RFC ls010 Simple-V Zero-Overhead Loop Prefix Subsystem </>

- Funded by NLnet under the Privacy and Enhanced Trust Programme, EU Horizon2020 Grant 825310, and NGI0 Entrust No 101069594

- https://www.sigarch.org/simd-instructions-considered-harmful/
- https://libre-soc.org/openpower/sv/
- https://libre-soc.org/openpower/sv/rfc/ls010/
- https://bugs.libre-soc.org/show_bug.cgi?id=1056
- https://git.openpower.foundation/isa/PowerISA/issues/64
- https://git.openpower.foundation/isa/PowerISA/issues/122
- https://libre-soc.org/openpower/sv/ls010/trial_addi/
- https://libre-soc.org/openpower/sv/ls010/hypothetical_addi/

Severity: Major

Status: New

Date: 04 Apr 2023. v2 TODO

Target: v3.2B

Source: v3.1B

Books and Section affected:

```
New Book: new Zero-Overhead-Loop
New Appendix, Zero-Overhead-Loop
```

Summary


```
Adds a Zero-Overhead-Loop Subsystem based on the Cray True-Scalable Vector concept
in a RISC-paradigm fashion. Total instructions six 5-bit XO, plus Prefix format (PO9).
```


Submitter: Luke Leighton (Libre-SOC)

Requester: Libre-SOC

Impact on processor:

```
Addition of new "Zero-Overhead-Loop-Control" DSP-style Vector-style
subsystem that in simple low-end (Embedded) systems may be minimalistically
and easily be implemented by inserting a new fully-independent Pipeline Stage
in between Decode and Issue, with very little disruption, and in higher
performance pre-existing Multi-Issue Out-of-Order systems seamlessly fits likewise
to significantly boost performance.
```

Impact on software:

```
Requires support for new instructions in assembler, debuggers, and related tools.
Dramatically reduces instructions. Requires introduction of term "High-Level Assembler"
```

Keywords:

```
Cray Supercomputing, Vectorization, Zero-Overhead-Loop-Control (ZOLC),
True-Scalable Vectors, Multi-Issue Out-of-Order, Sequential Programming Model,
Digital Signal Processing (DSP), High-level Assembler
```

Motivation

Just at the time when customers are asking for higher performance, the seductive lure of SIMD, as outlined in the sigarch “SIMD Considered Harmful” article, is getting out of control and damaging the reputation of mainstream general-purpose ISAs that offer it. A solution from 50 years ago exists in the form of Cray-Style True-Scalable Vectors. However the usual way that True-Scalable Vector ISAs are done *also* adds more instructions and complexifies the ISA. Simple-V takes a step back to a simpler era in computing from half a century ago: the Zilog Z80 CPIR and LDIR instructions, and the 8086 REP instruction, and brings them forward to Modern-day Computing. The result is a huge reduction in programming complexity, and a strong base to project the Power ISA back to the world’s most powerful Supercomputing ISA for at least the next two decades.

Notes and Observations:

Related RFCs are {RFC ls008} for the two Management instructions `setvl` and `svstep`, and {RFC ls009} for the REMAP Subsystem. Also {RFC ls001} is a Dependency as it introduces Primary Opcode 9 64-bit encoding. An additional RFC {RFC ls005.xlen} introduced XLEN on which SVP64 is also critically dependent, for Element-width Overrides.

Changes

Add the following entries to:

- A new “Vector Looping” Book
- New Vector-Looping Chapters
- New Vector-Looping Appendices

[[!tag opf_rfc]]

SVP64 Zero-Overhead Loop Prefix Subsystem </>

This document describes {Scalable Vectors for Power ISA} augmentation of the Power v3.0B ISA.

Credits and acknowledgements:

- Luke Leighton
- Jacob Lifshay
- Hendrik Boom
- Richard Wilbur
- Alexandre Oliva
- Cesar Strauss
- NLnet Foundation, for funding
- OpenPOWER Foundation
- Paul Mackerras
- Brad Frey
- Cathy May
- Toshaan Bharvani
- IBM for the Power ISA itself

Introduction </>

Simple-V is a type of Vectorization best described as a "Prefix Loop Subsystem" similar to the 5 decades-old Zilog Z80 `LDIR`[1] instruction and to the 8086 `REP`[2] Prefix instruction. More advanced features are similar to the Z80 `CPIR`[3] instruction.

Except where explicitly stated all bit numbers remain as in the rest of the Power ISA: in MSB0 form (the bits are numbered from 0 at the MSB on the left and counting up as you move rightwards to the LSB end). All bit ranges are inclusive (so `4:6` means bits 4, 5, and 6, in MSB0 order). **All register numbering and element numbering however is LSB0 ordering** which is a different convention from that used elsewhere in the Power ISA.

The SVP64 prefix always comes before the suffix in PC order and must be considered an independent "Defined Word-instruction"[4] that augments the behaviour of the following instruction (also a Defined Word-instruction), but does **not** change the actual Decoding of that following instruction just because it is Prefixed. Unlike EXT100-163, where the Suffix is considered an entirely new Opcode Space, SVP64-Prefixed instructions must never be treated or regarded as a different Opcode Space.

Two apparent exceptions to the above hard rule exist: SV Branch-Conditional operations and LD/ST-update "Post-Increment" Mode. Post-Increment was considered sufficiently high priority (significantly reducing hot-loop instruction count) that one bit in the Prefix is reserved for it (*Note the intention to release that bit and move Post-Increment instructions to EXT2xx, as part of [[sv/rfc/ls011]]*). Vectorized Branch-Conditional operations "embed" the original Scalar Branch-Conditional behaviour into a much more advanced variant that is highly suited to High-Performance Computation (HPC), Supercomputing, and parallel GPU Workloads.

Architectural Resource Allocation note: at present it is possible to perform partial parallel decode of the SVP64 24-bit Encoding Area at the same time as decoding of the Suffix. Multi-Issue Implementations may even Decode multiple 32-bit words in parallel and follow up with a second cycle of joining Prefix and Suffix "after-the-fact". Mixing and overlaying 64-bit Opcode Encodings into the {SVP64 24-bit Prefix}{Defined Word-instruction} space creates a hard dependency that catastrophically damages Multi-Issue Decoding by greatly complexifying Parallel Instruction-Length Detection. Therefore it has to be prohibited to accept RFCs which fundamentally violate the following hard requirement: ***under no circumstances*** *must the use of SVP64 24-bit Suffixes* ***also*** *imply a different Opcode space from* ***any*** *non-prefixed Word. Even RESERVED or Illegal Words must be Orthogonal.*

Subset implementations in hardware are permitted, as long as certain rules are followed, allowing for full soft-emulation including future revisions. Compliancy Subsets exist to ensure minimum levels of binary inter-operability expectations within certain environments. Details in the {SVP64 Appendix}.

SVP64 encoding features </>

A number of features need to be compacted into a very small space of only 24 bits:

- Independent per-register Scalar/Vector tagging and range extension on every register
- Element width overrides on both source and destination
- Predication on both source and destination
- Two different sources of predication: INT and CR Fields
- SV Modes including saturation (for Audio, Video and DSP), mapreduce, and fail-first mode.

Different classes of operations require different formats. The earlier sections cover the common formats and the five separate modes have their own section later:

- CR operations (crops),

[1] Zilog Z80 LDIR
[2] 8086 REP
[3] Zilog Z80 CPIR
[4] Defined Word-instruction: Power ISA v3.1 Section 1.6

- Arithmetic/Logical (termed “normal”),
- Load/Store Immediate,
- Load/Store Indexed,
- Branch-Conditional.

Definition of “PO9-Prefixed” </>

Used in the context of “A PO9-Prefixed Word” this is a new area similar to EXT100-163 that is shared between SVP64-Single, SVP64, 32 Vectorizable new Opcode areas EXT232-263, and a 32-bit area, EXT900, that is also Vectorizable and SVP64-Single extensible in future. See [[sv/po9_encoding]].

Definition of “SVP64-Prefix” </>

A 24-bit RISC-Paradigm Encoding area for Loop-Augmentation of the next “Defined Word-instruction-instruction”. Used in the context of “An SVP64-Prefixed Defined Word-instruction”, as separate and distinct from the 32-bit PO9-Prefix that *holds* a 24-bit SVP64 Prefix.

Definition of “Vectorizable” and “Unvectorizable” </>

“Vectorizable” Defined Word-instructions are Scalar instructions that benefit from SVP64 Loop-Prefixing. Conversely, any operation that inherently makes no sense if repeated in a Vector Loop is termed “Unvectorizable” or “Unvectorized”. Examples include `sc` or `sync` which have no registers. `mtmsr` is also classed as Unvectorizable because there is only one `MSR`.

Unvectorized instructions are required to be detected as such if Prefixed (either SVP64 or SVP64Single) and an Illegal Instruction Trap raised.

Architectural Note: Given that a “pre-classification” Decode Phase is required (identifying whether the Suffix - Defined Word-instruction - is Arithmetic/Logical, CR-op, Load/Store or Branch-Conditional), adding “Unvectorized” to this phase is not unreasonable.

Vectorizable Defined Word-instructions are **required** to be Vectorized, or they may not be permitted to be added at all to the Power ISA as Defined Word-instructions.

Engineering note: implementations may not choose to add Defined Word-instructions without also adding hardware support for SVP64-Prefixing of the same.

ISA Working Group note: Vectorized PackedSIMD instructions if ever proposed should be considered Unvectorizable and except in extreme mitigating circumstances rejected immediately.

Definition of Strict Element-Level Execution Order </>

Where Instruction Execution Order[5] guarantees the appearance of sequential execution of instructions, Simple-V requires a corresponding guarantee for Elements because in Simple-V “Execution of Elements” is synonymous with “Execution of instructions”.

Precise Interrupt Guarantees </>

Strict Instruction Execution Order is defined as giving the appearance, as far as programs are concerned, that instructions were executed strictly in the sequence that they occurred. A “Precise” out-of-order Micro-architecture goes to considerable lengths to ensure that this is the case.

Many Vector ISAs allow interrupts to occur in the middle of processing of large Vector operations, only under the condition that partial results are cleanly discarded, and continuation on return from the Trap Handler will restart the entire operation. The reason is that saving of full Architectural State is not practical. An example would be a Floating-Point Horizontal Sum instruction (very common in Vector ISAs) or a Dot Product instruction that specifies a higher degree of accuracy for the *internal* accumulator than the registers.

Simple-V operates on an entirely different paradigm from traditional Vector ISAs: as a “Sub-Execution Context”, where “Elements” are synonymous with Scalar instructions. With this in mind implementations must observe Strict **Element**-Level Execution Order[[#svp64_eeo]] at all times. *Any* element is Interruptible, and Architectural State may be fully preserved and restored regardless of that same State.

Engineering note: implementations are permitted have higher latency to perform context-switching (particularly if REMAP is active).

Interrupts still only save `MSR` and `PC` in `SRR0` and `SRR1` but the full SVP64 Architectural State may be saved and restored through manual copying of `SVSTATE` (and the four REMAP SPRs if in use at the time, which may be determined by `SVSTATE[32:46]` being non-zero).

Programmer’s note: Trap Handlers (and any stack-based context save/restore) must avoid the use of SVP64 Prefixed instructions to perform the necessary save/restore of Simple-V Architectural State (SPR SVSTATE), just as use of FPRs and VSRs is presently avoided. However once saved, and set to known-good, SVP64 Prefixed instructions may be used to save/restore GPRs, SPRs, FPRs and other state.

[5] Strict Instruction Execution Order is defined in Public v3.1 Book I Section 2.2

Programmer's note: SVSHAPE0-3 alters Element Execution Order, but only if activated in SVSHAPE. It is therefore technically possible in a Trap Handler to save SVSTATE (`mfspr t0, SVSTATE`), then clear bits 32-46. At this point it becomes safe to use SVP64 to save sequential batches of SPRs (`setvli MAXVL=VL=4; sv.mfspr *t0, *SVSHAPE0`)

The only major caveat for REMAP is that after an explicit change to Architectural State caused by writing to the Simple-V SPRs, some implementations may find it easier to take longer to calculate where in a given Schedule the re-mapping Indices were. Obvious examples include Interrupts occuring in the middle of a non-RADIX2 Matrix Multiply Schedule (5x3 by 3x3 for example), which will force some implementations to perform divide and modulo calculations.

An additional caveat involves Condition Register Fields when also used as Predicate Masks. An operation that overwrites the same CR Fields that are simultaneously being used as a Predicate Mask should exercise extreme care if the overwritten CR field element was needed by a subsequent Element for its Predicate Mask bit.

Some implementations may deploy Cray's technique of "Vector Chaining" (including in this case reading the CR field containing the Predicate bit until the very last moment), and consequently avoiding the risk of overwrite is the responsibility of the Programmer. `hphint` may be used here to good effect. Extra Special care is particularly needed here when using REMAP and also Vertical-First Mode.

The simplest option is to use Integer Predicate Masks but the caveats are stricter:

- In Vertical-First loops Programmers **must not** write to any Integers (r3, r0, r31) used as Predicate Masks. Doing so is `UNDEFINED` behaviour.
- An **entire** Vector is held up on Horizontal-First Mode if the Integer Predicate is still in in-flight Reservation Stations or pipelines. Speculative Vector Chained Execution mitigates delays but can be heavy on Reservation Station resources.

Register files, elements, and Element-width Overrides </>

The relationship between register files, elements, and element-width overrides is expressed as follows:

- register files are considered to be *byte-level* contiguous SRAMs, accessed exclusively in Little-Endian Byte-Order at all times
- elements are sequential contiguous unbounded arrays starting at the "address" of any given 64-bit GPR or FPR, numbered from 0 as the first, "spilling" into numerically-sequentially-increasing GPRs
- element-width overrides set the width of the *elements* in the sequentially-numbered contiguous array.

The relationship is best defined in Canonical form, below, in ANSI c as a union data structure. A key difference is that VSR elements are bounded fixed at 128-bit, where SVP64 elements are conceptually unbounded and only limited by the Maximum Vector Length.

Future specification note: SVP64 may be defined on top of VSRs in future. At which point VSX also gains conceptually unbounded VSR register elements

In the Upper Compliancy Levels of SVP64 the size of the GPR and FPR Register files are expanded from 32 to 128 entries, and the number of CR Fields expanded from CR0-CR7 to CR0-CR127. (Note: A future version of SVP64 is anticipated to extend the VSR register file).

Memory access remains exactly the same: the effects of `MSR.LE` remain exactly the same, affecting as they already do and remain **only** on the Load and Store memory-register operation byte-order, and having nothing to do with the ordering of the contents of register files or register-register arithmetic or logical operations.

The only major impact on Arithmetic and Logical operations is that all Scalar operations are defined, where practical and workable, to have three new widths: elwidth=32, elwidth=16, elwidth=8.

Architectural note: a future revision of SVP64 for VSX may have entirely different definitions of possible elwidths.

The default of elwidth=64 is the pre-existing (Scalar) behaviour which remains 100% unchanged. Thus, `addi` is now joined by a 32-bit, 16-bit, and 8-bit variant of `addi`, but the sole exclusive difference is the width. *In no way* is the actual `addi` instruction fundamentally altered to become an entirely different operation (such as a subtract or multiply). FP Operations elwidth overrides are also defined, as explained in the {SVP64 Appendix}.

