100.	X	I	547	xscvqpdp[o]	v3.0			VSX Scalar Convert with round Quad-Precision to Double-Precision format [with round to Odd]
111111 ..... 10110 ..... 11010 00100/	X	I	535	xscvdpqp	v3.0			VSX Scalar Convert Double-Precision to Quad-Precision format
111111 ..... 11001 ..... 11010 00100/	X	I	548	xscvqpsdz	v3.0			VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format
111111 .... 11011 00100/	X	I	569	xsiexpqp	v3.0			VSX Scalar Insert Exponent Quad-Precision
111111 ..... ////. ....000 00101.	X	I	634	xsrqpi[x]	v3.0			VSX Scalar Round Quad-Precision to Integral [Exact]
111111 ..... ////. 001 00101/	X	I	636	xsrqpxp	v3.0			VSX Scalar Round Quad-Precision to XP
111111 ..... ///// ///// 00001 00110.	X	I	173	mtfsb1[.]	P1			Move To FPSCR Bit 1
111111 ..... ///// ///// 00010 00110.	X	I	173	mtfsb0[.]	P1			Move To FPSCR Bit 0
111111 ...// ////. ..../ 00100 00110.	X	I	172	mtfsfi[.]	P1			Move To FPSCR Field Immediate
111111 .... 11010 00110/	X	I	151	fmrgow	v2.07			Floating Merge Odd Word
111111 .... 11110 00110/	X	I	151	fmrgew	v2.07			Floating Merge Even Word
111111 ..... 00000 ///// 10010 00111.	X	I	170	mffs[.]	P1			Move From FPSCR
111111 ..... 00001 ///// 10010 00111/	X	I	170	mffsce	v3.0B			Move From FPSCR & Clear Enables
111111 ..... 10100 ..... 10010 00111/	X	I	170	mffscdrn	v3.0B			Move From FPSCR Control & set DRN
111111 ..... 10101 //... 10010 00111/	X	I	170	mffscdrni	v3.0B			Move From FPSCR Control & set DRN Immediate
111111 ..... 10110 ..... 10010 00111/	X	I	170	mffscrn	v3.0B			Move From FPSCR Control & set RN
111111 ..... 10111 ///.. 10010 00111/	X	I	170	mffscrni	v3.0B			Move From FPSCR Control & set RN Immediate
111111 ..... 11000 ///// 10010 00111/	X	I	170	mffsl	v3.0B			Move From FPSCR Lightweight
111111 .... 10110 00111.	XFL	I	172	mtfsf[.]	P1			Move To FPSCR Fields
111111 .... 00000 01000.	X	I	150	fcpsgn[.]	v2.05			Floating Copy Sign
111111 ..... ///// ..... 00001 01000.	X	I	150	fneg[.]	P1			Floating Negate

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 17 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111111 ..... ///// ..... 00010 01000.	X	I	150	fmr[.]	P1			Floating Move Register
111111 ..... ///// ..... 00100 01000.	X	I	150	fnabs[.]	P1			Floating Negative Absolute Value
111111 ..... ///// ..... 01000 01000.	X	I	150	fabs[.]	P1			Floating Absolute
111111 ..... ///// ..... 01100 01000.	X	I	166	frin[.]	v2.02			Floating Round To Integer Nearest
111111 ..... ///// ..... 01101 01000.	X	I	166	friz[.]	v2.02			Floating Round To Integer Zero
111111 ..... ///// ..... 01110 01000.	X	I	166	frip[.]	v2.02			Floating Round To Integer Plus
111111 ..... ///// ..... 01111 01000.	X	I	166	frim[.]	v2.02			Floating Round To Integer Minus
111111 ..... ///// ..... 00000 01100.	X	I	159	frsp[.]	P1			Floating Round to Single-Precision
111111 ..... ///// ..... 00000 01110.	X	I	161	fctiw[.]	P2			Floating Convert with round Double-Precision To Signed Word format
111111 ..... ///// ..... 00100 01110.	X	I	162	fctiwu[.]	v2.06			Floating Convert with round Double-Precision To Unsigned Word format
111111 ..... ///// ..... 11001 01110.	X	I	159	fctid[.]	PPC			Floating Convert with round Double-Precision To Signed Doubleword format
111111 ..... ///// ..... 11010 01110.	X	I	163	fcfid[.]	PPC			Floating Convert with round Signed Doubleword to Double-Precision format
111111 ..... ///// ..... 11101 01110.	X	I	160	fctidu[.]	v2.06			Floating Convert with round Double-Precision To Unsigned Doubleword format
111111 ..... ///// ..... 11110 01110.	X	I	164	fcfidu[.]	v2.06			Floating Convert with round Unsigned Doubleword to Double-Precision format
111111 ..... ///// ..... 00000 01111.	X	I	162	fctiwz[.]	P2			Floating Convert with round to Zero Double-Precision To Signed Word format
111111 ..... ///// ..... 00100 01111.	X	I	163	fctiwuz[.]	v2.06			Floating Convert with round to Zero Double-Precision To Unsigned Word format
111111 ..... ///// ..... 11001 01111.	X	I	160	fctidz[.]	PPC			Floating Convert with round to Zero Double-Precision To Signed Doubleword format
111111 ..... ///// ..... 11101 01111.	X	I	161	fctiduz[.]	v2.06			Floating Convert with round to Zero Double-Precision To Unsigned Doubleword format
111111 .... ///// 10010.	A	I	153	fdiv[.]	P1			Floating Divide
111111 .... ///// 10100.	A	I	152	fsub[.]	P1			Floating Subtract
111111 .... ///// 10101.	A	I	152	fadd[.]	P1			Floating Add
111111 ..... ///// ..... ///// 10110.	A	I	154	fsqrt[.]	P2			Floating Square Root
111111 .... 10111.	A	I	168	fsel[.]	PPC			Floating Select
111111 ..... ///// ..... ///// 11000.	A	I	154	fre[.]	v2.02			Floating Reciprocal Estimate
111111 .... ///// ..... 11001.	A	I	153	fmul[.]	P1			Floating Multiply
111111 ..... ///// ..... ///// 11010.	A	I	155	frsqrte[.]	PPC			Floating Reciprocal Square Root Estimate
111111 .... 11100.	A	I	158	fmsub[.]	P1			Floating Multiply-Subtract
111111 .... 11101.	A	I	157	fmadd[.]	P1			Floating Multiply-Add
111111 .... 11110.	A	I	158	fnmsub[.]	P1			Floating Negative Multiply-Subtract
111111 .... 11111.	A	I	158	fnmadd[.]	P1			Floating Negative Multiply-Add

Figure 88. Power ISA AS Instruction Set Sorted by Opcode (Sheet 18 of 18)

1. Key to Instruction column.

/	Instruction bit that corresponds to a reserved field, must have a value of 0, otherwise invalid form.
-	Instruction bit that corresponds to an operand bit, may have a value of either 0 or 1.
0	Instruction bit having a value 0.
1	Instruction bit having a value 1.

2. Key to Version column.

P1	Instruction introduced in the POWER Architecture.
P2	Instruction introduced in the POWER2 Architecture.
PPC	Instruction introduced in the PowerPC Architecture prior to v2.00.
v2.00	Instruction introduced in the PowerPC Architecture Version 2.00.
v2.01	Instruction introduced in the PowerPC Architecture Version 2.01.
v2.02	Instruction introduced in the PowerPC Architecture Version 2.02.
v2.03	Instruction introduced in the Power ISA Architecture Version 2.03.
v2.04	Instruction introduced in the Power ISA Architecture Version 2.04.
v2.05	Instruction introduced in the Power ISA Architecture Version 2.05.
v2.06	Instruction introduced in the Power ISA Architecture Version 2.06.
v2.07	Instruction introduced in the Power ISA Architecture Version 2.07.
v3.0	Instruction introduced in the Power ISA Architecture Version 3.0.
v3.0B	Instruction introduced in the Power ISA Architecture Version 3.0B.

3. Key to Privilege column.

P	Denotes an instruction that is treated as privileged.
O	Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor, for mtspr), depending on the SPR or PMR number.
PI	Denotes an instruction that is illegal in privileged state.
H	Denotes an instruction that can be executed only in hypervisor state
U	Denotes an instruction that can be executed only in ultravisor state

4. Key to Mode Dependency column.
   Except as described below and in Section 1.11.3, “Effective Address Calculation”, in Book I, all instructions are independent of whether the processor is in 32-bit or 64-bit mode.

CT	If the instruction tests the Count Register, it tests the low-order 32 bits in 32-bit mode and all 64 bits in 64-bit mode.
SR	The setting of status registers (such as XER and CR0) is mode-dependent.
32	The instruction can be executed only in 32-bit mode.
64	The instruction can be executed only in 64-bit mode.

Appendix E. Power ISA Instruction Set Sorted by Version

This appendix lists all the instructions in the Power ISA, sorted in reverse order by ISA version.

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
011111 ....101 01010/	X	I	72	addex	v3.0B			Add Extended using alternate carry
111111 ..... 10100 ..... 10010 00111/	X	I	170	mffscdrn	v3.0B			Move From FPSCR Control & set DRN
111111 ..... 10101 //... 10010 00111/	X	I	170	mffscdrni	v3.0B			Move From FPSCR Control & set DRN Immediate
111111 ..... 00001 ///// 10010 00111/	X	I	170	mffsce	v3.0B			Move From FPSCR & Clear Enables
111111 ..... 10110 ..... 10010 00111/	X	I	170	mffscrn	v3.0B			Move From FPSCR Control & set RN
111111 ..... 10111 ///.. 10010 00111/	X	I	170	mffscrni	v3.0B			Move From FPSCR Control & set RN Immediate
111111 ..... 11000 ///// 10010 00111/	X	I	170	mffsl	v3.0B			Move From FPSCR Lightweight
011111 ..... ///// 11010 10010/	X	III	1028	slbiag	v3.0B	P		SLB Invalidate All Global
000100 100011	VA	I	289	vmsumudm	v3.0B			Vector Multiply-Sum Unsigned Doubleword Modulo
010011 .... 00010.	DX	I	68	addpcis	v3.0			Add PC Immediate Shifted
000100 ..... 00111 ..... 1.110 000001	VX	I	350	bcdcfn.	v3.0			Decimal Convert From National & record
000100 ..... 00010 ..... 1.110 000001	VX	I	354	bcdcfsq.	v3.0			Decimal Convert From Signed Quadword & record
000100 ..... 00110 ..... 1.110 000001	VX	I	351	bcdcfz.	v3.0			Decimal Convert From Zoned & record
000100 .... 01101 000001	VX	I	356	bcdcpsgn.	v3.0			Decimal CopySign & record
000100 ..... 00101 ..... 1/110 000001	VX	I	352	bcdctn.	v3.0			Decimal Convert To National & record
000100 ..... 00000 ..... 1/110 000001	VX	I	354	bcdctsq.	v3.0			Decimal Convert To Signed Quadword & record
000100 ..... 00100 ..... 1.110 000001	VX	I	353	bcdctz.	v3.0			Decimal Convert To Zoned & record
000100 .... 1.011 000001	VX	I	357	bcds.	v3.0			Decimal Shift & record
000100 ..... 11111 ..... 1.110 000001	VX	I	356	bcdsetsgn.	v3.0			Decimal Set Sign & record
000100 .... 1.111 000001	VX	I	359	bcdsr.	v3.0			Decimal Shift & Round & record
000100 .... 1.100 000001	VX	I	360	bcdtrunc.	v3.0			Decimal Truncate & record
000100 .... 1/010 000001	VX	I	358	bcdus.	v3.0			Decimal Unsigned Shift & record
000100 .... 1/101 000001	VX	I	361	bcdutrunc.	v3.0			Decimal Unsigned Truncate & record
011111 .../// 00111 00000/	X	I	88	cmpeqb	v3.0			Compare Equal Byte
011111 .../. 00110 00000/	X	I	87	cmprb	v3.0			Compare Ranged Byte
011111 .... ///// 10001 11010.	X	I	99	cnttzd[.]	v3.0			Count Trailing Zeros Doubleword
011111 .... ///// 10000 11010.	X	I	96	cnttzw[.]	v3.0			Count Trailing Zeros Word
011111 ////. 11000 00110/	X	II	855	copy	v3.0			Copy
011111 ///// ///// ///// 11010 00110/	X	II	856	cp_abort	v3.0			CP_Abort
011111 ..... ///.. ///// 10111 10011/	X	I	78	darn	v3.0			Deliver A Random Number
111011 .../// 10101 00011/	X	I	202	dtstsfi	v3.0			DFP Test Significance Immediate
111111 .../// 10101 00011/	X	I	202	dtstsfiq	v3.0			DFP Test Significance Immediate Quad
011111 .... 11011 1101..	XS	I	110	extswsli[.]	v3.0			Extend Sign Word & Shift Left Immediate
011111 .... 10011 00110/	X	II	860	ldat	v3.0			Load Doubleword ATomic
011111 .... 10010 00110/	X	II	860	lwat	v3.0			Load Word ATomic
111001 10	DS	I	480	lxsd	v3.0			Load VSX Scalar Doubleword
011111 .... 11000 01101.	X	I	482	lxsibzx	v3.0			Load VSX Scalar as Integer Byte & Zero Indexed
011111 .... 11001 01101.	X	I	482	lxsihzx	v3.0			Load VSX Scalar as Integer Halfword & Zero Indexed
111001 11	DS	I	485	lxssp	v3.0			Load VSX Scalar Single
111101 001	DQ	I	492	lxv	v3.0			Load VSX Vector
011111 .... 11011 01100.	X	I	487	lxvb16x	v3.0			Load VSX Vector Byte*16 Indexed

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 1 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
011111 .... 11001 01100.	X	I	495	lxvh8x	v3.0			Load VSX Vector Halfword*8 Indexed
011111 .... 01000 01101.	X	I	489	lxvl	v3.0			Load VSX Vector with Length
011111 .... 01001 01101.	X	I	491	lxvll	v3.0			Load VSX Vector Left-justified with Length
011111 .... 01011 01100.	X	I	497	lxvwsx	v3.0			Load VSX Vector Word & Splat Indexed
011111 .... 01000 01100.	X	I	492	lxvx	v3.0			Load VSX Vector Indexed
000100 110000	VA	I	80	maddhd	v3.0			Multiply-Add High Doubleword
000100 110001	VA	I	80	maddhdu	v3.0			Multiply-Add High Doubleword Unsigned
000100 110011	VA	I	80	maddld	v3.0			Multiply-Add Low Doubleword
011111 ...// ///// ///// 10010 00000/	X	I	120	mcrxrx	v3.0			Move XER to CR Extended
011111 .... ///// 01001 10011.	XX1	I	112	mfvsrld	v3.0			Move From VSR Lower Doubleword
011111 .... 11000 01001/	X	I	83	modsd	v3.0			Modulo Signed Doubleword
011111 .... 11000 01011/	X	I	77	modsw	v3.0			Modulo Signed Word
011111 .... 01000 01001/	X	I	83	modud	v3.0			Modulo Unsigned Doubleword
011111 .... 01000 01011/	X	I	77	moduw	v3.0			Modulo Unsigned Word
011111 ///// ///// ///// 11011 10110/	X	III	1132	msgsync	v3.0	HV		Message Synchronize
011111 .... 01101 10011.	XX1	I	115	mtvsrdd	v3.0			Move To VSR Double Doubleword
011111 .... ///// 01100 10011.	XX1	I	116	mtvsrws	v3.0			Move To VSR Word & Splat
011111 ////. .... 11100 00110.	X	II	855	paste[.]	v3.0			Paste
010011 ///// ///// ///// 00010 10010/	XL	III	953	rfscv	v3.0	P		Return From System Call Vectored
010001 ///// ///// ////. ..... .///01	SC	I	42	scv	v3.0			System Call Vectored
011111 ....// ///// 00100 000000	VX	I	122	setb	v3.0			Set Boolean
011111 ..... ///// 01110 10010/	X	III	1025	slbieg	v3.0	P		SLB Invalidate Entry Global
011111 ///// ///// ///// 01010 10010/	X	III	1032	slbsync	v3.0	P		SLB Synchronize
011111 .... 10111 00110/	X	II	862	stdat	v3.0			Store Doubleword ATomic
010011 ///// ///// ///// 01011 10010/	XL	III	958	stop	v3.0	P		Stop
011111 .... 10110 00110/	X	II	862	stwat	v3.0			Store Word ATomic
111101 10	DS	I	498	stxsd	v3.0			Store VSX Scalar Doubleword
011111 .... 11100 01101.	X	I	499	stxsibx	v3.0			Store VSX Scalar as Integer Byte Indexed
011111 .... 11101 01101.	X	I	499	stxsihx	v3.0			Store VSX Scalar as Integer Halfword Indexed
111101 11	DS	I	501	stxssp	v3.0			Store VSX Scalar Single-Precision
111101 101	DQ	I	507	stxv	v3.0			Store VSX Vector
011111 .... 11111 01100.	X	I	503	stxvb16x	v3.0			Store VSX Vector Byte*16 Indexed
011111 .... 11101 01100.	X	I	505	stxvh8x	v3.0			Store VSX Vector Halfword*8 Indexed
011111 .... 01100 01101.	X	I	507	stxvl	v3.0			Store VSX Vector with Length
011111 .... 01101 01101.	X	I	509	stxvll	v3.0			Store VSX Vector Left-justified with Length
011111 .... 01100 01100.	X	I	510	stxvx	v3.0			Store VSX Vector Indexed
000100 .... 10000 000011	VX	I	297	vabsdub	v3.0			Vector Absolute Difference Unsigned Byte
000100 .... 10001 000011	VX	I	297	vabsduh	v3.0			Vector Absolute Difference Unsigned Halfword
000100 .... 10010 000011	VX	I	298	vabsduw	v3.0			Vector Absolute Difference Unsigned Word
000100 .... 10111 001100	VX	I	346	vbpermd	v3.0			Vector Bit Permute Doubleword
000100 ..... 00000 ..... 11000 000010	VX	I	342	vclzlsbb	v3.0			Vector Count Leading Zero Least-Significant Bits Byte
000100 .... .0000 000111	VC	I	309	vcmpneb[.]	v3.0			Vector Compare Not Equal Byte
000100 .... .0001 000111	VC	I	310	vcmpneh[.]	v3.0			Vector Compare Not Equal Halfword
000100 .... .0010 000111	VC	I	311	vcmpnew[.]	v3.0			Vector Compare Not Equal Word
000100 .... .0100 000111	VC	I	309	vcmpnezb[.]	v3.0			Vector Compare Not Equal or Zero Byte
000100 .... .0101 000111	VC	I	310	vcmpnezh[.]	v3.0			Vector Compare Not Equal or Zero Halfword
000100 .... .0110 000111	VC	I	311	vcmpnezw[.]	v3.0			Vector Compare Not Equal or Zero Word
000100 ..... 11100 ..... 11000 000010	VX	I	341	vctzb	v3.0			Vector Count Trailing Zeros Byte
000100 ..... 11111 ..... 11000 000010	VX	I	341	vctzd	v3.0			Vector Count Trailing Zeros Doubleword
000100 ..... 11101 ..... 11000 000010	VX	I	341	vctzh	v3.0			Vector Count Trailing Zeros Halfword
000100 ..... 00001 ..... 11000 000010	VX	I	342	vctzlsbb	v3.0			Vector Count Trailing Zero Least-Significant Bits Byte
000100 ..... 11110 ..... 11000 000010	VX	I	341	vctzw	v3.0			Vector Count Trailing Zeros Word
000100 ..... /.... ..... 01011 001101	VX	I	267	vextractd	v3.0			Vector Extract Doubleword
000100 ..... /.... ..... 01000 001101	VX	I	267	vextractub	v3.0			Vector Extract Unsigned Byte
000100 ..... /.... ..... 01001 001101	VX	I	267	vextractuh	v3.0			Vector Extract Unsigned Halfword
000100 ..... /.... ..... 01010 001101	VX	I	267	vextractuw	v3.0			Vector Extract Unsigned Word

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 2 of 18)

Instruction[1] 0:5	6:10	11:15	16:20	21:25	26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
000100	.....	11000	.....	11000	000010	VX	I	294	vextsb2d	v3.0			Vector Extend Sign Byte to Doubleword
000100	.....	10000	.....	11000	000010	VX	I	294	vextsb2w	v3.0			Vector Extend Sign Byte to Word
000100	.....	11001	.....	11000	000010	VX	I	294	vextsh2d	v3.0			Vector Extend Sign Halfword to Doubleword
000100	.....	10001	.....	11000	000010	VX	I	294	vextsh2w	v3.0			Vector Extend Sign Halfword to Word
000100	.....	11010	.....	11000	000010	VX	I	294	vextsw2d	v3.0			Vector Extend Sign Word to Doubleword
000100	.....	.....	.....	11000	001101	VX	I	343	vextublx	v3.0			Vector Extract Unsigned Byte Left-Indexed
000100	.....	.....	.....	11100	001101	VX	I	343	vextubrx	v3.0			Vector Extract Unsigned Byte Right-Indexed
000100	.....	.....	.....	11001	001101	VX	I	343	vextuhlx	v3.0			Vector Extract Unsigned Halfword Left-Indexed
000100	.....	.....	.....	11101	001101	VX	I	343	vextuhrx	v3.0			Vector Extract Unsigned Halfword Right-Indexed
000100	.....	.....	.....	11010	001101	VX	I	344	vextuwlx	v3.0			Vector Extract Unsigned Word Left-Indexed
000100	.....	.....	.....	11110	001101	VX	I	344	vextuwrx	v3.0			Vector Extract Unsigned Word Right-Indexed
000100	.....	/....	.....	01100	001101	VX	I	268	vinsertb	v3.0			Vector Insert Byte
000100	.....	/....	.....	01111	001101	VX	I	268	vinsertd	v3.0			Vector Insert Doubleword
000100	.....	/....	.....	01101	001101	VX	I	268	vinserth	v3.0			Vector Insert Halfword
000100	.....	/....	.....	01110	001101	VX	I	268	vinsertw	v3.0			Vector Insert Word
000100	.....	.....	/////	00000	000001	VX	I	355	vmul10cuq	v3.0			Vector Multiply-by-10 & write Carry Unsigned Quadword
000100	.....	.....	.....	00001	000001	VX	I	355	vmul10ecuq	v3.0			Vector Multiply-by-10 Extended & write Carry Unsigned Quadword
000100	.....	.....	.....	01001	000001	VX	I	355	vmul10euq	v3.0			Vector Multiply-by-10 Extended Unsigned Quadword
000100	.....	.....	/////	01000	000001	VX	I	355	vmul10uq	v3.0			Vector Multiply-by-10 Unsigned Quadword
000100	.....	00111	.....	11000	000010	VX	I	293	vnegd	v3.0			Vector Negate Doubleword
000100	.....	00110	.....	11000	000010	VX	I	293	vnegw	v3.0			Vector Negate Word
000100	.....	.....	.....	.....	111011	VA	I	260	vpermr	v3.0			Vector Permute Right-indexed
000100	.....	01001	.....	11000	000010	VX	I	314	vprtybd	v3.0			Vector Parity Byte Doubleword
000100	.....	01010	.....	11000	000010	VX	I	314	vprtybq	v3.0			Vector Parity Byte Quadword
000100	.....	01000	.....	11000	000010	VX	I	314	vprtybw	v3.0			Vector Parity Byte Word
000100	.....	.....	.....	00011	000101	VX	I	320	vrldmi	v3.0			Vector Rotate Left Doubleword then Mask Insert
000100	.....	.....	.....	00111	000101	VX	I	320	vrldnm	v3.0			Vector Rotate Left Doubleword then AND with Mask
000100	.....	.....	.....	00010	000101	VX	I	319	vrlwmi	v3.0			Vector Rotate Left Word then Mask Insert
000100	.....	.....	.....	00110	000101	VX	I	319	vrlwnm	v3.0			Vector Rotate Left Word then AND with Mask
000100	.....	.....	.....	11101	000100	VX	I	265	vslv	v3.0			Vector Shift Left Variable
000100	.....	.....	.....	11100	000100	VX	I	265	vsrv	v3.0			Vector Shift Right Variable
011111	///..	/////	/////	00000	11110/	X	II	876	wait	v3.0			Wait for Interrupt
111111	.....	00000	.....	11001	00100/	X	I	512	xsabsqp	v3.0			VSX Scalar Absolute Quad-Precision
111111	.....	.....	.....	00000	00100.	X	I	520	xsaddqp[o]	v3.0			VSX Scalar Add Quad-Precision [with round to Odd]
111100	.....	.....	.....	00000	011...	XX3	I	524	xscmpeqdp	v3.0			VSX Scalar Compare Equal Double-Precision
111100	...//	.....	.....	00111	011../	XX3	I	522	xscmpexpdp	v3.0			VSX Scalar Compare Exponents Double-Precision
111111	...//	.....	.....	00101	00100/	X	I	523	xscmpexpqp	v3.0			VSX Scalar Compare Exponents Quad-Precision
111100	.....	.....	.....	00010	011...	XX3	I	525	xscmpgedp	v3.0			VSX Scalar Compare Greater Than or Equal Double-Precision
111100	.....	.....	.....	00001	011...	XX3	I	526	xscmpgtdp	v3.0			VSX Scalar Compare Greater Than Double-Precision
111111	...//	.....	.....	00100	00100/	X	I	529	xscmpoqp	v3.0			VSX Scalar Compare Ordered Quad-Precision
111111	...//	.....	.....	10100	00100/	X	I	532	xscmpuqp	v3.0			VSX Scalar Compare Unordered Quad-Precision
111111	.....	.....	.....	00011	00100/	X	I	533	xscpsgnqp	v3.0			VSX Scalar Copy Sign Quad-Precision
111100	.....	10001	.....	10101	1011..	XX2	I	534	xscvdphp	v3.0			VSX Scalar Convert with round Double-Precision to Half-Precision format
111111	.....	10110	.....	11010	00100/	X	I	535	xscvdpqp	v3.0			VSX Scalar Convert Double-Precision to Quad-Precision format
111100	.....	10000	.....	10101	1011..	XX2	I	546	xscvhpdp	v3.0			VSX Scalar Convert Half-Precision to Double-Precision format
111111	.....	10100	.....	11010	00100.	X	I	547	xscvqpdp[o]	v3.0			VSX Scalar Convert with round Quad-Precision to Double-Precision format [with round to Odd]
111111	.....	11001	.....	11010	00100/	X	I	548	xscvqpsdz	v3.0			VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format
111111	.....	01001	.....	11010	00100/	X	I	550	xscvqpswz	v3.0			VSX Scalar Convert with round to zero Quad-Precision to Signed Word format
111111	.....	10001	.....	11010	00100/	X	I	552	xscvqpudz	v3.0			VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format
111111	.....	00001	.....	11010	00100/	X	I	554	xscvqpuwz	v3.0			VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 3 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111111 ..... 01010 ..... 11010 00100/	X	I	556	xscvsdqp	v3.0			VSX Scalar Convert Signed Doubleword to Quad-Precision format
111111 ..... 00010 ..... 11010 00100/	X	I	560	xscvudqp	v3.0			VSX Scalar Convert Unsigned Doubleword to Quad-Precision format
111111 .... 10001 00100.	X	I	564	xsdivqp[o]	v3.0			VSX Scalar Divide Quad-Precision [with round to Odd]
111100 .... 11100 10110.	XX1	I	568	xsiexpdp	v3.0			VSX Scalar Insert Exponent Double-Precision
111111 .... 11011 00100/	X	I	569	xsiexpqp	v3.0			VSX Scalar Insert Exponent Quad-Precision
111111 .... 01100 00100.	X	I	576	xsmaddqp[o]	v3.0			VSX Scalar Multiply-Add Quad-Precision [with round to Odd]
111100 .... 10000 000...	XX3	I	581	xsmaxcdp	v3.0			VSX Scalar Maximum Type-C Double-Precision
111100 .... 10010 000...	XX3	I	583	xsmaxjdp	v3.0			VSX Scalar Maximum Type-J Double-Precision
111100 .... 10001 000...	XX3	I	587	xsmincdp	v3.0			VSX Scalar Minimum Type-C Double-Precision
111100 .... 10011 000...	XX3	I	589	xsminjdp	v3.0			VSX Scalar Minimum Type-J Double-Precision
111111 .... 01101 00100.	X	I	597	xsmsubqp[o]	v3.0			VSX Scalar Multiply-Subtract Quad-Precision [with round to Odd]
111111 .... 00001 00100.	X	I	602	xsmulqp[o]	v3.0			VSX Scalar Multiply Quad-Precision [with round to Odd]
111111 ..... 01000 ..... 11001 00100/	X	I	606	xsnabsqp	v3.0			VSX Scalar Negative Absolute Quad-Precision
111111 ..... 10000 ..... 11001 00100/	X	I	607	xsnegqp	v3.0			VSX Scalar Negate Quad-Precision
111111 .... 01110 00100.	X	I	616	xsnmaddqp[o]	v3.0			VSX Scalar Negative Multiply-Add Quad-Precision [with round to Odd]
111111 .... 01111 00100.	X	I	625	xsnmsubqp[o]	v3.0			VSX Scalar Negative Multiply-Subtract Quad-Precision [with round to Odd]
111111 ..... ////. ....000 00101.	X	I	634	xsrqpi[x]	v3.0			VSX Scalar Round Quad-Precision to Integral [Exact]
111111 ..... ////. 001 00101/	X	I	636	xsrqpxp	v3.0			VSX Scalar Round Quad-Precision to XP
111111 ..... 11011 ..... 11001 00100.	X	I	642	xssqrtqp[o]	v3.0			VSX Scalar Square Root Quad-Precision [with round to Odd]
111111 .... 10000 00100.	X	I	647	xssubqp[o]	v3.0			VSX Scalar Subtract Quad-Precision [with round to Odd]
111100 .... 10110 1010./	XX2	I	653	xststdcdp	v3.0			VSX Scalar Test Data Class Double-Precision
111111 .... 10110 00100/	X	I	654	xststdcqp	v3.0			VSX Scalar Test Data Class Quad-Precision
111100 .... 10010 1010./	XX2	I	655	xststdcsp	v3.0			VSX Scalar Test Data Class Single-Precision
111100 ..... 00000 ..... 10101 1011./	XX2	I	656	xsxexpdp	v3.0			VSX Scalar Extract Exponent Double-Precision
111111 ..... 00010 ..... 11001 00100/	X	I	656	xsxexpqp	v3.0			VSX Scalar Extract Exponent Quad-Precision
111100 ..... 00001 ..... 10101 1011./	XX2	I	657	xsxsigdp	v3.0			VSX Scalar Extract Significand Double-Precision
111111 ..... 10010 ..... 11001 00100/	X	I	657	xsxsigqp	v3.0			VSX Scalar Extract Significand Quad-Precision
111100 ..... 11000 ..... 11101 1011..	XX2	I	681	xvcvhpsp	v3.0			VSX Vector Convert Half-Precision to Single-Precision format
111100 ..... 11001 ..... 11101 1011..	XX2	I	683	xvcvsphp	v3.0			VSX Vector Convert with round Single-Precision to Half-Precision format
111100 .... 11111 000...	XX3	I	700	xviexpdp	v3.0			VSX Vector Insert Exponent Double-Precision
111100 .... 11011 000...	XX3	I	700	xviexpsp	v3.0			VSX Vector Insert Exponent Single-Precision
111100 .... 1111. 101...	XX2	I	760	xvtstdcdp	v3.0			VSX Vector Test Data Class Double-Precision
111100 .... 1101. 101...	XX2	I	761	xvtstdcsp	v3.0			VSX Vector Test Data Class Single-Precision
111100 ..... 00000 ..... 11101 1011..	XX2	I	762	xvxexpdp	v3.0			VSX Vector Extract Exponent Double-Precision
111100 ..... 01000 ..... 11101 1011..	XX2	I	762	xvxexpsp	v3.0			VSX Vector Extract Exponent Single-Precision
111100 ..... 00001 ..... 11101 1011..	XX2	I	763	xvxsigdp	v3.0			VSX Vector Extract Significand Double-Precision
111100 ..... 01001 ..... 11101 1011..	XX2	I	763	xvxsigsp	v3.0			VSX Vector Extract Significand Single-Precision
111100 ..... 10111 ..... 11101 1011..	XX2	I	764	xxbrd	v3.0			VSX Vector Byte-Reverse Doubleword
111100 ..... 00111 ..... 11101 1011..	XX2	I	764	xxbrh	v3.0			VSX Vector Byte-Reverse Halfword
111100 ..... 11111 ..... 11101 1011..	XX2	I	765	xxbrq	v3.0			VSX Vector Byte-Reverse Quadword
111100 ..... 01111 ..... 11101 1011..	XX2	I	765	xxbrw	v3.0			VSX Vector Byte-Reverse Word
111100 ..... /.... ..... 01010 0101..	XX2	I	766	xxextractuw	v3.0			VSX Vector Extract Unsigned Word
111100 ..... /.... ..... 01011 0101..	XX2	I	766	xxinsertw	v3.0			VSX Vector Insert Word
111100 .... 00011 010...	XX3	I	772	xxperm	v3.0			VSX Vector Permute
111100 .... 00111 010...	XX3	I	772	xxpermr	v3.0			VSX Vector Permute Right-indexed
111100 ..... 00... ..... 01011 01000.	XX1	I	774	xxspltib	v3.0			VSX Vector Splat Immediate Byte
000100 .... 1.000 000001	VX	I	348	bcdadd.	v2.07			Decimal Add Modulo & record
000100 .... 1.001 000001	VX	I	348	bcdsub.	v2.07			Decimal Subtract Modulo & record
010011 .... ///.. 10001 10000.	XL	I	39	bctar[l]	v2.07			Branch Conditional to BTAR [& Link]
011111 ///// ///// ///// 01101 01110/	X	I	909	clrbhrb	v2.07			Clear BHRB
111111 .... 11110 00110/	X	I	151	fmrgew	v2.07			Floating Merge Even Word
111111 .... 11010 00110/	X	I	151	fmrgow	v2.07			Floating Merge Odd Word