To be absolutely clear:

```
There are no conceptual arithmetic ordering or other changes over the
Scalar Power ISA definitions to registers or register files or to
arithmetic or Logical Operations, beyond element-width subdivision
```

Element offset numbering is naturally **LSB0-sequentially-incrementing from zero, not MSB0-incrementing** including when element-width overrides are used, at which point the elements progress through each register sequentially from the LSB end (confusingly numbered the highest in MSB0 ordering) and progress incrementally to the MSB end (confusingly numbered the lowest in MSB0 ordering).

When exclusively using MSB0-numbering, SVP64 becomes unnecessarily complex to both express and subsequently understand: the required conditional subtractions from 63, 31, 15 and 7 needed to express the fact that elements are LSB0-sequential unfortunately become a hostile minefield, obscuring both intent and meaning. Therefore for the purposes of this section the more natural **LSB0 numbering is assumed** and it is left to the reader to translate to MSB0 numbering.

The Canonical specification for how element-sequential numbering and element-width overrides is defined is expressed in the following c structure, assuming a Little-Endian system, and naturally using LSB0 numbering everywhere because the ANSI c specification is inherently LSB0. Note the deliberate similarity to how VSX register elements are defined, from Figure 97, Book I, Section 6.3, Page 258:

```
#pragma pack
typedef union {
    uint8_t actual_bytes[8];
    // all of these are very deliberately unbounded arrays
    // that intentionally "wrap" into subsequent actual_bytes...
    uint8_t  bytes[]; // elwidth 8
    uint16_t hwords[]; // elwidth 16
    uint32_t words[]; // elwidth 32
    uint64_t dwords[]; // elwidth 64

} el_reg_t;

// ... here, as packed statically-defined GPRs.
elreg_t int_regfile[128];

// use element 0 as the destination
void get_register_element(el_reg_t* el, int gpr, int element, int width) {
    switch (width) {
        case 64: el->dwords[0] = int_regfile[gpr].dwords[element];
        case 32: el->words[0] = int_regfile[gpr].words[element];
        case 16: el->hwords[0] = int_regfile[gpr].hwords[element];
        case 8 : el->bytes[0] = int_regfile[gpr].bytes[element];
    }
}

// use element 0 as the source
void set_register_element(el_reg_t* el, int gpr, int element, int width) {
    switch (width) {
        case 64: int_regfile[gpr].dwords[element] = el->dwords[0];
        case 32: int_regfile[gpr].words[element] = el->words[0];
        case 16: int_regfile[gpr].hwords[element] = el->hwords[0];
        case 8 : int_regfile[gpr].bytes[element] = el->bytes[0];
    }
}
```

Example Vector-looped add operation implementation when elwidths are 64-bit:

```
# vector-add RT, RA,RB using the "uint64_t" union member, "dwords"
for i in range(VL):
    int_regfile[RT].dword[i] = int_regfile[RA].dword[i] + int_regfile[RB].dword[i]
```

However if elwidth overrides are set to 16 for both source and destination:

```
# vector-add RT, RA, RB using the "uint64_t" union member "hwords"
for i in range(VL):
    int_regfile[RT].hwords[i] = int_regfile[RA].hwords[i] + int_regfile[RB].hwords[i]
```

The most fundamental aspect here to understand is that the wrapping into subsequent Scalar GPRs that occurs on larger-numbered elements including and especially on smaller element widths is **deliberate and intentional**. From this Canonical definition it should be clear that sequential elements begin at the LSB end of any given underlying Scalar GPR, progress to the MSB end, and then to the LSB end of the *next numerically-larger Scalar GPR*. In the example above if VL=5 and RT=1 then the contents of GPR(1) and GPR(2) will be as follows. For clarity in the table below:

- Both MSB0-ordered bitnumbering *and* LSB-ordered bitnumbering are shown
- The GPR-numbering is considered LSB0-ordered
- The Element-numbering (result0-result4) is LSB0-ordered
- Each of the results (result0-result4) are 16-bit
- “same” indicates “no change as a result of the Vectorized add”

MSB0:	0:15	16:31	32:47	48:63
LSB0:	63:48	47:32	31:16	15:0
GPR(0)	same	same	same	same
GPR(1)	result3	result2	result1	result0
GPR(2)	same	same	same	result4
GPR(3)	same	same	same	same
...	...	...	...	...
...	...	...	...	...

Note that the upper 48 bits of GPR(2) would **not** be modified due to the example having VL=5. Thus on “wrapping” - sequential progression from GPR(1) into GPR(2) - the 5th result modifies **only** the bottom 16 LSBs of GPR(1).

If the 16-bit operation were to be followed up with a 32-bit Vectorized Operation, the exact same contents would be viewed as follows:

```
| MSB0:  | 0:31                | 32:63               |
| LSB0:  | 63:32               | 31:0                |
|--------|---------------------|---------------------|
| GPR(0) | same                | same                |
| GPR(1) | (result3 || result2) | (result1 || result0) |
| GPR(2) | same                | (same    || result4) |
| GPR(3) | same                | same                |
| ...    | ...                 | ...                 |
| ...    | ...                 | ...                 |
```

In other words, this perspective really is no different from the situation where the actual Register File is treated as an Industry-standard byte-level-addressable Little-Endian-addressed SRAM. Note that this perspective does **not** involve `MSR.LE` in any way shape or form because `MSR.LE` is directly in control of the Memory-to-Register byte-ordering. This section is exclusively about how to correctly perceive Simple-V-Augmented **Register** Files.

Engineering note: to avoid a Read-Modify-Write at the register file it is strongly recommended to implement byte-level write-enable lines exactly as has been implemented in DRAM ICs for many decades. Additionally the predicate mask bit is advised to be associated with the element operation and alongside the result ultimately passed to the register file. When element-width is set to 64-bit the relevant predicate mask bit may be repeated eight times and pull all eight write-port byte-level lines HIGH. Clearly when element-width is set to 8-bit the relevant predicate mask bit corresponds directly with one single byte-level write-enable line. It is up to the Hardware Architect to then amortise (merge) elements together into both PredicatedSIMD Pipelines as well as simultaneous non-overlapping Register File writes, to achieve High Performance designs. Overall it helps to think of the GPR and FPR register files as being much more akin to a 64-bit-wide byte-level-addressable SRAM.

Comparative equivalent using VSR registers

For a comparative data point the VSR Registers may be expressed in the same fashion. The c code below is directly an expression of Figure 97 in Power ISA Public v3.1 Book I Section 6.3 page 258, *after compensating for MSB0 numbering in both bits and elements, adapting in full to LSB0 numbering, and obeying LE ordering*.

Crucial to understanding why the subtraction from 1,3,7,15 is present is because the Power ISA numbers VSX Registers elements also in MSB0 order. SVP64 very specifically numbers elements in **LSB0** order with the first element (numbered zero) being at the bitwise-numbered **LSB** end of the register, where VSX does the reverse: places the numerically-*highest* (last-numbered) element at the LSB end of the register.

```
#pragma pack
typedef union {
    // these do NOT match their Power ISA VSX numbering directly, they are all reversed
    // bytes[15] is actually VSR.byte[0] for example.  if this convention is not
    // followed then everything ends up in the wrong place
    uint8_t  bytes[16]; // elwidth 8, QTY 16 FIXED total
    uint16_t hwords[8]; // elwidth 16, QTY 8 FIXED total
    uint32_t words[4]; // elwidth 32, QTY 8 FIXED total
    uint64_t dwords[2]; // elwidth 64, QTY 2 FIXED total
    uint8_t actual_bytes[16]; // totals 128-bit
} el_reg_t;

elreg_t VSR_regfile[64];

static void check_num_elements(int elt, int width) {
    switch (width) {
        case 64: assert elt < 2;
        case 32: assert elt < 4;
        case 16: assert elt < 8;
        case 8 : assert elt < 16;
    }
}
void get_VSR_element(el_reg_t* el, int gpr, int elt, int width) {
    check_num_elements(elt, width);
    switch (width) {
        case 64: el->dwords[0] = VSR_regfile[gpr].dwords[1-elt];
        case 32: el->words[0] = VSR_regfile[gpr].words[3-elt];
        case 16: el->hwords[0] = VSR_regfile[gpr].hwords[7-elt];
        case 8 : el->bytes[0] = VSR_regfile[gpr].bytes[15-elt];
    }
}
void set_VSR_element(el_reg_t* el, int gpr, int elt, int width) {
    check_num_elements(elt, width);
    switch (width) {
        case 64: VSR_regfile[gpr].dwords[1-elt] = el->dwords[0];
        case 32: VSR_regfile[gpr].words[3-elt] = el->words[0];
```

```
            case 16: VSR_regfile[gpr].hwords[7-elt] = el->hwords[0];
            case 8 : VSR_regfile[gpr].bytes[15-elt] = el->bytes[0];
        }
    }
```

For VSR Registers one key difference is that the overlay of different element widths is clearly a *bounded static quantity*, whereas for Simple-V the elements are unrestrained and permitted to flow into *successive underlying Scalar registers*. This difference is absolutely critical to a full understanding of the entire Simple-V paradigm and why element-ordering, bit-numbering *and register numbering* are all so strictly defined.

Implementations are not permitted to violate the Canonical definition. Software will be critically relying on the wrapped (overflow) behaviour inherently implied by the unbounded variable-length c arrays.

Illustrating the exact same loop with the exact same effect as achieved by Simple-V we are first forced to create wrapper functions, to cater for the fact that VSR register elements are static bounded:

```
    int calc_VSR_reg_offs(int elt, int width) {
        switch (width) {
            case 64: return floor(elt / 2);
            case 32: return floor(elt / 4);
            case 16: return floor(elt / 8);
            case 8 : return floor(elt / 16);
        }
    }
    int calc_VSR_elt_offs(int elt, int width) {
        switch (width) {
            case 64: return (elt % 2);
            case 32: return (elt % 4);
            case 16: return (elt % 8);
            case 8 : return (elt % 16);
        }
    }
    void _set_VSR_element(el_reg_t* el, int gpr, int elt, int width) {
        int new_elt = calc_VSR_elt_offs(elt, width);
        int new_reg = calc_VSR_reg_offs(elt, width);
        set_VSR_element(el, gpr+new_reg, new_elt, width);
    }
```

And finally use these functions:

```
    # VSX-add RT, RA, RB using the "uint64_t" union member "hwords"
    for i in range(VL):
         el_reg_t result, ra, rb;
        _get_VSR_element(&ra, RA, i, 16);
        _get_VSR_element(&rb, RB, i, 16);
         result.hwords[0] = ra.hwords[0] + rb.hwords[0]; // use array 0 elements
        _set_VSR_element(&result, RT, i, 16);
```

Scalar Identity Behaviour </>

SVP64 is designed so that when the prefix is all zeros, and VL=1, no effect or influence occurs (no augmentation) such that all standard Power ISA v3.0/v3.1 instructions covered by the prefix are “unaltered”. This is termed `scalar identity behaviour` (based on the mathematical definition for “identity”, as in, “identity matrix” or better “identity transformation”).

Note that this is completely different from when VL=0. VL=0 turns all operations under its influence into `nops` (regardless of the prefix) whereas when VL=1 and the SV prefix is all zeros, the operation simply acts as if SV had not been applied at all to the instruction (an “identity transformation”).

The fact that `VL` is dynamic and can be set to any value at runtime based on program conditions and behaviour means very specifically that `scalar identity behaviour` is **not** a redundant encoding. If the only means by which VL could be set was by way of static-compiled immediates then this assertion would be false. VL should not be confused with MAXVL when understanding this key aspect of SimpleV.

Register Naming and size </>

As indicated above SV Registers are simply the GPR, FPR and CR register files extended linearly to larger sizes; SV Vectorization iterates sequentially through these registers (LSB0 sequential ordering from 0 to VL-1).

Where the integer regfile in standard scalar Power ISA v3.0B/v3.1B is r0 to r31, SV extends this range (in the Upper Compliancy Levels of SV) as r0 to r127. Likewise FP registers are extended to 128 (fp0 to fp127), and CR Fields are extended to 128 entries, CR0 thru CR127. In the Lower SV Compliancy Levels the quantity of registers remains the same in order to reduce implementation cost for Embedded systems.

The names of the registers therefore reflects a simple linear extension of the Power ISA v3.0B / v3.1B register naming, and in hardware this would be reflected by a linear increase in the size of the underlying SRAM used for the regfiles.

Note: when an EXTRA field (defined below) is zero, SV is deliberately designed so that the register fields are identical to as if SV was not in effect i.e. under these circumstances (EXTRA=0) the register field names RA, RB etc. are interpreted and treated as v3.0B / v3.1B scalar registers. This is part of `scalar identity behaviour` described above.

Condition Register(s)

The Scalar Power ISA Condition Register is a 64 bit register where the top 32 MSBs (numbered 0:31 in MSB0 numbering) are not used. This convention is *preserved* in SVP64 and an additional 15 Condition Registers provided in order to store the new CR Fields, CR8-CR15, CR16-CR23 etc. sequentially. The top 32 MSBs in each new SVP64 Condition Register are *also* not used: only the bottom 32 bits (numbered 32:63 in MSB0 numbering).

*Programmer's note: using `sv.mfcr` without element-width overrides to take into account the fact that the top 32 MSBs are zero and thus effectively doubling the number of GPR registers required to hold all 128 CR Fields would seem the only option because a source elwidth override to 32-bit would take only the bottom 16 LSBs of the Condition Register and set the top 16 LSBs to zeros. However in this case it is possible to use destination element-width overrides (for `sv.mfcr`. source overrides would be used on the GPR of `sv.mtocrf`), whereupon truncation of the 64-bit Condition Register(s) occurs, throwing away the zeros and storing the remaining (valid, desired) 32-bit values sequentially into (LSB0-convention) lower-numbered and upper-numbered halves of GPRs respectively. The programmer is expected to be aware however that the full width of the entire 64-bit Condition Register is considered to be "an element". This is **not** like any other Condition-Register instructions because all other CR instructions, on closer investigation, will be observed to all be CR-bit or CR-Field related. Thus a `VL` of 16 must be used*

Condition Register Fields as Predicate Masks

Condition Register Fields perform an additional duty in Simple-V: they are used for Predicate Masks. ARM's Scalar Instruction Set calls single-bit predication "Conditional Execution", and utilises Condition Codes for exactly this purpose to solve the problem caused by Branch Speculation. In a Vector ISA context the concept of Predication is naturally extended from single-bit to multi-bit, and the (well-known) benefits become all the more critical given that parallel branches in Vector ISAs are impossible (even a Vector ISA can only have Scalar branches).

However the Scalar Power ISA does not have Conditional Execution (for which, if it had ever been considered, Condition Register bits would be a perfect natural fit). Thus, when adding Predication using CR Fields via Simple-V it becomes a somewhat disruptive addition to the Power ISA.

To ameliorate this situation, particularly for pre-existing Hardware designs implementing up to Scalar Power ISA v3.1, some rules are set that allow those pre-existing designs not to require heavy modification to their existing Scalar pipelines. These rules effectively allow Hardware Architects to add the additional CR Fields CR8 to CR127 as if they were an **entirely separate register file**.

- any instruction involving more than 1 source 1 destination where one of the operands is a Condition Register is prohibited from using registers from both the CR0-7 group and the CR8-127 group at the same time.
- any instruction involving 1 source 1 destination where either the source or the destination is a Condition Register is prohibited from setting CR0-7 as a Vector.
- prohibitions are required to be enforced by raising Illegal Instruction Traps

Examples of permitted instructions:

```
sv.crand *cr8.eq, *cr16.le, *cr40.so # all CR8-CR127
sv.mfcr cr5, *cr40                   # only one source (CR40) copied to CR5
sv.mfcr *cr16, cr40                  # Vector-Splat CR40 onto CR16,17,18...
sv.mfcr *cr16, cr3                  # Vector-Splat CR3 onto CR16,17,18...
```

Examples of prohibited instructions:

```
sv.mfcr *cr0, cr40        # Vector-Splat onto CR0,1,2
sv.crand cr7, cr9, cr10   # crosses over between CR0-7 and CR8-127
```

Future expansion. </>

With the way that EXTRA fields are defined and applied to register fields, future versions of SV may involve 256 or greater registers in some way as long as the reputation of Power ISA for full backwards binary interoperability is preserved. Backwards binary compatibility may be achieved with a PCR bit (Program Compatibility Register) or an MSR bit analogous to SF. Further discussion is out of scope for this version of SVP64.

Additionally, a future variant of SVP64 will be applied to the Scalar (Quad-precision and 128-bit) VSX instructions. Element-width overrides are an opportunity to expand a future version of the Power ISA to 256-bit, 512-bit and 1024-bit operations, as well as doubling or quadrupling the number of VSX registers to 128 or 256. Again further discussion is out of scope for this version of SVP64.

SVP64 Remapped Encoding (`RM[0:23]`) </>

In the SVP64 Vector Prefix spaces, the 24 bits 8-31 are termed `RM`. Bits 32-37 are the Primary Opcode of the Suffix "Defined Word-instruction". 38-63 are the remainder of the Defined Word-instruction. Note that the new EXT232-263 SVP64 area it is obviously mandatory that bit 32 is required to be set to 1.

0-5	6	7	8-31	32-37	38-64	Description
PO	0	1	RM[0:23]	1nnnnn	xxxxxxxx	SVP64:EXT232-263
PO	1	1	RM[0:23]	nnnnnn	xxxxxxxx	SVP64:EXT000-063

It is important to note that unlike EXT1xx 64-bit prefixed instructions there is insufficient space in `RM` to provide identification of any SVP64 Fields without first partially decoding the 32-bit suffix. Similar to the "Forms" (X-Form, D-Form) the `RM` format is individually associated with every instruction. However this still does not adversely affect Multi-Issue Decoding because the identification of the *length* of anything in the 64-bit space has been kept brutally simple (EXT009), and further decoding of any number of 64-bit Encodings in parallel at that point is fully independent.

Extreme caution and care must be taken when extending SVP64 in future, to not create unnecessary relationships between prefix and suffix that could complicate decoding, adding latency.

Common RM fields </>

The following fields are common to all Remapped Encodings:

Field Name	Field bits	Description
MASKMODE	`0`	Execution (predication) Mask Kind
MASK	`1:3`	Execution Mask
SUBVL	`8:9`	Sub-vector length

The following fields are optional or encoded differently depending on context after decoding of the Scalar suffix:

Field Name	Field bits	Description
ELWIDTH	`4:5`	Element Width
ELWIDTH_SRC	`6:7`	Element Width for Source (or MASK_SRC in 2PM)
EXTRA	`10:18`	Register Extra encoding
MODE	`19:23`	changes Vector behaviour

- MODE changes the behaviour of the SV operation (result saturation, mapreduce)
- SUBVL groups elements together into vec2, vec3, vec4 for use in 3D and Audio/Video DSP work
- ELWIDTH and ELWIDTH_SRC overrides the instruction's destination and source operand width
- MASK (and MASK_SRC) and MASKMODE provide predication (two types of sources: scalar INT and Vector CR).
- Bits 10 to 18 (EXTRA) are further decoded depending on the RM category for the instruction, which is determined only by decoding the Scalar 32 bit suffix.

Similar to Power ISA `X-Form` etc. EXTRA bits are given designations, such as `RM-1P-3S1D` which indicates for this example that the operation is to be single-predicated and that there are 3 source operand EXTRA tags and one destination operand tag.

Note that if ELWIDTH != ELWIDTH_SRC this may result in reduced performance or increased latency in some implementations due to lane-crossing.

Mode </>

Mode is an augmentation of SV behaviour. Different types of instructions have different needs, similar to Power ISA v3.1 64 bit prefix 8LS and MTRR formats apply to different instruction types. Modes include Reduction, Iteration, arithmetic saturation, and Fail-First. More specific details in each section and in the {SVP64 Appendix}

- For condition register operations see {Condition Register Fields Mode}
- For LD/ST Modes, see {Load/Store Mode}.
- For Branch modes, see {Branch Mode}
- For arithmetic and logical, see {Arithmetic Mode}

ELWIDTH Encoding </>

Default behaviour is set to 0b00 so that zeros follow the convention of `scalar identity behaviour`. In this case it means that elwidth overrides are not applicable. Thus if a 32 bit instruction operates on 32 bit, `elwidth=0b00` specifies that this behaviour is unmodified. Likewise when a processor is switched from 64 bit to 32 bit mode, `elwidth=0b00` states that, again, the behaviour is not to be modified.

Only when elwidth is nonzero is the element width overridden to the explicitly required value.

Elwidth for Integers: </>

Value	Mnemonic	Description
00	DEFAULT	default behaviour for operation
01	`ELWIDTH=w`	Word: 32-bit integer
10	`ELWIDTH=h`	Halfword: 16-bit integer
11	`ELWIDTH=b`	Byte: 8-bit integer

This encoding is chosen such that the byte width may be computed as `8<<(3-ew)`

Elwidth for FP Registers: </>

Value	Mnemonic	Description
00	DEFAULT	default behaviour for FP operation
01	`ELWIDTH=f32`	32-bit IEEE 754 Single floating-point
10	`ELWIDTH=f16`	16-bit IEEE 754 Half floating-point
11	`ELWIDTH=bf16`	Reserved for `bf16`

Note: `bf16` is reserved for a future implementation of SV

Note that any IEEE754 FP operation in Power ISA ending in "s" (`fadds`) shall perform its operation at **half** the ELWIDTH then padded back out to ELWIDTH. `sv.fadds/ew=f32` shall perform an IEEE754 FP16 operation that is then "padded" to fill out to an IEEE754 FP32. When ELWIDTH=DEFAULT clearly the behaviour of `sv.fadds` is performed at 32-bit accuracy then padded back out to fit in IEEE754 FP64, exactly as for Scalar v3.0B "single" FP. Any FP operation ending in "s" where ELWIDTH=f16 or ELWIDTH=bf16 is reserved and must raise an illegal instruction (IEEE754 FP8 or BF8 are not defined).

Elwidth for CRs (no meaning) </>

Element-width overrides for CR Fields has no meaning. The bits are therefore used for other purposes, or when Rc=1, the Elwidth applies to the result being tested (a GPR or FPR), but not to the Vector of CR Fields.

SUBVL Encoding </>

The default for SUBVL is 1 and its encoding is 0b00 to indicate that SUBVL is effectively disabled (a SUBVL for-loop of only one element). this lines up in combination with all other "default is all zeros" behaviour.

Value	Mnemonic	Subvec	Description
00	`SUBVL=1`	single	Sub-vector length of 1
01	`SUBVL=2`	vec2	Sub-vector length of 2
10	`SUBVL=3`	vec3	Sub-vector length of 3
11	`SUBVL=4`	vec4	Sub-vector length of 4

The SUBVL encoding value may be thought of as an inclusive range of a sub-vector. SUBVL=2 represents a vec2, its encoding is 0b01, therefore this may be considered to be elements 0b00 to 0b01 inclusive.

Effectively, SUBVL is like a SIMD multiplier: instead of just 1 element operation issued, SUBVL element operations are issued (as an inner loop). The key difference between VL looping and SUBVL looping is that predication bits are applied per **group**, rather than by individual element.

Directly related to `subvl` is the `pack` and `unpack` Mode bits of `SVSTATE`.

MASK/MASK_SRC & MASKMODE Encoding </>

One bit (`MASKMODE`) indicates the mode: CR or Int predication. The two types may not be mixed.

Special note: to disable predication this field must be set to zero in combination with Integer Predication also being set to 0b000. this has the effect of enabling "all 1s" in the predicate mask, which is equivalent to "not having any predication at all".

`MASKMODE` may be set to one of 2 values:

Value	Description
0	MASK/MASK_SRC are encoded using Integer Predication
1	MASK/MASK_SRC are encoded using CR-based Predication

Integer Twin predication has a second set of 3 bits that uses the same encoding thus allowing either the same register (r3, r10 or r31) to be used for both src and dest, or different regs (one for src, one for dest).

Likewise CR based twin predication has a second set of 3 bits, allowing a different test to be applied.

Note that it cannot necessarily be assumed that Predicate Masks (whether INT or CR) are read in full *before* the operations proceed. In practice (for CR Fields) this creates an unnecessary block on parallelism, prohibiting "Vector Chaining". Therefore, it is up to the programmer to ensure that the CR field Elements used as Predicate Masks are not overwritten by any parallel Vector Loop. Doing so results in **UNDEFINED** behaviour, according to the definition outlined in the Power ISA v3.0B Specification.

Hardware Implementations are therefore free and clear to delay reading of individual CR fields until the actual predicated element operation needs to take place, safe in the knowledge that no programmer will have issued a Vector Instruction where previous elements could have overwritten (destroyed) not-yet-executed CR-Predicated element operations. This particularly is an issue when using REMAP, as the order in which CR-Field-based Predicate Mask bits could be read on a per-element execution basis could well conflict with the order in which prior elements wrote to the very same CR Field.

Additionally Programmers should avoid using r3 r10 or r30 as destination registers when these are also used as a Predicate Mask. Doing so is again UNDEFINED behaviour.

Usually in 2P `MASK_SRC` is exclusively in the EXTRA area. However for LD/ST-Indexed a different Encoding is required, designated `2PM`.

Integer Predication (MASKMODE=0) </>

When the predicate mode bit is zero the 3 bits are interpreted as below. Twin predication has an identical 3 bit field similarly encoded.

`MASK` and `MASK_SRC` may be set to one of 8 values, to provide the following meaning:

Value	Mnemonic	Element `i` enabled if:
000	ALWAYS	predicate effectively all 1s
001	1 « R3	`i == R3`
010	R3	`R3 & (1 << i)` is non-zero
011	~R3	`R3 & (1 << i)` is zero
100	R10	`R10 & (1 << i)` is non-zero
101	~R10	`R10 & (1 << i)` is zero
110	R30	`R30 & (1 << i)` is non-zero
111	~R30	`R30 & (1 << i)` is zero

r10 and r30 are at the high end of temporary and unused registers, so as not to interfere with register allocation from ABIs.

CR-based Predication (MASKMODE=1) </>

When the predicate mode bit is one the 3 bits are interpreted as below. Twin predication has an identical 3 bit field similarly encoded.

`MASK` and `MASK_SRC` may be set to one of 8 values, to provide the following meaning:

Value	Mnemonic	Element `i` is enabled if
000	lt	`CR[offs+i].LT` is set
001	nl/ge	`CR[offs+i].LT` is clear
010	gt	`CR[offs+i].GT` is set
011	ng/le	`CR[offs+i].GT` is clear
100	eq	`CR[offs+i].EQ` is set
101	ne	`CR[offs+i].EQ` is clear
110	so/un	`CR[offs+i].FU` is set
111	ns/nu	`CR[offs+i].FU` is clear

`offs` is defined as CR32 (4x8) so as to mesh cleanly with Vectorized Rc=1 operations (see below). Rc=1 operations start from CR8 (TBD).

The CR Predicates chosen must start on a boundary that Vectorized CR operations can access cleanly, in full. With EXTRA2 restricting starting points to multiples of 8 (CR0, CR8, CR16...) both Vectorized Rc=1 and CR Predicate Masks have to be adapted to fit on these boundaries as well.

Extra Remapped Encoding </>

Shows all instruction-specific fields in the Remapped Encoding `RM[10:18]` for all instruction variants. Note that due to the very tight space, the encoding mode is *not* included in the prefix itself. The mode is "applied", similar to Power ISA "Forms" (X-Form, D-Form) on a per-instruction basis, and, like "Forms" are given a des-

ignation (below) of the form `RM-nP-nSnD`. The full list of which instructions use which remaps is here {SVP64 Augmentation Table}.

Please note the following:

```
Machine-readable CSV files have been autogenerated which will make the
task of creating SV-aware ISA decoders, documentation, assembler tools
compiler tools Simulators documentation all aspects of SVP64 easier
and less prone to mistakes.  Please avoid manual re-creation of
information from the written specification wording in this chapter,
and use the CSV files or use the Canonical tool which creates the CSV
files, named sv_analysis.py. The information contained within
sv_analysis.py is considered to be part of this Specification, even
encoded as it is in python3.
```

The mappings are part of the SVP64 Specification in exactly the same way as X-Form, D-Form. New Scalar instructions added to the Power ISA will need a corresponding SVP64 Mapping, which can be derived by-rote from examining the Register "Profile" of the instruction.

There are two categories: Single and Twin Predication. Due to space considerations further subdivision of Single Predication is based on whether the number of src operands is 2 or 3. With only 9 bits available some compromises have to be made.

- `RM-1P-3S1D` Single Predication dest/src1/2/3, applies to 4-operand instructions (fmadd, isel, madd).
- `RM-1P-2S1D` Single Predication dest/src1/2 applies to 3-operand instructions (src1 src2 dest)
- `RM-2P-1S1D` Twin Predication (src=1, dest=1)
- `RM-2P-2S1D` Twin Predication (src=2, dest=1) primarily for LDST (Indexed)
- `RM-2P-1S2D` Twin Predication (src=1, dest=2) primarily for LDST Update
- `RM-2PM-2S1D` Twin Predication (src=2, dest=1) for LD/ST Update (Indexed)

The `2PM` designation uses bits 6 and 7 as well as the 9 EXTRA bits in order to extend two registers to EXTRA3, sacrificing destination elwidths in the process. `MASK_SRC` has a different encoding in `2PM`.

RM-1P-3S1D </>

Field Name	Field bits	Description
Rdest_EXTRA2	`10:11`	extends Rdest (R*_EXTRA2 Encoding)
Rsrc1_EXTRA2	`12:13`	extends Rsrc1 (R*_EXTRA2 Encoding)
Rsrc2_EXTRA2	`14:15`	extends Rsrc2 (R*_EXTRA2 Encoding)
Rsrc3_EXTRA2	`16:17`	extends Rsrc3 (R*_EXTRA2 Encoding)
EXTRA2_MODE	`18`	used by `divmod2du` and `maddedu` for RS

These are for 3 operand in and either 1 or 2 out instructions. 3-in 1-out includes `madd RT,RA,RB,RC`. (DRAFT) instructions such as `maddedu` have an implicit second destination, RS, the selection of which is determined by bit 18.