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 4 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
011111 /.... .... 00000 10110/	X	II	840	icbt	v2.07			Instruction Cache Block Touch
011111 .... 01000 10100.	X	I	871	lqarx	v2.07			Load Quadword And Reserve Indexed
011111 .... 00010 01100.	X	I	483	lxsiwax	v2.07			Load VSX Scalar as Integer Word Algebraic Indexed
011111 .... 00000 01100.	X	I	484	lxsiwzx	v2.07			Load VSX Scalar as Integer Word & Zero Indexed
011111 .... 10000 01100.	X	I	485	lxsspx	v2.07			Load VSX Scalar Single-Precision Indexed
011111 .... 01001 01110/	X	I	909	mfbhrbe	v2.07			Move From BHRB
011111 .... ///// 00001 10011.	XX1	I	112	mfvsrd	v2.07			Move From VSR Doubleword
011111 .... ///// 00011 10011.	XX1	I	113	mfvsrwz	v2.07			Move From VSR Word & Zero
011111 ///// ///// 00111 01110/	X	III	1130	msgclr	v2.07	HV		Message Clear
011111 ///// ///// 00101 01110/	X	III	1132	msgclrp	v2.07	P		Message Clear Privileged
011111 ///// ///// 00110 01110/	X	III	1129	msgsnd	v2.07	HV		Message Send
011111 ///// ///// 00100 01110/	X	III	1131	msgsndp	v2.07	P		Message Send Privileged
011111 .... ///// 00101 10011.	XX1	I	114	mtvsrd	v2.07			Move To VSR Doubleword
011111 .... ///// 00110 10011.	XX1	I	114	mtvsrwa	v2.07			Move To VSR Word Algebraic
011111 .... ///// 00111 10011.	XX1	I	115	mtvsrwz	v2.07			Move To VSR Word & Zero
010011 ///// ///// ////. 00100 10010/	XL	I	905	rfebb	v2.07			Return from Event Based Branch
011111 .... 00101 101101	X	I	872	stqcx.	v2.07			Store Quadword Conditional Indexed & record
011111 .... 00100 01100.	X	I	500	stxsiwx	v2.07			Store VSX Scalar as Integer Word Indexed
011111 .... 10100 01100.	X	I	502	stxsspx	v2.07			Store VSX Scalar Single-Precision Indexed
011111 ///// ..... ///// 11100 011101	X	II	892	tabort.	v2.07			Transaction Abort & record
011111 .... 11001 011101	X	II	894	tabortdc.	v2.07			Transaction Abort Doubleword Conditional & record
011111 .... 11011 011101	X	II	894	tabortdci.	v2.07			Transaction Abort Doubleword Conditional Immediate & record
011111 .... 11000 011101	X	II	893	tabortwc.	v2.07			Transaction Abort Word Conditional & record
011111 .... 11010 011101	X	II	893	tabortwci.	v2.07			Transaction Abort Word Conditional Immediate & record
011111 .///. ///// ///// 10100 011101	X	II	890	tbegin.	v2.07			Transaction Begin & record
011111 .../// ///// ///// 10110 01110/	X	II	895	tcheck	v2.07			Transaction Check & record
011111 .//// ///// ///// 10101 01110/	X	II	891	tend.	v2.07			Transaction End & record
011111 ///// ///// ///// 11111 011101	X	II	970	trechkpt.	v2.07			Transaction Recheckpoint & record
011111 ///// ..... ///// 11101 011101	X	II	969	treclaim.	v2.07			Transaction Reclaim & record
011111 ////. ///// ///// 10111 01110/	X	II	895	tsr.	v2.07			Transaction Suspend or Resume & record
000100 .... 00101 000000	VX	I	273	vaddcuq	v2.07			Vector Add & write Carry Unsigned Quadword
000100 111101	VA	I	273	vaddecuq	v2.07			Vector Add Extended & write Carry Unsigned Quadword
000100 111100	VA	I	273	vaddeuqm	v2.07			Vector Add Extended Unsigned Quadword Modulo
000100 .... 00011 000000	VX	I	270	vaddudm	v2.07			Vector Add Unsigned Doubleword Modulo
000100 .... 00100 000000	VX	I	270	vadduqm	v2.07			Vector Add Unsigned Quadword Modulo
000100 .... 10101 001100	VX	I	346	vbpermq	v2.07			Vector Bit Permute Quadword
000100 .... 10100 001000	VX	I	333	vcipher	v2.07			Vector AES Cipher
000100 .... 10100 001001	VX	I	333	vcipherlast	v2.07			Vector AES Cipher Last
000100 ..... ///// 11100 000010	VX	I	340	vclzb	v2.07			Vector Count Leading Zeros Byte
000100 ..... ///// 11111 000010	VX	I	340	vclzd	v2.07			Vector Count Leading Zeros Doubleword
000100 ..... ///// 11101 000010	VX	I	340	vclzh	v2.07			Vector Count Leading Zeros Halfword
000100 ..... ///// 11110 000010	VX	I	340	vclzw	v2.07			Vector Count Leading Zeros Word
000100 .... .0011 000111	VC	I	304	vcmpequd[.]	v2.07			Vector Compare Equal To Unsigned Doubleword
000100 .... .1111 000111	VC	I	305	vcmpgtsd[.]	v2.07			Vector Compare Greater Than Signed Doubleword
000100 .... .1011 000111	VC	I	307	vcmpgtud[.]	v2.07			Vector Compare Greater Than Unsigned Doubleword
000100 .... 11010 000100	VX	I	312	veqv	v2.07			Vector Equivalence
000100 ..... ///// 10100 001100	VX	I	339	vgbbd	v2.07			Vector Gather Bits by Byte by Doubleword
000100 .... 00111 000010	VX	I	299	vmaxsd	v2.07			Vector Maximum Signed Doubleword
000100 .... 00011 000010	VX	I	299	vmaxud	v2.07			Vector Maximum Unsigned Doubleword
000100 .... 01111 000010	VX	I	301	vminsd	v2.07			Vector Minimum Signed Doubleword
000100 .... 01011 000010	VX	I	301	vminud	v2.07			Vector Minimum Unsigned Doubleword
000100 .... 11110 001100	VX	I	257	vmrgew	v2.07			Vector Merge Even Word
000100 .... 11010 001100	VX	I	257	vmrgow	v2.07			Vector Merge Odd Word
000100 .... 01110 001000	VX	I	283	vmulesw	v2.07			Vector Multiply Even Signed Word
000100 .... 01010 001000	VX	I	283	vmuleuw	v2.07			Vector Multiply Even Unsigned Word
000100 .... 00110 001000	VX	I	283	vmulosw	v2.07			Vector Multiply Odd Signed Word

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 5 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
000100 .... 00010 001000	VX	I	283	vmulouw	v2.07			Vector Multiply Odd Unsigned Word
000100 .... 00010 001001	VX	I	284	vmuluwm	v2.07			Vector Multiply Unsigned Word Modulo
000100 .... 10110 000100	VX	I	312	vnand	v2.07			Vector NAND
000100 .... 10101 001000	VX	I	334	vncipher	v2.07			Vector AES Inverse Cipher
000100 .... 10101 001001	VX	I	334	vncipherlast	v2.07			Vector AES Inverse Cipher Last
000100 .... 10101 000100	VX	I	313	vorc	v2.07			Vector OR with Complement
000100 101101	VA	I	338	vpermxor	v2.07			Vector Permute & Exclusive-OR
000100 .... 10111 001110	VX	I	248	vpksdss	v2.07			Vector Pack Signed Doubleword Signed Saturate
000100 .... 10101 001110	VX	I	249	vpksdus	v2.07			Vector Pack Signed Doubleword Unsigned Saturate
000100 .... 10001 001110	VX	I	251	vpkudum	v2.07			Vector Pack Unsigned Doubleword Unsigned Modulo
000100 .... 10011 001110	VX	I	251	vpkudus	v2.07			Vector Pack Unsigned Doubleword Unsigned Saturate
000100 .... 10000 001000	VX	I	336	vpmsumb	v2.07			Vector Polynomial Multiply-Sum Byte
000100 .... 10011 001000	VX	I	336	vpmsumd	v2.07			Vector Polynomial Multiply-Sum Doubleword
000100 .... 10001 001000	VX	I	337	vpmsumh	v2.07			Vector Polynomial Multiply-Sum Halfword
000100 .... 10010 001000	VX	I	337	vpmsumw	v2.07			Vector Polynomial Multiply-Sum Word
000100 ..... ///// 11100 000011	VX	I	345	vpopcntb	v2.07			Vector Population Count Byte
000100 ..... ///// 11111 000011	VX	I	345	vpopcntd	v2.07			Vector Population Count Doubleword
000100 ..... ///// 11101 000011	VX	I	345	vpopcnth	v2.07			Vector Population Count Halfword
000100 ..... ///// 11110 000011	VX	I	345	vpopcntw	v2.07			Vector Population Count Word
000100 .... 00011 000100	VX	I	315	vrld	v2.07			Vector Rotate Left Doubleword
000100 .... ///// 10111 001000	VX	I	334	vsbox	v2.07			Vector AES S-Box
000100 .... 11011 000010	VX	I	335	vshasigmad	v2.07			Vector SHA-512 Sigma Doubleword
000100 .... 11010 000010	VX	I	335	vshasigmaw	v2.07			Vector SHA-256 Sigma Word
000100 .... 10111 000100	VX	I	316	vsld	v2.07			Vector Shift Left Doubleword
000100 .... 01111 000100	VX	I	318	vsrad	v2.07			Vector Shift Right Algebraic Doubleword
000100 .... 11011 000100	VX	I	317	vsrd	v2.07			Vector Shift Right Doubleword
000100 .... 10101 000000	VX	I	279	vsubcuq	v2.07			Vector Subtract & write Carry Unsigned Quadword
000100 111111	VA	I	279	vsubecuq	v2.07			Vector Subtract Extended & write Carry Unsigned Quadword
000100 111110	VA	I	279	vsubeuqm	v2.07			Vector Subtract Extended Unsigned Quadword Modulo
000100 .... 10011 000000	VX	I	277	vsubudm	v2.07			Vector Subtract Unsigned Doubleword Modulo
000100 .... 10100 000000	VX	I	279	vsubuqm	v2.07			Vector Subtract Unsigned Quadword Modulo
000100 ..... ///// 11001 001110	VX	I	254	vupkhsw	v2.07			Vector Unpack High Signed Word
000100 ..... ///// 11011 001110	VX	I	254	vupklsw	v2.07			Vector Unpack Low Signed Word
111100 .... 00000 000...	XX3	I	518	xsaddsp	v2.07			VSX Scalar Add Single-Precision
111100 ..... ///// ..... 10000 1011..	XX2	I	537	xscvdpspn	v2.07			VSX Scalar Convert Double-Precision to Single-Precision Non-signalling format
111100 ..... ///// ..... 10100 1011..	XX2	I	558	xscvspdpn	v2.07			VSX Scalar Convert Single-Precision to Double-Precision Non-signalling format
111100 ..... ///// ..... 10011 1000..	XX2	I	559	xscvsxdsp	v2.07			VSX Scalar Convert with round Signed Doubleword to Single-Precision format
111100 ..... ///// ..... 10010 1000..	XX2	I	561	xscvuxdsp	v2.07			VSX Scalar Convert with round Unsigned Doubleword to Single-Precision format
111100 .... 00011 000...	XX3	I	566	xsdivsp	v2.07			VSX Scalar Divide Single-Precision
111100 .... 00000 001...	XX3	I	573	xsmaddasp	v2.07			VSX Scalar Multiply-Add Type-A Single-Precision
111100 .... 00001 001...	XX3	I	573	xsmaddmsp	v2.07			VSX Scalar Multiply-Add Type-M Single-Precision
111100 .... 00010 001...	XX3	I	594	xsmsubasp	v2.07			VSX Scalar Multiply-Subtract Type-A Single-Precision
111100 .... 00011 001...	XX3	I	594	xsmsubmsp	v2.07			VSX Scalar Multiply-Subtract Type-M Single-Precision
111100 .... 00010 000...	XX3	I	604	xsmulsp	v2.07			VSX Scalar Multiply Single-Precision
111100 .... 10000 001...	XX3	I	613	xsnmaddasp	v2.07			VSX Scalar Negative Multiply-Add Type-A Single-Precision
111100 .... 10001 001...	XX3	I	613	xsnmaddmsp	v2.07			VSX Scalar Negative Multiply-Add Type-M Single-Precision
111100 .... 10010 001...	XX3	I	622	xsnmsubasp	v2.07			VSX Scalar Negative Multiply-Subtract Type-A Single-Precision
111100 .... 10011 001...	XX3	I	622	xsnmsubmsp	v2.07			VSX Scalar Negative Multiply-Subtract Type-M Single-Precision
111100 ..... ///// ..... 00001 1010..	XX2	I	633	xsresp	v2.07			VSX Scalar Reciprocal Estimate Single-Precision
111100 ..... ///// ..... 10001 1001..	XX2	I	638	xsrsp	v2.07			VSX Scalar Round Double-Precision to Single-Precision
111100 ..... ///// ..... 00000 1010..	XX2	I	640	xsrsqrtesp	v2.07			VSX Scalar Reciprocal Square Root Estimate Single-Precision
111100 ..... ///// ..... 00000 1011..	XX2	I	644	xssqrtsp	v2.07			VSX Scalar Square Root Single-Precision
111100 .... 00001 000...	XX3	I	649	xssubsp	v2.07			VSX Scalar Subtract Single-Precision

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 6 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111100 .... 10111 010...	XX3	I	768	xxleqv	v2.07			VSX Vector Logical Equivalence
111100 .... 10110 010...	XX3	I	768	xxlnand	v2.07			VSX Vector Logical NAND
111100 .... 10101 010...	XX3	I	769	xxlorc	v2.07			VSX Vector Logical OR with Complement
011111 .... /0010 01010/	XO	I	111	addg6s	v2.06			Add & Generate Sixes
011111 .... 00111 11100/	X	I	100	bpermd	v2.06			Bit Permute Doubleword
011111 .... ///// 01001 11010/	X	I	111	cbcdtd	v2.06			Convert Binary Coded Decimal To Declets
011111 .... ///// 01000 11010/	X	I	111	cdtbcd	v2.06			Convert Declets To Binary Coded Decimal
111011 ..... ///// ..... 11001 00010.	X	I	215	dcffix[.]	v2.06			DFP Convert From Fixed
011111 .... .1101 01001.	XO	I	82	divde[o][.]	v2.06		SR	Divide Doubleword Extended
011111 .... .1100 01001.	XO	I	82	divdeu[o][.]	v2.06		SR	Divide Doubleword Extended Unsigned
011111 .... .1101 01011.	XO	I	75	divwe[o][.]	v2.06		SR	Divide Word Extended
011111 .... .1100 01011.	XO	I	75	divweu[o][.]	v2.06		SR	Divide Word Extended Unsigned
111011 ..... ///// ..... 11010 01110.	X	I	164	fcfids[.]	v2.06			Floating Convert with round Signed Doubleword to Single-Precision format
111111 ..... ///// ..... 11110 01110.	X	I	164	fcfidu[.]	v2.06			Floating Convert with round Unsigned Doubleword to Double-Precision format
111011 ..... ///// ..... 11110 01110.	X	I	165	fcfidus[.]	v2.06			Floating Convert with round Unsigned Doubleword to Single-Precision format
111111 ..... ///// ..... 11101 01110.	X	I	160	fctidu[.]	v2.06			Floating Convert with round Double-Precision To Unsigned Doubleword format
111111 ..... ///// ..... 11101 01111.	X	I	161	fctiduz[.]	v2.06			Floating Convert with round to Zero Double-Precision To Unsigned Doubleword format
111111 ..... ///// ..... 00100 01110.	X	I	162	fctiwu[.]	v2.06			Floating Convert with round Double-Precision To Unsigned Word format
111111 ..... ///// ..... 00100 01111.	X	I	163	fctiwuz[.]	v2.06			Floating Convert with round to Zero Double-Precision To Unsigned Word format
111111 .../// 00100 00000/	X	I	156	ftdiv	v2.06			Floating Test for software Divide
111111 .../// ///// 00101 00000/	X	I	156	ftsqrt	v2.06			Floating Test for software Square Root
011111 .... 00001 10100.	X	II	864	lbarx	v2.06			Load Byte And Reserve Indexed
011111 .... 10000 10100/	X	I	61	ldbrx	v2.06			Load Doubleword Byte-Reverse Indexed
011111 .... 11011 10111/	X	I	143	lfiwzx	v2.06			Load Floating as Integer Word & Zero Indexed
011111 .... 00011 10100.	X	II	865	lharx	v2.06			Load Halfword And Reserve Indexed Xform
011111 .... 10010 01100.	X	I	480	lxsdx	v2.06			Load VSX Scalar Doubleword Indexed
011111 .... 11010 01100.	X	I	488	lxvd2x	v2.06			Load VSX Vector Doubleword*2 Indexed
011111 .... 01010 01100.	X	I	494	lxvdsx	v2.06			Load VSX Vector Doubleword & Splat Indexed
011111 .... 11000 01100.	X	I	496	lxvw4x	v2.06			Load VSX Vector Word*4 Indexed
011111 .... ///// 01111 11010/	X	I	99	popcntd	v2.06			Population Count Doubleword
011111 .... ///// 01011 11010/	X	I	97	popcntw	v2.06			Population Count Words
011111 .... 10101 101101	X	II	866	stbcx.	v2.06			Store Byte Conditional Indexed & record
011111 .... 10100 10100/	X	I	61	stdbrx	v2.06			Store Doubleword Byte-Reverse Indexed
011111 .... 10110 101101	X	II	867	sthcx.	v2.06			Store Halfword Conditional Indexed & record
011111 .... 10110 01100.	X	I	498	stxsdx	v2.06			Store VSX Scalar Doubleword Indexed
011111 .... 11110 01100.	X	I	504	stxvd2x	v2.06			Store VSX Vector Doubleword*2 Indexed
011111 .... 11100 01100.	X	I	506	stxvw4x	v2.06			Store VSX Vector Word*4 Indexed
111100 ..... ///// ..... 10101 1001..	XX2	I	512	xsabsdp	v2.06			VSX Scalar Absolute Double-Precision
111100 .... 00100 000...	XX3	I	513	xsadddp	v2.06			VSX Scalar Add Double-Precision
111100 .../// .... 00101 011../	XX3	I	527	xscmpodp	v2.06			VSX Scalar Compare Ordered Double-Precision
111100 .../// .... 00100 011../	XX3	I	530	xscmpudp	v2.06			VSX Scalar Compare Unordered Double-Precision
111100 .... 10110 000...	XX3	I	533	xscpsgndp	v2.06			VSX Scalar Copy Sign Double-Precision
111100 ..... ///// ..... 10000 1001..	XX2	I	536	xscvdpsp	v2.06			VSX Scalar Convert with round Double-Precision to Single-Precision format
111100 ..... ///// ..... 10101 1000..	XX2	I	537	xscvdpsxds	v2.06			VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format
111100 ..... ///// ..... 00101 1000..	XX2	I	540	xscvdpsxws	v2.06			VSX Scalar Convert with round to zero Double-Precision to Signed Word format
111100 ..... ///// ..... 10100 1000..	XX2	I	542	xscvdpuxds	v2.06			VSX Scalar Convert with round to zero Double-Precision to Unsigned Doubleword format
111100 ..... ///// ..... 00100 1000..	XX2	I	544	xscvdpuxws	v2.06			VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 7 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111100 ..... ///// ..... 10100 1001..	XX2	I	557	xscvspdp	v2.06			VSX Scalar Convert Single-Precision to Double-Precision format
111100 ..... ///// ..... 10111 1000..	XX2	I	559	xscvsxddp	v2.06			VSX Scalar Convert with round Signed Doubleword to Double-Precision format
111100 ..... ///// ..... 10110 1000..	XX2	I	561	xscvuxddp	v2.06			VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format
111100 .... 00111 000...	XX3	I	562	xsdivdp	v2.06			VSX Scalar Divide Double-Precision
111100 .... 00100 001...	XX3	I	570	xsmaddadp	v2.06			VSX Scalar Multiply-Add Type-A Double-Precision
111100 .... 00101 001...	XX3	I	570	xsmaddmdp	v2.06			VSX Scalar Multiply-Add Type-M Double-Precision
111100 .... 10100 000...	XX3	I	579	xsmaxdp	v2.06			VSX Scalar Maximum Double-Precision
111100 .... 10101 000...	XX3	I	585	xsmindp	v2.06			VSX Scalar Minimum Double-Precision
111100 .... 00110 001...	XX3	I	591	xsmsubadp	v2.06			VSX Scalar Multiply-Subtract Type-A Double-Precision
111100 .... 00111 001...	XX3	I	591	xsmsubmdp	v2.06			VSX Scalar Multiply-Subtract Type-M Double-Precision
111100 .... 00110 000...	XX3	I	600	xsmuldp	v2.06			VSX Scalar Multiply Double-Precision
111100 ..... ///// ..... 10110 1001..	XX2	I	606	xsnabsdp	v2.06			VSX Scalar Negative Absolute Double-Precision
111100 ..... ///// ..... 10111 1001..	XX2	I	607	xsnegdp	v2.06			VSX Scalar Negate Double-Precision
111100 .... 10100 001...	XX3	I	608	xsnmaddadp	v2.06			VSX Scalar Negative Multiply-Add Type-A Double-Precision
111100 .... 10101 001...	XX3	I	608	xsnmaddmdp	v2.06			VSX Scalar Negative Multiply-Add Type-M Double-Precision
111100 .... 10110 001...	XX3	I	619	xsnmsubadp	v2.06			VSX Scalar Negative Multiply-Subtract Type-A Double-Precision
111100 .... 10111 001...	XX3	I	619	xsnmsubmdp	v2.06			VSX Scalar Negative Multiply-Subtract Type-M Double-Precision
111100 ..... ///// ..... 00100 1001..	XX2	I	628	xsrdpi	v2.06			VSX Scalar Round Double-Precision to Integral
111100 ..... ///// ..... 00110 1011..	XX2	I	629	xsrdpic	v2.06			VSX Scalar Round Double-Precision to Integral using Current rounding mode
111100 ..... ///// ..... 00111 1001..	XX2	I	630	xsrdpim	v2.06			VSX Scalar Round Double-Precision to Integral toward -Infinity
111100 ..... ///// ..... 00110 1001..	XX2	I	630	xsrdpip	v2.06			VSX Scalar Round Double-Precision to Integral toward +Infinity
111100 ..... ///// ..... 00101 1001..	XX2	I	631	xsrdpiz	v2.06			VSX Scalar Round Double-Precision to Integral toward Zero
111100 ..... ///// ..... 00101 1010..	XX2	I	632	xsredp	v2.06			VSX Scalar Reciprocal Estimate Double-Precision
111100 ..... ///// ..... 00100 1010..	XX2	I	639	xsrsqrtedp	v2.06			VSX Scalar Reciprocal Square Root Estimate Double-Precision
111100 ..... ///// ..... 00100 1011..	XX2	I	641	xssqrtdp	v2.06			VSX Scalar Square Root Double-Precision
111100 .... 00101 000...	XX3	I	645	xssubdp	v2.06			VSX Scalar Subtract Double-Precision
111100 ...// .... 00111 101../	XX3	I	651	xstdivdp	v2.06			VSX Scalar Test for software Divide Double-Precision
111100 ...// ///// 00110 1010./	XX2	I	652	xstsqrtdp	v2.06			VSX Scalar Test for software Square Root Double-Precision
111100 ..... ///// ..... 11101 1001..	XX2	I	658	xvabsdp	v2.06			VSX Vector Absolute Double-Precision
111100 ..... ///// ..... 11001 1001..	XX2	I	658	xvabssp	v2.06			VSX Vector Absolute Single-Precision
111100 .... 01100 000...	XX3	I	659	xvadddp	v2.06			VSX Vector Add Double-Precision
111100 .... 01000 000...	XX3	I	663	xvaddsp	v2.06			VSX Vector Add Single-Precision
111100 .... .1100 011...	XX3	I	665	xvcmpeqdp[.]	v2.06			VSX Vector Compare Equal Double-Precision
111100 .... .1000 011...	XX3	I	666	xvcmpeqsp[.]	v2.06			VSX Vector Compare Equal Single-Precision
111100 .... .1110 011...	XX3	I	667	xvcmpgedp[.]	v2.06			VSX Vector Compare Greater Than or Equal Double-Precision
111100 .... .1010 011...	XX3	I	668	xvcmpgesp[.]	v2.06			VSX Vector Compare Greater Than or Equal Single-Precision
111100 .... .1101 011...	XX3	I	669	xvcmpgtdp[.]	v2.06			VSX Vector Compare Greater Than Double-Precision
111100 .... .1001 011...	XX3	I	670	xvcmpgtsp[.]	v2.06			VSX Vector Compare Greater Than Single-Precision
111100 .... 11110 000...	XX3	I	671	xvcpsgndp	v2.06			VSX Vector Copy Sign Double-Precision
111100 .... 11010 000...	XX3	I	671	xvcpsgnsp	v2.06			VSX Vector Copy Sign Single-Precision
111100 ..... ///// ..... 11000 1001..	XX2	I	672	xvcvdpsp	v2.06			VSX Vector Convert with round Double-Precision to Single-Precision format
111100 ..... ///// ..... 11101 1000..	XX2	I	673	xvcvdpsxds	v2.06			VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format
111100 ..... ///// ..... 01101 1000..	XX2	I	675	xvcvdpsxws	v2.06			VSX Vector Convert with round to zero Double-Precision to Signed Word format
111100 ..... ///// ..... 11100 1000..	XX2	I	677	xvcvdpuxds	v2.06			VSX Vector Convert with round to zero Double-Precision to Unsigned Doubleword format
111100 ..... ///// ..... 01100 1000..	XX2	I	679	xvcvdpuxws	v2.06			VSX Vector Convert with round to zero Double-Precision to Unsigned Word format
111100 ..... ///// ..... 11100 1001..	XX2	I	682	xvcvspdp	v2.06			VSX Vector Convert Single-Precision to Double-Precision format
111100 ..... ///// ..... 11001 1000..	XX2	I	684	xvcvspsxds	v2.06			VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format
111100 ..... ///// ..... 01001 1000..	XX2	I	686	xvcvspsxws	v2.06			VSX Vector Convert with round to zero Single-Precision to Signed Word format

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 8 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111100 ..... ///// ..... 11000 1000..	XX2	I	688	xvcvspuxds	v2.06			VSX Vector Convert with round to zero Single-Precision to Unsigned Doubleword format
111100 ..... ///// ..... 01000 1000..	XX2	I	690	xvcvspuxws	v2.06			VSX Vector Convert with round to zero Single-Precision to Unsigned Word format
111100 ..... ///// ..... 11111 1000..	XX2	I	692	xvcvsxddp	v2.06			VSX Vector Convert with round Signed Doubleword to Double-Precision format
111100 ..... ///// ..... 11011 1000..	XX2	I	692	xvcvsxdsp	v2.06			VSX Vector Convert with round Signed Doubleword to Single-Precision format
111100 ..... ///// ..... 01111 1000..	XX2	I	693	xvcvsxwdp	v2.06			VSX Vector Convert Signed Word to Double-Precision format
111100 ..... ///// ..... 01011 1000..	XX2	I	693	xvcvsxwsp	v2.06			VSX Vector Convert with round Signed Word to Single-Precision format
111100 ..... ///// ..... 11110 1000..	XX2	I	694	xvcvuxddp	v2.06			VSX Vector Convert with round Unsigned Doubleword to Double-Precision format
111100 ..... ///// ..... 11010 1000..	XX2	I	694	xvcvuxdsp	v2.06			VSX Vector Convert with round Unsigned Doubleword to Single-Precision format
111100 ..... ///// ..... 01110 1000..	XX2	I	695	xvcvuxwdp	v2.06			VSX Vector Convert Unsigned Word to Double-Precision format
111100 ..... ///// ..... 01010 1000..	XX2	I	695	xvcvuxwsp	v2.06			VSX Vector Convert with round Unsigned Word to Single-Precision format
111100 .... 01111 000...	XX3	I	696	xvdivdp	v2.06			VSX Vector Divide Double-Precision
111100 .... 01011 000...	XX3	I	698	xvdivsp	v2.06			VSX Vector Divide Single-Precision
111100 .... 01100 001...	XX3	I	701	xvmaddadp	v2.06			VSX Vector Multiply-Add Type-A Double-Precision
111100 .... 01000 001...	XX3	I	704	xvmaddasp	v2.06			VSX Vector Multiply-Add Type-A Single-Precision
111100 .... 01101 001...	XX3	I	701	xvmaddmdp	v2.06			VSX Vector Multiply-Add Type-M Double-Precision
111100 .... 01001 001...	XX3	I	704	xvmaddmsp	v2.06			VSX Vector Multiply-Add Type-M Single-Precision
111100 .... 11100 000...	XX3	I	707	xvmaxdp	v2.06			VSX Vector Maximum Double-Precision
111100 .... 11000 000...	XX3	I	709	xvmaxsp	v2.06			VSX Vector Maximum Single-Precision
111100 .... 11101 000...	XX3	I	711	xvmindp	v2.06			VSX Vector Minimum Double-Precision
111100 .... 11001 000...	XX3	I	713	xvminsp	v2.06			VSX Vector Minimum Single-Precision
111100 .... 01110 001...	XX3	I	715	xvmsubadp	v2.06			VSX Vector Multiply-Subtract Type-A Double-Precision
111100 .... 01010 001...	XX3	I	718	xvmsubasp	v2.06			VSX Vector Multiply-Subtract Type-A Single-Precision
111100 .... 01111 001...	XX3	I	715	xvmsubmdp	v2.06			VSX Vector Multiply-Subtract Type-M Double-Precision
111100 .... 01011 001...	XX3	I	718	xvmsubmsp	v2.06			VSX Vector Multiply-Subtract Type-M Single-Precision
111100 .... 01110 000...	XX3	I	721	xvmuldp	v2.06			VSX Vector Multiply Double-Precision
111100 .... 01010 000...	XX3	I	723	xvmulsp	v2.06			VSX Vector Multiply Single-Precision
111100 ..... ///// ..... 11110 1001..	XX2	I	725	xvnabsdp	v2.06			VSX Vector Negative Absolute Double-Precision
111100 ..... ///// ..... 11010 1001..	XX2	I	725	xvnabssp	v2.06			VSX Vector Negative Absolute Single-Precision
111100 ..... ///// ..... 11111 1001..	XX2	I	726	xvnegdp	v2.06			VSX Vector Negate Double-Precision
111100 ..... ///// ..... 11011 1001..	XX2	I	726	xvnegsp	v2.06			VSX Vector Negate Single-Precision
111100 .... 11100 001...	XX3	I	727	xvnmaddadp	v2.06			VSX Vector Negative Multiply-Add Type-A Double-Precision
111100 .... 11000 001...	XX3	I	732	xvnmaddasp	v2.06			VSX Vector Negative Multiply-Add Type-A Single-Precision
111100 .... 11101 001...	XX3	I	727	xvnmaddmdp	v2.06			VSX Vector Negative Multiply-Add Type-M Double-Precision
111100 .... 11001 001...	XX3	I	732	xvnmaddmsp	v2.06			VSX Vector Negative Multiply-Add Type-M Single-Precision
111100 .... 11110 001...	XX3	I	735	xvnmsubadp	v2.06			VSX Vector Negative Multiply-Subtract Type-A Double-Precision
111100 .... 11010 001...	XX3	I	738	xvnmsubasp	v2.06			VSX Vector Negative Multiply-Subtract Type-A Single-Precision
111100 .... 11111 001...	XX3	I	735	xvnmsubmdp	v2.06			VSX Vector Negative Multiply-Subtract Type-M Double-Precision
111100 .... 11011 001...	XX3	I	738	xvnmsubmsp	v2.06			VSX Vector Negative Multiply-Subtract Type-M Single-Precision
111100 ..... ///// ..... 01100 1001..	XX2	I	741	xvrdpi	v2.06			VSX Vector Round Double-Precision to Integral
111100 ..... ///// ..... 01110 1011..	XX2	I	741	xvrdpic	v2.06			VSX Vector Round Double-Precision to Integral using Current rounding mode
111100 ..... ///// ..... 01111 1001..	XX2	I	742	xvrdpim	v2.06			VSX Vector Round Double-Precision to Integral toward -Infinity
111100 ..... ///// ..... 01110 1001..	XX2	I	742	xvrdpip	v2.06			VSX Vector Round Double-Precision to Integral toward +Infinity
111100 ..... ///// ..... 01101 1001..	XX2	I	743	xvrdpiz	v2.06			VSX Vector Round Double-Precision to Integral toward Zero
111100 ..... ///// ..... 01101 1010..	XX2	I	744	xvredp	v2.06			VSX Vector Reciprocal Estimate Double-Precision
111100 ..... ///// ..... 01001 1010..	XX2	I	745	xvresp	v2.06			VSX Vector Reciprocal Estimate Single-Precision
111100 ..... ///// ..... 01000 1001..	XX2	I	746	xvrspi	v2.06			VSX Vector Round Single-Precision to Integral
111100 ..... ///// ..... 01010 1011..	XX2	I	746	xvrspic	v2.06			VSX Vector Round Single-Precision to Integral using Current rounding mode
111100 ..... ///// ..... 01011 1001..	XX2	I	747	xvrspim	v2.06			VSX Vector Round Single-Precision to Integral toward -Infinity

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 9 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111100 ..... ///// ..... 01010 1001..	XX2	I	747	xvrspip	v2.06			VSX Vector Round Single-Precision to Integral toward +Infinity
111100 ..... ///// ..... 01001 1001..	XX2	I	748	xvrspiz	v2.06			VSX Vector Round Single-Precision to Integral toward Zero
111100 ..... ///// ..... 01100 1010..	XX2	I	748	xvrsqrtedp	v2.06			VSX Vector Reciprocal Square Root Estimate Double-Precision
111100 ..... ///// ..... 01000 1010..	XX2	I	750	xvrsqrtesp	v2.06			VSX Vector Reciprocal Square Root Estimate Single-Precision
111100 ..... ///// ..... 01100 1011..	XX2	I	751	xvsqrtdp	v2.06			VSX Vector Square Root Double-Precision
111100 ..... ///// ..... 01000 1011..	XX2	I	752	xvsqrtsp	v2.06			VSX Vector Square Root Single-Precision
111100 .... 01101 000...	XX3	I	753	xvsubdp	v2.06			VSX Vector Subtract Double-Precision
111100 .... 01001 000...	XX3	I	755	xvsubsp	v2.06			VSX Vector Subtract Single-Precision
111100 ...// .... 01111 101../	XX3	I	757	xvtdivdp	v2.06			VSX Vector Test for software Divide Double-Precision
111100 ...// .... 01011 101../	XX3	I	758	xvtdivsp	v2.06			VSX Vector Test for software Divide Single-Precision
111100 ...// ///// 01110 1010./	XX2	I	759	xvtsqrtdp	v2.06			VSX Vector Test for software Square Root Double-Precision
111100 ...// ///// 01010 1010./	XX2	I	759	xvtsqrtsp	v2.06			VSX Vector Test for software Square Root Single-Precision
111100 .... 10000 010...	XX3	I	767	xxland	v2.06			VSX Vector Logical AND
111100 .... 10001 010...	XX3	I	767	xxlandc	v2.06			VSX Vector Logical AND with Complement
111100 .... 10100 010...	XX3	I	769	xxlnor	v2.06			VSX Vector Logical NOR
111100 .... 10010 010...	XX3	I	770	xxlor	v2.06			VSX Vector Logical OR
111100 .... 10011 010...	XX3	I	770	xxlxor	v2.06			VSX Vector Logical XOR
111100 .... 00010 010...	XX3	I	771	xxmrghw	v2.06			VSX Vector Merge Word High
111100 .... 00110 010...	XX3	I	771	xxmrglw	v2.06			VSX Vector Merge Word Low
111100 .... 0..01 010...	XX3	I	773	xxpermdi	v2.06			VSX Vector Doubleword Permute Immediate
111100 .... 11....	XX4	I	773	xxsel	v2.06			VSX Vector Select
111100 .... 0..00 010...	XX3	I	774	xxsldwi	v2.06			VSX Vector Shift Left Double by Word Immediate
111100 ..... ///.. ..... 01010 0100..	XX2	I	774	xxspltw	v2.06			VSX Vector Splat Word
011111 .... 01111 11100/	X	I	97	cmpb	v2.05			Compare Byte
111011 .... 00000 00010.	X	I	193	dadd[.]	v2.05			DFP Add
111111 .... 00000 00010.	X	I	193	daddq[.]	v2.05			DFP Add Quad
111111 ..... ///// ..... 11001 00010.	X	I	215	dcffixq[.]	v2.05			DFP Convert From Fixed Quad
111011 ...// 00100 00010/	X	I	199	dcmpo	v2.05			DFP Compare Ordered
111111 ...// 00100 00010/	X	I	199	dcmpoq	v2.05			DFP Compare Ordered Quad
111011 ...// 10100 00010/	X	I	198	dcmpu	v2.05			DFP Compare Unordered
111111 ...// 10100 00010/	X	I	198	dcmpuq	v2.05			DFP Compare Unordered Quad
111011 ..... ///// ..... 01000 00010.	X	I	213	dctdp[.]	v2.05			DFP Convert To DFP Long
111011 ..... ///// ..... 01001 00010.	X	I	215	dctfix[.]	v2.05			DFP Convert To Fixed
111111 ..... ///// ..... 01001 00010.	X	I	215	dctfixq[.]	v2.05			DFP Convert To Fixed Quad
111111 ..... ///// ..... 01000 00010.	X	I	213	dctqpq[.]	v2.05			DFP Convert To DFP Extended
111011 ....// ..... 01010 00010.	X	I	217	ddedpd[.]	v2.05			DFP Decode DPD To BCD
111111 ....// ..... 01010 00010.	X	I	217	ddedpdq[.]	v2.05			DFP Decode DPD To BCD Quad
111011 .... 10001 00010.	X	I	196	ddiv[.]	v2.05			DFP Divide
111111 .... 10001 00010.	X	I	196	ddivq[.]	v2.05			DFP Divide Quad
111011 ..... .//// ..... 11010 00010.	X	I	217	denbcd[.]	v2.05			DFP Encode BCD To DPD
111111 ..... .//// ..... 11010 00010.	X	I	217	denbcdq[.]	v2.05			DFP Encode BCD To DPD Quad
111011 .... 11011 00010.	X	I	218	diex[.]	v2.05			DFP Insert Exponent
111111 .... 11011 00010.	X	I	218	diexq[.]	v2.05			DFP Insert Exponent Quad
111011 .... 00001 00010.	X	I	195	dmul[.]	v2.05			DFP Multiply
111111 .... 00001 00010.	X	I	195	dmulq[.]	v2.05			DFP Multiply Quad
111011 ....000 00011.	Z23	I	204	dqua[.]	v2.05			DFP Quantize
111011 ....010 00011.	Z23	I	203	dquai[.]	v2.05			DFP Quantize Immediate
111111 ....010 00011.	Z23	I	203	dquaiq[.]	v2.05			DFP Quantize Immediate Quad
111111 ....000 00011.	Z23	I	204	dquaq[.]	v2.05			DFP Quantize Quad
111111 ..... ///// ..... 11000 00010.	X	I	214	drdpq[.]	v2.05			DFP Round To DFP Long
111011 ..... ////. ....111 00011.	Z23	I	211	drintn[.]	v2.05			DFP Round To FP Integer Without Inexact
111111 ..... ////. ....111 00011.	Z23	I	211	drintnq[.]	v2.05			DFP Round To FP Integer Without Inexact Quad
111011 ..... ////. ....011 00011.	Z23	I	209	drintx[.]	v2.05			DFP Round To FP Integer With Inexact
111111 ..... ////. ....011 00011.	Z23	I	209	drintxq[.]	v2.05			DFP Round To FP Integer With Inexact Quad
111011 ....001 00011.	Z23	I	206	drrnd[.]	v2.05			DFP Reround
111111 ....001 00011.	Z23	I	206	drrndq[.]	v2.05			DFP Reround Quad

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 10 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111011 ..... ///// ..... 11000 00010.	X	I	214	drsp[.]	v2.05			DFP Round To DFP Short
111011 .... .0010 00010.	Z22	I	220	dscli[.]	v2.05			DFP Shift Significand Left Immediate
111111 .... .0010 00010.	Z22	I	220	dscliq[.]	v2.05			DFP Shift Significand Left Immediate Quad
111011 .... .0011 00010.	Z22	I	220	dscri[.]	v2.05			DFP Shift Significand Right Immediate
111111 .... .0011 00010.	Z22	I	220	dscriq[.]	v2.05			DFP Shift Significand Right Immediate Quad
111011 .... 10000 00010.	X	I	193	dsub[.]	v2.05			DFP Subtract
111111 .... 10000 00010.	X	I	193	dsubq[.]	v2.05			DFP Subtract Quad
111011 .../// .... .0110 00010/	Z22	I	200	dtstdc	v2.05			DFP Test Data Class
111111 .../// .... .0110 00010/	Z22	I	200	dtstdcq	v2.05			DFP Test Data Class Quad
111011 .../// .... .0111 00010/	Z22	I	200	dtstdg	v2.05			DFP Test Data Group
111111 .../// .... .0111 00010/	Z22	I	200	dtstdgq	v2.05			DFP Test Data Group Quad
111011 .../// 00101 00010/	X	I	201	dtstex	v2.05			DFP Test Exponent
111111 .../// 00101 00010/	X	I	201	dtstexq	v2.05			DFP Test Exponent Quad
111011 .../// 10101 00010/	X	I	202	dtstsf	v2.05			DFP Test Significance
111111 .../// 10101 00010/	X	I	202	dtstsfq	v2.05			DFP Test Significance Quad
111011 ..... ///// ..... 01011 00010.	X	I	218	dxex[.]	v2.05			DFP Extract Exponent
111111 ..... ///// ..... 01011 00010.	X	I	218	dxexq[.]	v2.05			DFP Extract Exponent Quad
111111 .... 00000 01000.	X	I	150	fcpsgn[.]	v2.05			Floating Copy Sign
011111 .... 11010 10101/	X	III	966	lbzcix	v2.05	HV		Load Byte & Zero Caching Inhibited Indexed
011111 .... 11011 10101/	X	III	966	ldcix	v2.05	HV		Load Doubleword Caching Inhibited Indexed
111001 00	DS	I	149	lfdp	v2.05			Load Floating Double Pair
011111 .... 11000 10111/	X	I	149	lfdpx	v2.05			Load Floating Double Pair Indexed
011111 .... 11010 10111/	X	I	143	lfiwax	v2.05			Load Floating as Integer Word Algebraic Indexed
011111 .... 11001 10101/	X	III	966	lhzcix	v2.05	HV		Load Halfword & Zero Caching Inhibited Indexed
011111 .... 11000 10101/	X	III	966	lwzcix	v2.05	HV		Load Word & Zero Caching Inhibited Indexed
011111 .... ///// 00101 11010/	X	I	98	prtyd	v2.05			Parity Doubleword
011111 .... ///// 00100 11010/	X	I	98	prtyw	v2.05			Parity Word
011111 ..... ///// 11110 100111	X	III	1031	slbfee.	v2.05	P	SR	SLB Find Entry ESID & record
011111 .... 11110 10101/	X	III	967	stbcix	v2.05	HV		Store Byte Caching Inhibited Indexed
011111 .... 11111 10101/	X	III	967	stdcix	v2.05	HV		Store Doubleword Caching Inhibited Indexed
111101 00	DS	I	149	stfdp	v2.05			Store Floating Double Pair
011111 .... 11100 10111/	X	I	149	stfdpx	v2.05			Store Floating Double Pair Indexed
011111 .... 11101 10101/	X	III	967	sthcix	v2.05	HV		Store Halfword Caching Inhibited Indexed
011111 .... 11100 10101/	X	III	967	stwcix	v2.05	HV		Store Word Caching Inhibited Indexed
011010 00000 00000 00000 00000 000000	D	I	93	xnop	v2.05			Executed No Operation
011111 .... 01111/	A	I	91	isel	v2.03			Integer Select
111000	DQ	I	58	lq	v2.03			Load Quadword
011111 .... 00000 00111/	X	I	242	lvebx	v2.03			Load Vector Element Byte Indexed
011111 .... 00001 00111/	X	I	242	lvehx	v2.03			Load Vector Element Halfword Indexed
011111 .... 00010 00111/	X	I	243	lvewx	v2.03			Load Vector Element Word Indexed
011111 .... 00000 00110/	X	I	247	lvsl	v2.03			Load Vector for Shift Left
011111 .... 00001 00110/	X	I	247	lvsr	v2.03			Load Vector for Shift Right
011111 .... 00011 00111/	X	I	243	lvx	v2.03			Load Vector Indexed
011111 .... 01011 00111/	X	I	243	lvxl	v2.03			Load Vector Indexed Last
000100 ..... ///// ///// 11000 000100	VX	I	362	mfvscr	v2.03			Move From VSCR
000100 ///// ///// 11001 000100	VX	I	362	mtvscr	v2.03			Move To VSCR
111110 10	DS	I	59	stq	v2.03			Store Quadword
011111 .... 00100 00111/	X	I	245	stvebx	v2.03			Store Vector Element Byte Indexed
011111 .... 00101 00111/	X	I	245	stvehx	v2.03			Store Vector Element Halfword Indexed
011111 .... 00110 00111/	X	I	246	stvewx	v2.03			Store Vector Element Word Indexed
011111 .... 00111 00111/	X	I	246	stvx	v2.03			Store Vector Indexed
011111 .... 01111 00111/	X	I	246	stvxl	v2.03			Store Vector Indexed Last
011111 ..... /.... ..... 01000 10010/	X	III	1038	tlbiel	v2.03	P	64	TLB Invalidate Entry Local
000100 .... 00110 000000	VX	I	269	vaddcuw	v2.03			Vector Add & Write Carry-Out Unsigned Word
000100 .... 00000 001010	VX	I	321	vaddfp	v2.03			Vector Add Floating-Point
000100 .... 01100 000000	VX	I	269	vaddsbs	v2.03			Vector Add Signed Byte Saturate

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 11 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
000100 .... 01101 000000	VX	I	269	vaddshs	v2.03			Vector Add Signed Halfword Saturate
000100 .... 01110 000000	VX	I	270	vaddsws	v2.03			Vector Add Signed Word Saturate
000100 .... 00000 000000	VX	I	270	vaddubm	v2.03			Vector Add Unsigned Byte Modulo
000100 .... 01000 000000	VX	I	272	vaddubs	v2.03			Vector Add Unsigned Byte Saturate
000100 .... 00001 000000	VX	I	271	vadduhm	v2.03			Vector Add Unsigned Halfword Modulo
000100 .... 01001 000000	VX	I	272	vadduhs	v2.03			Vector Add Unsigned Halfword Saturate
000100 .... 00010 000000	VX	I	271	vadduwm	v2.03			Vector Add Unsigned Word Modulo
000100 .... 01010 000000	VX	I	272	vadduws	v2.03			Vector Add Unsigned Word Saturate
000100 .... 10000 000100	VX	I	312	vand	v2.03			Vector Logical AND
000100 .... 10001 000100	VX	I	312	vandc	v2.03			Vector Logical AND with Complement
000100 .... 10100 000010	VX	I	295	vavgsb	v2.03			Vector Average Signed Byte
000100 .... 10101 000010	VX	I	295	vavgsh	v2.03			Vector Average Signed Halfword
000100 .... 10110 000010	VX	I	295	vavgsw	v2.03			Vector Average Signed Word
000100 .... 10000 000010	VX	I	296	vavgub	v2.03			Vector Average Unsigned Byte
000100 .... 10001 000010	VX	I	296	vavguh	v2.03			Vector Average Unsigned Halfword
000100 .... 10010 000010	VX	I	296	vavguw	v2.03			Vector Average Unsigned Word
000100 .... 01101 001010	VX	I	325	vcfsx	v2.03			Vector Convert with round to nearest Signed Word format to FP
000100 .... 01100 001010	VX	I	325	vcfux	v2.03			Vector Convert with round to nearest Unsigned Word format to FP
000100 .... .1111 000110	VC	I	328	vcmpbfp[.]	v2.03			Vector Compare Bounds Floating-Point
000100 .... .0011 000110	VC	I	329	vcmpeqfp[.]	v2.03			Vector Compare Equal To Floating-Point
000100 .... .0000 000110	VC	I	303	vcmpequb[.]	v2.03			Vector Compare Equal To Unsigned Byte
000100 .... .0001 000110	VC	I	303	vcmpequh[.]	v2.03			Vector Compare Equal To Unsigned Halfword
000100 .... .0010 000110	VC	I	304	vcmpequw[.]	v2.03			Vector Compare Equal To Unsigned Word
000100 .... .0111 000110	VC	I	329	vcmpgefp[.]	v2.03			Vector Compare Greater Than or Equal To Floating-Point
000100 .... .1011 000110	VC	I	330	vcmpgtfp[.]	v2.03			Vector Compare Greater Than Floating-Point
000100 .... .1100 000110	VC	I	305	vcmpgtsb[.]	v2.03			Vector Compare Greater Than Signed Byte
000100 .... .1101 000110	VC	I	306	vcmpgtsh[.]	v2.03			Vector Compare Greater Than Signed Halfword
000100 .... .1110 000110	VC	I	306	vcmpgtsw[.]	v2.03			Vector Compare Greater Than Signed Word
000100 .... .1000 000110	VC	I	307	vcmpgtub[.]	v2.03			Vector Compare Greater Than Unsigned Byte
000100 .... .1001 000110	VC	I	308	vcmpgtuh[.]	v2.03			Vector Compare Greater Than Unsigned Halfword
000100 .... .1010 000110	VC	I	308	vcmpgtuw[.]	v2.03			Vector Compare Greater Than Unsigned Word
000100 .... 01111 001010	VX	I	324	vctsxs	v2.03			Vector Convert with round to zero FP To Signed Word format Saturate
000100 .... 01110 001010	VX	I	324	vctuxs	v2.03			Vector Convert with round to zero FP To Unsigned Word format Saturate
000100 ..... ///// 00110 001010	VX	I	331	vexptefp	v2.03			Vector 2 Raised to the Exponent Estimate Floating-Point
000100 .... 00111 001010	VX	I	331	vlogefp	v2.03			Vector Log Base 2 Estimate Floating-Point
000100 101110	VA	I	322	vmaddfp	v2.03			Vector Multiply-Add Floating-Point
000100 .... 10000 001010	VX	I	323	vmaxfp	v2.03			Vector Maximum Floating-Point
000100 .... 00100 000010	VX	I	299	vmaxsb	v2.03			Vector Maximum Signed Byte
000100 .... 00101 000010	VX	I	300	vmaxsh	v2.03			Vector Maximum Signed Halfword
000100 .... 00110 000010	VX	I	300	vmaxsw	v2.03			Vector Maximum Signed Word
000100 .... 00000 000010	VX	I	299	vmaxub	v2.03			Vector Maximum Unsigned Byte
000100 .... 00001 000010	VX	I	300	vmaxuh	v2.03			Vector Maximum Unsigned Halfword
000100 .... 00010 000010	VX	I	300	vmaxuw	v2.03			Vector Maximum Unsigned Word
000100 100000	VA	I	285	vmhaddshs	v2.03			Vector Multiply-High-Add Signed Halfword Saturate
000100 100001	VA	I	285	vmhraddshs	v2.03			Vector Multiply-High-Round-Add Signed Halfword Saturate
000100 .... 10001 001010	VX	I	323	vminfp	v2.03			Vector Minimum Floating-Point
000100 .... 01100 000010	VX	I	301	vminsb	v2.03			Vector Minimum Signed Byte
000100 .... 01101 000010	VX	I	302	vminsh	v2.03			Vector Minimum Signed Halfword
000100 .... 01110 000010	VX	I	302	vminsw	v2.03			Vector Minimum Signed Word
000100 .... 01000 000010	VX	I	301	vminub	v2.03			Vector Minimum Unsigned Byte
000100 .... 01001 000010	VX	I	302	vminuh	v2.03			Vector Minimum Unsigned Halfword
000100 .... 01010 000010	VX	I	302	vminuw	v2.03			Vector Minimum Unsigned Word
000100 100010	VA	I	286	vmladduhm	v2.03			Vector Multiply-Low-Add Unsigned Halfword Modulo
000100 .... 00000 001100	VX	I	255	vmrghb	v2.03			Vector Merge High Byte