RM-1P-2S1D </>

Field Name	Field bits	Description
Rdest_EXTRA3	`10:12`	extends Rdest
Rsrc1_EXTRA3	`13:15`	extends Rsrc1
Rsrc2_EXTRA3	`16:18`	extends Rsrc3

These are for 2 operand 1 dest instructions, such as `add RT, RA, RB`. However also included are unusual instructions with an implicit dest that is identical to its src reg, such as `rlwinmi`.

Normally, with instructions such as `rlwinmi`, the scalar v3.0B ISA would not have sufficient bit fields to allow an alternative destination. With SV however this becomes possible. Therefore, the fact that the dest is implicitly also a src should not mislead: due to the *prefix* they are different SV regs.

- `rlwimi RA, RS, ...`
- Rsrc1_EXTRA3 applies to RS as the first src
- Rsrc2_EXTRA3 applies to RA as the second src
- Rdest_EXTRA3 applies to RA to create an **independent** dest.

With the addition of the EXTRA bits, the three registers each may be *independently* made vector or scalar, and be independently augmented to 7 bits in length.

RM-2P-1S1D/2S </>

Field Name	Field bits	Description
Rdest_EXTRA3	`10:12`	extends Rdest

Field Name	Field bits	Description
Rsrc1_EXTRA3	13:15	extends Rsrc1
MASK_SRC	16:18	Execution Mask for Source

RM-2P-2S is for stw etc. and is Rsrc1 Rsrc2.

Field Name	Field bits	Description
Rsrc1_EXTRA3	10:12	extends Rsrc1
Rsrc2_EXTRA3	13:15	extends Rsrc2
MASK_SRC	16:18	Execution Mask for Source

RM-1P-2S1D </>

single-predicate, three registers (2 read, 1 write)

Field Name	Field bits	Description
Rdest_EXTRA3	10:12	extends Rdest
Rsrc1_EXTRA3	13:15	extends Rsrc1
Rsrc2_EXTRA3	16:18	extends Rsrc2

RM-2P-2S1D/1S2D/3S </>

The primary purpose for this encoding is for Twin Predication on LOAD and STORE operations. see {Load/Store Mode} for detailed analysis.

RM-2P-2S1D:

Field Name	Field bits	Description
Rdest_EXTRA2	10:11	extends Rdest (R*_EXTRA2 Encoding)
Rsrc1_EXTRA2	12:13	extends Rsrc1 (R*_EXTRA2 Encoding)
Rsrc2_EXTRA2	14:15	extends Rsrc2 (R*_EXTRA2 Encoding)
MASK_SRC	16:18	Execution Mask for Source

RM-2P-1S2D:

For RM-2P-1S2D dest2 is in bits 14:15

Field Name	Field bits	Description
Rdest_EXTRA2	10:11	extends Rdest (R*_EXTRA2 Encoding)
Rsrc1_EXTRA2	12:13	extends Rsrc1 (R*_EXTRA2 Encoding)
Rdest2_EXTRA2	14:15	extends Rdest22 (R*_EXTRA2 Encoding)
MASK_SRC	16:18	Execution Mask for Source

RM-2P-3S:

Also that for RM-2P-3S (to cover stdx etc.) the names are switched to 3 src: Rsrc1_EXTRA2, Rsrc2_EXTRA2, Rsrc3_EXTRA2.

Field Name	Field bits	Description
Rsrc1_EXTRA2	10:11	extends Rsrc1 (R*_EXTRA2 Encoding)
Rsrc2_EXTRA2	12:13	extends Rsrc2 (R*_EXTRA2 Encoding)
Rsrc3_EXTRA2	14:15	extends Rsrc3 (R*_EXTRA2 Encoding)
MASK_SRC	16:18	Execution Mask for Source

Note also that LD with update indexed, which takes 2 src and creates 2 dest registers (e.g. lhaux RT,RA,RB), does not have room for 4 registers and also Twin Predication. Therefore these are treated as RM-2P-2S1D and the src spec for RA is also used for the same RA as a dest.

Note that if ELWIDTH != ELWIDTH_SRC this may result in reduced performance or increased latency in some implementations due to lane-crossing.

RM-2PM-2S1D/1S2D/3S </>

The primary purpose for this encoding is for Twin Predication on LOAD and STORE operations providing EXTRA3 for RT, RA and RS. see {Load/Store Mode} for detailed analysis.

RM-2PM-2S1D:

RT or RS requires EXTRA3, RA requires EXTRA3, but for RB EXTRA2 will suffice. `MASK_SRC` may be read from the bits normally used for dest-elwidth.

Field Name	Field bits	Description
Rdest_EXTRA3	`10:12`	extends Rdest (R*_EXTRA2 Encoding)
Rsrc1_EXTRA3	`13:15`	extends Rsrc1 (R*_EXTRA2 Encoding)
Rsrc2_EXTRA2	`16:17`	extends Rsrc2 (R*_EXTRA2 Encoding)
MASK_SRC	`6:7,18`	Execution Mask for Source

R*_EXTRA2/3 </>

EXTRA is the means by which two things are achieved:

1. Registers are marked as either Vector *or Scalar*
2. Register field numbers (limited typically to 5 bit) are extended in range, both for Scalar and Vector.

The register files are therefore extended:

- INT (GPR) is extended from r0-31 to r0-127
- FP (FPR) is extended from fp0-32 to fp0-fp127
- CR Fields are extended from CR0-7 to CR0-127

However due to pressure in `RM.EXTRA` not all these registers are accessible by all instructions, particularly those with a large number of operands (`madd`, `isel`).

In the following tables register numbers are constructed from the standard v3.0B / v3.1B 32 bit register field (RA, FRA) and the EXTRA2 or EXTRA3 field from the SV Prefix, determined by the specific RM-xx-yyyy designation for a given instruction. The prefixing is arranged so that interoperability between prefixing and nonprefixing of scalar registers is direct and convenient (when the EXTRA field is all zeros).

A pseudocode algorithm explains the relationship, for INT/FP (see {SVP64 Appendix} for CRs)

```
if extra3_mode:
    spec = EXTRA3
elif EXTRA2[0]:  # vector mode, can express even registers in r0-126
    spec = EXTRA2 << 1  # same as EXTRA3, shifted
else:            # scalar mode, can express registers in r0-63
    spec = (EXTRA2[0] << 2) | EXTRA2[1]
if spec[0]: # vector
     return (RA << 2) | spec[1:2]
else:         # scalar
     return (spec[1:2] << 5) | RA
```

Future versions may extend to 256 by shifting Vector numbering up. Scalar will not be altered.

Note that in some cases the range of starting points for Vectors is limited.

INT/FP EXTRA3 </>

If EXTRA3 is zero, maps to “scalar identity” (scalar Power ISA field naming).

Fields are as follows:

- Value: R_EXTRA3
- Mode: register is tagged as scalar or vector
- Range/Inc: the range of registers accessible from this EXTRA encoding, and the “increment” (accessibility). “/4” means that this EXTRA encoding may only give access (starting point) every 4th register.
- MSB..LSB: the bit field showing how the register opcode field combines with EXTRA to give (extend) the register number (GPR)

Encoding shown in LSB0: MSB down to LSB (MSB 6..0 LSB)

Value	Mode	Range/Inc	6..0
000	Scalar	`r0-r31/1`	`0b00 RA`
001	Scalar	`r32-r63/1`	`0b01 RA`
010	Scalar	`r64-r95/1`	`0b10 RA`
011	Scalar	`r96-r127/1`	`0b11 RA`
100	Vector	`r0-r124/4`	`RA 0b00`
101	Vector	`r1-r125/4`	`RA 0b01`
110	Vector	`r2-r126/4`	`RA 0b10`
111	Vector	`r3-r127/4`	`RA 0b11`

INT/FP EXTRA2 </>

If EXTRA2 is zero will map to “scalar identity behaviour” i.e Scalar Power ISA register naming:

Encoding shown in LSB0: MSB down to LSB (MSB 6..0 LSB)

Value	Mode	Range/inc	6..0
00	Scalar	r0-r31/1	0b00 RA
01	Scalar	r32-r63/1	0b01 RA
10	Vector	r0-r124/4	RA 0b00
11	Vector	r2-r126/4	RA 0b10

Note that unlike in EXTRA3, in EXTRA2:

- the GPR Vectors may only start from r0, r2, r4, r6, r8 and likewise FPR Vectors.
- the GPR Scalars may only go from r0, r1, r2.. r63 and likewise FPR Scalars.

as there is insufficient bits to cover the full range.

CR Field EXTRA3 </>

CR Field encoding is essentially the same but made more complex due to CRs being bit-based, because the application of SVP64 element-numbering applies to the CR *Field* numbering not the CR register *bit* numbering. Note that Vectors may only start from CR0, CR4, CR8, CR12, CR16, CR20... and Scalars may only go from CR0, CR1, ... CR31

Encoding shown in LSB0: MSB down to LSB (MSB 8..5 4..2 1..0 LSB), BA ranges are in MSB0.

For a 5-bit operand (BA, BB, BT):

Value	Mode	Range/Inc	8..5	4..2	1..0
000	Scalar	CR0-CR7/1	0b0000	BA[0:2]	BA[3:4]
001	Scalar	CR8-CR15/1	0b0001	BA[0:2]	BA[3:4]
010	Scalar	CR16-CR23/1	0b0010	BA[0:2]	BA[3:4]
011	Scalar	CR24-CR31/1	0b0011	BA[0:2]	BA[3:4]
100	Vector	CR0-CR112/16	BA[0:2] 0	0b000	BA[3:4]
101	Vector	CR4-CR116/16	BA[0:2] 0	0b100	BA[3:4]
110	Vector	CR8-CR120/16	BA[0:2] 1	0b000	BA[3:4]
111	Vector	CR12-CR124/16	BA[0:2] 1	0b100	BA[3:4]

For a 3-bit operand (e.g. BFA):

Value	Mode	Range/Inc	6..3	2..0
000	Scalar	CR0-CR7/1	0b0000	BFA
001	Scalar	CR8-CR15/1	0b0001	BFA
010	Scalar	CR16-CR23/1	0b0010	BFA
011	Scalar	CR24-CR31/1	0b0011	BFA
100	Vector	CR0-CR112/16	BFA 0	0b000
101	Vector	CR4-CR116/16	BFA 0	0b100
110	Vector	CR8-CR120/16	BFA 1	0b000
111	Vector	CR12-CR124/16	BFA 1	0b100

CR EXTRA2 </>

CR encoding is essentially the same but made more complex due to CRs being bit-based, because the application of SVP64 element-numbering applies to the CR *Field* numbering not the CR register *bit* numbering. Note that Vectors may only start from CR0, CR8, CR16, CR24, CR32...

Encoding shown in LSB0: MSB down to LSB (MSB 8..5 4..2 1..0 LSB), BA ranges are in MSB0.

For a 5-bit operand (BA, BB, BC):

Value	Mode	Range/Inc	8..5	4..2	1..0
00	Scalar	CR0-CR7/1	0b0000	BA[0:2]	BA[3:4]
01	Scalar	CR8-CR15/1	0b0001	BA[0:2]	BA[3:4]
10	Vector	CR0-CR112/16	BA[0:2] 0	0b000	BA[3:4]
11	Vector	CR8-CR120/16	BA[0:2] 1	0b000	BA[3:4]

For a 3-bit operand (e.g. BFA):

Value	Mode	Range/Inc	6..3	2..0
00	Scalar	CR0-CR7/1	0b0000	BFA
01	Scalar	CR8-CR15/1	0b0001	BFA
10	Vector	CR0-CR112/16	BFA 0	0b000
11	Vector	CR8-CR120/16	BFA 1	0b000

Normal SVP64 Modes, for Arithmetic and Logical Operations </>

- https://bugs.libre-soc.org/show_bug.cgi?id=574
- https://bugs.libre-soc.org/show_bug.cgi?id=558#c47
- https://bugs.libre-soc.org/show_bug.cgi?id=936 write on failfirst
- {SVP64 Chapter}

Normal SVP64 Mode covers Arithmetic and Logical operations to provide suitable additional behaviour. The Mode field is bits 19-23 of the {SVP64 Chapter} RM Field.

Table of contents:

[[!toc]]

Mode </>

Mode is an augmentation of SV behaviour, providing additional functionality. Some of these alterations are element-based (saturation), others are Vector-based (mapreduce, fail-on-first).

{Load/Store Mode}, {Condition Register Fields Mode} and {Branch Mode} are covered separately: the following Modes apply to Arithmetic and Logical SVP64 operations:

- **simple** mode is straight vectorization. No augmentations: the vector comprises an array of independently created results.
- **ffirst** or data-dependent fail-on-first: see separate section. The vector may be truncated depending on certain criteria. *VL is altered as a result*.
- **sat mode** or saturation: clamps each element result to a min/max rather than overflows / wraps. Allows signed and unsigned clamping for both INT and FP.
- **reduce mode**. If used correctly, a mapreduce (or a prefix sum) is performed. See {SVP64 Appendix}. Note that there are comprehensive caveats when using this mode, and it should not be confused with the Parallel Reduction {REMAP subsystem}. Also care is needed with `hphint`.

Note that ffirst and reduce modes are not anticipated to be high-performance in some implementations. ffirst due to interactions with VL, and reduce due to it creating overlapping operations in many of its uses. simple and saturate are however inter-element independent and may easily be parallelised to give high performance, regardless of the value of VL.

The Mode table for Arithmetic and Logical operations, being bits 19-23 of SVP64 `RM`, is laid out as follows:

0-1	2	3 4	description
0 0	0	dz sz	simple mode
0 0	1	RG 0	scalar reduce mode (mapreduce)
0 0	1	/ 1	reserved
1 0	N	dz sz	sat mode: N=0/1 u/s
VLi 1	inv	CR-bit	Rc=1: ffirst CR sel
VLi 1	inv	zz RC1	Rc=0: ffirst z/nonz

Fields:

- **sz / dz** source-zeroing, destination-zeroing. if predication is enabled will put zeros into the dest (or as src in the case of twin pred) when the predicate bit is zero. Otherwise the element is ignored or skipped, depending on context.
- **zz**: both sz and dz are set equal to this flag
- **inv CR bit** just as in branches (BO) these bits allow testing of a CR bit and whether it is set (inv=0) or unset (inv=1)
- **RG** inverts the Vector Loop order (VL-1 downto 0) rather than the normal 0..VL-1
- **N** sets signed/unsigned saturation.
- **RC1** as if Rc=1, on operations that do not have it (typically Logical)
- **VLi** VL inclusive: in fail-first mode, the truncation of VL *includes* the current element at the failure point rather than excludes it from the count.

For LD/ST Modes, see {Load/Store Mode}. For Condition Registers see {Condition Register Fields Mode}. For Branch modes, see {Branch Mode}.

Rounding, clamp and saturate </>

See {Audio and Video Opcodes} for relevant opcodes and use-cases.

To help ensure for example that audio quality is not compromised by overflow, “saturation” is provided, as well as a way to detect when saturation occurred if desired (Rc=1). When Rc=1 there will be a *vector* of CRs, one CR per element in the result (Note: this is different from VSX which has a single CR per block).