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 12 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
000100 .... 00001 001100	VX	I	255	vmrghh	v2.03			Vector Merge High Halfword
000100 .... 00010 001100	VX	I	256	vmrghw	v2.03			Vector Merge High Word
000100 .... 00100 001100	VX	I	255	vmrglb	v2.03			Vector Merge Low Byte
000100 .... 00101 001100	VX	I	255	vmrglh	v2.03			Vector Merge Low Halfword
000100 .... 00110 001100	VX	I	256	vmrglw	v2.03			Vector Merge Low Word
000100 100101	VA	I	287	vmsummbm	v2.03			Vector Multiply-Sum Mixed Byte Modulo
000100 101000	VA	I	287	vmsumshm	v2.03			Vector Multiply-Sum Signed Halfword Modulo
000100 101001	VA	I	288	vmsumshs	v2.03			Vector Multiply-Sum Signed Halfword Saturate
000100 100100	VA	I	286	vmsumubm	v2.03			Vector Multiply-Sum Unsigned Byte Modulo
000100 100110	VA	I	288	vmsumuhm	v2.03			Vector Multiply-Sum Unsigned Halfword Modulo
000100 100111	VA	I	289	vmsumuhs	v2.03			Vector Multiply-Sum Unsigned Halfword Saturate
000100 .... 01100 001000	VX	I	281	vmulesb	v2.03			Vector Multiply Even Signed Byte
000100 .... 01101 001000	VX	I	282	vmulesh	v2.03			Vector Multiply Even Signed Halfword
000100 .... 01000 001000	VX	I	281	vmuleub	v2.03			Vector Multiply Even Unsigned Byte
000100 .... 01001 001000	VX	I	282	vmuleuh	v2.03			Vector Multiply Even Unsigned Halfword
000100 .... 00100 001000	VX	I	281	vmulosb	v2.03			Vector Multiply Odd Signed Byte
000100 .... 00101 001000	VX	I	282	vmulosh	v2.03			Vector Multiply Odd Signed Halfword
000100 .... 00000 001000	VX	I	281	vmuloub	v2.03			Vector Multiply Odd Unsigned Byte
000100 .... 00001 001000	VX	I	282	vmulouh	v2.03			Vector Multiply Odd Unsigned Halfword
000100 101111	VA	I	322	vnmsubfp	v2.03			Vector Negative Multiply-Subtract Floating-Point
000100 .... 10100 000100	VX	I	313	vnor	v2.03			Vector Logical NOR
000100 .... 10010 000100	VX	I	313	vor	v2.03			Vector Logical OR
000100 101011	VA	I	260	vperm	v2.03			Vector Permute
000100 .... 01100 001110	VX	I	248	vpkpx	v2.03			Vector Pack Pixel
000100 .... 00110 001110	VX	I	249	vpkshss	v2.03			Vector Pack Signed Halfword Signed Saturate
000100 .... 00100 001110	VX	I	250	vpkshus	v2.03			Vector Pack Signed Halfword Unsigned Saturate
000100 .... 00111 001110	VX	I	250	vpkswss	v2.03			Vector Pack Signed Word Signed Saturate
000100 .... 00101 001110	VX	I	251	vpkswus	v2.03			Vector Pack Signed Word Unsigned Saturate
000100 .... 00000 001110	VX	I	251	vpkuhum	v2.03			Vector Pack Unsigned Halfword Unsigned Modulo
000100 .... 00010 001110	VX	I	252	vpkuhus	v2.03			Vector Pack Unsigned Halfword Unsigned Saturate
000100 .... 00001 001110	VX	I	252	vpkuwum	v2.03			Vector Pack Unsigned Word Unsigned Modulo
000100 .... 00011 001110	VX	I	252	vpkuwus	v2.03			Vector Pack Unsigned Word Unsigned Saturate
000100 ..... ///// 00100 001010	VX	I	332	vrefp	v2.03			Vector Reciprocal Estimate Floating-Point
000100 ..... ///// 01011 001010	VX	I	326	vrfim	v2.03			Vector Round to Floating-Point Integral toward -Infinity
000100 ..... ///// 01000 001010	VX	I	326	vrfin	v2.03			Vector Round to Floating-Point Integral Nearest
000100 ..... ///// 01010 001010	VX	I	326	vrfip	v2.03			Vector Round to Floating-Point Integral toward +Infinity
000100 ..... ///// 01001 001010	VX	I	327	vrfiz	v2.03			Vector Round to Floating-Point Integral toward Zero
000100 .... 00000 000100	VX	I	315	vrlb	v2.03			Vector Rotate Left Byte
000100 .... 00001 000100	VX	I	315	vrlh	v2.03			Vector Rotate Left Halfword
000100 .... 00010 000100	VX	I	315	vrlw	v2.03			Vector Rotate Left Word
000100 ..... ///// 00101 001010	VX	I	332	vrsqrtefp	v2.03			Vector Reciprocal Square Root Estimate Floating-Point
000100 101010	VA	I	261	vsel	v2.03			Vector Select
000100 .... 00111 000100	VX	I	264	vsl	v2.03			Vector Shift Left
000100 .... 00100 000100	VX	I	316	vslb	v2.03			Vector Shift Left Byte
000100 .... /.... 101100	VA	I	263	vsldoi	v2.03			Vector Shift Left Double by Octet Immediate
000100 .... 00101 000100	VX	I	316	vslh	v2.03			Vector Shift Left Halfword
000100 .... 10000 001100	VX	I	264	vslo	v2.03			Vector Shift Left by Octet
000100 .... 00110 000100	VX	I	316	vslw	v2.03			Vector Shift Left Word
000100 ..... /.... ..... 01000 001100	VX	I	258	vspltb	v2.03			Vector Splat Byte
000100 ..... //... ..... 01001 001100	VX	I	258	vsplth	v2.03			Vector Splat Halfword
000100 .... ///// 01100 001100	VX	I	259	vspltisb	v2.03			Vector Splat Immediate Signed Byte
000100 .... ///// 01101 001100	VX	I	259	vspltish	v2.03			Vector Splat Immediate Signed Halfword
000100 .... ///// 01110 001100	VX	I	259	vspltisw	v2.03			Vector Splat Immediate Signed Word
000100 ..... ///.. ..... 01010 001100	VX	I	258	vspltw	v2.03			Vector Splat Word
000100 .... 01011 000100	VX	I	264	vsr	v2.03			Vector Shift Right
000100 .... 01100 000100	VX	I	318	vsrab	v2.03			Vector Shift Right Algebraic Byte

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 13 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
000100 .... 01101 000100	VX	I	318	vsrah	v2.03			Vector Shift Right Algebraic Halfword
000100 .... 01110 000100	VX	I	318	vsraw	v2.03			Vector Shift Right Algebraic Word
000100 .... 01000 000100	VX	I	317	vsrb	v2.03			Vector Shift Right Byte
000100 .... 01001 000100	VX	I	317	vsrh	v2.03			Vector Shift Right Halfword
000100 .... 10001 001100	VX	I	264	vsro	v2.03			Vector Shift Right by Octet
000100 .... 01010 000100	VX	I	317	vsrw	v2.03			Vector Shift Right Word
000100 .... 10110 000000	VX	I	275	vsubcuw	v2.03			Vector Subtract & Write Carry-Out Unsigned Word
000100 .... 00001 001010	VX	I	321	vsubfp	v2.03			Vector Subtract Floating-Point
000100 .... 11100 000000	VX	I	275	vsubsbs	v2.03			Vector Subtract Signed Byte Saturate
000100 .... 11101 000000	VX	I	275	vsubshs	v2.03			Vector Subtract Signed Halfword Saturate
000100 .... 11110 000000	VX	I	276	vsubsws	v2.03			Vector Subtract Signed Word Saturate
000100 .... 10000 000000	VX	I	277	vsububm	v2.03			Vector Subtract Unsigned Byte Modulo
000100 .... 11000 000000	VX	I	278	vsububs	v2.03			Vector Subtract Unsigned Byte Saturate
000100 .... 10001 000000	VX	I	277	vsubuhm	v2.03			Vector Subtract Unsigned Halfword Modulo
000100 .... 11001 000000	VX	I	278	vsubuhs	v2.03			Vector Subtract Unsigned Halfword Saturate
000100 .... 10010 000000	VX	I	277	vsubuwm	v2.03			Vector Subtract Unsigned Word Modulo
000100 .... 11010 000000	VX	I	278	vsubuws	v2.03			Vector Subtract Unsigned Word Saturate
000100 .... 11010 001000	VX	I	290	vsum2sws	v2.03			Vector Sum across Half Signed Word Saturate
000100 .... 11100 001000	VX	I	291	vsum4sbs	v2.03			Vector Sum across Quarter Signed Byte Saturate
000100 .... 11001 001000	VX	I	291	vsum4shs	v2.03			Vector Sum across Quarter Signed Halfword Saturate
000100 .... 11000 001000	VX	I	292	vsum4ubs	v2.03			Vector Sum across Quarter Unsigned Byte Saturate
000100 .... 11110 001000	VX	I	290	vsumsws	v2.03			Vector Sum across Signed Word Saturate
000100 ..... ///// 01101 001110	VX	I	253	vupkhpx	v2.03			Vector Unpack High Pixel
000100 ..... ///// 01000 001110	VX	I	254	vupkhsb	v2.03			Vector Unpack High Signed Byte
000100 ..... ///// 01001 001110	VX	I	254	vupkhsh	v2.03			Vector Unpack High Signed Halfword
000100 ..... ///// 01111 001110	VX	I	253	vupklpx	v2.03			Vector Unpack Low Pixel
000100 ..... ///// 01010 001110	VX	I	254	vupklsb	v2.03			Vector Unpack Low Signed Byte
000100 ..... ///// 01011 001110	VX	I	254	vupklsh	v2.03			Vector Unpack Low Signed Halfword
000100 .... 10011 000100	VX	I	313	vxor	v2.03			Vector Logical XOR
111111 ..... ///// ..... ///// 11000.	A	I	154	fre[.]	v2.02			Floating Reciprocal Estimate
111111 ..... ///// ..... 01111 01000.	X	I	166	frim[.]	v2.02			Floating Round To Integer Minus
111111 ..... ///// ..... 01100 01000.	X	I	166	frin[.]	v2.02			Floating Round To Integer Nearest
111111 ..... ///// ..... 01110 01000.	X	I	166	frip[.]	v2.02			Floating Round To Integer Plus
111111 ..... ///// ..... 01101 01000.	X	I	166	friz[.]	v2.02			Floating Round To Integer Zero
111011 ..... ///// ..... ///// 11010.	A	I	155	frsqrtes[.]	v2.02			Floating Reciprocal Square Root Estimate Single
010011 ///// ///// ///// 01000 10010/	XL	III	956	hrfid	v2.02	HV		Return From Interrupt Doubleword Hypervisor
011111 .... ///// 00011 11010/	X	I	97	popcntb	v2.02			Population Count Byte
011111 ..... 1.... ..../ 00000 10011/	XFX	I	122	mfocrf	v2.01			Move From One CR Field
011111 ..... 1.... ..../ 00100 10000/	XFX	I	121	mtocrf	v2.01			Move To One CR Field
011111 ..... ///// 11100 10011/	X	III	1031	slbmfee	v2.00	P		SLB Move From Entry ESID
011111 ..... ///// 11010 10011/	X	III	1030	slbmfev	v2.00	P		SLB Move From Entry VSID
011111 ..... ///// 01100 10010/	X	III	1029	slbmte	v2.00	P		SLB Move To Entry
011111 .... ///// 00001 11010.	X	I	99	cntlzd[.]	PPC		SR	Count Leading Zeros Doubleword
011111 ///.. .... 00010 10110/	X	II	852	dcbf	PPC			Data Cache Block Flush
011111 ///// 00001 10110/	X	II	851	dcbst	PPC			Data Cache Block Store
011111 .... 01000 10110/	X	II	849	dcbt	PPC			Data Cache Block Touch
011111 .... 00111 10110/	X	II	850	dcbtst	PPC			Data Cache Block Touch for Store
011111 .... .1111 01001.	XO	I	81	divd[o][.]	PPC		SR	Divide Doubleword
011111 .... .1110 01001.	XO	I	81	divdu[o][.]	PPC		SR	Divide Doubleword Unsigned
011111 .... .1111 01011.	XO	I	74	divw[o][.]	PPC		SR	Divide Word
011111 .... .1110 01011.	XO	I	74	divwu[o][.]	PPC		SR	Divide Word Unsigned
011111 ///// ///// ///// 11010 10110/	X	II	875	eieio	PPC			Enforce In-order Execution of I/O
011111 .... ///// 11101 11010.	X	I	96	extsb[.]	PPC		SR	Extend Sign Byte
011111 .... ///// 11110 11010.	X	I	99	extsw[.]	PPC		SR	Extend Sign Word
111011 .... ///// 10101.	A	I	152	fadds[.]	PPC			Floating Add Single
111111 ..... ///// ..... 11010 01110.	X	I	163	fcfid[.]	PPC			Floating Convert with round Signed Doubleword to Double-Precision format

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 14 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111111 ..... ///// ..... 11001 01110.	X	I	159	fctid[.]	PPC			Floating Convert with round Double-Precision To Signed Doubleword format
111111 ..... ///// ..... 11001 01111.	X	I	160	fctidz[.]	PPC			Floating Convert with round to Zero Double-Precision To Signed Doubleword format
111011 .... ///// 10010.	A	I	153	fdivs[.]	PPC			Floating Divide Single
111011 .... 11101.	A	I	157	fmadds[.]	PPC			Floating Multiply-Add Single
111011 .... 11100.	A	I	158	fmsubs[.]	PPC			Floating Multiply-Subtract Single
111011 .... ///// ..... 11001.	A	I	153	fmuls[.]	PPC			Floating Multiply Single
111011 .... 11111.	A	I	158	fnmadds[.]	PPC			Floating Negative Multiply-Add Single
111011 .... 11110.	A	I	158	fnmsubs[.]	PPC			Floating Negative Multiply-Subtract Single
111011 ..... ///// ..... ///// 11000.	A	I	154	fres[.]	PPC			Floating Reciprocal Estimate Single
111111 ..... ///// ..... ///// 11010.	A	I	155	frsqrte[.]	PPC			Floating Reciprocal Square Root Estimate
111111 .... 10111.	A	I	168	fsel[.]	PPC			Floating Select
111011 ..... ///// ..... ///// 10110.	A	I	154	fsqrts[.]	PPC			Floating Square Root Single
111011 .... ///// 10100.	A	I	152	fsubs[.]	PPC			Floating Subtract Single
011111 ///// 11110 10110/	X	II	840	icbi	PPC			Instruction Cache Block Invalidate
111010 00	DS	I	53	ld	PPC			Load Doubleword
011111 .... 00010 10100/	X	II	869	ldarx	PPC			Load Doubleword And Reserve Indexed
111010 01	DS	I	53	ldu	PPC			Load Doubleword with Update
011111 .... 00001 10101/	X	I	53	ldux	PPC			Load Doubleword with Update Indexed
011111 .... 00000 10101/	X	I	53	ldx	PPC			Load Doubleword Indexed
111010 10	DS	I	52	lwa	PPC			Load Word Algebraic
011111 .... 00000 10100/	X	II	865	lwarx	PPC			Load Word & Reserve Indexed
011111 .... 01011 10101/	X	I	52	lwaux	PPC			Load Word Algebraic with Update Indexed
011111 .... 01010 10101/	X	I	52	lwax	PPC			Load Word Algebraic Indexed
011111 .... 01011 10011/	X	II	898	mftb	PPC			Move From Time Base
011111 ..... ////. ///// 00101 10010/	X	III	978	mtmsrd	PPC	P		Move To MSR Doubleword
011111 .... /0010 01001.	XO	I	79	mulhd[.]	PPC		SR	Multiply High Doubleword
011111 .... /0000 01001.	XO	I	79	mulhdu[.]	PPC		SR	Multiply High Doubleword Unsigned
011111 .... /0010 01011.	XO	I	73	mulhw[.]	PPC		SR	Multiply High Word
011111 .... /0000 01011.	XO	I	73	mulhwu[.]	PPC		SR	Multiply High Word Unsigned
011111 .... .0111 01001.	XO	I	79	mulld[o][.]	PPC		SR	Multiply Low Doubleword
010011 ///// ///// ///// 00000 10010/	XL	III	955	rfid	PPC	P		Return from Interrupt Doubleword
011110 .... .1000.	MDS	I	104	rldcl[.]	PPC		SR	Rotate Left Doubleword then Clear Left
011110 .... .1001.	MDS	I	104	rldcr[.]	PPC		SR	Rotate Left Doubleword then Clear Right
011110 .... .010..	MD	I	105	rldic[.]	PPC		SR	Rotate Left Doubleword Immediate then Clear
011110 .... .000..	MD	I	105	rldicl[.]	PPC		SR	Rotate Left Doubleword Immediate then Clear Left
011110 .... .001..	MD	I	106	rldicr[.]	PPC		SR	Rotate Left Doubleword Immediate then Clear Right
011110 .... .011..	MD	I	106	rldimi[.]	PPC		SR	Rotate Left Doubleword Immediate then Mask Insert
010001 ///// ///// ////. ..... .///1/	SC	I	42	sc	PPC			System Call
011111 //... ///// ///// 01111 10010/	X	III	1026	slbia	PPC	P		SLB Invalidate All
011111 ///// ///// 01101 10010/	X	III	1024	slbie	PPC	P		SLB Invalidate Entry
011111 .... 00000 11011.	X	I	109	sld[.]	PPC		SR	Shift Left Doubleword
011111 .... 11000 11010.	X	I	110	srad[.]	PPC		SR	Shift Right Algebraic Doubleword
011111 .... 11001 1101..	XS	I	110	sradi[.]	PPC		SR	Shift Right Algebraic Doubleword Immediate
011111 .... 10000 11011.	X	I	109	srd[.]	PPC		SR	Shift Right Doubleword
111110 00	DS	I	57	std	PPC			Store Doubleword
011111 .... 00110 101101	X	II	869	stdcx.	PPC			Store Doubleword Conditional Indexed & record
111110 01	DS	I	57	stdu	PPC			Store Doubleword with Update
011111 .... 00101 10101/	X	I	57	stdux	PPC			Store Doubleword with Update Indexed
011111 .... 00100 10101/	X	I	57	stdx	PPC			Store Doubleword Indexed
011111 .... 11110 10111/	X	I	147	stfiwx	PPC			Store Floating as Integer Word Indexed
011111 .... 00100 101101	X	II	868	stwcx.	PPC			Store Word Conditional Indexed & record
011111 .... .0001 01000.	XO	I	69	subf[o][.]	PPC		SR	Subtract From
011111 .... 00010 00100/	X	I	91	td	PPC			Trap Doubleword
000010	D	I	91	tdi	PPC			Trap Doubleword Immediate
011111 ///// ///// ///// 10001 10110/	X	III	1042	tlbsync	PPC	HV/P		TLB Synchronize

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 15 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111111 ..... ///// ..... 00000 01110.	X	I	161	fctiw[.]	P2			Floating Convert with round Double-Precision To Signed Word format
111111 ..... ///// ..... 00000 01111.	X	I	162	fctiwz[.]	P2			Floating Convert with round to Zero Double-Precision To Signed Word format
111111 ..... ///// ..... ///// 10110.	A	I	154	fsqrt[.]	P2			Floating Square Root
011111 .... .1000 01010.	XO	I	69	add[o][.]	P1		SR	Add
011111 .... .0000 01010.	XO	I	70	addc[o][.]	P1		SR	Add Carrying
011111 .... .0100 01010.	XO	I	71	adde[o][.]	P1		SR	Add Extended
001110	D	I	67	addi	P1			Add Immediate
001100	D	I	69	addic	P1		SR	Add Immediate Carrying
001101	D	I	69	addic.	P1		SR	Add Immediate Carrying & record
001111	D	I	67	addis	P1			Add Immediate Shifted
011111 .... ///// .0111 01010.	XO	I	71	addme[o][.]	P1		SR	Add to Minus One Extended
011111 .... ///// .0110 01010.	XO	I	72	addze[o][.]	P1		SR	Add to Zero Extended
011111 .... 00000 11100.	X	I	94	and[.]	P1		SR	AND
011111 .... 00001 11100.	X	I	95	andc[.]	P1		SR	AND with Complement
011100	D	I	92	andi.	P1		SR	AND Immediate & record
011101	D	I	92	andis.	P1		SR	AND Immediate Shifted & record
010010	I	I	37	b[l][a]	P1			Branch [& Link] [Absolute]
010000	B	I	37	bc[l][a]	P1		CT	Branch Conditional [& Link] [Absolute]
010011 .... ///.. 10000 10000.	XL	I	38	bcctr[l]	P1		CT	Branch Conditional to CTR [& Link]
010011 .... ///.. 00000 10000.	XL	I	38	bclr[l]	P1		CT	Branch Conditional to LR [& Link]
011111 .../. 00000 00000/	X	I	85	cmp	P1			Compare
001011 .../.	D	I	85	cmpi	P1			Compare Immediate
011111 .../. 00001 00000/	X	I	86	cmpl	P1			Compare Logical
001010 .../.	D	I	86	cmpli	P1			Compare Logical Immediate
011111 .... ///// 00000 11010.	X	I	96	cntlzw[.]	P1		SR	Count Leading Zeros Word
010011 .... 01000 00001/	XL	I	40	crand	P1			CR AND
010011 .... 00100 00001/	XL	I	41	crandc	P1			CR AND with Complement
010011 .... 01001 00001/	XL	I	41	creqv	P1			CR Equivalent
010011 .... 00111 00001/	XL	I	40	crnand	P1			CR NAND
010011 .... 00001 00001/	XL	I	41	crnor	P1			CR NOR
010011 .... 01110 00001/	XL	I	40	cror	P1			CR OR
010011 .... 01101 00001/	XL	I	41	crorc	P1			CR OR with Complement
010011 .... 00110 00001/	XL	I	40	crxor	P1			CR XOR
011111 ///// 11111 10110/	X	II	851	dcbz	P1			Data Cache Block Zero
011111 .... 01000 11100.	X	I	95	eqv[.]	P1		SR	Equivalent
011111 .... ///// 11100 11010.	X	I	96	extsh[.]	P1		SR	Extend Sign Halfword
111111 ..... ///// ..... 01000 01000.	X	I	150	fabs[.]	P1			Floating Absolute
111111 .... ///// 10101.	A	I	152	fadd[.]	P1			Floating Add
111111 .../// 00001 00000/	X	I	167	fcmpo	P1			Floating Compare Ordered
111111 .../// 00000 00000/	X	I	167	fcmpu	P1			Floating Compare Unordered
111111 .... ///// 10010.	A	I	153	fdiv[.]	P1			Floating Divide
111111 .... 11101.	A	I	157	fmadd[.]	P1			Floating Multiply-Add
111111 ..... ///// ..... 00010 01000.	X	I	150	fmr[.]	P1			Floating Move Register
111111 .... 11100.	A	I	158	fmsub[.]	P1			Floating Multiply-Subtract
111111 .... ///// ..... 11001.	A	I	153	fmul[.]	P1			Floating Multiply
111111 ..... ///// ..... 00100 01000.	X	I	150	fnabs[.]	P1			Floating Negative Absolute Value
111111 ..... ///// ..... 00001 01000.	X	I	150	fneg[.]	P1			Floating Negate
111111 .... 11111.	A	I	158	fnmadd[.]	P1			Floating Negative Multiply-Add
111111 .... 11110.	A	I	158	fnmsub[.]	P1			Floating Negative Multiply-Subtract
111111 ..... ///// ..... 00000 01100.	X	I	159	frsp[.]	P1			Floating Round to Single-Precision
111111 .... ///// 10100.	A	I	152	fsub[.]	P1			Floating Subtract
010011 ///// ///// ///// 00100 10110/	XL	II	863	isync	P1			Instruction Synchronize
100010	D	I	48	lbz	P1			Load Byte & Zero
100011	D	I	48	lbzu	P1			Load Byte & Zero with Update
011111 .... 00011 10111/	X	I	48	lbzux	P1			Load Byte & Zero with Update Indexed

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 16 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
011111 .... 00010 10111/	X	I	48	lbzx	P1			Load Byte & Zero Indexed
110010	D	I	142	lfd	P1			Load Floating Double
110011	D	I	142	lfdu	P1			Load Floating Double with Update
011111 .... 10011 10111/	X	I	143	lfdux	P1			Load Floating Double with Update Indexed
011111 .... 10010 10111/	X	I	142	lfdx	P1			Load Floating Double Indexed
110000	D	I	140	lfs	P1			Load Floating Single
110001	D	I	141	lfsu	P1			Load Floating Single with Update
011111 .... 10001 10111/	X	I	142	lfsux	P1			Load Floating Single with Update Indexed
011111 .... 10000 10111/	X	I	141	lfsx	P1			Load Floating Single Indexed
101010	D	I	50	lha	P1			Load Halfword Algebraic
101011	D	I	50	lhau	P1			Load Halfword Algebraic with Update
011111 .... 01011 10111/	X	I	50	lhaux	P1			Load Halfword Algebraic with Update Indexed
011111 .... 01010 10111/	X	I	50	lhax	P1			Load Halfword Algebraic Indexed
011111 .... 11000 10110/	X	I	60	lhbrx	P1			Load Halfword Byte-Reverse Indexed
101000	D	I	49	lhz	P1			Load Halfword & Zero
101001	D	I	49	lhzu	P1			Load Halfword & Zero with Update
011111 .... 01001 10111/	X	I	49	lhzux	P1			Load Halfword & Zero with Update Indexed
011111 .... 01000 10111/	X	I	49	lhzx	P1			Load Halfword & Zero Indexed
101110	D	I	62	lmw	P1			Load Multiple Word
011111 .... 10010 10101/	X	I	64	lswi	P1			Load String Word Immediate
011111 .... 10000 10101/	X	I	64	lswx	P1			Load String Word Indexed
011111 .... 10000 10110/	X	I	60	lwbrx	P1			Load Word Byte-Reverse Indexed
100000	D	I	51	lwz	P1			Load Word & Zero
100001	D	I	51	lwzu	P1			Load Word & Zero with Update
011111 .... 00001 10111/	X	I	51	lwzux	P1			Load Word & Zero with Update Indexed
011111 .... 00000 10111/	X	I	51	lwzx	P1			Load Word & Zero Indexed
010011 ..// ..// ///// 00000 00000/	XL	I	41	mcrf	P1			Move CR Field
111111 ..// ..// ///// 00010 00000/	X	I	171	mcrfs	P1			Move To CR from FPSCR
011111 ..... 0//// ///// 00000 10011/	XFX	I	122	mfcr	P1			Move From CR
111111 ..... 00000 ///// 10010 00111.	X	I	170	mffs[.]	P1			Move From FPSCR
011111 ..... ///// ///// 00010 10011/	X	III	979	mfmsr	P1	P		Move From MSR
011111 .... 01010 10011/	X	X	119 975	mfspr	P1	O		Move From SPR
011111 ..... 0.... ..../ 00100 10000/	XFX	I	121	mtcrf	P1			Move To CR Fields
111111 ..... ///// ///// 00010 00110.	X	I	173	mtfsb0[.]	P1			Move To FPSCR Bit 0
111111 ..... ///// ///// 00001 00110.	X	I	173	mtfsb1[.]	P1			Move To FPSCR Bit 1
111111 .... 10110 00111.	XFL	I	172	mtfsf[.]	P1			Move To FPSCR Fields
111111 ..// ////. ..../ 00100 00110.	X	I	172	mtfsfi[.]	P1			Move To FPSCR Field Immediate
011111 ..... ////. ///// 00100 10010/	X	III	977	mtmsr	P1	P		Move To MSR
011111 .... 01110 10011/	X	X	117 974	mtspr	P1	O		Move To SPR
000111	D	I	73	mulli	P1			Multiply Low Immediate
011111 .... .0111 01011.	XO	I	73	mullw[o][.]	P1		SR	Multiply Low Word
011111 .... 01110 11100.	X	I	94	nand[.]	P1		SR	NAND
011111 .... ///// .0011 01000.	XO	I	72	neg[o][.]	P1		SR	Negate
011111 .... 00011 11100.	X	I	95	nor[.]	P1		SR	NOR
011111 .... 01101 11100.	X	I	94	or[.]	P1		SR	OR
011111 .... 01100 11100.	X	I	95	orc[.]	P1		SR	OR with Complement
011000	D	I	92	ori	P1			OR Immediate
011001	D	I	93	oris	P1			OR Immediate Shifted
010100	M	I	103	rlwimi[.]	P1		SR	Rotate Left Word Immediate then Mask Insert
010101	M	I	102	rlwinm[.]	P1		SR	Rotate Left Word Immediate then AND with Mask
010111	M	I	103	rlwnm[.]	P1		SR	Rotate Left Word then AND with Mask
011111 .... 00000 11000.	X	I	107	slw[.]	P1		SR	Shift Left Word
011111 .... 11000 11000.	X	I	108	sraw[.]	P1		SR	Shift Right Algebraic Word
011111 .... 11001 11000.	X	I	108	srawi[.]	P1		SR	Shift Right Algebraic Word Immediate
011111 .... 10000 11000.	X	I	107	srw[.]	P1		SR	Shift Right Word

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 17 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
100110	D	I	54	stb	P1			Store Byte
100111	D	I	54	stbu	P1			Store Byte with Update
011111 .... 00111 10111/	X	I	54	stbux	P1			Store Byte with Update Indexed
011111 .... 00110 10111/	X	I	54	stbx	P1			Store Byte Indexed
110110	D	I	146	stfd	P1			Store Floating Double
110111	D	I	146	stfdu	P1			Store Floating Double with Update
011111 .... 10111 10111/	X	I	146	stfdux	P1			Store Floating Double with Update Indexed
011111 .... 10110 10111/	X	I	146	stfdx	P1			Store Floating Double Indexed
110100	D	I	145	stfs	P1			Store Floating Single
110101	D	I	145	stfsu	P1			Store Floating Single with Update
011111 .... 10101 10111/	X	I	145	stfsux	P1			Store Floating Single with Update Indexed
011111 .... 10100 10111/	X	I	145	stfsx	P1			Store Floating Single Indexed
101100	D	I	55	sth	P1			Store Halfword
011111 .... 11100 10110/	X	I	60	sthbrx	P1			Store Halfword Byte-Reverse Indexed
101101	D	I	55	sthu	P1			Store Halfword with Update
011111 .... 01101 10111/	X	I	55	sthux	P1			Store Halfword with Update Indexed
011111 .... 01100 10111/	X	I	55	sthx	P1			Store Halfword Indexed
101111	D	I	62	stmw	P1			Store Multiple Word
011111 .... 10110 10101/	X	I	65	stswi	P1			Store String Word Immediate
011111 .... 10100 10101/	X	I	65	stswx	P1			Store String Word Indexed
100100	D	I	56	stw	P1			Store Word
011111 .... 10100 10110/	X	I	60	stwbrx	P1			Store Word Byte-Reverse Indexed
100101	D	I	56	stwu	P1			Store Word with Update
011111 .... 00101 10111/	X	I	56	stwux	P1			Store Word with Update Indexed
011111 .... 00100 10111/	X	I	56	stwx	P1			Store Word Indexed
011111 .... .0000 01000.	XO	I	70	subfc[o][.]	P1		SR	Subtract From Carrying
011111 .... .0100 01000.	XO	I	71	subfe[o][.]	P1		SR	Subtract From Extended
001000	D	I	70	subfic	P1		SR	Subtract From Immediate Carrying
011111 .... ///// .0111 01000.	XO	I	71	subfme[o][.]	P1		SR	Subtract From Minus One Extended
011111 .... ///// .0110 01000.	XO	I	72	subfze[o][.]	P1		SR	Subtract From Zero Extended
011111 ///.. ///// ///// 10010 10110/	X	II	873	sync	P1			Synchronize
011111 ..... /.... ..... 01001 10010/	X	III	1034	tlbie	P1	HV	64	TLB Invalidate Entry
011111 .... 00000 00100/	X	I	90	tw	P1			Trap Word
000011	D	I	90	twi	P1			Trap Word Immediate
011111 .... 01001 11100.	X	I	94	xor[.]	P1		SR	XOR
011010	D	I	93	xori	P1			XOR Immediate
011011	D	I	93	xoris	P1			XOR Immediate Shifted

Figure 89. Power ISA AS Instruction Set Sorted by Version (Sheet 18 of 18)

1. Key to Instruction column.

/	Instruction bit that corresponds to a reserved field, must have a value of 0, otherwise invalid form.
-	Instruction bit that corresponds to an operand bit, may have a value of either 0 or 1.
0	Instruction bit having a value 0.
1	Instruction bit having a value 1.

2. Key to Version column.

P1	Instruction introduced in the POWER Architecture.
P2	Instruction introduced in the POWER2 Architecture.
PPC	Instruction introduced in the PowerPC Architecture prior to v2.00.
v2.00	Instruction introduced in the PowerPC Architecture Version 2.00.
v2.01	Instruction introduced in the PowerPC Architecture Version 2.01.
v2.02	Instruction introduced in the PowerPC Architecture Version 2.02.
v2.03	Instruction introduced in the Power ISA Architecture Version 2.03.
v2.04	Instruction introduced in the Power ISA Architecture Version 2.04.
v2.05	Instruction introduced in the Power ISA Architecture Version 2.05.
v2.06	Instruction introduced in the Power ISA Architecture Version 2.06.
v2.07	Instruction introduced in the Power ISA Architecture Version 2.07.
v3.0	Instruction introduced in the Power ISA Architecture Version 3.0.
v3.0B	Instruction introduced in the Power ISA Architecture Version 3.0B.

3. Key to Privilege column.

P	Denotes an instruction that is treated as privileged.
O	Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor, for mtspr), depending on the SPR or PMR number.
PI	Denotes an instruction that is illegal in privileged state.
H	Denotes an instruction that can be executed only in hypervisor state
U	Denotes an instruction that can be executed only in ultravisor state

4. Key to Mode Dependency column.
   Except as described below and in Section 1.11.3, “Effective Address Calculation”, in Book I, all instructions are independent of whether the processor is in 32-bit or 64-bit mode.

CT	If the instruction tests the Count Register, it tests the low-order 32 bits in 32-bit mode and all 64 bits in 64-bit mode.
SR	The setting of status registers (such as XER and CR0) is mode-dependent.
32	The instruction can be executed only in 32-bit mode.
64	The instruction can be executed only in 64-bit mode.

Appendix F. Power ISA Instruction Set Sorted by Mnemonic

This appendix lists all the instructions in the Power ISA, sorted by mnemonic.

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
011111 .... .1000 01010.	XO	I	69	add[o][.]	P1		SR	Add
011111 .... .0000 01010.	XO	I	70	addc[o][.]	P1		SR	Add Carrying
011111 .... .0100 01010.	XO	I	71	adde[o][.]	P1		SR	Add Extended
011111 ....101 01010/	X	I	72	addex	v3.0B			Add Extended using alternate carry
011111 .... /0010 01010/	XO	I	111	addg6s	v2.06			Add & Generate Sixes
001110	D	I	67	addi	P1			Add Immediate
001100	D	I	69	addic	P1		SR	Add Immediate Carrying
001101	D	I	69	addic.	P1		SR	Add Immediate Carrying & record
001111	D	I	67	addis	P1			Add Immediate Shifted
011111 .... ///// .0111 01010.	XO	I	71	addme[o][.]	P1		SR	Add to Minus One Extended
010011 .... 00010.	DX	I	68	addpcis	v3.0			Add PC Immediate Shifted
011111 .... ///// .0110 01010.	XO	I	72	addze[o][.]	P1		SR	Add to Zero Extended
011111 .... 00000 11100.	X	I	94	and[.]	P1		SR	AND
011111 .... 00001 11100.	X	I	95	andc[.]	P1		SR	AND with Complement
011100	D	I	92	andi.	P1		SR	AND Immediate & record
011101	D	I	92	andis.	P1		SR	AND Immediate Shifted & record
010010	I	I	37	b[l][a]	P1			Branch [& Link] [Absolute]
010000	B	I	37	bc[l][a]	P1		CT	Branch Conditional [& Link] [Absolute]
010011 .... ///.. 10000 10000.	XL	I	38	bcctr[l]	P1		CT	Branch Conditional to CTR [& Link]
000100 .... 1.000 000001	VX	I	348	bcdadd.	v2.07			Decimal Add Modulo & record
000100 ..... 00111 ..... 1.110 000001	VX	I	350	bcdcfn.	v3.0			Decimal Convert From National & record
000100 ..... 00010 ..... 1.110 000001	VX	I	354	bcdcfsq.	v3.0			Decimal Convert From Signed Quadword & record
000100 ..... 00110 ..... 1.110 000001	VX	I	351	bcdcfz.	v3.0			Decimal Convert From Zoned & record
000100 .... 01101 000001	VX	I	356	bcdcpsgn.	v3.0			Decimal CopySign & record
000100 ..... 00101 ..... 1/110 000001	VX	I	352	bcdctn.	v3.0			Decimal Convert To National & record
000100 ..... 00000 ..... 1/110 000001	VX	I	354	bcdctsq.	v3.0			Decimal Convert To Signed Quadword & record
000100 ..... 00100 ..... 1.110 000001	VX	I	353	bcdctz.	v3.0			Decimal Convert To Zoned & record
000100 .... 1.011 000001	VX	I	357	bcds.	v3.0			Decimal Shift & record
000100 ..... 11111 ..... 1.110 000001	VX	I	356	bcdsetsgn.	v3.0			Decimal Set Sign & record
000100 .... 1.111 000001	VX	I	359	bcdsr.	v3.0			Decimal Shift & Round & record
000100 .... 1.001 000001	VX	I	348	bcdsub.	v2.07			Decimal Subtract Modulo & record
000100 .... 1.100 000001	VX	I	360	bcdtrunc.	v3.0			Decimal Truncate & record
000100 .... 1/010 000001	VX	I	358	bcdus.	v3.0			Decimal Unsigned Shift & record
000100 .... 1/101 000001	VX	I	361	bcdutrunc.	v3.0			Decimal Unsigned Truncate & record
010011 .... ///.. 00000 10000.	XL	I	38	bclr[l]	P1		CT	Branch Conditional to LR [& Link]
010011 .... ///.. 10001 10000.	XL	I	39	bctar[l]	v2.07			Branch Conditional to BTAR [& Link]
011111 .... 00111 11100/	X	I	100	bpermd	v2.06			Bit Permute Doubleword
011111 .... ///// 01001 11010/	X	I	111	cbcdtd	v2.06			Convert Binary Coded Decimal To Declets
011111 .... ///// 01000 11010/	X	I	111	cdtbcd	v2.06			Convert Declets To Binary Coded Decimal
011111 ///// ///// ///// 01101 01110/	X	I	909	clrbhrb	v2.07			Clear BHRB
011111 .../. 00000 00000/	X	I	85	cmp	P1			Compare

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 1 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
011111 .... 01111 11100/	X	I	97	cmpb	v2.05			Compare Byte
011111 ...// 00111 00000/	X	I	88	cmpeqb	v3.0			Compare Equal Byte
001011 .../.	D	I	85	cmpi	P1			Compare Immediate
011111 .../. 00001 00000/	X	I	86	cmpl	P1			Compare Logical
001010 .../.	D	I	86	cmpli	P1			Compare Logical Immediate
011111 .../. 00110 00000/	X	I	87	cmprb	v3.0			Compare Ranged Byte
011111 .... ///// 00001 11010.	X	I	99	cntlzd[.]	PPC		SR	Count Leading Zeros Doubleword
011111 .... ///// 00000 11010.	X	I	96	cntlzw[.]	P1		SR	Count Leading Zeros Word
011111 .... ///// 10001 11010.	X	I	99	cnttzd[.]	v3.0			Count Trailing Zeros Doubleword
011111 .... ///// 10000 11010.	X	I	96	cnttzw[.]	v3.0			Count Trailing Zeros Word
011111 ////. 11000 00110/	X	II	855	copy	v3.0			Copy
011111 ///// ///// ///// 11010 00110/	X	II	856	cp_abort	v3.0			CP_Abort
010011 .... 01000 00001/	XL	I	40	crand	P1			CR AND
010011 .... 00100 00001/	XL	I	41	crandc	P1			CR AND with Complement
010011 .... 01001 00001/	XL	I	41	creqv	P1			CR Equivalent
010011 .... 00111 00001/	XL	I	40	crnand	P1			CR NAND
010011 .... 00001 00001/	XL	I	41	crnor	P1			CR NOR
010011 .... 01110 00001/	XL	I	40	cror	P1			CR OR
010011 .... 01101 00001/	XL	I	41	crorc	P1			CR OR with Complement
010011 .... 00110 00001/	XL	I	40	crxor	P1			CR XOR
111011 .... 00000 00010.	X	I	193	dadd[.]	v2.05			DFP Add
111111 .... 00000 00010.	X	I	193	daddq[.]	v2.05			DFP Add Quad
011111 ..... ///.. ///// 10111 10011/	X	I	78	darn	v3.0			Deliver A Random Number
011111 ///.. .... 00010 10110/	X	II	852	dcbf	PPC			Data Cache Block Flush
011111 ///// 00001 10110/	X	II	851	dcbst	PPC			Data Cache Block Store
011111 .... 01000 10110/	X	II	849	dcbt	PPC			Data Cache Block Touch
011111 .... 00111 10110/	X	II	850	dcbtst	PPC			Data Cache Block Touch for Store
011111 ///// 11111 10110/	X	II	851	dcbz	P1			Data Cache Block Zero
111011 ..... ///// ..... 11001 00010.	X	I	215	dcffix[.]	v2.06			DFP Convert From Fixed
111111 ..... ///// ..... 11001 00010.	X	I	215	dcffixq[.]	v2.05			DFP Convert From Fixed Quad
111011 ...// 00100 00010/	X	I	199	dcmpo	v2.05			DFP Compare Ordered
111111 ...// 00100 00010/	X	I	199	dcmpoq	v2.05			DFP Compare Ordered Quad
111011 ...// 10100 00010/	X	I	198	dcmpu	v2.05			DFP Compare Unordered
111111 ...// 10100 00010/	X	I	198	dcmpuq	v2.05			DFP Compare Unordered Quad
111011 ..... ///// ..... 01000 00010.	X	I	213	dctdp[.]	v2.05			DFP Convert To DFP Long
111011 ..... ///// ..... 01001 00010.	X	I	215	dctfix[.]	v2.05			DFP Convert To Fixed
111111 ..... ///// ..... 01001 00010.	X	I	215	dctfixq[.]	v2.05			DFP Convert To Fixed Quad
111111 ..... ///// ..... 01000 00010.	X	I	213	dctqpq[.]	v2.05			DFP Convert To DFP Extended
111011 ....// ..... 01010 00010.	X	I	217	ddedpd[.]	v2.05			DFP Decode DPD To BCD
111111 ....// ..... 01010 00010.	X	I	217	ddedpdq[.]	v2.05			DFP Decode DPD To BCD Quad
111011 .... 10001 00010.	X	I	196	ddiv[.]	v2.05			DFP Divide
111111 .... 10001 00010.	X	I	196	ddivq[.]	v2.05			DFP Divide Quad
111011 ..... .//// ..... 11010 00010.	X	I	217	denbcd[.]	v2.05			DFP Encode BCD To DPD
111111 ..... .//// ..... 11010 00010.	X	I	217	denbcdq[.]	v2.05			DFP Encode BCD To DPD Quad
111011 .... 11011 00010.	X	I	218	diex[.]	v2.05			DFP Insert Exponent
111111 .... 11011 00010.	X	I	218	diexq[.]	v2.05			DFP Insert Exponent Quad
011111 .... .1111 01001.	XO	I	81	divd[o][.]	PPC		SR	Divide Doubleword
011111 .... .1101 01001.	XO	I	82	divde[o][.]	v2.06		SR	Divide Doubleword Extended
011111 .... .1100 01001.	XO	I	82	divdeu[o][.]	v2.06		SR	Divide Doubleword Extended Unsigned
011111 .... .1110 01001.	XO	I	81	divdu[o][.]	PPC		SR	Divide Doubleword Unsigned
011111 .... .1111 01011.	XO	I	74	divw[o][.]	PPC		SR	Divide Word
011111 .... .1101 01011.	XO	I	75	divwe[o][.]	v2.06		SR	Divide Word Extended
011111 .... .1100 01011.	XO	I	75	divweu[o][.]	v2.06		SR	Divide Word Extended Unsigned
011111 .... .1110 01011.	XO	I	74	divwu[o][.]	PPC		SR	Divide Word Unsigned
111011 .... 00001 00010.	X	I	195	dmul[.]	v2.05			DFP Multiply
111111 .... 00001 00010.	X	I	195	dmulq[.]	v2.05			DFP Multiply Quad

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 2 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111011 ....000 00011.	Z23	I	204	dqua[.]	v2.05			DFP Quantize
111011 ....010 00011.	Z23	I	203	dquai[.]	v2.05			DFP Quantize Immediate
111111 ....010 00011.	Z23	I	203	dquaiq[.]	v2.05			DFP Quantize Immediate Quad
111111 ....000 00011.	Z23	I	204	dquaq[.]	v2.05			DFP Quantize Quad
111111 ..... ///// ..... 11000 00010.	X	I	214	drdpq[.]	v2.05			DFP Round To DFP Long
111011 ..... ////. ....111 00011.	Z23	I	211	drintn[.]	v2.05			DFP Round To FP Integer Without Inexact
111111 ..... ////. ....111 00011.	Z23	I	211	drintnq[.]	v2.05			DFP Round To FP Integer Without Inexact Quad
111011 ..... ////. ....011 00011.	Z23	I	209	drintx[.]	v2.05			DFP Round To FP Integer With Inexact
111111 ..... ////. ....011 00011.	Z23	I	209	drintxq[.]	v2.05			DFP Round To FP Integer With Inexact Quad
111011 ....001 00011.	Z23	I	206	drrnd[.]	v2.05			DFP Reround
111111 ....001 00011.	Z23	I	206	drrndq[.]	v2.05			DFP Reround Quad
111011 ..... ///// ..... 11000 00010.	X	I	214	drsp[.]	v2.05			DFP Round To DFP Short
111011 .... .0010 00010.	Z22	I	220	dscli[.]	v2.05			DFP Shift Significand Left Immediate
111111 .... .0010 00010.	Z22	I	220	dscliq[.]	v2.05			DFP Shift Significand Left Immediate Quad
111011 .... .0011 00010.	Z22	I	220	dscri[.]	v2.05			DFP Shift Significand Right Immediate
111111 .... .0011 00010.	Z22	I	220	dscriq[.]	v2.05			DFP Shift Significand Right Immediate Quad
111011 .... 10000 00010.	X	I	193	dsub[.]	v2.05			DFP Subtract
111111 .... 10000 00010.	X	I	193	dsubq[.]	v2.05			DFP Subtract Quad
111011 .....// .... .0110 00010/	Z22	I	200	dtstdc	v2.05			DFP Test Data Class
111111 .....// .... .0110 00010/	Z22	I	200	dtstdcq	v2.05			DFP Test Data Class Quad
111011 .....// .... .0111 00010/	Z22	I	200	dtstdg	v2.05			DFP Test Data Group
111111 .....// .... .0111 00010/	Z22	I	200	dtstdgq	v2.05			DFP Test Data Group Quad
111011 .....// 00101 00010/	X	I	201	dtstex	v2.05			DFP Test Exponent
111111 .....// 00101 00010/	X	I	201	dtstexq	v2.05			DFP Test Exponent Quad
111011 .....// 10101 00010/	X	I	202	dtstsf	v2.05			DFP Test Significance
111011 .....// 10101 00011/	X	I	202	dtstsfi	v3.0			DFP Test Significance Immediate
111111 .....// 10101 00011/	X	I	202	dtstsfiq	v3.0			DFP Test Significance Immediate Quad
111111 .....// 10101 00010/	X	I	202	dtstsfq	v2.05			DFP Test Significance Quad
111011 ..... ///// ..... 01011 00010.	X	I	218	dxex[.]	v2.05			DFP Extract Exponent
111111 ..... ///// ..... 01011 00010.	X	I	218	dxexq[.]	v2.05			DFP Extract Exponent Quad
011111 ///// ///// ///// 11010 10110/	X	II	875	eieio	PPC			Enforce In-order Execution of I/O
011111 .... 01000 11100.	X	I	95	eqv[.]	P1		SR	Equivalent
011111 .... ///// 11101 11010.	X	I	96	extsb[.]	PPC		SR	Extend Sign Byte
011111 .... ///// 11100 11010.	X	I	96	extsh[.]	P1		SR	Extend Sign Halfword
011111 .... ///// 11110 11010.	X	I	99	extsw[.]	PPC		SR	Extend Sign Word
011111 .... 11011 1101..	XS	I	110	extswsli[.]	v3.0			Extend Sign Word & Shift Left Immediate
111111 ..... ///// ..... 01000 01000.	X	I	150	fabs[.]	P1			Floating Absolute
111111 .... ///// 10101.	A	I	152	fadd[.]	P1			Floating Add
111011 .... ///// 10101.	A	I	152	fadds[.]	PPC			Floating Add Single
111111 ..... ///// ..... 11010 01110.	X	I	163	fcfid[.]	PPC			Floating Convert with round Signed Doubleword to Double-Precision format
111011 ..... ///// ..... 11010 01110.	X	I	164	fcfids[.]	v2.06			Floating Convert with round Signed Doubleword to Single-Precision format
111111 ..... ///// ..... 11110 01110.	X	I	164	fcfidu[.]	v2.06			Floating Convert with round Unsigned Doubleword to Double-Precision format
111011 ..... ///// ..... 11110 01110.	X	I	165	fcfidus[.]	v2.06			Floating Convert with round Unsigned Doubleword to Single-Precision format
111111 .....// 00001 00000/	X	I	167	fcmpo	P1			Floating Compare Ordered
111111 .....// 00000 00000/	X	I	167	fcmpu	P1			Floating Compare Unordered
111111 .... 00000 01000.	X	I	150	fcpsgn[.]	v2.05			Floating Copy Sign
111111 ..... ///// ..... 11001 01110.	X	I	159	fctid[.]	PPC			Floating Convert with round Double-Precision To Signed Doubleword format
111111 ..... ///// ..... 11101 01110.	X	I	160	fctidu[.]	v2.06			Floating Convert with round Double-Precision To Unsigned Doubleword format
111111 ..... ///// ..... 11101 01111.	X	I	161	fctiduz[.]	v2.06			Floating Convert with round to Zero Double-Precision To Unsigned Doubleword format
111111 ..... ///// ..... 11001 01111.	X	I	160	fctidz[.]	PPC			Floating Convert with round to Zero Double-Precision To Signed Doubleword format

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 3 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111111 ..... ///// ..... 00000 01110.	X	I	161	fctiw[.]	P2			Floating Convert with round Double-Precision To Signed Word format
111111 ..... ///// ..... 00100 01110.	X	I	162	fctiwu[.]	v2.06			Floating Convert with round Double-Precision To Unsigned Word format
111111 ..... ///// ..... 00100 01111.	X	I	163	fctiwuz[.]	v2.06			Floating Convert with round to Zero Double-Precision To Unsigned Word format
111111 ..... ///// ..... 00000 01111.	X	I	162	fctiwz[.]	P2			Floating Convert with round to Zero Double-Precision To Signed Word format
111111 .... ///// 10010.	A	I	153	fdiv[.]	P1			Floating Divide
111011 .... ///// 10010.	A	I	153	fdivs[.]	PPC			Floating Divide Single
111111 .... 11101.	A	I	157	fmadd[.]	P1			Floating Multiply-Add
111011 .... 11101.	A	I	157	fmadds[.]	PPC			Floating Multiply-Add Single
111111 ..... ///// ..... 00010 01000.	X	I	150	fmr[.]	P1			Floating Move Register
111111 .... 11110 00110/	X	I	151	fmrgew	v2.07			Floating Merge Even Word
111111 .... 11010 00110/	X	I	151	fmrgow	v2.07			Floating Merge Odd Word
111111 .... 11100.	A	I	158	fmsub[.]	P1			Floating Multiply-Subtract
111011 .... 11100.	A	I	158	fmsubs[.]	PPC			Floating Multiply-Subtract Single
111111 .... ///// ..... 11001.	A	I	153	fmul[.]	P1			Floating Multiply
111011 .... ///// ..... 11001.	A	I	153	fmuls[.]	PPC			Floating Multiply Single
111111 ..... ///// ..... 00100 01000.	X	I	150	fnabs[.]	P1			Floating Negative Absolute Value
111111 ..... ///// ..... 00001 01000.	X	I	150	fneg[.]	P1			Floating Negate
111111 .... 11111.	A	I	158	fnmadd[.]	P1			Floating Negative Multiply-Add
111011 .... 11111.	A	I	158	fnmadds[.]	PPC			Floating Negative Multiply-Add Single
111111 .... 11110.	A	I	158	fnmsub[.]	P1			Floating Negative Multiply-Subtract
111011 .... 11110.	A	I	158	fnmsubs[.]	PPC			Floating Negative Multiply-Subtract Single
111111 ..... ///// ..... ///// 11000.	A	I	154	fre[.]	v2.02			Floating Reciprocal Estimate
111011 ..... ///// ..... ///// 11000.	A	I	154	fres[.]	PPC			Floating Reciprocal Estimate Single
111111 ..... ///// ..... 01111 01000.	X	I	166	frim[.]	v2.02			Floating Round To Integer Minus
111111 ..... ///// ..... 01100 01000.	X	I	166	frin[.]	v2.02			Floating Round To Integer Nearest
111111 ..... ///// ..... 01110 01000.	X	I	166	frip[.]	v2.02			Floating Round To Integer Plus
111111 ..... ///// ..... 01101 01000.	X	I	166	friz[.]	v2.02			Floating Round To Integer Zero
111111 ..... ///// ..... 00000 01100.	X	I	159	frsp[.]	P1			Floating Round to Single-Precision
111111 ..... ///// ..... ///// 11010.	A	I	155	frsqrte[.]	PPC			Floating Reciprocal Square Root Estimate
111011 ..... ///// ..... ///// 11010.	A	I	155	frsqrtes[.]	v2.02			Floating Reciprocal Square Root Estimate Single
111111 .... 10111.	A	I	168	fsel[.]	PPC			Floating Select
111111 ..... ///// ..... ///// 10110.	A	I	154	fsqrt[.]	P2			Floating Square Root
111011 ..... ///// ..... ///// 10110.	A	I	154	fsqrts[.]	PPC			Floating Square Root Single
111111 .... ///// 10100.	A	I	152	fsub[.]	P1			Floating Subtract
111011 .... ///// 10100.	A	I	152	fsubs[.]	PPC			Floating Subtract Single
111111 .../// 00100 00000/	X	I	156	ftdiv	v2.06			Floating Test for software Divide
111111 .../// ///// 00101 00000/	X	I	156	ftsqrt	v2.06			Floating Test for software Square Root
010011 ///// ///// ///// 01000 10010/	XL	III	956	hrfid	v2.02	HV		Return From Interrupt Doubleword Hypervisor
011111 ///// 11110 10110/	X	II	840	icbi	PPC			Instruction Cache Block Invalidate
011111 /.... .... 00000 10110/	X	II	840	icbt	v2.07			Instruction Cache Block Touch
011111 .... 01111/	A	I	91	isel	v2.03			Integer Select
010011 ///// ///// ///// 00100 10110/	XL	II	863	isync	P1			Instruction Synchronize
011111 .... 00001 10100.	X	II	864	lbarx	v2.06			Load Byte And Reserve Indexed
100010	D	I	48	lbz	P1			Load Byte & Zero
011111 .... 11010 10101/	X	III	966	lbzcix	v2.05	HV		Load Byte & Zero Caching Inhibited Indexed
100011	D	I	48	lbzu	P1			Load Byte & Zero with Update
011111 .... 00011 10111/	X	I	48	lbzux	P1			Load Byte & Zero with Update Indexed
011111 .... 00010 10111/	X	I	48	lbzx	P1			Load Byte & Zero Indexed
111010 00	DS	I	53	ld	PPC			Load Doubleword
011111 .... 00010 10100/	X	II	869	ldarx	PPC			Load Doubleword And Reserve Indexed
011111 .... 10011 00110/	X	II	860	ldat	v3.0			Load Doubleword ATomic
011111 .... 10000 10100/	X	I	61	ldbrx	v2.06			Load Doubleword Byte-Reverse Indexed
011111 .... 11011 10101/	X	III	966	ldcix	v2.05	HV		Load Doubleword Caching Inhibited Indexed

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 4 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111010 01	DS	I	53	ldu	PPC			Load Doubleword with Update
011111 .... 00001 10101/	X	I	53	ldux	PPC			Load Doubleword with Update Indexed
011111 .... 00000 10101/	X	I	53	ldx	PPC			Load Doubleword Indexed
110010	D	I	142	lfd	P1			Load Floating Double
111001 00	DS	I	149	lfdp	v2.05			Load Floating Double Pair
011111 .... 11000 10111/	X	I	149	lfdpx	v2.05			Load Floating Double Pair Indexed
110011	D	I	142	lfdu	P1			Load Floating Double with Update
011111 .... 10011 10111/	X	I	143	lfdux	P1			Load Floating Double with Update Indexed
011111 .... 10010 10111/	X	I	142	lfdx	P1			Load Floating Double Indexed
011111 .... 11010 10111/	X	I	143	lfiwax	v2.05			Load Floating as Integer Word Algebraic Indexed
011111 .... 11011 10111/	X	I	143	lfiwzx	v2.06			Load Floating as Integer Word & Zero Indexed
110000	D	I	140	lfs	P1			Load Floating Single
110001	D	I	141	lfsu	P1			Load Floating Single with Update
011111 .... 10001 10111/	X	I	142	lfsux	P1			Load Floating Single with Update Indexed
011111 .... 10000 10111/	X	I	141	lfsx	P1			Load Floating Single Indexed
101010	D	I	50	lha	P1			Load Halfword Algebraic
011111 .... 00011 10100.	X	II	865	lharx	v2.06			Load Halfword And Reserve Indexed Xform
101011	D	I	50	lhau	P1			Load Halfword Algebraic with Update
011111 .... 01011 10111/	X	I	50	lhaux	P1			Load Halfword Algebraic with Update Indexed
011111 .... 01010 10111/	X	I	50	lhax	P1			Load Halfword Algebraic Indexed
011111 .... 11000 10110/	X	I	60	lhbrx	P1			Load Halfword Byte-Reverse Indexed
101000	D	I	49	lhz	P1			Load Halfword & Zero
011111 .... 11001 10101/	X	III	966	lhzcix	v2.05	HV		Load Halfword & Zero Caching Inhibited Indexed
101001	D	I	49	lhzu	P1			Load Halfword & Zero with Update
011111 .... 01001 10111/	X	I	49	lhzux	P1			Load Halfword & Zero with Update Indexed
011111 .... 01000 10111/	X	I	49	lhzx	P1			Load Halfword & Zero Indexed
101110	D	I	62	lmw	P1			Load Multiple Word
111000	DQ	I	58	lq	v2.03			Load Quadword
011111 .... 01000 10100.	X	I	871	lqarx	v2.07			Load Quadword And Reserve Indexed
011111 .... 10010 10101/	X	I	64	lswi	P1			Load String Word Immediate
011111 .... 10000 10101/	X	I	64	lswx	P1			Load String Word Indexed
011111 .... 00000 00111/	X	I	242	lvebx	v2.03			Load Vector Element Byte Indexed
011111 .... 00001 00111/	X	I	242	lvehx	v2.03			Load Vector Element Halfword Indexed
011111 .... 00010 00111/	X	I	243	lvewx	v2.03			Load Vector Element Word Indexed
011111 .... 00000 00110/	X	I	247	lvsl	v2.03			Load Vector for Shift Left
011111 .... 00001 00110/	X	I	247	lvsr	v2.03			Load Vector for Shift Right
011111 .... 00011 00111/	X	I	243	lvx	v2.03			Load Vector Indexed
011111 .... 01011 00111/	X	I	243	lvxl	v2.03			Load Vector Indexed Last
111010 10	DS	I	52	lwa	PPC			Load Word Algebraic
011111 .... 00000 10100/	X	II	865	lwarx	PPC			Load Word & Reserve Indexed
011111 .... 10010 00110/	X	II	860	lwat	v3.0			Load Word ATomic
011111 .... 01011 10101/	X	I	52	lwaux	PPC			Load Word Algebraic with Update Indexed
011111 .... 01010 10101/	X	I	52	lwax	PPC			Load Word Algebraic Indexed
011111 .... 10000 10110/	X	I	60	lwbrx	P1			Load Word Byte-Reverse Indexed
100000	D	I	51	lwz	P1			Load Word & Zero
011111 .... 11000 10101/	X	III	966	lwzcix	v2.05	HV		Load Word & Zero Caching Inhibited Indexed
100001	D	I	51	lwzu	P1			Load Word & Zero with Update
011111 .... 00001 10111/	X	I	51	lwzux	P1			Load Word & Zero with Update Indexed
011111 .... 00000 10111/	X	I	51	lwzx	P1			Load Word & Zero Indexed
111001 10	DS	I	480	lxsd	v3.0			Load VSX Scalar Doubleword
011111 .... 10010 01100.	X	I	480	lxsdx	v2.06			Load VSX Scalar Doubleword Indexed
011111 .... 11000 01101.	X	I	482	lxsibzx	v3.0			Load VSX Scalar as Integer Byte & Zero Indexed
011111 .... 11001 01101.	X	I	482	lxsihzx	v3.0			Load VSX Scalar as Integer Halfword & Zero Indexed
011111 .... 00010 01100.	X	I	483	lxsiwax	v2.07			Load VSX Scalar as Integer Word Algebraic Indexed
011111 .... 00000 01100.	X	I	484	lxsiwzx	v2.07			Load VSX Scalar as Integer Word & Zero Indexed
111001 11	DS	I	485	lxssp	v3.0			Load VSX Scalar Single