When N=0 the result is saturated to within the maximum range of an unsigned value. For integer ops this will be 0 to 2^elwidth-1. Similar logic applies to FP operations, with the result being saturated to maximum rather than returning INF, and the minimum to +0.0

When N=1 the same occurs except that the result is saturated to the min or max of a signed result, and for FP to the min and max value rather than returning +/- INF.

When Rc=1, the CR "overflow" bit is set on the CR associated with the element, to indicate whether saturation occurred. Note that due to the hugely detrimental effect it has on parallel processing, XER.SO is **ignored** completely and is **not** brought into play here. The CR overflow bit is therefore simply set to zero if saturation did not occur, and to one if it did. This behaviour (ignoring XER.SO) is actually optional in the SFFS Compliancy Subset: for SVP64 it is made mandatory *but only on Vectorized instructions*.

Note also that saturate on operations that set OE=1 must raise an Illegal Instruction due to the conflicting use of the CR.so bit for storing if saturation occurred. Vectorized Integer Operations that produce a Carry-Out (CA, CA32): these two bits will be `UNDEFINED` if saturation is also requested.

Note that the operation takes place at the maximum bitwidth (max of src and dest elwidth) and that truncation occurs to the range of the dest elwidth.

Programmer's Note: Post-analysis of the Vector of CRs to find out if any given element hit saturation may be done using a mapreduced CR op (cror), or by using the new crrweird instruction with Rc=1, which will transfer the required CR bits to a scalar integer and update CR0, which will allow testing the scalar integer for nonzero. See {CR Weird ops}. Alternatively, a Data-Dependent Fail-First may be used to truncate the Vector Length to non-saturated elements, greatly increasing the productivity of parallelised inner hot-loops.

Reduce mode </>

Reduction in SVP64 is similar in essence to other Vector Processing ISAs, but leverages the underlying scalar Base v3.0B operations. Thus it is more a convention that the programmer may utilise to give the appearance and effect of a Horizontal Vector Reduction. Due to the unusual decoupling it is also possible to perform prefix-sum (Fibonacci Series) in certain circumstances. Details are in the {SVP64 Appendix}

Reduce Mode should not be confused with Parallel Reduction {REMAP subsystem}. As explained in the {SVP64 Appendix} Reduce Mode switches off the check which would normally stop looping if the result register is scalar. Thus, the result scalar register, if also used as a source scalar, may be used to perform sequential accumulation. This *deliberately* sets up a chain of Register Hazard Dependencies (which advanced hardware may optimise out), whereas Parallel Reduce {REMAP subsystem} deliberately issues a Tree-Schedule of operations that may be parallelised.

Hardware architectural note: implementations may optimise out the Hazard Dependency chain as long as Sequential Program Execution Order is preserved. Easy examples include Reduction on Logical OR or AND operations.

Horizontal Parallelism Hint

`SVSTATE.hphint` declares to hardware that groups of elements up to this size are 100% independent (free of all Hazards inter-element but not inter-group). With Reduction literally creating Dependency Hazards on every element-level sub-instruction it is pretty clear that setting `hphint` *at all* would cause data corruption. However `sv.add *r0, *r4, *r0` for example clearly leaves room for four parallel elements. Programmers must be aware of this and exercise caution.

Data-dependent Fail-on-first </>

Data-dependent fail-on-first is CR-field-driven and is completely separate and distinct from LD/ST Fail-First (also known as Fault-First). Note in each case the assumption is that vector elements are required to appear to be executed in sequential Program Order. When REMAP is not active, element 0 would be the first.

Arithmetic/Logical Data-driven (CR-field-driven) fail-on-first performs a test ofvthe result, similar to Branch-Conditional `BO` field testing, and if the test fails, the Vector Loop operation terminates, and VL is truncated to either the *previous* element or the current one, depending on whether VLi (VL "inclusive") is clear or set, respectively.

Thus the new VL comprises a contiguous vector of results, all of which pass the testing criteria (equal to zero, less than zero etc as defined by the CR-bit test). When Rc=1 the Condition Regster Field for the element just tested is always written out (regardless of VLi).

- **VLi=0** Only elements that passed the test are written out. When Rc=1 the co-result CR Field element is written out (even if the current test failed). Vector length is truncated to "elements that passed"
- **VLi=1** Elements that were *tested* are written out. When Rc=1 the co-result CR Field element is written out. Vector length is truncated to "elements tested up to the first fail point"

Note: when VLi is clear, the behaviour at first seems counter-intuitive. A result is calculated but if the test fails it is prohibited from being actually written. This becomes intuitive again when it is remembered that the length that VL is set to is the number of written* elements, and only when VLI is set will the current element be included in that count.*

The CR-based data-driven fail-on-first is "new" and not found in ARM SVE or RVV. At the same time it is "old" because it is almost identical to a generalised form of Z80's `CPIR` instruction. It is extremely useful for reducing instruction count, however requires speculative execution involving modifications of VL to get high performance implementations. An additional mode (RC1=1) allows instructions that would not normally have an Rc=1 mode to at least be tested for zero or non-zero. The CR is stored (and the CR.eq bit tested against the `inv` field). If the CR.eq bit is equal to `inv` then the Vector is truncated and the loop ends.

VLi is only available as an option when `Rc=0` (or for instructions which do not have Rc). When set, the current element is always also included in the count (the new length that VL will be set to). This may be useful in combination with "inv" to truncate the Vector to *exclude* elements that fail a test, or, in the case of implementations of strncpy, to include the terminating zero.

In CR-based data-driven fail-on-first there is only the option to select and test one bit of each CR (just as with branch BO). For more complex tests this may be insufficient. If that is the case, a vectorized crop such as crand, cror or {CR Weird ops} crweirder may be used, and ffirst applied to the crop instead of to the arithmetic vector. Note that crops are covered by the {Condition Register Fields Mode} Mode format.

Use of Fail-on-first with Vertical-First Mode is not prohibited but is not really recommended. The effect of truncating VL may have unintended and unexpected consequences on subsequent instructions. VLi set will be fine: it is when VLi is clear that problems may be faced.

Programmer's note: `VLi` is only accessible in normal operations which in turn limits the CR field bit-testing to only `EQ/NE`. {Condition Register Fields Mode} are not so limited. Thus it is possible to use for example `sv.cror/ff=gt/vli *0,*0,*0`, which is not a nop because it allows Fail-First Mode to perform a test and truncate VL.

*Hardware implementor's note: effective Sequential Program Order must be preserved. Speculative Execution is perfectly permitted as long as the speculative elements are held back from writing to register files (kept in Resevation Stations), until such time as the relevant CR Field bit(s) has been analysed. All Speculative elements sequentially beyond the test-failure point **MUST** be cancelled. This is no different from standard Out-of-Order Execution and the modification effort to efficiently support Data-Dependent Fail-First within a pre-existing Multi-Issue Out-of-Order Engine is anticipated to be minimal. In-Order systems on the other hand are expected, unavoidably, to be low-performance unless they also make use of `SVSTATE.hphint` and exploit it to safely implement rudimentary Shadow-Commit-Hold normally only found in Out-of-Order systems.*

Two extremely important aspects of ffirst are:

- LDST ffirst may never set VL equal to zero. This because on the first element an exception must be raised "as normal".
- CR-based data-dependent ffirst on the other hand **can** set VL equal to zero. When VL is set zero due to the first element failing the CR bit-test, all subsequent vectorized operations are effectively `nops` which is *precisely the desired and intended behaviour*.

The second crucial aspect, compared to LDST Ffirst:

- LD/ST Failfirst may (beyond the initial first element conditions) truncate VL for any architecturally suitable reason. Beyond the first element LD/ST Failfirst is arbitrarily speculative and 100% non-deterministic.
- CR-based data-dependent first on the other hand MUST NOT truncate VL arbitrarily to a length decided by the hardware: VL MUST only be truncated based explicitly on whether a test fails. This because it is a precise Deterministic test on which algorithms can and will rely.

Floating-point Exceptions

When Floating-point exceptions are enabled VL must be truncated at the point where the Exception appears not to have occurred. If `VLi` is set then VL must include the faulting element, and thus the faulting element will always raise its exception. If however `VLi` is clear then VL **excludes** the faulting element and thus the exception will **never** be raised.

Although very strongly discouraged the Exception Mode that permits Floating Point Exception notification to arrive too late to unwind is permitted (under protest, due it violating the otherwise 100% Deterministic nature of Data-dependent Fail-first).

Use of lax FP Exception Notification Mode could result in parallel computations proceeding with invalid results that have to be explicitly detected, whereas with the strict FP Execption Mode enabled, FFirst truncates VL, allows subsequent parallel computation to avoid the exceptions entirely

Data-dependent fail-first on CR operations (crand etc) </>

Operations that actually produce or alter CR Field as a result have their own SVP64 Mode, described in {Condition Register Fields Mode}.

[[!tag standards]]

SV Load and Store </>

This section describes how Standard Load/Store Defined Word-instructions are exploited as Element-level Load/Stores and augmented to create direct equivalents of Vector Load/Store instructions.

Modes overview </>

Vectorization of Load and Store requires creation, from scalar operations, a number of different modes:

- **fixed aka "unit" stride** - contiguous sequence with no gaps
- **element strided** - sequential but regularly offset, with gaps
- **vector indexed** - vector of base addresses and vector of offsets
- **Speculative Fault-first** - where it makes sense to do so
- **Data-Dependent Fail-First** - Conditional truncation of Vector Length
- **Structure Packing** - covered in SV by {REMAP subsystem} and Pack/Unpack Mode.

Despite being constructed from Scalar LD/ST none of these Modes exist or make sense in any Scalar ISA. They ***only*** *exist in Vector ISAs and are a critical part of its value*.

Also included in SVP64 LD/ST is Element-width overrides and Twin-Predication.

Note also that Indexed {REMAP subsystem} mode may be applied to both Scalar LD/ST Immediate Defined Word-instructions *and* LD/ST Indexed Defined Word-instructions. LD/ST-Indexed should not be conflated with Indexed REMAP mode: clarification is provided below.

Determining the LD/ST Modes

A minor complication (caused by the retro-fitting of modern Vector features to a Scalar ISA) is that certain features do not exactly make sense or are considered a security risk. Fault-first on Vector Indexed would allow attackers to probe large numbers of pages from userspace, where strided Fault-first (by creating contiguous sequential LDs likely to be in the same Page) does not.

In addition, reduce mode makes no sense. Realistically we need an alternative table definition for {SVP64 Chapter} `RM.MODE`. The following modes make sense:

- simple (no augmentation)
- Fault-first (where Vector Indexed is banned)
- Data-dependent Fail-First (extremely useful for Linked-List pointer-chasing)
- Signed Effective Address computation (Vector Indexed only, on RB)

More than that however it is necessary to fit the usual Vector ISA capabilities onto both Power ISA LD/ST with immediate and to LD/ST Indexed. They present subtly different Mode tables, which, due to lack of space, have the following quirks:

- LD/ST Immediate has no individual control over src/dest zeroing, whereas LD/ST Indexed does.
- LD/ST Immediate has saturation but LD/ST Indexed does not.

Format and fields </>

Fields used in tables below:

- **zz**: both sz and dz are set equal to this flag. If predication is enabled will put zeros into the dest (or as src in the case of twin pred) when the predicate bit is zero. otherwise the element is ignored or skipped, depending on context.
- **inv CR bit** just as in branches (BO) these bits allow testing of a CR bit and whether it is set (inv=0) or unset (inv=1)
- **RC1** as if Rc=1, stores CRs *but not the result*
- **SEA** - Signed Effective Address, if enabled performs sign-extension on registers that have been reduced due to elwidth overrides
- **PI** - post-increment mode (applies to LD/ST with update only). the Effective Address utilised is always just RA, i.e. the computation of EA is stored in RA **after** it is actually used.
- **LF** - Load/Store Fail or Fault First: for any reason Load or Store Vectors may be truncated to (at least) one element, and VL altered to indicate such.
- **VLi** - Inclusive Data-Dependent Fail-First: the failing element is included in the Truncated Vector.
- **els** - Element-strided Mode: the element index (after REMAP) is multiplied by the immediate offset (or Scalar RB for Indexed). Restrictions apply.

When VLi=0 on Store Operations the Memory update does **not** take place on the element that failed. EA does **not** update into RA on Load/Store with Update instructions either.

LD/ST immediate

The table for {SVP64 Chapter} for `immed(RA)` which is `RM.MODE` (bits 19:23 of `RM`) is:

0	1	2	3 4	description
els	0	PI	zz LF	post-increment and Fault-First
VLi	1	inv	CR-bit	Data-Dependent ffirst CR sel

The `els` bit is only relevant when `RA.isvec` is clear: this indicates whether stride is unit or element:

```
if RA.isvec:
    svctx.ldstmode = indexed
elif els == 0:
    svctx.ldstmode = unitstride
elif immediate != 0:
    svctx.ldstmode = elementstride
```

An immediate of zero is a safety-valve to allow `LD-VSPLAT`: in effect the multiplication of the immediate-offset by zero results in reading from the exact same memory location, *even with a Vector register*. (Normally this type of behaviour is reserved for the mapreduce modes)

For `LD-VSPLAT`, on non-cache-inhibited Loads, the read can occur just the once and be copied, rather than hitting the Data Cache multiple times with the same memory read at the same location. The benefit of Cache-inhibited LD-splats is that it allows for memory-mapped peripherals to have multiple data values read in quick succession and stored in sequentially numbered registers (but, see Note below).

For non-cache-inhibited ST from a vector source onto a scalar destination: with the Vector loop effectively creating multiple memory writes to the same location, we can deduce that the last of these will be the "successful" one. Thus, implementations are free and clear to optimise out the overwriting STs, leaving just the last one as the "winner". Bear in mind that predicate masks will skip some elements (in source non-zeroing mode). Cache-inhibited ST operations on the other hand **MUST** write out a Vector source multiple successive times to the exact same Scalar destination. Just like Cache-inhibited LDs, multiple values may be written out in quick succession to a memory-mapped peripheral from sequentially-numbered registers.

Note that any memory location may be Cache-inhibited (Power ISA v3.1, Book III, 1.6.1, p1033)

Programmer's Note: an immediate also with a Scalar source as a "VSPLAT" mode is simply not possible: there are not enough Mode bits. One single Scalar Load operation may be used instead, followed by any arithmetic operation (including a simple mv) in "Splat" mode.

LD/ST Indexed

The modes for `RA+RB` indexed version are slightly different but are the same `RM.MODE` bits (19:23 of `RM`):

0	1	2	3 4	description
els	0	PI	zz SEA	post-increment and Fault-First
VLi	1	inv	CR-bit	Data-Dependent ffirst CR sel

Vector Indexed Strided Mode is qualified as follows:

```
if els and !RA.isvec and !RB.isvec:
    svctx.ldstmode = elementstride
```

A summary of the effect of Vectorization of src or dest:

```
imm(RA)  RT.v   RA.v   no stride allowed
imm(RA)  RT.s   RA.v   no stride allowed
imm(RA)  RT.v   RA.s   stride-select allowed
imm(RA)  RT.s   RA.s   not vectorized
RA,RB    RT.v  {RA|RB}.v Standard Indexed
RA,RB    RT.s  {RA|RB}.v Indexed but single LD (no VSPLAT)
RA,RB    RT.v  {RA&RB}.s VSPLAT possible. stride selectable
RA,RB    RT.s  {RA&RB}.s not vectorized (scalar identity)
```

Signed Effective Address computation is only relevant for Vector Indexed Mode, when elwidth overrides are applied. The source override applies to RB, and before adding to RA in order to calculate the Effective Address, if SEA is set then RB is sign-extended from elwidth bits to the full 64 bits. For other Modes (ffirst), all EA computation with elwidth overrides is unsigned. RA is *never* altered (not truncated) by element-width overrides.