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 5 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
011111 .... 10000 01100.	X	I	485	lxsspx	v2.07			Load VSX Scalar Single-Precision Indexed
111101 001	DQ	I	492	lxv	v3.0			Load VSX Vector
011111 .... 11011 01100.	X	I	487	lxvb16x	v3.0			Load VSX Vector Byte*16 Indexed
011111 .... 11010 01100.	X	I	488	lxvd2x	v2.06			Load VSX Vector Doubleword*2 Indexed
011111 .... 01010 01100.	X	I	494	lxvdsx	v2.06			Load VSX Vector Doubleword & Splat Indexed
011111 .... 11001 01100.	X	I	495	lxvh8x	v3.0			Load VSX Vector Halfword*8 Indexed
011111 .... 01000 01101.	X	I	489	lxvl	v3.0			Load VSX Vector with Length
011111 .... 01001 01101.	X	I	491	lxvll	v3.0			Load VSX Vector Left-justified with Length
011111 .... 11000 01100.	X	I	496	lxvw4x	v2.06			Load VSX Vector Word*4 Indexed
011111 .... 01011 01100.	X	I	497	lxvwsx	v3.0			Load VSX Vector Word & Splat Indexed
011111 .... 01000 01100.	X	I	492	lxvx	v3.0			Load VSX Vector Indexed
000100 110000	VA	I	80	maddhd	v3.0			Multiply-Add High Doubleword
000100 110001	VA	I	80	maddhdu	v3.0			Multiply-Add High Doubleword Unsigned
000100 110011	VA	I	80	maddld	v3.0			Multiply-Add Low Doubleword
010011 ...// ...// ///// 00000 00000/	XL	I	41	mcrf	P1			Move CR Field
111111 ...// ...// ///// 00010 00000/	X	I	171	mcrfs	P1			Move To CR from FPSCR
011111 ...// ///// ///// 10010 00000/	X	I	120	mcrxrx	v3.0			Move XER to CR Extended
011111 .... 01001 01110/	X	I	909	mfbhrbe	v2.07			Move From BHRB
011111 ..... 0/// ///// 00000 10011/	XFX	I	122	mfcr	P1			Move From CR
111111 ..... 00000 ///// 10010 00111.	X	I	170	mffs[.]	P1			Move From FPSCR
111111 ..... 10100 ..... 10010 00111/	X	I	170	mffscdrn	v3.0B			Move From FPSCR Control & set DRN
111111 ..... 10101 //... 10010 00111/	X	I	170	mffscdrni	v3.0B			Move From FPSCR Control & set DRN Immediate
111111 ..... 00001 ///// 10010 00111/	X	I	170	mffsce	v3.0B			Move From FPSCR & Clear Enables
111111 ..... 10110 ..... 10010 00111/	X	I	170	mffscrn	v3.0B			Move From FPSCR Control & set RN
111111 ..... 10111 ///.. 10010 00111/	X	I	170	mffscrni	v3.0B			Move From FPSCR Control & set RN Immediate
111111 ..... 11000 ///// 10010 00111/	X	I	170	mffsl	v3.0B			Move From FPSCR Lightweight
011111 ..... ///// ///// 00010 10011/	X	III	979	mfmsr	P1	P		Move From MSR
011111 ..... 1.... ..../ 00000 10011/	XFX	I	122	mfocrf	v2.01			Move From One CR Field
011111 .... 01010 10011/	X	X	119 975	mfspr	P1	O		Move From SPR
011111 .... 01011 10011/	X	II	898	mftb	PPC			Move From Time Base
000100 ..... ///// ///// 11000 000100	VX	I	362	mfvscr	v2.03			Move From VSCR
011111 .... ///// 00001 10011.	XX1	I	112	mfvsrd	v2.07			Move From VSR Doubleword
011111 .... ///// 01001 10011.	XX1	I	112	mfvsrld	v3.0			Move From VSR Lower Doubleword
011111 .... ///// 00011 10011.	XX1	I	113	mfvsrwz	v2.07			Move From VSR Word & Zero
011111 .... 11000 01001/	X	I	83	modsd	v3.0			Modulo Signed Doubleword
011111 .... 11000 01011/	X	I	77	modsw	v3.0			Modulo Signed Word
011111 .... 01000 01001/	X	I	83	modud	v3.0			Modulo Unsigned Doubleword
011111 .... 01000 01011/	X	I	77	moduw	v3.0			Modulo Unsigned Word
011111 ///// ///// 00111 01110/	X	III	1130	msgclr	v2.07	HV		Message Clear
011111 ///// ///// 00101 01110/	X	III	1132	msgclrp	v2.07	P		Message Clear Privileged
011111 ///// ///// 00110 01110/	X	III	1129	msgsnd	v2.07	HV		Message Send
011111 ///// ///// 00100 01110/	X	III	1131	msgsndp	v2.07	P		Message Send Privileged
011111 ///// ///// ///// 11011 10110/	X	III	1132	msgsync	v3.0	HV		Message Synchronize
011111 ..... 0.... ..../ 00100 10000/	XFX	I	121	mtcrf	P1			Move To CR Fields
111111 ..... ///// ///// 00010 00110.	X	I	173	mtfsb0[.]	P1			Move To FPSCR Bit 0
111111 ..... ///// ///// 00001 00110.	X	I	173	mtfsb1[.]	P1			Move To FPSCR Bit 1
111111 .... 10110 00111.	XFL	I	172	mtfsf[.]	P1			Move To FPSCR Fields
111111 ...// ////. ..../ 00100 00110.	X	I	172	mtfsfi[.]	P1			Move To FPSCR Field Immediate
011111 ..... ////. ///// 00100 10010/	X	III	977	mtmsr	P1	P		Move To MSR
011111 ..... ////. ///// 00101 10010/	X	III	978	mtmsrd	PPC	P		Move To MSR Doubleword
011111 ..... 1.... ..../ 00100 10000/	XFX	I	121	mtocrf	v2.01			Move To One CR Field
011111 .... 01110 10011/	X	X	117 974	mtspr	P1	O		Move To SPR
000100 ///// ///// 11001 000100	VX	I	362	mtvscr	v2.03			Move To VSCR
011111 .... ///// 00101 10011.	XX1	I	114	mtvsrd	v2.07			Move To VSR Doubleword
011111 .... 01101 10011.	XX1	I	115	mtvsrdd	v3.0			Move To VSR Double Doubleword

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 6 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
011111 .... ///// 00110 10011.	XX1	I	114	mtvsrwa	v2.07			Move To VSR Word Algebraic
011111 .... ///// 01100 10011.	XX1	I	116	mtvsrws	v3.0			Move To VSR Word & Splat
011111 .... ///// 00111 10011.	XX1	I	115	mtvsrwz	v2.07			Move To VSR Word & Zero
011111 .... /0010 01001.	XO	I	79	mulhd[.]	PPC		SR	Multiply High Doubleword
011111 .... /0000 01001.	XO	I	79	mulhdu[.]	PPC		SR	Multiply High Doubleword Unsigned
011111 .... /0010 01011.	XO	I	73	mulhw[.]	PPC		SR	Multiply High Word
011111 .... /0000 01011.	XO	I	73	mulhwu[.]	PPC		SR	Multiply High Word Unsigned
011111 .... .0111 01001.	XO	I	79	mulld[o][.]	PPC		SR	Multiply Low Doubleword
000111	D	I	73	mulli	P1			Multiply Low Immediate
011111 .... .0111 01011.	XO	I	73	mullw[o][.]	P1		SR	Multiply Low Word
011111 .... 01110 11100.	X	I	94	nand[.]	P1		SR	NAND
011111 .... ///// .0011 01000.	XO	I	72	neg[o][.]	P1		SR	Negate
011111 .... 00011 11100.	X	I	95	nor[.]	P1		SR	NOR
011111 .... 01101 11100.	X	I	94	or[.]	P1		SR	OR
011111 .... 01100 11100.	X	I	95	orc[.]	P1		SR	OR with Complement
011000	D	I	92	ori	P1			OR Immediate
011001	D	I	93	oris	P1			OR Immediate Shifted
011111 ////. .... 11100 00110.	X	II	855	paste[.]	v3.0			Paste
011111 .... ///// 00011 11010/	X	I	97	popcntb	v2.02			Population Count Byte
011111 .... ///// 01111 11010/	X	I	99	popcntd	v2.06			Population Count Doubleword
011111 .... ///// 01011 11010/	X	I	97	popcntw	v2.06			Population Count Words
011111 .... ///// 00101 11010/	X	I	98	prtyd	v2.05			Parity Doubleword
011111 .... ///// 00100 11010/	X	I	98	prtyw	v2.05			Parity Word
010011 ///// ///// ////. 00100 10010/	XL	I	905	rfebb	v2.07			Return from Event Based Branch
010011 ///// ///// ///// 00000 10010/	XL	III	955	rfid	PPC	P		Return from Interrupt Doubleword
010011 ///// ///// ///// 00010 10010/	XL	III	953	rfscv	v3.0	P		Return From System Call Vectored
011110 .... .1000.	MDS	I	104	rldcl[.]	PPC		SR	Rotate Left Doubleword then Clear Left
011110 .... .1001.	MDS	I	104	rldcr[.]	PPC		SR	Rotate Left Doubleword then Clear Right
011110 .... .010..	MD	I	105	rldic[.]	PPC		SR	Rotate Left Doubleword Immediate then Clear
011110 .... .000..	MD	I	105	rldicl[.]	PPC		SR	Rotate Left Doubleword Immediate then Clear Left
011110 .... .001..	MD	I	106	rldicr[.]	PPC		SR	Rotate Left Doubleword Immediate then Clear Right
011110 .... .011..	MD	I	106	rldimi[.]	PPC		SR	Rotate Left Doubleword Immediate then Mask Insert
010100	M	I	103	rlwimi[.]	P1		SR	Rotate Left Word Immediate then Mask Insert
010101	M	I	102	rlwinm[.]	P1		SR	Rotate Left Word Immediate then AND with Mask
010111	M	I	103	rlwnm[.]	P1		SR	Rotate Left Word then AND with Mask
010001 ///// ///// ////. ..... .///1/	SC	I	42	sc	PPC			System Call
010001 ///// ///// ////. ..... .///01	SC	I	42	scv	v3.0			System Call Vectored
011111 ....// ///// 00100 000000	VX	I	122	setb	v3.0			Set Boolean
011111 ..... ///// 11110 100111	X	III	1031	slbfee.	v2.05	P	SR	SLB Find Entry ESID & record
011111 //... ///// ///// 01111 10010/	X	III	1026	slbia	PPC	P		SLB Invalidate All
011111 ..... ///// 11010 10010/	X	III	1028	slbiag	v3.0B	P		SLB Invalidate All Global
011111 ///// ///// 01101 10010/	X	III	1024	slbie	PPC	P		SLB Invalidate Entry
011111 ..... ///// 01110 10010/	X	III	1025	slbieg	v3.0	P		SLB Invalidate Entry Global
011111 ..... ///// 11100 10011/	X	III	1031	slbmfee	v2.00	P		SLB Move From Entry ESID
011111 ..... ///// 11010 10011/	X	III	1030	slbmfev	v2.00	P		SLB Move From Entry VSID
011111 ..... ///// 01100 10010/	X	III	1029	slbmte	v2.00	P		SLB Move To Entry
011111 ///// ///// ///// 01010 10010/	X	III	1032	slbsync	v3.0	P		SLB Synchronize
011111 .... 00000 11011.	X	I	109	sld[.]	PPC		SR	Shift Left Doubleword
011111 .... 00000 11000.	X	I	107	slw[.]	P1		SR	Shift Left Word
011111 .... 11000 11010.	X	I	110	srad[.]	PPC		SR	Shift Right Algebraic Doubleword
011111 .... 11001 1101..	XS	I	110	sradi[.]	PPC		SR	Shift Right Algebraic Doubleword Immediate
011111 .... 11000 11000.	X	I	108	sraw[.]	P1		SR	Shift Right Algebraic Word
011111 .... 11001 11000.	X	I	108	srawi[.]	P1		SR	Shift Right Algebraic Word Immediate
011111 .... 10000 11011.	X	I	109	srd[.]	PPC		SR	Shift Right Doubleword
011111 .... 10000 11000.	X	I	107	srw[.]	P1		SR	Shift Right Word
100110	D	I	54	stb	P1			Store Byte

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 7 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
011111 .... 11110 10101/	X	III	967	stbcix	v2.05	HV		Store Byte Caching Inhibited Indexed
011111 .... 10101 101101	X	II	866	stbcx.	v2.06			Store Byte Conditional Indexed & record
100111	D	I	54	stbu	P1			Store Byte with Update
011111 .... 00111 10111/	X	I	54	stbux	P1			Store Byte with Update Indexed
011111 .... 00110 10111/	X	I	54	stbx	P1			Store Byte Indexed
111110 00	DS	I	57	std	PPC			Store Doubleword
011111 .... 10111 00110/	X	II	862	stdat	v3.0			Store Doubleword ATomic
011111 .... 10100 10100/	X	I	61	stdbrx	v2.06			Store Doubleword Byte-Reverse Indexed
011111 .... 11111 10101/	X	III	967	stdcix	v2.05	HV		Store Doubleword Caching Inhibited Indexed
011111 .... 00110 101101	X	II	869	stdcx.	PPC			Store Doubleword Conditional Indexed & record
111110 01	DS	I	57	stdu	PPC			Store Doubleword with Update
011111 .... 00101 10101/	X	I	57	stdux	PPC			Store Doubleword with Update Indexed
011111 .... 00100 10101/	X	I	57	stdx	PPC			Store Doubleword Indexed
110110	D	I	146	stfd	P1			Store Floating Double
111101 00	DS	I	149	stfdp	v2.05			Store Floating Double Pair
011111 .... 11100 10111/	X	I	149	stfdpx	v2.05			Store Floating Double Pair Indexed
110111	D	I	146	stfdu	P1			Store Floating Double with Update
011111 .... 10111 10111/	X	I	146	stfdux	P1			Store Floating Double with Update Indexed
011111 .... 10110 10111/	X	I	146	stfdx	P1			Store Floating Double Indexed
011111 .... 11110 10111/	X	I	147	stfiwx	PPC			Store Floating as Integer Word Indexed
110100	D	I	145	stfs	P1			Store Floating Single
110101	D	I	145	stfsu	P1			Store Floating Single with Update
011111 .... 10101 10111/	X	I	145	stfsux	P1			Store Floating Single with Update Indexed
011111 .... 10100 10111/	X	I	145	stfsx	P1			Store Floating Single Indexed
101100	D	I	55	sth	P1			Store Halfword
011111 .... 11100 10110/	X	I	60	sthbrx	P1			Store Halfword Byte-Reverse Indexed
011111 .... 11101 10101/	X	III	967	sthcix	v2.05	HV		Store Halfword Caching Inhibited Indexed
011111 .... 10110 101101	X	II	867	sthcx.	v2.06			Store Halfword Conditional Indexed & record
101101	D	I	55	sthu	P1			Store Halfword with Update
011111 .... 01101 10111/	X	I	55	sthux	P1			Store Halfword with Update Indexed
011111 .... 01100 10111/	X	I	55	sthx	P1			Store Halfword Indexed
101111	D	I	62	stmw	P1			Store Multiple Word
010011 ///// ///// ///// 01011 10010/	XL	III	958	stop	v3.0	P		Stop
111110 10	DS	I	59	stq	v2.03			Store Quadword
011111 .... 00101 101101	X	I	872	stqcx.	v2.07			Store Quadword Conditional Indexed & record
011111 .... 10110 10101/	X	I	65	stswi	P1			Store String Word Immediate
011111 .... 10100 10101/	X	I	65	stswx	P1			Store String Word Indexed
011111 .... 00100 00111/	X	I	245	stvebx	v2.03			Store Vector Element Byte Indexed
011111 .... 00101 00111/	X	I	245	stvehx	v2.03			Store Vector Element Halfword Indexed
011111 .... 00110 00111/	X	I	246	stvewx	v2.03			Store Vector Element Word Indexed
011111 .... 00111 00111/	X	I	246	stvx	v2.03			Store Vector Indexed
011111 .... 01111 00111/	X	I	246	stvxl	v2.03			Store Vector Indexed Last
100100	D	I	56	stw	P1			Store Word
011111 .... 10110 00110/	X	II	862	stwat	v3.0			Store Word ATomic
011111 .... 10100 10110/	X	I	60	stwbrx	P1			Store Word Byte-Reverse Indexed
011111 .... 11100 10101/	X	III	967	stwcix	v2.05	HV		Store Word Caching Inhibited Indexed
011111 .... 00100 101101	X	II	868	stwcx.	PPC			Store Word Conditional Indexed & record
100101	D	I	56	stwu	P1			Store Word with Update
011111 .... 00101 10111/	X	I	56	stwux	P1			Store Word with Update Indexed
011111 .... 00100 10111/	X	I	56	stwx	P1			Store Word Indexed
111101 10	DS	I	498	stxsd	v3.0			Store VSX Scalar Doubleword
011111 .... 10110 01100.	X	I	498	stxsdx	v2.06			Store VSX Scalar Doubleword Indexed
011111 .... 11100 01101.	X	I	499	stxsibx	v3.0			Store VSX Scalar as Integer Byte Indexed
011111 .... 11101 01101.	X	I	499	stxsihx	v3.0			Store VSX Scalar as Integer Halfword Indexed
011111 .... 00100 01100.	X	I	500	stxsiwx	v2.07			Store VSX Scalar as Integer Word Indexed
111101 11	DS	I	501	stxssp	v3.0			Store VSX Scalar Single-Precision

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 8 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
011111 .... 10100 01100.	X	I	502	stxsspx	v2.07			Store VSX Scalar Single-Precision Indexed
111101 101	DQ	I	507	stxv	v3.0			Store VSX Vector
011111 .... 11111 01100.	X	I	503	stxvb16x	v3.0			Store VSX Vector Byte*16 Indexed
011111 .... 11110 01100.	X	I	504	stxvd2x	v2.06			Store VSX Vector Doubleword*2 Indexed
011111 .... 11101 01100.	X	I	505	stxvh8x	v3.0			Store VSX Vector Halfword*8 Indexed
011111 .... 01100 01101.	X	I	507	stxvl	v3.0			Store VSX Vector with Length
011111 .... 01101 01101.	X	I	509	stxvll	v3.0			Store VSX Vector Left-justified with Length
011111 .... 11100 01100.	X	I	506	stxvw4x	v2.06			Store VSX Vector Word*4 Indexed
011111 .... 01100 01100.	X	I	510	stxvx	v3.0			Store VSX Vector Indexed
011111 .... .0001 01000.	XO	I	69	subf[o][.]	PPC		SR	Subtract From
011111 .... .0000 01000.	XO	I	70	subfc[o][.]	P1		SR	Subtract From Carrying
011111 .... .0100 01000.	XO	I	71	subfe[o][.]	P1		SR	Subtract From Extended
001000	D	I	70	subfic	P1		SR	Subtract From Immediate Carrying
011111 .... ///// .0111 01000.	XO	I	71	subfme[o][.]	P1		SR	Subtract From Minus One Extended
011111 .... ///// .0110 01000.	XO	I	72	subfze[o][.]	P1		SR	Subtract From Zero Extended
011111 ///.. ///// ///// 10010 10110/	X	II	873	sync	P1			Synchronize
011111 ///// ..... ///// 11100 011101	X	II	892	tabort.	v2.07			Transaction Abort & record
011111 .... 11001 011101	X	II	894	tabortdc.	v2.07			Transaction Abort Doubleword Conditional & record
011111 .... 11011 011101	X	II	894	tabortdci.	v2.07			Transaction Abort Doubleword Conditional Immediate & record
011111 .... 11000 011101	X	II	893	tabortwc.	v2.07			Transaction Abort Word Conditional & record
011111 .... 11010 011101	X	II	893	tabortwci.	v2.07			Transaction Abort Word Conditional Immediate & record
011111 .///. ///// ///// 10100 011101	X	II	890	tbegin.	v2.07			Transaction Begin & record
011111 ...// ///// ///// 10110 01110/	X	II	895	tcheck	v2.07			Transaction Check & record
011111 .... 00010 00100/	X	I	91	td	PPC			Trap Doubleword
000010	D	I	91	tdi	PPC			Trap Doubleword Immediate
011111 .//// ///// ///// 10101 01110/	X	II	891	tend.	v2.07			Transaction End & record
011111 ..... /.... ..... 01001 10010/	X	III	1034	tlbie	P1	HV	64	TLB Invalidate Entry
011111 ..... /.... ..... 01000 10010/	X	III	1038	tlbiel	v2.03	P	64	TLB Invalidate Entry Local
011111 ///// ///// ///// 10001 10110/	X	III	1042	tlbsync	PPC	HV/P		TLB Synchronize
011111 ///// ///// ///// 11111 011101	X	II	970	trechkpt.	v2.07			Transaction Recheckpoint & record
011111 ///// ..... ///// 11101 011101	X	II	969	treclaim.	v2.07			Transaction Reclaim & record
011111 ////. ///// ///// 10111 01110/	X	II	895	tsr.	v2.07			Transaction Suspend or Resume & record
011111 .... 00000 00100/	X	I	90	tw	P1			Trap Word
000011	D	I	90	twi	P1			Trap Word Immediate
000100 .... 10000 000011	VX	I	297	vabsdub	v3.0			Vector Absolute Difference Unsigned Byte
000100 .... 10001 000011	VX	I	297	vabsduh	v3.0			Vector Absolute Difference Unsigned Halfword
000100 .... 10010 000011	VX	I	298	vabsduw	v3.0			Vector Absolute Difference Unsigned Word
000100 .... 00101 000000	VX	I	273	vaddcuq	v2.07			Vector Add & write Carry Unsigned Quadword
000100 .... 00110 000000	VX	I	269	vaddcuw	v2.03			Vector Add & Write Carry-Out Unsigned Word
000100 111101	VA	I	273	vaddecuq	v2.07			Vector Add Extended & write Carry Unsigned Quadword
000100 111100	VA	I	273	vaddeuqm	v2.07			Vector Add Extended Unsigned Quadword Modulo
000100 .... 00000 001010	VX	I	321	vaddfp	v2.03			Vector Add Floating-Point
000100 .... 01100 000000	VX	I	269	vaddsbs	v2.03			Vector Add Signed Byte Saturate
000100 .... 01101 000000	VX	I	269	vaddshs	v2.03			Vector Add Signed Halfword Saturate
000100 .... 01110 000000	VX	I	270	vaddsws	v2.03			Vector Add Signed Word Saturate
000100 .... 00000 000000	VX	I	270	vaddubm	v2.03			Vector Add Unsigned Byte Modulo
000100 .... 01000 000000	VX	I	272	vaddubs	v2.03			Vector Add Unsigned Byte Saturate
000100 .... 00011 000000	VX	I	270	vaddudm	v2.07			Vector Add Unsigned Doubleword Modulo
000100 .... 00001 000000	VX	I	271	vadduhm	v2.03			Vector Add Unsigned Halfword Modulo
000100 .... 01001 000000	VX	I	272	vadduhs	v2.03			Vector Add Unsigned Halfword Saturate
000100 .... 00100 000000	VX	I	270	vadduqm	v2.07			Vector Add Unsigned Quadword Modulo
000100 .... 00010 000000	VX	I	271	vadduwm	v2.03			Vector Add Unsigned Word Modulo
000100 .... 01010 000000	VX	I	272	vadduws	v2.03			Vector Add Unsigned Word Saturate
000100 .... 10000 000100	VX	I	312	vand	v2.03			Vector Logical AND
000100 .... 10001 000100	VX	I	312	vandc	v2.03			Vector Logical AND with Complement
000100 .... 10100 000010	VX	I	295	vavgsb	v2.03			Vector Average Signed Byte

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 9 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
000100 .... 10101 000010	VX	I	295	vavgsh	v2.03			Vector Average Signed Halfword
000100 .... 10110 000010	VX	I	295	vavgsw	v2.03			Vector Average Signed Word
000100 .... 10000 000010	VX	I	296	vavgub	v2.03			Vector Average Unsigned Byte
000100 .... 10001 000010	VX	I	296	vavguh	v2.03			Vector Average Unsigned Halfword
000100 .... 10010 000010	VX	I	296	vavguw	v2.03			Vector Average Unsigned Word
000100 .... 10111 001100	VX	I	346	vbpermd	v3.0			Vector Bit Permute Doubleword
000100 .... 10101 001100	VX	I	346	vbpermq	v2.07			Vector Bit Permute Quadword
000100 .... 01101 001010	VX	I	325	vcfsx	v2.03			Vector Convert with round to nearest Signed Word format to FP
000100 .... 01100 001010	VX	I	325	vcfux	v2.03			Vector Convert with round to nearest Unsigned Word format to FP
000100 .... 10100 001000	VX	I	333	vcipher	v2.07			Vector AES Cipher
000100 .... 10100 001001	VX	I	333	vcipherlast	v2.07			Vector AES Cipher Last
000100 ..... ///// 11100 000010	VX	I	340	vclzb	v2.07			Vector Count Leading Zeros Byte
000100 ..... ///// 11111 000010	VX	I	340	vclzd	v2.07			Vector Count Leading Zeros Doubleword
000100 ..... ///// 11101 000010	VX	I	340	vclzh	v2.07			Vector Count Leading Zeros Halfword
000100 ..... 00000 ..... 11000 000010	VX	I	342	vclzlsbb	v3.0			Vector Count Leading Zero Least-Significant Bits Byte
000100 ..... ///// 11110 000010	VX	I	340	vclzw	v2.07			Vector Count Leading Zeros Word
000100 .... .1111 000110	VC	I	328	vcmpbfp[.]	v2.03			Vector Compare Bounds Floating-Point
000100 .... .0011 000110	VC	I	329	vcmpeqfp[.]	v2.03			Vector Compare Equal To Floating-Point
000100 .... .0000 000110	VC	I	303	vcmpequb[.]	v2.03			Vector Compare Equal To Unsigned Byte
000100 .... .0011 000111	VC	I	304	vcmpequd[.]	v2.07			Vector Compare Equal To Unsigned Doubleword
000100 .... .0001 000110	VC	I	303	vcmpequh[.]	v2.03			Vector Compare Equal To Unsigned Halfword
000100 .... .0010 000110	VC	I	304	vcmpequw[.]	v2.03			Vector Compare Equal To Unsigned Word
000100 .... .0111 000110	VC	I	329	vcmpgefp[.]	v2.03			Vector Compare Greater Than or Equal To Floating-Point
000100 .... .1011 000110	VC	I	330	vcmpgtfp[.]	v2.03			Vector Compare Greater Than Floating-Point
000100 .... .1100 000110	VC	I	305	vcmpgtsb[.]	v2.03			Vector Compare Greater Than Signed Byte
000100 .... .1111 000111	VC	I	305	vcmpgtsd[.]	v2.07			Vector Compare Greater Than Signed Doubleword
000100 .... .1101 000110	VC	I	306	vcmpgtsh[.]	v2.03			Vector Compare Greater Than Signed Halfword
000100 .... .1110 000110	VC	I	306	vcmpgtsw[.]	v2.03			Vector Compare Greater Than Signed Word
000100 .... .1000 000110	VC	I	307	vcmpgtub[.]	v2.03			Vector Compare Greater Than Unsigned Byte
000100 .... .1011 000111	VC	I	307	vcmpgtud[.]	v2.07			Vector Compare Greater Than Unsigned Doubleword
000100 .... .1001 000110	VC	I	308	vcmpgtuh[.]	v2.03			Vector Compare Greater Than Unsigned Halfword
000100 .... .1010 000110	VC	I	308	vcmpgtuw[.]	v2.03			Vector Compare Greater Than Unsigned Word
000100 .... .0000 000111	VC	I	309	vcmpneb[.]	v3.0			Vector Compare Not Equal Byte
000100 .... .0001 000111	VC	I	310	vcmpneh[.]	v3.0			Vector Compare Not Equal Halfword
000100 .... .0010 000111	VC	I	311	vcmpnew[.]	v3.0			Vector Compare Not Equal Word
000100 .... .0100 000111	VC	I	309	vcmpnezb[.]	v3.0			Vector Compare Not Equal or Zero Byte
000100 .... .0101 000111	VC	I	310	vcmpnezh[.]	v3.0			Vector Compare Not Equal or Zero Halfword
000100 .... .0110 000111	VC	I	311	vcmpnezw[.]	v3.0			Vector Compare Not Equal or Zero Word
000100 .... 01111 001010	VX	I	324	vctsxs	v2.03			Vector Convert with round to zero FP To Signed Word format Saturate
000100 .... 01110 001010	VX	I	324	vctuxs	v2.03			Vector Convert with round to zero FP To Unsigned Word format Saturate
000100 ..... 11100 ..... 11000 000010	VX	I	341	vctzb	v3.0			Vector Count Trailing Zeros Byte
000100 ..... 11111 ..... 11000 000010	VX	I	341	vctzd	v3.0			Vector Count Trailing Zeros Doubleword
000100 ..... 11101 ..... 11000 000010	VX	I	341	vctzh	v3.0			Vector Count Trailing Zeros Halfword
000100 ..... 00001 ..... 11000 000010	VX	I	342	vctzlsbb	v3.0			Vector Count Trailing Zero Least-Significant Bits Byte
000100 ..... 11110 ..... 11000 000010	VX	I	341	vctzw	v3.0			Vector Count Trailing Zeros Word
000100 .... 11010 000100	VX	I	312	veqv	v2.07			Vector Equivalence
000100 ..... ///// 00110 001010	VX	I	331	vexptefp	v2.03			Vector 2 Raised to the Exponent Estimate Floating-Point
000100 ..... /.... ..... 01011 001101	VX	I	267	vextractd	v3.0			Vector Extract Doubleword
000100 ..... /.... ..... 01000 001101	VX	I	267	vextractub	v3.0			Vector Extract Unsigned Byte
000100 ..... /.... ..... 01001 001101	VX	I	267	vextractuh	v3.0			Vector Extract Unsigned Halfword
000100 ..... /.... ..... 01010 001101	VX	I	267	vextractuw	v3.0			Vector Extract Unsigned Word
000100 ..... 11000 ..... 11000 000010	VX	I	294	vextsb2d	v3.0			Vector Extend Sign Byte to Doubleword
000100 ..... 10000 ..... 11000 000010	VX	I	294	vextsb2w	v3.0			Vector Extend Sign Byte to Word
000100 ..... 11001 ..... 11000 000010	VX	I	294	vextsh2d	v3.0			Vector Extend Sign Halfword to Doubleword