Note that cache-inhibited LD/ST when VSPLAT is activated will perform **multiple** LD/ST operations, sequentially. Even with scalar src a Cache-inhibited LD will read the same memory location *multiple times*, storing the result in successive Vector destination registers. This because the cache-inhibit instructions are typically used to read and write memory-mapped peripherals. If a genuine cache-inhibited LD-VSPLAT is required then a single *scalar* cache-inhibited LD should be performed, followed by a VSPLAT-augmented mv, copying the one *scalar* value into multiple register destinations.

Note also that cache-inhibited VSPLAT with Data-Dependent Fail-First is possible. This allows for example to issue a massive batch of memory-mapped peripheral reads, stopping at the first NULL-terminated character and truncating VL to that point. No branch is needed to issue that large burst of LDs, which may be valuable in Embedded scenarios.

Vectorization of Scalar Power ISA v3.0B </>

Scalar Power ISA Load/Store operations may be seen from [[isa/fixedload]] and [[isa/fixedstore]] pseudocode to be of the form:

```
lbux RT, RA, RB
EA <- (RA) + (RB)
RT <- MEM(EA)
```

and for immediate variants:

```
lb RT,D(RA)
EA <- RA + EXTS(D)
RT <- MEM(EA)
```

Thus in the first example, the source registers may each be independently marked as scalar or vector, and likewise the destination; in the second example only the one source and one dest may be marked as scalar or vector.

Thus we can see that Vector Indexed may be covered, and, as demonstrated with the pseudocode below, the immediate can be used to give unit stride or element stride. With there being no way to tell which from the Power v3.0B Scalar opcode alone, the choice is provided instead by the SV Context.

```
# LD not VLD!  format - ldop RT, immed(RA)
# op_width: lb=1, lh=2, lw=4, ld=8
op_load(RT, RA, op_width, immed, svctx, RAupdate):
  ps = get_pred_val(FALSE, RA); # predication on src
  pd = get_pred_val(FALSE, RT); # ... AND on dest
  for (i=0, j=0, u=0; i < VL && j < VL;):
    # skip nonpredicates elements
    if (RA.isvec) while (!(ps & 1<<i)) i++;
    if (RAupdate.isvec) while (!(ps & 1<<u)) u++;
    if (RT.isvec) while (!(pd & 1<<j)) j++;
    if postinc:
        offs = 0; # added afterwards
        if RA.isvec: srcbase = ireg[RA+i]
        else         srcbase = ireg[RA]
    elif svctx.ldstmode == elementstride:
      # element stride mode
      srcbase = ireg[RA]
      offs = i * immed              # j*immed for a ST
    elif svctx.ldstmode == unitstride:
      # unit stride mode
      srcbase = ireg[RA]
      offs = immed + (i * op_width) # j*op_width for ST
    elif RA.isvec:
      # quirky Vector indexed mode but with an immediate
      srcbase = ireg[RA+i]
      offs = immed;
    else
      # standard scalar mode (but predicated)
      # no stride multiplier means VSPLAT mode
      srcbase = ireg[RA]
      offs = immed

    # compute EA
    EA = srcbase + offs
    # load from memory
    ireg[RT+j] <= MEM[EA];
    # check post-increment of EA
    if postinc: EA = srcbase + immed;
    # update RA?
    if RAupdate: ireg[RAupdate+u] = EA;
    if (!RT.isvec)
        break # destination scalar, end now
    if (RA.isvec) i++;
    if (RAupdate.isvec) u++;
    if (RT.isvec) j++;
```

Indexed LD is:

```
# format: ldop RT, RA, RB
function op_ldx(RT, RA, RB, RAupdate=False) # LD not VLD!
  ps = get_pred_val(FALSE, RA); # predication on src
  pd = get_pred_val(FALSE, RT); # ... AND on dest
  for (i=0, j=0, k=0, u=0; i < VL && j < VL && k < VL):
    # skip nonpredicated RA, RB and RT
    if (RA.isvec) while (!(ps & 1<<i)) i++;
    if (RAupdate.isvec) while (!(ps & 1<<u)) u++;
    if (RB.isvec) while (!(ps & 1<<k)) k++;
    if (RT.isvec) while (!(pd & 1<<j)) j++;
    if svctx.ldstmode == elementstride:
```

```
        EA = ireg[RA] + ireg[RB]*j   # register-strided
    else
        EA = ireg[RA+i] + ireg[RB+k] # indexed address
    if RAupdate: ireg[RAupdate+u] = EA
    ireg[RT+j] <= MEM[EA];
    if (!RT.isvec)
        break # destination scalar, end immediately
    if (RA.isvec) i++;
    if (RAupdate.isvec) u++;
    if (RB.isvec) k++;
    if (RT.isvec) j++;
```

Note that Element-Strided uses the Destination Step because with both sources being Scalar as a prerequisite condition of activation of Element-Stride Mode, the source step (being Scalar) would never advance.

Note in both cases that {SVP64 Chapter} allows RA-as-a-dest in "update" mode (`ldux`) to be effectively a *completely different* register from RA-as-a-source. This because there is room in svp64 to extend RA-as-src as well as RA-as-dest, both independently as scalar or vector *and* independently extending their range.

*Programmer's note: being able to set RA-as-a-source as separate from RA-as-a-destination as Scalar is **extremely valuable** once it is remembered that Simple-V element operations must be in Program Order, especially in loops, for saving on multiple address computations. Care does have to be taken however that RA-as-src is not overwritten by RA-as-dest unless intentionally desired, especially in element-strided Mode.*

LD/ST Indexed vs Indexed REMAP </>

Unfortunately the word "Indexed" is used twice in completely different contexts, potentially causing confusion.

- There has existed instructions in the Power ISA `ld RT,RA,RB` since its creation: these are called "LD/ST Indexed" instructions and their name and meaning is well-established.
- There now exists, in Simple-V, a {REMAP subsystem} mode called "Indexed" Mode that can be applied to *any* instruction **including those named LD/ST Indexed**.

Whilst it may be costly in terms of register reads to allow REMAP Indexed Mode to be applied to any Vectorized LD/ST Indexed operation such as `sv.ld *RT,RA,*RB`, or even misleadingly labelled as redundant, firstly the strict application of the RISC Paradigm that Simple-V follows makes it awkward to consider *preventing* the application of Indexed REMAP to such operations, and secondly they are not actually the same at all.

Indexed REMAP, as applied to RB in the instruction `sv.ld *RT,RA,*RB` effectively performs an *in-place* reordering of the offsets, RB. To achieve the same effect without Indexed REMAP would require taking a *copy* of the Vector of offsets starting at RB, manually explicitly reordering them, and finally using the copy of reordered offsets in a non-REMAP'ed `sv.ld`. Using non-strided LD as an example, pseudocode showing what actually occurs, where the pseudocode for `indexed_remap` may be found in {REMAP subsystem}:

```
# sv.ld *RT,RA,*RB with Index REMAP applied to RB
for i in 0..VL-1:
    if remap.indexed:
        rb_idx = indexed_remap(i) # remap
    else:
        rb_idx = i # use the index as-is
    EA = GPR(RA) + GPR(RB+rb_idx)
    GPR(RT+i) = MEM(EA, 8)
```

Thus it can be seen that the use of Indexed REMAP saves copying and manual reordering of the Vector of RB offsets.

LD/ST ffirst (Fault-First) </>

LD/ST ffirst treats the first LD/ST in a vector (element 0 if REMAP is not active and predication is not applied) as an ordinary one, with all behaviour with respect to Interrupts Exceptions Page Faults Memory Management being identical in every regard to Scalar v3.0 Power ISA LD/ST. However for elements 1 and above, if an exception would occur, then VL is **truncated** to the previous element: the exception is **not** then raised because the LD/ST that would otherwise have caused an exception is *required* to be cancelled. Additionally an implementor may choose to truncate VL for any arbitrary reason *except for the very first*.

ffirst LD/ST to multiple pages via a Vectorized Index base is considered a security risk due to the abuse of probing multiple pages in rapid succession and getting speculative feedback on which pages would fail. Therefore Vector Indexed LD/ST is prohibited entirely, and the Mode bit instead used for element-strided LD/ST.

```
for(i = 0; i < VL; i++)
    reg[rt + i] = mem[reg[ra] + i * reg[rb]];
```

High security implementations where any kind of speculative probing of memory pages is considered a risk should take advantage of the fact that implementations may truncate VL at any point, without requiring software to be rewritten and made non-portable. Such implementations may choose to *always* set VL=1 which will have the effect of terminating any speculative probing (and also adversely affect performance), but will at least not require applications to be rewritten.

Low-performance simpler hardware implementations may also choose (always) to also set VL=1 as the bare minimum compliant implementation of LD/ST Fail-First. It is however critically important to remember that the first element LD/ST **MUST** be treated as an ordinary LD/ST, i.e. **MUST** raise exceptions exactly like an ordinary LD/ST.

For ffirst LD/STs, VL may be truncated arbitrarily to a nonzero value for any implementation-specific reason. For example: it is perfectly reasonable for implementations to alter VL when ffirst LD or ST operations are initiated on a nonaligned boundary, such that within a loop the subsequent iteration of that loop begins the following ffirst LD/ST operations on an aligned boundary such as the beginning of a cache line, or beginning of a Virtual Memory page. Likewise, to reduce workloads or balance resources.

When Predication is used, the "first" element is considered to be the first non-predicated element rather than specifically `srcstep=0`.

Vertical-First Mode is slightly strange in that only one element at a time is ever executed anyway. Given that programmers may legitimately choose to alter srcstep and dststep in non-sequential order as part of explicit loops, it is neither possible nor safe to make speculative assumptions about future LD/STs. Therefore, Fail-First LD/ST in Vertical-First is `UNDEFINED`. This is very different from Arithmetic (Data-dependent) FFirst where Vertical-First Mode is fully deterministic, not speculative.

Data-Dependent Fail-First (not Fail/Fault-First) </>

Not to be confused with Fail/Fault First, Data-Fail-First performs an additional check on the data, and if the test fails then VL is truncated and further looping terminates. This is precisely the same as Arithmetic Data-Dependent Fail-First, the only difference being that the result comes from the LD/ST rather than from an Arithmetic operation.

Also a crucial difference between Arithmetic and LD/ST Data-Dependent Fail-First: except for Store-Conditional a 4-bit Condition Register Field test is created for testing purposes *but not stored* (thus there is no RC1 Mode as there is in Arithmetic). The reason why a CR Field is not stored is because Load/Store, particularly the Update instructions, is already expensive in register terms, and adding an extra Vector write would be too costly in hardware.

Programmer's note: Programmers may use Data-Dependent Load with a test to truncate VL, and may then follow up with a `sv.cmpi` or other operation. The important aspect is that the Vector Load truncated on finding a NULL pointer, for example.

Programmer's note: Load-with-Update may be used to update the register used in Effective Address computation of th next element. This may be used to perform single-linked-list walking, where Data-Dependent Fail-First terminates and truncates the Vector at the first NULL.

Load/Store Data-Dependent Fail-First, VLi=0

In the case of Store operations there is a quirk when VLi (VL inclusive is "Valid") is clear. Bear in mind the criteria is that the truncated Vector of results, when VLi is clear, must all pass the "test", but when VLi is set the *current failed test* is permitted to be included. Thus, the actual update (store) to Memory is **not permitted to take place** should the test fail.

Additionally in any Load/Store with Update instruction, when VLi=0 and a test fails then RA does **not** receive a copy of the Effective Address. Hardware implementations with Out-of-Order Micro-Architectures should use speculative Shadow-Hold and Cancellation (or other Transactional Rollback mechanism) when the test fails.

- **Load, VLi=0**: perform the Memory Load, do not put the result into the regfile yet (or EA into RA). Test the Loaded data: if fail do not store the Load in the register file (or EA into RA). Otherwise proceed with updating regfiles. VL is truncated to "only elements that passed the test"
- **Store, VLi=0**: even before the Store takes place, perform the test on the data to *be* stored. If fail do not proceed with the Store at all. VL is truncated to "only elements that passed the test"

Load/Store Data-Dependent Fail-First, VLi=1

By contrast if VLi=1 and the test fails, the Store may proceed *and then* looping terminates. In this way, when Inclusive the Vector of Truncated results contains the first-failed data (including RA on Updates)

- **Load, VLi=1**: perform the Memory Load, complete it in full (including EA into RA). Test the Loaded data: if fail then VL is truncated to "elements tested".
- **Store, VLi=0**: same as Load. Perform the Store in full and after-the-fact carry out the test of the original data requested to be stored. If fail then VL is truncated to "elements tested".

Below is an example of loading the starting addresses of Linked-List nodes. If VLi=1 it will load the NULL pointer into the Vector of results. If however VLi=0 it will *exclude* the NULL pointer by truncating VL to one Element earlier (only loading non-NULL data into registers).

Programmer's Note: by also setting the RC1 qualifier as well as setting VLi=1 it is possible to establish a Predicate Mask such that the first zero in the predicate will be the NULL pointer

```
RT=1 # vec - deliberately overlaps by one with RA
RA=0 # vec - first one is valid, contains ptr
imm = 8 # offset_of(ptr->next)
for i in range(VL):
    # this part is the Scalar Defined Word-instruction (standard scalar ld operation)
    EA = GPR(RA+i) + imm          # ptr + offset(next)
```

```
data = MEM(EA, 8)             # 64-bit address of ptr->next
# was a normal vector-ld up to this point. now the Data-Fail-First
cr_test = conditions(data)
if Rc=1 or RC1: CR.field(i) = cr_test # only store if Rc=1/RC1
action_load = True
if cr_test.EQ == testbit:             # check if zero
    if VLI then
       VL = i+1            # update VL, inclusive
    else
       VL = i              # update VL, exclusive current
       action_load = False # current load excluded
    stop = True            # stop looping
if action_load:
   GPR(RT+i) = data        # happens to be read on next loop!
if stop: break
```

Data-Dependent Fail-First on Store-Conditional (Rc=1)

There are very few instructions that allow Rc=1 for Load/Store: one of those is the `stdcx.` and other Atomic Store-Conditional instructions. With Simple-V being a loop around Scalar instructions strictly obeying Scalar Program Order a Horizontal-First Fail-First loop on an Atomic Store-Conditional will always fail the second and all other Store-Conditional instructions because Load-Reservation and Store-Conditional are required to be executed in pairs.

By contrast, in Vertical-First Mode it is in fact possible to issue the pairs, and consequently allowing Vectorized Data-Dependent Fail-First is useful.

Programmer's note: Care should be taken when VL is truncated in Vertical-First Mode.