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 10 of 18)

Instruction[1] 0:5	6:10	11:15	16:20	21:25	26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
000100	.....	10001	.....	11000	000010	VX	I	294	vextsh2w	v3.0			Vector Extend Sign Halfword to Word
000100	.....	11010	.....	11000	000010	VX	I	294	vextsw2d	v3.0			Vector Extend Sign Word to Doubleword
000100	.....	.....	.....	11000	001101	VX	I	343	vextublx	v3.0			Vector Extract Unsigned Byte Left-Indexed
000100	.....	.....	.....	11100	001101	VX	I	343	vextubrx	v3.0			Vector Extract Unsigned Byte Right-Indexed
000100	.....	.....	.....	11001	001101	VX	I	343	vextuhlx	v3.0			Vector Extract Unsigned Halfword Left-Indexed
000100	.....	.....	.....	11101	001101	VX	I	343	vextuhrx	v3.0			Vector Extract Unsigned Halfword Right-Indexed
000100	.....	.....	.....	11010	001101	VX	I	344	vextuwlx	v3.0			Vector Extract Unsigned Word Left-Indexed
000100	.....	.....	.....	11110	001101	VX	I	344	vextuwrx	v3.0			Vector Extract Unsigned Word Right-Indexed
000100	.....	/////	.....	10100	001100	VX	I	339	vgbbd	v2.07			Vector Gather Bits by Byte by Doubleword
000100	.....	/....	.....	01100	001101	VX	I	268	vinsertb	v3.0			Vector Insert Byte
000100	.....	/....	.....	01111	001101	VX	I	268	vinsertd	v3.0			Vector Insert Doubleword
000100	.....	/....	.....	01101	001101	VX	I	268	vinserth	v3.0			Vector Insert Halfword
000100	.....	/....	.....	01110	001101	VX	I	268	vinsertw	v3.0			Vector Insert Word
000100	.....	.....	.....	00111	001010	VX	I	331	vlogefp	v2.03			Vector Log Base 2 Estimate Floating-Point
000100	.....	.....	.....	.....	101110	VA	I	322	vmaddfp	v2.03			Vector Multiply-Add Floating-Point
000100	.....	.....	.....	10000	001010	VX	I	323	vmaxfp	v2.03			Vector Maximum Floating-Point
000100	.....	.....	.....	00100	000010	VX	I	299	vmaxsb	v2.03			Vector Maximum Signed Byte
000100	.....	.....	.....	00111	000010	VX	I	299	vmaxsd	v2.07			Vector Maximum Signed Doubleword
000100	.....	.....	.....	00101	000010	VX	I	300	vmaxsh	v2.03			Vector Maximum Signed Halfword
000100	.....	.....	.....	00110	000010	VX	I	300	vmaxsw	v2.03			Vector Maximum Signed Word
000100	.....	.....	.....	00000	000010	VX	I	299	vmaxub	v2.03			Vector Maximum Unsigned Byte
000100	.....	.....	.....	00011	000010	VX	I	299	vmaxud	v2.07			Vector Maximum Unsigned Doubleword
000100	.....	.....	.....	00001	000010	VX	I	300	vmaxuh	v2.03			Vector Maximum Unsigned Halfword
000100	.....	.....	.....	00010	000010	VX	I	300	vmaxuw	v2.03			Vector Maximum Unsigned Word
000100	.....	.....	.....	.....	100000	VA	I	285	vmhaddshs	v2.03			Vector Multiply-High-Add Signed Halfword Saturate
000100	.....	.....	.....	.....	100001	VA	I	285	vmhraddshs	v2.03			Vector Multiply-High-Round-Add Signed Halfword Saturate
000100	.....	.....	.....	10001	001010	VX	I	323	vminfp	v2.03			Vector Minimum Floating-Point
000100	.....	.....	.....	01100	000010	VX	I	301	vminsb	v2.03			Vector Minimum Signed Byte
000100	.....	.....	.....	01111	000010	VX	I	301	vminsd	v2.07			Vector Minimum Signed Doubleword
000100	.....	.....	.....	01101	000010	VX	I	302	vminsh	v2.03			Vector Minimum Signed Halfword
000100	.....	.....	.....	01110	000010	VX	I	302	vminsw	v2.03			Vector Minimum Signed Word
000100	.....	.....	.....	01000	000010	VX	I	301	vminub	v2.03			Vector Minimum Unsigned Byte
000100	.....	.....	.....	01011	000010	VX	I	301	vminud	v2.07			Vector Minimum Unsigned Doubleword
000100	.....	.....	.....	01001	000010	VX	I	302	vminuh	v2.03			Vector Minimum Unsigned Halfword
000100	.....	.....	.....	01010	000010	VX	I	302	vminuw	v2.03			Vector Minimum Unsigned Word
000100	.....	.....	.....	.....	100010	VA	I	286	vmladduhm	v2.03			Vector Multiply-Low-Add Unsigned Halfword Modulo
000100	.....	.....	.....	11110	001100	VX	I	257	vmrgew	v2.07			Vector Merge Even Word
000100	.....	.....	.....	00000	001100	VX	I	255	vmrghb	v2.03			Vector Merge High Byte
000100	.....	.....	.....	00001	001100	VX	I	255	vmrghh	v2.03			Vector Merge High Halfword
000100	.....	.....	.....	00010	001100	VX	I	256	vmrghw	v2.03			Vector Merge High Word
000100	.....	.....	.....	00100	001100	VX	I	255	vmrglb	v2.03			Vector Merge Low Byte
000100	.....	.....	.....	00101	001100	VX	I	255	vmrglh	v2.03			Vector Merge Low Halfword
000100	.....	.....	.....	00110	001100	VX	I	256	vmrglw	v2.03			Vector Merge Low Word
000100	.....	.....	.....	11010	001100	VX	I	257	vmrgow	v2.07			Vector Merge Odd Word
000100	.....	.....	.....	.....	100101	VA	I	287	vmsummbm	v2.03			Vector Multiply-Sum Mixed Byte Modulo
000100	.....	.....	.....	.....	101000	VA	I	287	vmsumshm	v2.03			Vector Multiply-Sum Signed Halfword Modulo
000100	.....	.....	.....	.....	101001	VA	I	288	vmsumshs	v2.03			Vector Multiply-Sum Signed Halfword Saturate
000100	.....	.....	.....	.....	100100	VA	I	286	vmsumubm	v2.03			Vector Multiply-Sum Unsigned Byte Modulo
000100	.....	.....	.....	.....	100011	VA	I	289	vmsumudm	v3.0B			Vector Multiply-Sum Unsigned Doubleword Modulo
000100	.....	.....	.....	.....	100110	VA	I	288	vmsumuhm	v2.03			Vector Multiply-Sum Unsigned Halfword Modulo
000100	.....	.....	.....	.....	100111	VA	I	289	vmsumuhs	v2.03			Vector Multiply-Sum Unsigned Halfword Saturate
000100	.....	.....	/////	00000	000001	VX	I	355	vmul10cuq	v3.0			Vector Multiply-by-10 & write Carry Unsigned Quadword
000100	.....	.....	.....	00001	000001	VX	I	355	vmul10ecuq	v3.0			Vector Multiply-by-10 Extended & write Carry Unsigned Quadword
000100	.....	.....	.....	01001	000001	VX	I	355	vmul10euq	v3.0			Vector Multiply-by-10 Extended Unsigned Quadword
000100	.....	.....	/////	01000	000001	VX	I	355	vmul10uq	v3.0			Vector Multiply-by-10 Unsigned Quadword
000100	.....	.....	.....	01100	001000	VX	I	281	vmulesb	v2.03			Vector Multiply Even Signed Byte

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 11 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
000100 .... 01101 001000	VX	I	282	vmulesh	v2.03			Vector Multiply Even Signed Halfword
000100 .... 01110 001000	VX	I	283	vmulesw	v2.07			Vector Multiply Even Signed Word
000100 .... 01000 001000	VX	I	281	vmuleub	v2.03			Vector Multiply Even Unsigned Byte
000100 .... 01001 001000	VX	I	282	vmuleuh	v2.03			Vector Multiply Even Unsigned Halfword
000100 .... 01010 001000	VX	I	283	vmuleuw	v2.07			Vector Multiply Even Unsigned Word
000100 .... 00100 001000	VX	I	281	vmulosb	v2.03			Vector Multiply Odd Signed Byte
000100 .... 00101 001000	VX	I	282	vmulosh	v2.03			Vector Multiply Odd Signed Halfword
000100 .... 00110 001000	VX	I	283	vmulosw	v2.07			Vector Multiply Odd Signed Word
000100 .... 00000 001000	VX	I	281	vmuloub	v2.03			Vector Multiply Odd Unsigned Byte
000100 .... 00001 001000	VX	I	282	vmulouh	v2.03			Vector Multiply Odd Unsigned Halfword
000100 .... 00010 001000	VX	I	283	vmulouw	v2.07			Vector Multiply Odd Unsigned Word
000100 .... 00010 001001	VX	I	284	vmuluwm	v2.07			Vector Multiply Unsigned Word Modulo
000100 .... 10110 000100	VX	I	312	vnand	v2.07			Vector NAND
000100 .... 10101 001000	VX	I	334	vncipher	v2.07			Vector AES Inverse Cipher
000100 .... 10101 001001	VX	I	334	vncipherlast	v2.07			Vector AES Inverse Cipher Last
000100 ..... 00111 ..... 11000 000010	VX	I	293	vnegd	v3.0			Vector Negate Doubleword
000100 ..... 00110 ..... 11000 000010	VX	I	293	vnegw	v3.0			Vector Negate Word
000100 101111	VA	I	322	vnmsubfp	v2.03			Vector Negative Multiply-Subtract Floating-Point
000100 .... 10100 000100	VX	I	313	vnor	v2.03			Vector Logical NOR
000100 .... 10010 000100	VX	I	313	vor	v2.03			Vector Logical OR
000100 .... 10101 000100	VX	I	313	vorc	v2.07			Vector OR with Complement
000100 101011	VA	I	260	vperm	v2.03			Vector Permute
000100 111011	VA	I	260	vpermr	v3.0			Vector Permute Right-indexed
000100 101101	VA	I	338	vpermxor	v2.07			Vector Permute & Exclusive-OR
000100 .... 01100 001110	VX	I	248	vpkpx	v2.03			Vector Pack Pixel
000100 .... 10111 001110	VX	I	248	vpksdss	v2.07			Vector Pack Signed Doubleword Signed Saturate
000100 .... 10101 001110	VX	I	249	vpksdus	v2.07			Vector Pack Signed Doubleword Unsigned Saturate
000100 .... 00110 001110	VX	I	249	vpkshss	v2.03			Vector Pack Signed Halfword Signed Saturate
000100 .... 00100 001110	VX	I	250	vpkshus	v2.03			Vector Pack Signed Halfword Unsigned Saturate
000100 .... 00111 001110	VX	I	250	vpkswss	v2.03			Vector Pack Signed Word Signed Saturate
000100 .... 00101 001110	VX	I	251	vpkswus	v2.03			Vector Pack Signed Word Unsigned Saturate
000100 .... 10001 001110	VX	I	251	vpkudum	v2.07			Vector Pack Unsigned Doubleword Unsigned Modulo
000100 .... 10011 001110	VX	I	251	vpkudus	v2.07			Vector Pack Unsigned Doubleword Unsigned Saturate
000100 .... 00000 001110	VX	I	251	vpkuhum	v2.03			Vector Pack Unsigned Halfword Unsigned Modulo
000100 .... 00010 001110	VX	I	252	vpkuhus	v2.03			Vector Pack Unsigned Halfword Unsigned Saturate
000100 .... 00001 001110	VX	I	252	vpkuwum	v2.03			Vector Pack Unsigned Word Unsigned Modulo
000100 .... 00011 001110	VX	I	252	vpkuwus	v2.03			Vector Pack Unsigned Word Unsigned Saturate
000100 .... 10000 001000	VX	I	336	vpmsumb	v2.07			Vector Polynomial Multiply-Sum Byte
000100 .... 10011 001000	VX	I	336	vpmsumd	v2.07			Vector Polynomial Multiply-Sum Doubleword
000100 .... 10001 001000	VX	I	337	vpmsumh	v2.07			Vector Polynomial Multiply-Sum Halfword
000100 .... 10010 001000	VX	I	337	vpmsumw	v2.07			Vector Polynomial Multiply-Sum Word
000100 ..... ///// 11100 000011	VX	I	345	vpopcntb	v2.07			Vector Population Count Byte
000100 ..... ///// 11111 000011	VX	I	345	vpopcntd	v2.07			Vector Population Count Doubleword
000100 ..... ///// 11101 000011	VX	I	345	vpopcnth	v2.07			Vector Population Count Halfword
000100 ..... ///// 11110 000011	VX	I	345	vpopcntw	v2.07			Vector Population Count Word
000100 ..... 01001 ..... 11000 000010	VX	I	314	vprtybd	v3.0			Vector Parity Byte Doubleword
000100 ..... 01010 ..... 11000 000010	VX	I	314	vprtybq	v3.0			Vector Parity Byte Quadword
000100 ..... 01000 ..... 11000 000010	VX	I	314	vprtybw	v3.0			Vector Parity Byte Word
000100 ..... ///// 00100 001010	VX	I	332	vrefp	v2.03			Vector Reciprocal Estimate Floating-Point
000100 ..... ///// 01011 001010	VX	I	326	vrfim	v2.03			Vector Round to Floating-Point Integral toward -Infinity
000100 ..... ///// 01000 001010	VX	I	326	vrfin	v2.03			Vector Round to Floating-Point Integral Nearest
000100 ..... ///// 01010 001010	VX	I	326	vrfip	v2.03			Vector Round to Floating-Point Integral toward +Infinity
000100 ..... ///// 01001 001010	VX	I	327	vrfiz	v2.03			Vector Round to Floating-Point Integral toward Zero
000100 .... 00000 000100	VX	I	315	vrlb	v2.03			Vector Rotate Left Byte
000100 .... 00011 000100	VX	I	315	vrld	v2.07			Vector Rotate Left Doubleword
000100 .... 00011 000101	VX	I	320	vrldmi	v3.0			Vector Rotate Left Doubleword then Mask Insert

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 12 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
000100 .... 00111 000101	VX	I	320	vrldnm	v3.0			Vector Rotate Left Doubleword then AND with Mask
000100 .... 00001 000100	VX	I	315	vrlh	v2.03			Vector Rotate Left Halfword
000100 .... 00010 000100	VX	I	315	vrlw	v2.03			Vector Rotate Left Word
000100 .... 00010 000101	VX	I	319	vrlwmi	v3.0			Vector Rotate Left Word then Mask Insert
000100 .... 00110 000101	VX	I	319	vrlwnm	v3.0			Vector Rotate Left Word then AND with Mask
000100 ..... ///// 00101 001010	VX	I	332	vrsqrtefp	v2.03			Vector Reciprocal Square Root Estimate Floating-Point
000100 .... ///// 10111 001000	VX	I	334	vsbox	v2.07			Vector AES S-Box
000100 101010	VA	I	261	vsel	v2.03			Vector Select
000100 .... 11011 000010	VX	I	335	vshasigmad	v2.07			Vector SHA-512 Sigma Doubleword
000100 .... 11010 000010	VX	I	335	vshasigmaw	v2.07			Vector SHA-256 Sigma Word
000100 .... 00111 000100	VX	I	264	vsl	v2.03			Vector Shift Left
000100 .... 00100 000100	VX	I	316	vslb	v2.03			Vector Shift Left Byte
000100 .... 10111 000100	VX	I	316	vsld	v2.07			Vector Shift Left Doubleword
000100 .... /.... 101100	VA	I	263	vsldoi	v2.03			Vector Shift Left Double by Octet Immediate
000100 .... 00101 000100	VX	I	316	vslh	v2.03			Vector Shift Left Halfword
000100 .... 10000 001100	VX	I	264	vslo	v2.03			Vector Shift Left by Octet
000100 .... 11101 000100	VX	I	265	vslv	v3.0			Vector Shift Left Variable
000100 .... 00110 000100	VX	I	316	vslw	v2.03			Vector Shift Left Word
000100 ..... /.... ..... 01000 001100	VX	I	258	vspltb	v2.03			Vector Splat Byte
000100 ..... //... ..... 01001 001100	VX	I	258	vsplth	v2.03			Vector Splat Halfword
000100 .... ///// 01100 001100	VX	I	259	vspltisb	v2.03			Vector Splat Immediate Signed Byte
000100 .... ///// 01101 001100	VX	I	259	vspltish	v2.03			Vector Splat Immediate Signed Halfword
000100 .... ///// 01110 001100	VX	I	259	vspltisw	v2.03			Vector Splat Immediate Signed Word
000100 ..... ///.. ..... 01010 001100	VX	I	258	vspltw	v2.03			Vector Splat Word
000100 .... 01011 000100	VX	I	264	vsr	v2.03			Vector Shift Right
000100 .... 01100 000100	VX	I	318	vsrab	v2.03			Vector Shift Right Algebraic Byte
000100 .... 01111 000100	VX	I	318	vsrad	v2.07			Vector Shift Right Algebraic Doubleword
000100 .... 01101 000100	VX	I	318	vsrah	v2.03			Vector Shift Right Algebraic Halfword
000100 .... 01110 000100	VX	I	318	vsraw	v2.03			Vector Shift Right Algebraic Word
000100 .... 01000 000100	VX	I	317	vsrb	v2.03			Vector Shift Right Byte
000100 .... 11011 000100	VX	I	317	vsrd	v2.07			Vector Shift Right Doubleword
000100 .... 01001 000100	VX	I	317	vsrh	v2.03			Vector Shift Right Halfword
000100 .... 10001 001100	VX	I	264	vsro	v2.03			Vector Shift Right by Octet
000100 .... 11100 000100	VX	I	265	vsrv	v3.0			Vector Shift Right Variable
000100 .... 01010 000100	VX	I	317	vsrw	v2.03			Vector Shift Right Word
000100 .... 10101 000000	VX	I	279	vsubcuq	v2.07			Vector Subtract & write Carry Unsigned Quadword
000100 .... 10110 000000	VX	I	275	vsubcuw	v2.03			Vector Subtract & Write Carry-Out Unsigned Word
000100 111111	VA	I	279	vsubecuq	v2.07			Vector Subtract Extended & write Carry Unsigned Quadword
000100 111110	VA	I	279	vsubeuqm	v2.07			Vector Subtract Extended Unsigned Quadword Modulo
000100 .... 00001 001010	VX	I	321	vsubfp	v2.03			Vector Subtract Floating-Point
000100 .... 11100 000000	VX	I	275	vsubsbs	v2.03			Vector Subtract Signed Byte Saturate
000100 .... 11101 000000	VX	I	275	vsubshs	v2.03			Vector Subtract Signed Halfword Saturate
000100 .... 11110 000000	VX	I	276	vsubsws	v2.03			Vector Subtract Signed Word Saturate
000100 .... 10000 000000	VX	I	277	vsububm	v2.03			Vector Subtract Unsigned Byte Modulo
000100 .... 11000 000000	VX	I	278	vsububs	v2.03			Vector Subtract Unsigned Byte Saturate
000100 .... 10011 000000	VX	I	277	vsubudm	v2.07			Vector Subtract Unsigned Doubleword Modulo
000100 .... 10001 000000	VX	I	277	vsubuhm	v2.03			Vector Subtract Unsigned Halfword Modulo
000100 .... 11001 000000	VX	I	278	vsubuhs	v2.03			Vector Subtract Unsigned Halfword Saturate
000100 .... 10100 000000	VX	I	279	vsubuqm	v2.07			Vector Subtract Unsigned Quadword Modulo
000100 .... 10010 000000	VX	I	277	vsubuwm	v2.03			Vector Subtract Unsigned Word Modulo
000100 .... 11010 000000	VX	I	278	vsubuws	v2.03			Vector Subtract Unsigned Word Saturate
000100 .... 11010 001000	VX	I	290	vsum2sws	v2.03			Vector Sum across Half Signed Word Saturate
000100 .... 11100 001000	VX	I	291	vsum4sbs	v2.03			Vector Sum across Quarter Signed Byte Saturate
000100 .... 11001 001000	VX	I	291	vsum4shs	v2.03			Vector Sum across Quarter Signed Halfword Saturate
000100 .... 11000 001000	VX	I	292	vsum4ubs	v2.03			Vector Sum across Quarter Unsigned Byte Saturate
000100 .... 11110 001000	VX	I	290	vsumsws	v2.03			Vector Sum across Signed Word Saturate

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 13 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
000100 ..... ///// 01101 001110	VX	I	253	vupkhpx	v2.03			Vector Unpack High Pixel
000100 ..... ///// 01000 001110	VX	I	254	vupkhsb	v2.03			Vector Unpack High Signed Byte
000100 ..... ///// 01001 001110	VX	I	254	vupkhsh	v2.03			Vector Unpack High Signed Halfword
000100 ..... ///// 11001 001110	VX	I	254	vupkhsw	v2.07			Vector Unpack High Signed Word
000100 ..... ///// 01111 001110	VX	I	253	vupklpx	v2.03			Vector Unpack Low Pixel
000100 ..... ///// 01010 001110	VX	I	254	vupklsb	v2.03			Vector Unpack Low Signed Byte
000100 ..... ///// 01011 001110	VX	I	254	vupklsh	v2.03			Vector Unpack Low Signed Halfword
000100 ..... ///// 11011 001110	VX	I	254	vupklsw	v2.07			Vector Unpack Low Signed Word
000100 .... 10011 000100	VX	I	313	vxor	v2.03			Vector Logical XOR
011111 ///.. ///// ///// 00000 11110/	X	II	876	wait	v3.0			Wait for Interrupt
011010 00000 00000 00000 00000 000000	D	I	93	xnop	v2.05			Executed No Operation
011111 .... 01001 11100.	X	I	94	xor[.]	P1		SR	XOR
011010	D	I	93	xori	P1			XOR Immediate
011011	D	I	93	xoris	P1			XOR Immediate Shifted
111100 ..... ///// ..... 10101 1001..	XX2	I	512	xsabsdp	v2.06			VSX Scalar Absolute Double-Precision
111111 ..... 00000 ..... 11001 00100/	X	I	512	xsabsqp	v3.0			VSX Scalar Absolute Quad-Precision
111100 .... 00100 000...	XX3	I	513	xsadddp	v2.06			VSX Scalar Add Double-Precision
111111 .... 00000 00100.	X	I	520	xsaddqp[o]	v3.0			VSX Scalar Add Quad-Precision [with round to Odd]
111100 .... 00000 000...	XX3	I	518	xsaddsp	v2.07			VSX Scalar Add Single-Precision
111100 .... 00000 011...	XX3	I	524	xscmpeqdp	v3.0			VSX Scalar Compare Equal Double-Precision
111100 ...// .... 00111 011../	XX3	I	522	xscmpexpdp	v3.0			VSX Scalar Compare Exponents Double-Precision
111111 ...// 00101 00100/	X	I	523	xscmpexpqp	v3.0			VSX Scalar Compare Exponents Quad-Precision
111100 .... 00010 011...	XX3	I	525	xscmpgedp	v3.0			VSX Scalar Compare Greater Than or Equal Double-Precision
111100 .... 00001 011...	XX3	I	526	xscmpgtdp	v3.0			VSX Scalar Compare Greater Than Double-Precision
111100 ...// .... 00101 011../	XX3	I	527	xscmpodp	v2.06			VSX Scalar Compare Ordered Double-Precision
111111 ...// 00100 00100/	X	I	529	xscmpoqp	v3.0			VSX Scalar Compare Ordered Quad-Precision
111100 ...// .... 00100 011../	XX3	I	530	xscmpudp	v2.06			VSX Scalar Compare Unordered Double-Precision
111111 ...// 10100 00100/	X	I	532	xscmpuqp	v3.0			VSX Scalar Compare Unordered Quad-Precision
111100 .... 10110 000...	XX3	I	533	xscpsgndp	v2.06			VSX Scalar Copy Sign Double-Precision
111111 .... 00011 00100/	X	I	533	xscpsgnqp	v3.0			VSX Scalar Copy Sign Quad-Precision
111100 ..... 10001 ..... 10101 1011..	XX2	I	534	xscvdphp	v3.0			VSX Scalar Convert with round Double-Precision to Half-Precision format
111111 ..... 10110 ..... 11010 00100/	X	I	535	xscvdpqp	v3.0			VSX Scalar Convert Double-Precision to Quad-Precision format
111100 ..... ///// ..... 10000 1001..	XX2	I	536	xscvdpsp	v2.06			VSX Scalar Convert with round Double-Precision to Single-Precision format
111100 ..... ///// ..... 10000 1011..	XX2	I	537	xscvdpspn	v2.07			VSX Scalar Convert Double-Precision to Single-Precision Non-signalling format
111100 ..... ///// ..... 10101 1000..	XX2	I	537	xscvdpsxds	v2.06			VSX Scalar Convert with round to zero Double-Precision to Signed Doubleword format
111100 ..... ///// ..... 00101 1000..	XX2	I	540	xscvdpsxws	v2.06			VSX Scalar Convert with round to zero Double-Precision to Signed Word format
111100 ..... ///// ..... 10100 1000..	XX2	I	542	xscvdpuxds	v2.06			VSX Scalar Convert with round to zero Double-Precision to Unsigned Doubleword format
111100 ..... ///// ..... 00100 1000..	XX2	I	544	xscvdpuxws	v2.06			VSX Scalar Convert with round to zero Double-Precision to Unsigned Word format
111100 ..... 10000 ..... 10101 1011..	XX2	I	546	xscvhpdp	v3.0			VSX Scalar Convert Half-Precision to Double-Precision format
111111 ..... 10100 ..... 11010 00100.	X	I	547	xscvqpdp[o]	v3.0			VSX Scalar Convert with round Quad-Precision to Double-Precision format [with round to Odd]
111111 ..... 11001 ..... 11010 00100/	X	I	548	xscvqpsdz	v3.0			VSX Scalar Convert with round to zero Quad-Precision to Signed Doubleword format
111111 ..... 01001 ..... 11010 00100/	X	I	550	xscvqpswz	v3.0			VSX Scalar Convert with round to zero Quad-Precision to Signed Word format
111111 ..... 10001 ..... 11010 00100/	X	I	552	xscvqpudz	v3.0			VSX Scalar Convert with round to zero Quad-Precision to Unsigned Doubleword format
111111 ..... 00001 ..... 11010 00100/	X	I	554	xscvqpuwz	v3.0			VSX Scalar Convert with round to zero Quad-Precision to Unsigned Word format
111111 ..... 01010 ..... 11010 00100/	X	I	556	xscvsdqp	v3.0			VSX Scalar Convert Signed Doubleword to Quad-Precision format
111100 ..... ///// ..... 10100 1001..	XX2	I	557	xscvspdp	v2.06			VSX Scalar Convert Single-Precision to Double-Precision format