Future potential

Although Rc=1 on LD/ST is a rare occurrence at present, future versions of Power ISA *might* conceivably have Rc=1 LD/ST Scalar instructions, and with the SVP64 Vectorization Prefixing being itself a RISC-paradigm that is itself fully-independent of the Scalar Suffix Defined Word-instructions, prohibiting the possibility of Rc=1 Data-Dependent Mode on future potential LD/ST operations is not strategically sound.

LOAD/STORE Elwidths </>

Loads and Stores are almost unique in that the Power Scalar ISA provides a width for the operation (lb, lh, lw, ld). Only `extsb` and others like it provide an explicit operation width. There are therefore *three* widths involved:

- operation width (lb=8, lh=16, lw=32, ld=64)
- src element width override (8/16/32/default)
- destination element width override (8/16/32/default)

Some care is therefore needed to express and make clear the transformations, which are expressly in this order:

- Calculate the Effective Address from RA at full width but (on Indexed Load) allow srcwidth overrides on RB
- Load at the operation width (lb/lh/lw/ld) as usual
- byte-reversal as usual
- zero-extension or truncation from operation width to dest elwidth
- place result in destination at dest elwidth

In order to respect Power v3.0B Scalar behaviour the memory side is treated effectively as completely separate and distinct from SV augmentation. This is primarily down to quirks surrounding LE/BE and byte-reversal.

It is rather unfortunately possible to request an elwidth override on the memory side which does not mesh with the overridden operation width: these result in `UNDEFINED` behaviour. The reason is that the effect of attempting a 64-bit `sv.ld` operation with a source elwidth override of 8/16/32 would result in overlapping memory requests, particularly on unit and element strided operations. Thus it is `UNDEFINED` when the elwidth is smaller than the memory operation width. Examples include `sv.lw/sw=16/els` which requests (overlapping) 4-byte memory reads offset from each other at 2-byte intervals. Store likewise is also `UNDEFINED` where the dest elwidth override is less than the operation width.

Note the following regarding the pseudocode to follow:

- `scalar identity behaviour` SV Context parameter conditions turn this into a straight absolute fully-compliant Scalar v3.0B LD operation
- `brev` selects whether the operation is the byte-reversed variant (`ldbrx` rather than `ld`)
- `op_width` specifies the operation width (`lb`, `lh`, `lw`, `ld`) as a "normal" part of Scalar v3.0B LD
- `imm_offs` specifies the immediate offset `ld r3, imm_offs(r5)`, again as a "normal" part of Scalar v3.0B LD
- `svctx` specifies the SV Context and includes VL as well as source and destination elwidth overrides.

Below is the pseudocode for Unit-Strided LD (which includes Vector capability). Observe in particular that RA, as the base address in both Immediate and Indexed LD/ST, does not have element-width overriding applied to it.

Note that predication, predication-zeroing, and other modes have all been removed, for clarity and simplicity:

```
# LD not VLD!
# this covers unit stride mode and a type of vector offset
function op_ld(RT, RA, op_width, imm_offs, svctx)
  for (int i = 0, int j = 0; i < svctx.VL && j < svctx.VL):
    if not svctx.unit/el-strided:
        # strange vector mode, compute 64 bit address which is
        # not polymorphic! elwidth hardcoded to 64 here
        srcbase = get_polymorphed_reg(RA, 64, i)
    else:
        # unit / element stride mode, compute 64 bit address
        srcbase = get_polymorphed_reg(RA, 64, 0)
        # adjust for unit/el-stride
        srcbase += .... uses op_width here

    # read the underlying memory
    memread <= MEM(srcbase + imm_offs, op_width)

    # truncate/extend to over-ridden dest width.
    memread = adjust_wid(memread, op_width, svctx.elwidth)

    # takes care of inserting memory-read (now correctly byteswapped)
    # into regfile underlying LE-defined order, into the right place
    # using Element-Packing starting at register RT, respecting destination
    # element bitwidth, and the element index (j)
    set_polymorphed_reg(RT, svctx.elwidth, j, memread)

    # increments both src and dest element indices (no predication here)
    i++;
    j++;
```

Note above that the source elwidth is *not used at all* in LD-immediate: RA never has elwidth overrides, leaving the elwidth free for truncation/extension of the result.

For LD/Indexed, the key is that in the calculation of the Effective Address, RA has no elwidth override but RB does. Pseudocode below is simplified for clarity: predication and all modes are removed:

```
# LD not VLD! ld*rx if brev else ld*
function op_ld(RT, RA, RB, op_width, svctx, brev)
  for (int i = 0, int j = 0; i < svctx.VL && j < svctx.VL):
    if not svctx.el-strided:
        # RA not polymorphic! elwidth hardcoded to 64 here
        srcbase = get_polymorphed_reg(RA, 64, i)
    else:
        # element stride mode, again RA not polymorphic
        srcbase = get_polymorphed_reg(RA, 64, 0)
    # RB *is* polymorphic
    offs = get_polymorphed_reg(RB, svctx.src_elwidth, i)
    # sign-extend
    if svctx.SEA: offs = sext(offs, svctx.src_elwidth, 64)

    # takes care of (merges) processor LE/BE and ld/ldbrx
    bytereverse = brev XNOR MSR.LE

    # read the underlying memory
    memread <= MEM(srcbase + offs, op_width)

    # optionally performs byteswap at op width
    if (bytereverse):
        memread = byteswap(memread, op_width)

    # truncate/extend to over-ridden dest width.
    dest_width = op_width if RT.isvec else 64
    memread = adjust_wid(memread, op_width, dest_width)

    # takes care of inserting memory-read (now correctly byteswapped)
    # into regfile underlying LE-defined order, into the right place
    # within the NEON-like register, respecting destination element
    # bitwidth, and the element index (j)
    set_polymorphed_reg(RT, destwidth, j, memread)

    # increments both src and dest element indices (no predication here)
    i++;
    j++;
```

Programmer's note: with no destination elwidth override the destination width must be implicitly ascertained. The assumption is that if the destination is a Scalar that the entire 64-bit register must be written, thus the width is extended to 64-bit. If however the destination is a Vector then it is deemed appropriate to use the LD/ST width and to perform contiguous register element packing at that width. The justification for doing so is that if further sign-extension or saturation is required after a LD, these may be performed by a follow-up instruction that uses a source elwidth override matching the exact width of the LD operation. Correspondingly for a ST a destination elwidth override on a prior instruction may match the exact width of the ST instruction.

Remapped LD/ST </>

In the {REMAP subsystem} page the concept of "Remapping" is described. Whilst it is expensive to set up (2 64-bit opcodes minimum) it provides a way to arbitrarily perform 1D, 2D and 3D "remapping" of up to 64 elements worth of LDs or STs. The usual interest in such re-mapping is for example in separating out 24-bit RGB channel data into separate contiguous registers. NEON covers this as shown in the diagram below:

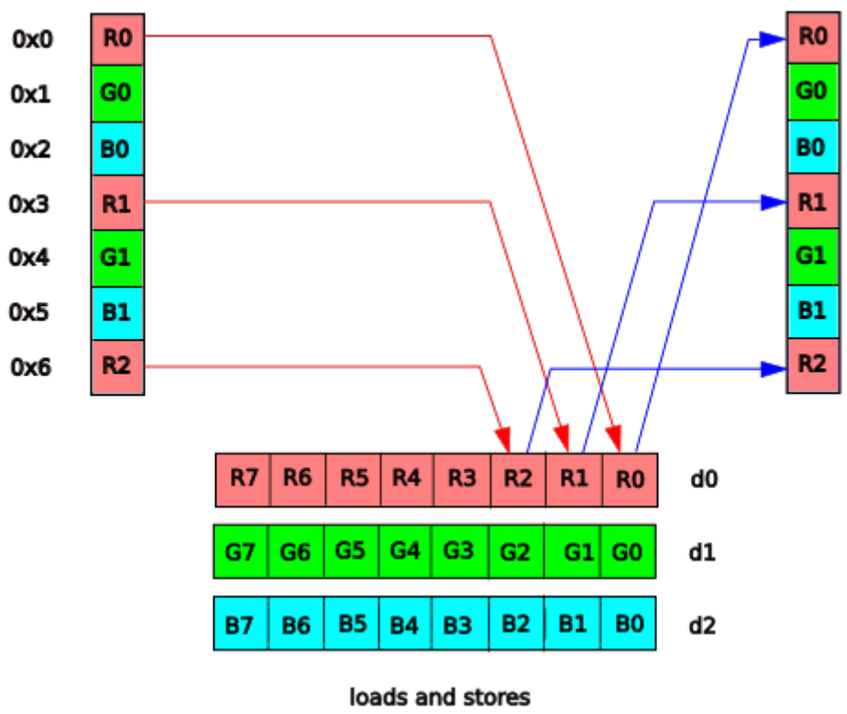

Figure 1: Load/Store remap

REMAP easily covers this capability, and with dest elwidth overrides and saturation may do so with built-in conversion that would normally require additional width-extension, sign-extension and min/max Vectorized instructions as post-processing stages.

Thus we do not need to provide specialist LD/ST "Structure Packed" opcodes because the generic abstracted concept of "Remapping", when applied to LD/ST, will give that same capability, with far more flexibility.

It is worth noting that Pack/Unpack Modes of SVSTATE, which may be established through `svstep`, are also an easy way to perform regular Structure Packing, at the vec2/vec3/vec4 granularity level. Beyond that, REMAP will need to be used.

Parallel Reduction REMAP

No REMAP Schedule is prohibited in SVP64 because the RISC-paradigm Prefix is completely separate from the RISC-paradigm Scalar Defined Word-instructions. Although obscure there does exist the outside possibility that a potential use for Parallel Reduction Schedules on LD/ST would find a use in Computer Science. Readers are invited to contact the authors of this document if one is ever found.

[[!tag standards]]

SVP64 Branch Conditional behaviour </>

Please note: although similar, SVP64 Branch instructions should be considered completely separate and distinct from standard scalar OpenPOWER-approved v3.0B branches. **v3.0B branches are in no way impacted, altered, changed or modified in any way, shape or form by the SVP64 Vectorized Variants**.

It is also extremely important to note that Branches are the sole pseudo-exception in SVP64 to `Scalar Identity Behaviour`. SVP64 Branches contain additional modes that are useful for scalar operations (i.e. even when VL=1 or when using single-bit predication).

Rationale </>

Scalar 3.0B Branch Conditional operations, `bc`, `bctar` etc. test a Condition Register. However for parallel processing it is simply impossible to perform multiple independent branches: the Program Counter simply cannot branch to multiple destinations based on multiple conditions. The best that can be done is to test multiple Conditions and make a decision of a *single* branch, based on analysis of a *Vector* of CR Fields which have just been calculated from a *Vector* of results.

In 3D Shader binaries, which are inherently parallelised and predicated, testing all or some results and branching based on multiple tests is extremely common, and a fundamental part of Shader Compilers. Example: without such multi-condition test-and-branch, if a predicate mask is all zeros a large batch of instructions may be masked out to `nop`, and it would waste CPU cycles to run them. 3D GPU ISAs can test for this scenario and, with the appropriate predicate-analysis instruction, jump over fully-masked-out operations, by spotting that *all* Conditions are false.

Unless Branches are aware and capable of such analysis, additional instructions would be required which perform Horizontal Cumulative analysis of Vectorized Condition Register Fields, in order to reduce the Vector of CR Fields down to one single yes or no decision that a Scalar-only v3.0B Branch-Conditional could cope with. Such instructions would be unavoidable, required, and costly by comparison to a single Vector-aware Branch. Therefore, in order to be commercially competitive, `sv.bc` and other Vector-aware Branch Conditional instructions are a high priority for 3D GPU (and OpenCL-style) workloads.

Given that Power ISA v3.0B is already quite powerful, particularly the Condition Registers and their interaction with Branches, there are opportunities to create extremely flexible and compact Vectorized Branch behaviour. In addition, the side-effects (updating of CTR, truncation of VL, described below) make it a useful instruction even if the branch points to the next instruction (no actual branch).

Overview </>

When considering an "array" of branch-tests, there are four primarily-useful modes: AND, OR, NAND and NOR of all Conditions. NAND and NOR may be synthesised from AND and OR by inverting `BO[1]` which just leaves two modes:

- Branch takes place on the **first** CR Field test to succeed (a Great Big OR of all condition tests). Exit occurs on the first **successful** test.
- Branch takes place only if **all** CR field tests succeed: a Great Big AND of all condition tests. Exit occurs on the first **failed** test.

Early-exit is enacted such that the Vectorized Branch does not perform needless extra tests, which will help reduce reads on the Condition Register file.

*Note: Early-exit is **MANDATORY** (required) behaviour. Branches **MUST** exit at the first sequentially-encountered failure point, for exactly the same reasons for which it is mandatory in programming languages doing early-exit: to avoid damaging side-effects and to provide deterministic behaviour. Speculative testing of Condition Register Fields is permitted, as is speculative calculation of CTR, as long as, as usual in any Out-of-Order microarchitecture, that speculative testing is cancelled should an early-exit occur. i.e. the speculation must be "precise": Program Order must be preserved*

Also note that when early-exit occurs in Horizontal-first Mode, srcstep, dststep etc. are all reset, ready to begin looping from the beginning for the next instruction. However for Vertical-first Mode srcstep etc. are incremented "as usual" i.e. an early-exit has no special impact, regardless of whether the branch occurred or not. This can leave srcstep etc. in what may be considered an unusual state on exit from a loop and it is up to the programmer to reset srcstep, dststep etc. to known-good values *(easily achieved with `setvl`)*.

Additional useful behaviour involves two primary Modes (both of which may be enabled and combined):

- **VLSET Mode**: identical to Data-Dependent Fail-First Mode for Arithmetic SVP64 operations, with more flexibility and a close interaction and integration into the underlying base Scalar v3.0B Branch instruction. Truncation of VL takes place around the early-exit point.
- **CTR-test Mode**: gives much more flexibility over when and why CTR is decremented, including options to decrement if a Condition test succeeds *or if it fails*.

With these side-effects, basic Boolean Logic Analysis advises that it is important to provide a means to enact them each based on whether testing succeeds *or fails*. This results in a not-insignificant number of additional Mode Augmentation bits, accompanying VLSET and CTR-test Modes respectively.

Predicate skipping or zeroing may, as usual with SVP64, be controlled by `sz`. Where the predicate is masked out and zeroing is enabled, then in such circumstances the same Boolean Logic Analysis dictates that rather

than testing only against zero, the option to test against one is also prudent. This introduces a new immediate field, SNZ, which works in conjunction with sz.

Vectorized Branches can be used in either SVP64 Horizontal-First or Vertical-First Mode. Essentially, at an element level, the behaviour is identical in both Modes, although the ALL bit is meaningless in Vertical-First Mode.

It is also important to bear in mind that, fundamentally, Vectorized Branch-Conditional is still extremely close to the Scalar v3.0B Branch-Conditional instructions, and that the same v3.0B Scalar Branch-Conditional instructions are still *completely separate and independent*, being unaltered and unaffected by their SVP64 variants in every conceivable way.