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 14 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111100 ..... ///// ..... 10100 1011..	XX2	I	558	xscvspdpn	v2.07			VSX Scalar Convert Single-Precision to Double-Precision Non-signalling format
111100 ..... ///// ..... 10111 1000..	XX2	I	559	xscvsxddp	v2.06			VSX Scalar Convert with round Signed Doubleword to Double-Precision format
111100 ..... ///// ..... 10011 1000..	XX2	I	559	xscvsxdsp	v2.07			VSX Scalar Convert with round Signed Doubleword to Single-Precision format
111111 ..... 00010 ..... 11010 00100/	X	I	560	xscvudqp	v3.0			VSX Scalar Convert Unsigned Doubleword to Quad-Precision format
111100 ..... ///// ..... 10110 1000..	XX2	I	561	xscvuxddp	v2.06			VSX Scalar Convert with round Unsigned Doubleword to Double-Precision format
111100 ..... ///// ..... 10010 1000..	XX2	I	561	xscvuxdsp	v2.07			VSX Scalar Convert with round Unsigned Doubleword to Single-Precision format
111100 .... 00111 000...	XX3	I	562	xsdivdp	v2.06			VSX Scalar Divide Double-Precision
111111 .... 10001 00100.	X	I	564	xsdivqp[o]	v3.0			VSX Scalar Divide Quad-Precision [with round to Odd]
111100 .... 00011 000...	XX3	I	566	xsdivsp	v2.07			VSX Scalar Divide Single-Precision
111100 .... 11100 10110.	XX1	I	568	xsiexpdp	v3.0			VSX Scalar Insert Exponent Double-Precision
111111 .... 11011 00100/	X	I	569	xsiexpqp	v3.0			VSX Scalar Insert Exponent Quad-Precision
111100 .... 00100 001...	XX3	I	570	xsmaddadp	v2.06			VSX Scalar Multiply-Add Type-A Double-Precision
111100 .... 00000 001...	XX3	I	573	xsmaddasp	v2.07			VSX Scalar Multiply-Add Type-A Single-Precision
111100 .... 00101 001...	XX3	I	570	xsmaddmdp	v2.06			VSX Scalar Multiply-Add Type-M Double-Precision
111100 .... 00001 001...	XX3	I	573	xsmaddmsp	v2.07			VSX Scalar Multiply-Add Type-M Single-Precision
111111 .... 01100 00100.	X	I	576	xsmaddqp[o]	v3.0			VSX Scalar Multiply-Add Quad-Precision [with round to Odd]
111100 .... 10000 000...	XX3	I	581	xsmaxcdp	v3.0			VSX Scalar Maximum Type-C Double-Precision
111100 .... 10100 000...	XX3	I	579	xsmaxdp	v2.06			VSX Scalar Maximum Double-Precision
111100 .... 10010 000...	XX3	I	583	xsmaxjdp	v3.0			VSX Scalar Maximum Type-J Double-Precision
111100 .... 10001 000...	XX3	I	587	xsmincdp	v3.0			VSX Scalar Minimum Type-C Double-Precision
111100 .... 10101 000...	XX3	I	585	xsmindp	v2.06			VSX Scalar Minimum Double-Precision
111100 .... 10011 000...	XX3	I	589	xsminjdp	v3.0			VSX Scalar Minimum Type-J Double-Precision
111100 .... 00110 001...	XX3	I	591	xsmsubadp	v2.06			VSX Scalar Multiply-Subtract Type-A Double-Precision
111100 .... 00010 001...	XX3	I	594	xsmsubasp	v2.07			VSX Scalar Multiply-Subtract Type-A Single-Precision
111100 .... 00111 001...	XX3	I	591	xsmsubmdp	v2.06			VSX Scalar Multiply-Subtract Type-M Double-Precision
111100 .... 00011 001...	XX3	I	594	xsmsubmsp	v2.07			VSX Scalar Multiply-Subtract Type-M Single-Precision
111111 .... 01101 00100.	X	I	597	xsmsubqp[o]	v3.0			VSX Scalar Multiply-Subtract Quad-Precision [with round to Odd]
111100 .... 00110 000...	XX3	I	600	xsmuldp	v2.06			VSX Scalar Multiply Double-Precision
111111 .... 00001 00100.	X	I	602	xsmulqp[o]	v3.0			VSX Scalar Multiply Quad-Precision [with round to Odd]
111100 .... 00010 000...	XX3	I	604	xsmulsp	v2.07			VSX Scalar Multiply Single-Precision
111100 ..... ///// ..... 10110 1001..	XX2	I	606	xsnabsdp	v2.06			VSX Scalar Negative Absolute Double-Precision
111111 ..... 01000 ..... 11001 00100/	X	I	606	xsnabsqp	v3.0			VSX Scalar Negative Absolute Quad-Precision
111100 ..... ///// ..... 10111 1001..	XX2	I	607	xsnegdp	v2.06			VSX Scalar Negate Double-Precision
111111 ..... 10000 ..... 11001 00100/	X	I	607	xsnegqp	v3.0			VSX Scalar Negate Quad-Precision
111100 .... 10100 001...	XX3	I	608	xsnmaddadp	v2.06			VSX Scalar Negative Multiply-Add Type-A Double-Precision
111100 .... 10000 001...	XX3	I	613	xsnmaddasp	v2.07			VSX Scalar Negative Multiply-Add Type-A Single-Precision
111100 .... 10101 001...	XX3	I	608	xsnmaddmdp	v2.06			VSX Scalar Negative Multiply-Add Type-M Double-Precision
111100 .... 10001 001...	XX3	I	613	xsnmaddmsp	v2.07			VSX Scalar Negative Multiply-Add Type-M Single-Precision
111111 .... 01110 00100.	X	I	616	xsnmaddqp[o]	v3.0			VSX Scalar Negative Multiply-Add Quad-Precision [with round to Odd]
111100 .... 10110 001...	XX3	I	619	xsnmsubadp	v2.06			VSX Scalar Negative Multiply-Subtract Type-A Double-Precision
111100 .... 10010 001...	XX3	I	622	xsnmsubasp	v2.07			VSX Scalar Negative Multiply-Subtract Type-A Single-Precision
111100 .... 10111 001...	XX3	I	619	xsnmsubmdp	v2.06			VSX Scalar Negative Multiply-Subtract Type-M Double-Precision
111100 .... 10011 001...	XX3	I	622	xsnmsubmsp	v2.07			VSX Scalar Negative Multiply-Subtract Type-M Single-Precision
111111 .... 01111 00100.	X	I	625	xsnmsubqp[o]	v3.0			VSX Scalar Negative Multiply-Subtract Quad-Precision [with round to Odd]
111100 ..... ///// ..... 00100 1001..	XX2	I	628	xsrdpi	v2.06			VSX Scalar Round Double-Precision to Integral
111100 ..... ///// ..... 00110 1011..	XX2	I	629	xsrdpic	v2.06			VSX Scalar Round Double-Precision to Integral using Current rounding mode
111100 ..... ///// ..... 00111 1001..	XX2	I	630	xsrdpim	v2.06			VSX Scalar Round Double-Precision to Integral toward -Infinity
111100 ..... ///// ..... 00110 1001..	XX2	I	630	xsrdpip	v2.06			VSX Scalar Round Double-Precision to Integral toward +Infinity
111100 ..... ///// ..... 00101 1001..	XX2	I	631	xsrdpiz	v2.06			VSX Scalar Round Double-Precision to Integral toward Zero

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 15 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111100 ..... ///// ..... 00101 1010..	XX2	I	632	xsredp	v2.06			VSX Scalar Reciprocal Estimate Double-Precision
111100 ..... ///// ..... 00001 1010..	XX2	I	633	xsresp	v2.07			VSX Scalar Reciprocal Estimate Single-Precision
111111 ..... ////. ....000 00101.	X	I	634	xsrqpi[x]	v3.0			VSX Scalar Round Quad-Precision to Integral [Exact]
111111 ..... ////. 001 00101/	X	I	636	xsrqpxp	v3.0			VSX Scalar Round Quad-Precision to XP
111100 ..... ///// ..... 10001 1001..	XX2	I	638	xsrsp	v2.07			VSX Scalar Round Double-Precision to Single-Precision
111100 ..... ///// ..... 00100 1010..	XX2	I	639	xsrsqrtedp	v2.06			VSX Scalar Reciprocal Square Root Estimate Double-Precision
111100 ..... ///// ..... 00000 1010..	XX2	I	640	xsrsqrtesp	v2.07			VSX Scalar Reciprocal Square Root Estimate Single-Precision
111100 ..... ///// ..... 00100 1011..	XX2	I	641	xssqrtdp	v2.06			VSX Scalar Square Root Double-Precision
111111 ..... 11011 ..... 11001 00100.	X	I	642	xssqrtqp[o]	v3.0			VSX Scalar Square Root Quad-Precision [with round to Odd]
111100 ..... ///// ..... 00000 1011..	XX2	I	644	xssqrtsp	v2.07			VSX Scalar Square Root Single-Precision
111100 .... 00101 000...	XX3	I	645	xssubdp	v2.06			VSX Scalar Subtract Double-Precision
111111 .... 10000 00100.	X	I	647	xssubqp[o]	v3.0			VSX Scalar Subtract Quad-Precision [with round to Odd]
111100 .... 00001 000...	XX3	I	649	xssubsp	v2.07			VSX Scalar Subtract Single-Precision
111100 ...// .... 00111 101../	XX3	I	651	xstdivdp	v2.06			VSX Scalar Test for software Divide Double-Precision
111100 ...// ///// 00110 1010./	XX2	I	652	xstsqrtdp	v2.06			VSX Scalar Test for software Square Root Double-Precision
111100 .... 10110 1010./	XX2	I	653	xststdcdp	v3.0			VSX Scalar Test Data Class Double-Precision
111111 .... 10110 00100/	X	I	654	xststdcqp	v3.0			VSX Scalar Test Data Class Quad-Precision
111100 .... 10010 1010./	XX2	I	655	xststdcsp	v3.0			VSX Scalar Test Data Class Single-Precision
111100 ..... 00000 ..... 10101 1011./	XX2	I	656	xsxexpdp	v3.0			VSX Scalar Extract Exponent Double-Precision
111111 ..... 00010 ..... 11001 00100/	X	I	656	xsxexpqp	v3.0			VSX Scalar Extract Exponent Quad-Precision
111100 ..... 00001 ..... 10101 1011./	XX2	I	657	xsxsigdp	v3.0			VSX Scalar Extract Significand Double-Precision
111111 ..... 10010 ..... 11001 00100/	X	I	657	xsxsigqp	v3.0			VSX Scalar Extract Significand Quad-Precision
111100 ..... ///// ..... 11101 1001..	XX2	I	658	xvabsdp	v2.06			VSX Vector Absolute Double-Precision
111100 ..... ///// ..... 11001 1001..	XX2	I	658	xvabssp	v2.06			VSX Vector Absolute Single-Precision
111100 .... 01100 000...	XX3	I	659	xvadddp	v2.06			VSX Vector Add Double-Precision
111100 .... 01000 000...	XX3	I	663	xvaddsp	v2.06			VSX Vector Add Single-Precision
111100 .... .1100 011...	XX3	I	665	xvcmpeqdp[.]	v2.06			VSX Vector Compare Equal Double-Precision
111100 .... .1000 011...	XX3	I	666	xvcmpeqsp[.]	v2.06			VSX Vector Compare Equal Single-Precision
111100 .... .1110 011...	XX3	I	667	xvcmpgedp[.]	v2.06			VSX Vector Compare Greater Than or Equal Double-Precision
111100 .... .1010 011...	XX3	I	668	xvcmpgesp[.]	v2.06			VSX Vector Compare Greater Than or Equal Single-Precision
111100 .... .1101 011...	XX3	I	669	xvcmpgtdp[.]	v2.06			VSX Vector Compare Greater Than Double-Precision
111100 .... .1001 011...	XX3	I	670	xvcmpgtsp[.]	v2.06			VSX Vector Compare Greater Than Single-Precision
111100 .... 11110 000...	XX3	I	671	xvcpsgndp	v2.06			VSX Vector Copy Sign Double-Precision
111100 .... 11010 000...	XX3	I	671	xvcpsgnsp	v2.06			VSX Vector Copy Sign Single-Precision
111100 ..... ///// ..... 11000 1001..	XX2	I	672	xvcvdpsp	v2.06			VSX Vector Convert with round Double-Precision to Single-Precision format
111100 ..... ///// ..... 11101 1000..	XX2	I	673	xvcvdpsxds	v2.06			VSX Vector Convert with round to zero Double-Precision to Signed Doubleword format
111100 ..... ///// ..... 01101 1000..	XX2	I	675	xvcvdpsxws	v2.06			VSX Vector Convert with round to zero Double-Precision to Signed Word format
111100 ..... ///// ..... 11100 1000..	XX2	I	677	xvcvdpuxds	v2.06			VSX Vector Convert with round to zero Double-Precision to Unsigned Doubleword format
111100 ..... ///// ..... 01100 1000..	XX2	I	679	xvcvdpuxws	v2.06			VSX Vector Convert with round to zero Double-Precision to Unsigned Word format
111100 ..... 11000 ..... 11101 1011..	XX2	I	681	xvcvhpsp	v3.0			VSX Vector Convert Half-Precision to Single-Precision format
111100 ..... ///// ..... 11100 1001..	XX2	I	682	xvcvspdp	v2.06			VSX Vector Convert Single-Precision to Double-Precision format
111100 ..... 11001 ..... 11101 1011..	XX2	I	683	xvcvsphp	v3.0			VSX Vector Convert with round Single-Precision to Half-Precision format
111100 ..... ///// ..... 11001 1000..	XX2	I	684	xvcvspsxds	v2.06			VSX Vector Convert with round to zero Single-Precision to Signed Doubleword format
111100 ..... ///// ..... 01001 1000..	XX2	I	686	xvcvspsxws	v2.06			VSX Vector Convert with round to zero Single-Precision to Signed Word format
111100 ..... ///// ..... 11000 1000..	XX2	I	688	xvcvspuxds	v2.06			VSX Vector Convert with round to zero Single-Precision to Unsigned Doubleword format
111100 ..... ///// ..... 01000 1000..	XX2	I	690	xvcvspuxws	v2.06			VSX Vector Convert with round to zero Single-Precision to Unsigned Word format
111100 ..... ///// ..... 11111 1000..	XX2	I	692	xvcvsxddp	v2.06			VSX Vector Convert with round Signed Doubleword to Double-Precision format

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 16 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111100 ..... ///// ..... 11011 1000..	XX2	I	692	xvcvsxdsp	v2.06			VSX Vector Convert with round Signed Doubleword to Single-Precision format
111100 ..... ///// ..... 01111 1000..	XX2	I	693	xvcvsxwdp	v2.06			VSX Vector Convert Signed Word to Double-Precision format
111100 ..... ///// ..... 01011 1000..	XX2	I	693	xvcvsxwsp	v2.06			VSX Vector Convert with round Signed Word to Single-Precision format
111100 ..... ///// ..... 11110 1000..	XX2	I	694	xvcvuxddp	v2.06			VSX Vector Convert with round Unsigned Doubleword to Double-Precision format
111100 ..... ///// ..... 11010 1000..	XX2	I	694	xvcvuxdsp	v2.06			VSX Vector Convert with round Unsigned Doubleword to Single-Precision format
111100 ..... ///// ..... 01110 1000..	XX2	I	695	xvcvuxwdp	v2.06			VSX Vector Convert Unsigned Word to Double-Precision format
111100 ..... ///// ..... 01010 1000..	XX2	I	695	xvcvuxwsp	v2.06			VSX Vector Convert with round Unsigned Word to Single-Precision format
111100 .... 01111 000...	XX3	I	696	xvdivdp	v2.06			VSX Vector Divide Double-Precision
111100 .... 01011 000...	XX3	I	698	xvdivsp	v2.06			VSX Vector Divide Single-Precision
111100 .... 11111 000...	XX3	I	700	xviexpdp	v3.0			VSX Vector Insert Exponent Double-Precision
111100 .... 11011 000...	XX3	I	700	xviexpsp	v3.0			VSX Vector Insert Exponent Single-Precision
111100 .... 01100 001...	XX3	I	701	xvmaddadp	v2.06			VSX Vector Multiply-Add Type-A Double-Precision
111100 .... 01000 001...	XX3	I	704	xvmaddasp	v2.06			VSX Vector Multiply-Add Type-A Single-Precision
111100 .... 01101 001...	XX3	I	701	xvmaddmdp	v2.06			VSX Vector Multiply-Add Type-M Double-Precision
111100 .... 01001 001...	XX3	I	704	xvmaddmsp	v2.06			VSX Vector Multiply-Add Type-M Single-Precision
111100 .... 11100 000...	XX3	I	707	xvmaxdp	v2.06			VSX Vector Maximum Double-Precision
111100 .... 11000 000...	XX3	I	709	xvmaxsp	v2.06			VSX Vector Maximum Single-Precision
111100 .... 11101 000...	XX3	I	711	xvmindp	v2.06			VSX Vector Minimum Double-Precision
111100 .... 11001 000...	XX3	I	713	xvminsp	v2.06			VSX Vector Minimum Single-Precision
111100 .... 01110 001...	XX3	I	715	xvmsubadp	v2.06			VSX Vector Multiply-Subtract Type-A Double-Precision
111100 .... 01010 001...	XX3	I	718	xvmsubasp	v2.06			VSX Vector Multiply-Subtract Type-A Single-Precision
111100 .... 01111 001...	XX3	I	715	xvmsubmdp	v2.06			VSX Vector Multiply-Subtract Type-M Double-Precision
111100 .... 01011 001...	XX3	I	718	xvmsubmsp	v2.06			VSX Vector Multiply-Subtract Type-M Single-Precision
111100 .... 01110 000...	XX3	I	721	xvmuldp	v2.06			VSX Vector Multiply Double-Precision
111100 .... 01010 000...	XX3	I	723	xvmulsp	v2.06			VSX Vector Multiply Single-Precision
111100 ..... ///// ..... 11110 1001..	XX2	I	725	xvnabsdp	v2.06			VSX Vector Negative Absolute Double-Precision
111100 ..... ///// ..... 11010 1001..	XX2	I	725	xvnabssp	v2.06			VSX Vector Negative Absolute Single-Precision
111100 ..... ///// ..... 11111 1001..	XX2	I	726	xvnegdp	v2.06			VSX Vector Negate Double-Precision
111100 ..... ///// ..... 11011 1001..	XX2	I	726	xvnegsp	v2.06			VSX Vector Negate Single-Precision
111100 .... 11100 001...	XX3	I	727	xvnmaddadp	v2.06			VSX Vector Negative Multiply-Add Type-A Double-Precision
111100 .... 11000 001...	XX3	I	732	xvnmaddasp	v2.06			VSX Vector Negative Multiply-Add Type-A Single-Precision
111100 .... 11101 001...	XX3	I	727	xvnmaddmdp	v2.06			VSX Vector Negative Multiply-Add Type-M Double-Precision
111100 .... 11001 001...	XX3	I	732	xvnmaddmsp	v2.06			VSX Vector Negative Multiply-Add Type-M Single-Precision
111100 .... 11110 001...	XX3	I	735	xvnmsubadp	v2.06			VSX Vector Negative Multiply-Subtract Type-A Double-Precision
111100 .... 11010 001...	XX3	I	738	xvnmsubasp	v2.06			VSX Vector Negative Multiply-Subtract Type-A Single-Precision
111100 .... 11111 001...	XX3	I	735	xvnmsubmdp	v2.06			VSX Vector Negative Multiply-Subtract Type-M Double-Precision
111100 .... 11011 001...	XX3	I	738	xvnmsubmsp	v2.06			VSX Vector Negative Multiply-Subtract Type-M Single-Precision
111100 ..... ///// ..... 01100 1001..	XX2	I	741	xvrdpi	v2.06			VSX Vector Round Double-Precision to Integral
111100 ..... ///// ..... 01110 1011..	XX2	I	741	xvrdpic	v2.06			VSX Vector Round Double-Precision to Integral using Current rounding mode
111100 ..... ///// ..... 01111 1001..	XX2	I	742	xvrdpim	v2.06			VSX Vector Round Double-Precision to Integral toward -Infinity
111100 ..... ///// ..... 01110 1001..	XX2	I	742	xvrdpip	v2.06			VSX Vector Round Double-Precision to Integral toward +Infinity
111100 ..... ///// ..... 01101 1001..	XX2	I	743	xvrdpiz	v2.06			VSX Vector Round Double-Precision to Integral toward Zero
111100 ..... ///// ..... 01101 1010..	XX2	I	744	xvredp	v2.06			VSX Vector Reciprocal Estimate Double-Precision
111100 ..... ///// ..... 01001 1010..	XX2	I	745	xvresp	v2.06			VSX Vector Reciprocal Estimate Single-Precision
111100 ..... ///// ..... 01000 1001..	XX2	I	746	xvrspi	v2.06			VSX Vector Round Single-Precision to Integral
111100 ..... ///// ..... 01010 1011..	XX2	I	746	xvrspic	v2.06			VSX Vector Round Single-Precision to Integral using Current rounding mode
111100 ..... ///// ..... 01011 1001..	XX2	I	747	xvrspim	v2.06			VSX Vector Round Single-Precision to Integral toward -Infinity
111100 ..... ///// ..... 01010 1001..	XX2	I	747	xvrspip	v2.06			VSX Vector Round Single-Precision to Integral toward +Infinity
111100 ..... ///// ..... 01001 1001..	XX2	I	748	xvrspiz	v2.06			VSX Vector Round Single-Precision to Integral toward Zero
111100 ..... ///// ..... 01100 1010..	XX2	I	748	xvrsqrtedp	v2.06			VSX Vector Reciprocal Square Root Estimate Double-Precision
111100 ..... ///// ..... 01000 1010..	XX2	I	750	xvrsqrtesp	v2.06			VSX Vector Reciprocal Square Root Estimate Single-Precision

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 17 of 18)

Instruction[1] 0:5 6:10 11:15 16:20 21:25 26:31	Format	Book	Page	Mnemonic	Version[2]	Privilege[3]	Mode Dep[4]	Name
111100 ..... ///// ..... 01100 1011..	XX2	I	751	xvsqrtdp	v2.06			VSX Vector Square Root Double-Precision
111100 ..... ///// ..... 01000 1011..	XX2	I	752	xvsqrtsp	v2.06			VSX Vector Square Root Single-Precision
111100 .... 01101 000...	XX3	I	753	xvsubdp	v2.06			VSX Vector Subtract Double-Precision
111100 .... 01001 000...	XX3	I	755	xvsubsp	v2.06			VSX Vector Subtract Single-Precision
111100 .../// .... 01111 101../	XX3	I	757	xvtdivdp	v2.06			VSX Vector Test for software Divide Double-Precision
111100 .../// .... 01011 101../	XX3	I	758	xvtdivsp	v2.06			VSX Vector Test for software Divide Single-Precision
111100 .../// ///// 01110 1010./	XX2	I	759	xvtsqrtdp	v2.06			VSX Vector Test for software Square Root Double-Precision
111100 .../// ///// 01010 1010./	XX2	I	759	xvtsqrtsp	v2.06			VSX Vector Test for software Square Root Single-Precision
111100 .... 1111. 101...	XX2	I	760	xvtstdcdp	v3.0			VSX Vector Test Data Class Double-Precision
111100 .... 1101. 101...	XX2	I	761	xvtstdcsp	v3.0			VSX Vector Test Data Class Single-Precision
111100 ..... 00000 ..... 11101 1011..	XX2	I	762	xvxexpdp	v3.0			VSX Vector Extract Exponent Double-Precision
111100 ..... 01000 ..... 11101 1011..	XX2	I	762	xvxexpsp	v3.0			VSX Vector Extract Exponent Single-Precision
111100 ..... 00001 ..... 11101 1011..	XX2	I	763	xvxsigdp	v3.0			VSX Vector Extract Significand Double-Precision
111100 ..... 01001 ..... 11101 1011..	XX2	I	763	xvxsigsp	v3.0			VSX Vector Extract Significand Single-Precision
111100 ..... 10111 ..... 11101 1011..	XX2	I	764	xxbrd	v3.0			VSX Vector Byte-Reverse Doubleword
111100 ..... 00111 ..... 11101 1011..	XX2	I	764	xxbrh	v3.0			VSX Vector Byte-Reverse Halfword
111100 ..... 11111 ..... 11101 1011..	XX2	I	765	xxbrq	v3.0			VSX Vector Byte-Reverse Quadword
111100 ..... 01111 ..... 11101 1011..	XX2	I	765	xxbrw	v3.0			VSX Vector Byte-Reverse Word
111100 ..... /.... ..... 01010 0101..	XX2	I	766	xxextractuw	v3.0			VSX Vector Extract Unsigned Word
111100 ..... /.... ..... 01011 0101..	XX2	I	766	xxinsertw	v3.0			VSX Vector Insert Word
111100 .... 10000 010...	XX3	I	767	xxland	v2.06			VSX Vector Logical AND
111100 .... 10001 010...	XX3	I	767	xxlandc	v2.06			VSX Vector Logical AND with Complement
111100 .... 10111 010...	XX3	I	768	xxleqv	v2.07			VSX Vector Logical Equivalence
111100 .... 10110 010...	XX3	I	768	xxlnand	v2.07			VSX Vector Logical NAND
111100 .... 10100 010...	XX3	I	769	xxlnor	v2.06			VSX Vector Logical NOR
111100 .... 10010 010...	XX3	I	770	xxlor	v2.06			VSX Vector Logical OR
111100 .... 10101 010...	XX3	I	769	xxlorc	v2.07			VSX Vector Logical OR with Complement
111100 .... 10011 010...	XX3	I	770	xxlxor	v2.06			VSX Vector Logical XOR
111100 .... 00010 010...	XX3	I	771	xxmrghw	v2.06			VSX Vector Merge Word High
111100 .... 00110 010...	XX3	I	771	xxmrglw	v2.06			VSX Vector Merge Word Low
111100 .... 00011 010...	XX3	I	772	xxperm	v3.0			VSX Vector Permute
111100 .... 0..01 010...	XX3	I	773	xxpermdi	v2.06			VSX Vector Doubleword Permute Immediate
111100 .... 00111 010...	XX3	I	772	xxpermr	v3.0			VSX Vector Permute Right-indexed
111100 .... 11....	XX4	I	773	xxsel	v2.06			VSX Vector Select
111100 .... 0..00 010...	XX3	I	774	xxsldwi	v2.06			VSX Vector Shift Left Double by Word Immediate
111100 ..... 00... ..... 01011 01000.	XX1	I	774	xxspltib	v3.0			VSX Vector Splat Immediate Byte
111100 ..... ///.. ..... 01010 0100..	XX2	I	774	xxspltw	v2.06			VSX Vector Splat Word

Figure 90. Power ISA AS Instruction Set Sorted by Mnemonic (Sheet 18 of 18)

1. Key to Instruction column.

/	Instruction bit that corresponds to a reserved field, must have a value of 0, otherwise invalid form.
-	Instruction bit that corresponds to an operand bit, may have a value of either 0 or 1.
0	Instruction bit having a value 0.
1	Instruction bit having a value 1.

2. Key to Version column.

P1	Instruction introduced in the POWER Architecture.
P2	Instruction introduced in the POWER2 Architecture.
PPC	Instruction introduced in the PowerPC Architecture prior to v2.00.
v2.00	Instruction introduced in the PowerPC Architecture Version 2.00.
v2.01	Instruction introduced in the PowerPC Architecture Version 2.01.
v2.02	Instruction introduced in the PowerPC Architecture Version 2.02.
v2.03	Instruction introduced in the Power ISA Architecture Version 2.03.
v2.04	Instruction introduced in the Power ISA Architecture Version 2.04.
v2.05	Instruction introduced in the Power ISA Architecture Version 2.05.
v2.06	Instruction introduced in the Power ISA Architecture Version 2.06.
v2.07	Instruction introduced in the Power ISA Architecture Version 2.07.
v3.0	Instruction introduced in the Power ISA Architecture Version 3.0.
v3.0B	Instruction introduced in the Power ISA Architecture Version 3.0B.

3. Key to Privilege column.

P	Denotes an instruction that is treated as privileged.
O	Denotes an instruction that is treated as privileged or nonprivileged (or hypervisor, for mtspr), depending on the SPR or PMR number.
PI	Denotes an instruction that is illegal in privileged state.
H	Denotes an instruction that can be executed only in hypervisor state
U	Denotes an instruction that can be executed only in ultravisor state

4. Key to Mode Dependency column.
   Except as described below and in Section 1.11.3, “Effective Address Calculation”, in Book I, all instructions are independent of whether the processor is in 32-bit or 64-bit mode.

CT	If the instruction tests the Count Register, it tests the low-order 32 bits in 32-bit mode and all 64 bits in 64-bit mode.
SR	The setting of status registers (such as XER and CR0) is mode-dependent.
32	The instruction can be executed only in 32-bit mode.
64	The instruction can be executed only in 64-bit mode.

Last Page - End of Document