*Programming note: One important point is that SVP64 instructions are 64 bit. (8 bytes not 4). This needs to be taken into consideration when computing branch offsets: the offset is relative to the start of the instruction, which **includes** the SVP64 Prefix*

Format and fields </>

With element-width overrides being meaningless for Condition Register Fields, bits 4 thru 7 of SVP64 RM may be used for additional Mode bits.

SVP64 RM MODE (includes repurposing ELWIDTH bits 4:5, and ELWIDTH_SRC bits 6-7 for *alternate* uses) for Branch Conditional:

4	5	6	7	17	18	19	20	21	22 23	description
ALL	SNZ	/	/	SL	SLu	0	0	/	LRu sz	simple mode
ALL	SNZ	/	VSb	SL	SLu	0	1	VLI	LRu sz	VLSET mode
ALL	SNZ	CTi	/	SL	SLu	1	0	/	LRu sz	CTR-test mode
ALL	SNZ	CTi	VSb	SL	SLu	1	1	VLI	LRu sz	CTR-test+VLSET mode

Brief description of fields:

- **sz=1** if predication is enabled and sz=1 and a predicate element bit is zero, SNZ will be substituted in place of the CR bit selected by BI, as the Condition tested. Contrast this with normal SVP64 sz=1 behaviour, where *only* a zero is put in place of masked-out predicate bits.
- **sz=0** When sz=0 skipping occurs as usual on masked-out elements, but unlike all other SVP64 behaviour which entirely skips an element with no related side-effects at all, there are certain special circumstances where CTR may be decremented. See CTR-test Mode, below.
- **ALL** when set, all branch conditional tests must pass in order for the branch to succeed. When clear, it is the first sequentially encountered successful test that causes the branch to succeed. This is identical behaviour to how programming languages perform early-exit on Boolean Logic chains.
- **VLI** VLSET is identical to Data-dependent Fail-First mode. In VLSET mode, VL *may* (depending on VSb) be truncated. If VLI (Vector Length Inclusive) is clear, VL is truncated to *exclude* the current element, otherwise it is included. SVSTATE.MVL is not altered: only VL.
- **SL** identical to LR except applicable to SVSTATE. If SL is set, SVSTATE is transferred to SVLR (conditionally on whether SLu is set).
- **SLu**: SVSTATE Link Update, like LRu except applies to SVSTATE.
- **LRu**: Link Register Update, used in conjunction with LK=1 to make LR update conditional
- **VSb** In VLSET Mode, after testing, if VSb is set, VL is truncated if the test succeeds. If VSb is clear, VL is truncated if a test *fails*. Masked-out (skipped) bits are not considered part of testing when sz=0
- **CTi** CTR inversion. CTR-test Mode normally decrements per element tested. CTR inversion decrements if a test *fails*. Only relevant in CTR-test Mode.

LRu and CTR-test modes are where SVP64 Branches subtly differ from Scalar v3.0B Branches. sv.bcl for example will always update LR, whereas sv.bcl/lru will only update LR if the branch succeeds.

Of special interest is that when using ALL Mode (Great Big AND of all Condition Tests), if VL=0, which is rare but can occur in Data-Dependent Modes, the Branch will always take place because there will be no failing Condition Tests to prevent it. Likewise when not using ALL Mode (Great Big OR of all Condition Tests) and VL=0 the Branch is guaranteed not to occur because there will be no *successful* Condition Tests to make it happen.

Vectorized CR Field numbering, and Scalar behaviour </>

It is important to keep in mind that just like all SVP64 instructions, the BI field of the base v3.0B Branch Conditional instruction may be extended by SVP64 EXTRA augmentation, as well as be marked as either Scalar or Vector. It is also crucially important to keep in mind that for CRs, SVP64 sequentially increments the CR *Field* numbers. CR *Fields* are treated as elements, not bit-numbers of the CR *register*.

The BI operand of Branch Conditional operations is five bits, in scalar v3.0B this would select one bit of the 32 bit CR, comprising eight CR Fields of 4 bits each. In SVP64 there are 16 32 bit CRs, containing 128 4-bit CR Fields. Therefore, the 2 LSBs of BI select the bit from the CR Field (EQ LT GT SO), and the top 3 bits are extended to either scalar or vector and to select CR Fields 0..127 as specified in SVP64 {SVP64 Appendix}.

When the CR Fields selected by SVP64-Augmented BI is marked as scalar, then as the usual SVP64 rules apply: the Vector loop ends at the first element tested (the first CR *Field*), after taking predication into consideration.

Thus, also as usual, when a predicate mask is given, and BI marked as scalar, and sz is zero, srcstep skips forward to the first non-zero predicated element, and only that one element is tested.

In other words, the fact that this is a Branch Operation (instead of an arithmetic one) does not result, ultimately, in significant changes as to how SVP64 is fundamentally applied, except with respect to:

- the unique properties associated with conditionally changing the Program Counter (aka "a Branch"), resulting in early-out opportunities
- CTR-testing

Both are outlined below, in later sections.

Horizontal-First and Vertical-First Modes </>

In SVP64 Horizontal-First Mode, the first failure in ALL mode (Great Big AND) results in early exit: no more updates to CTR occur (if requested); no branch occurs, and LR is not updated (if requested). Likewise for non-ALL mode (Great Big Or) on first success early exit also occurs, however this time with the Branch proceeding. In both cases the testing of the Vector of CRs should be done in linear sequential order (or in REMAP re-sequenced order): such that tests that are sequentially beyond the exit point are *not* carried out. (*Note: it is standard practice in Programming languages to exit early from conditional tests, however a little unusual to consider in an ISA that is designed for Parallel Vector Processing. The reason is to have strictly-defined guaranteed behaviour*)

In Vertical-First Mode, setting the ALL bit results in UNDEFINED behaviour. Given that only one element is being tested at a time in Vertical-First Mode, a test designed to be done on multiple bits is meaningless.

Description and Modes </>

Predication in both INT and CR modes may be applied to sv.bc and other SVP64 Branch Conditional operations, exactly as they may be applied to other SVP64 operations. When sz is zero, any masked-out Branch-element operations are not included in condition testing, exactly like all other SVP64 operations, *including* side-effects such as potentially updating LR or CTR, which will also be skipped. There is *one* exception here, which is when BO[2]=0, sz=0, CTR-test=0, CTi=1 and the relevant element predicate mask bit is also zero: under these special circumstances CTR will also decrement.

When sz is non-zero, this normally requests insertion of a zero in place of the input data, when the relevant predicate mask bit is zero. This would mean that a zero is inserted in place of CR[BI+32] for testing against BO, which may not be desirable in all circumstances. Therefore, an extra field is provided SNZ, which, if set, will insert a **one** in place of a masked-out element, instead of a zero.

(*Note: Both options are provided because it is useful to deliberately cause the Branch-Conditional Vector testing to fail at a specific point, controlled by the Predicate mask. This is particularly useful in VLSET mode, which will truncate SVSTATE.VL at the point of the first failed test.*)

Normally, CTR mode will decrement once per Condition Test, resulting under normal circumstances that CTR reduces by up to VL in Horizontal-First Mode. Just as when v3.0B Branch-Conditional saves at least one instruction on tight inner loops through auto-decrementation of CTR, likewise it is also possible to save instruction count for SVP64 loops in both Vertical-First and Horizontal-First Mode, particularly in circumstances where there is conditional interaction between the element computation and testing, and the continuation (or otherwise) of a given loop. The potential combinations of interactions is why CTR testing options have been added.

Also, the unconditional bit BO[0] is still relevant when Predication is applied to the Branch because in ALL mode all nonmasked bits have to be tested, and when sz=0 skipping occurs. Even when VLSET mode is not used, CTR may still be decremented by the total number of nonmasked elements, acting in effect as either a popcount or cntlz depending on which mode bits are set. In short, Vectorized Branch becomes an extremely powerful tool.

Micro-Architectural Implementation Note: *when implemented on top of a Multi-Issue Out-of-Order Engine it is possible to pass a copy of the predicate and the prerequisite CR Fields to all Branch Units, as well as the current value of CTR at the time of multi-issue, and for each Branch Unit to compute how many times CTR would be subtracted, in a fully-deterministic and parallel fashion. A SIMD-based Branch Unit, receiving and processing multiple CR Fields covered by multiple predicate bits, would do the exact same thing. Obviously, however, if CTR is modified within any given loop (mtctr) the behaviour of CTR is no longer deterministic.*

Link Register Update </>

For a Scalar Branch, unconditional updating of the Link Register LR is useful and practical. However, if a loop of CR Fields is tested, unconditional updating of LR becomes problematic.

For example when using bclr with LRu=1,LK=0 in Horizontal-First Mode, LR's value will be unconditionally overwritten after the first element, such that for execution (testing) of the second element, LR has the value CIA+8. This is covered in the bclrl example, in a later section.

The addition of a LRu bit modifies behaviour in conjunction with LK, as follows:

- sv.bc When LRu=0,LK=0, Link Register is not updated
- sv.bcl When LRu=0,LK=1, Link Register is updated unconditionally
- sv.bcl/lru When LRu=1,LK=1, Link Register will only be updated if the Branch Condition fails.
- sv.bc/lru When LRu=1,LK=0, Link Register will only be updated if the Branch Condition succeeds.

This avoids destruction of LR during loops (particularly Vertical-First ones).

SVLR and SVSTATE

For precisely the reasons why `LK=1` was added originally to the Power ISA, with SVSTATE being a peer of the Program Counter it becomes necessary to also add an SVLR (SVSTATE Link Register) and corresponding control bits `SL` and `SLu`.

CTR-test </>

Where a standard Scalar v3.0B branch unconditionally decrements CTR when `BO[2]` is clear, CTR-test Mode introduces more flexibility which allows CTR to be used for many more types of Vector loops constructs.

CTR-test mode and CTi interaction is as follows: note that `BO[2]` is still required to be clear for CTR decrements to be considered, exactly as is the case in Scalar Power ISA v3.0B

- **CTR-test=0, CTi=0**: CTR decrements on a per-element basis if `BO[2]` is zero. Masked-out elements when `sz=0` are skipped (i.e. CTR is *not* decremented when the predicate bit is zero and `sz=0`).
- **CTR-test=0, CTi=1**: CTR decrements on a per-element basis if `BO[2]` is zero and a masked-out element is skipped (`sz=0` and predicate bit is zero). This one special case is the **opposite** of other combinations, as well as being completely different from normal SVP64 `sz=0` behaviour)
- **CTR-test=1, CTi=0**: CTR decrements on a per-element basis if `BO[2]` is zero and the Condition Test succeeds. Masked-out elements when `sz=0` are skipped (including not decrementing CTR)
- **CTR-test=1, CTi=1**: CTR decrements on a per-element basis if `BO[2]` is zero and the Condition Test *fails*. Masked-out elements when `sz=0` are skipped (including not decrementing CTR)

`CTR-test=0, CTi=1, sz=0` requires special emphasis because it is the only time in the entirety of SVP64 that has side-effects when a predicate mask bit is clear. **All** other SVP64 operations entirely skip an element when sz=0 and a predicate mask bit is zero. It is also critical to emphasise that in this unusual mode, no other side-effects occur: **only** CTR is decremented, i.e. the rest of the Branch operation is skipped.

VLSET Mode </>

VLSET Mode truncates the Vector Length so that subsequent instructions operate on a reduced Vector Length. This is similar to Data-dependent Fail-First and LD/ST Fail-First, where for VLSET the truncation occurs at the Branch decision-point.

Interestingly, due to the side-effects of `VLSET` mode it is actually useful to use Branch Conditional even to perform no actual branch operation, i.e to point to the instruction after the branch. Truncation of VL would thus conditionally occur yet control flow alteration would not.

`VLSET` mode with Vertical-First is particularly unusual. Vertical-First is designed to be used for explicit looping, where an explicit call to `svstep` is required to move both srcstep and dststep on to the next element, until VL (or other condition) is reached. Vertical-First Looping is expected (required) to terminate if the end of the Vector, VL, is reached. If however that loop is terminated early because VL is truncated, VLSET with Vertical-First becomes meaningless. Resolving this would require two branches: one Conditional, the other branching unconditionally to create the loop, where the Conditional one jumps over it.

Therefore, with `VSb`, the option to decide whether truncation should occur if the branch succeeds *or* if the branch condition fails allows for the flexibility required. This allows a Vertical-First Branch to *either* be used as a branch-back (loop) *or* as part of a conditional exit or function call from *inside* a loop, and for VLSET to be integrated into both types of decision-making.

In the case of a Vertical-First branch-back (loop), with `VSb=0` the branch takes place if success conditions are met, but on exit from that loop (branch condition fails), VL will be truncated. This is extremely useful.

`VLSET` mode with Horizontal-First when `VSb=0` is still useful, because it can be used to truncate VL to the first predicated (non-masked-out) element.

The truncation point for VL, when VLi is clear, must not include skipped elements that preceded the current element being tested. Example: `sz=0, VLi=0, predicate mask = 0b110010` and the Condition Register failure point is at CR Field element 4.

- Testing at element 0 is skipped because its predicate bit is zero
- Testing at element 1 passed
- Testing elements 2 and 3 are skipped because their respective predicate mask bits are zero
- Testing element 4 fails therefore VL is truncated to **2** not 4 due to elements 2 and 3 being skipped.

If `sz=1` in the above example *then* VL would have been set to 4 because in non-zeroing mode the zero'd elements are still effectively part of the Vector (with their respective elements set to `SNZ`)

If `VLI=1` then VL would be set to 5 regardless of sz, due to being inclusive of the element actually being tested.

VLSET and CTR-test combined </>

If both CTR-test and VLSET Modes are requested, it is important to observe the correct order. What occurs depends on whether VLi is enabled, because VLi affects the length, VL.

If VLi (VL truncate inclusive) is set:

1. compute the test including whether CTR triggers

2. (optionally) decrement CTR
3. (optionally) truncate VL (VSb inverts the decision)
4. decide (based on step 1) whether to terminate looping (including not executing step 5)
5. decide whether to branch.

If VLi is clear, then when a test fails that element and any following it should **not** be considered part of the Vector. Consequently:

1. compute the branch test including whether CTR triggers
2. if the test fails against VSb, truncate VL to the *previous* element, and terminate looping. No further steps executed.
3. (optionally) decrement CTR
4. decide whether to branch.

Boolean Logic combinations </>

In a Scalar ISA, Branch-Conditional testing even of vector results may be performed through inversion of tests. NOR of all tests may be performed by inversion of the scalar condition and branching *out* from the scalar loop around elements, using scalar operations.

In a parallel (Vector) ISA it is the ISA itself which must perform the prerequisite logic manipulation. Thus for SVP64 there are an extraordinary number of nesessary combinations which provide completely different and useful behaviour. Available options to combine:

- `BO[0]` to make an unconditional branch would seem irrelevant if it were not for predication and for side-effects (CTR Mode for example)
- Enabling CTR-test Mode and setting `BO[2]` can still result in the Branch taking place, not because the Condition Test itself failed, but because CTR reached zero **because**, as required by CTR-test mode, CTR was decremented as a **result** of Condition Tests failing.
- `BO[1]` to select whether the CR bit being tested is zero or nonzero
- `R30` and `~R30` and other predicate mask options including CR and inverted CR bit testing
- `sz` and `SNZ` to insert either zeros or ones in place of masked-out predicate bits
- `ALL` or `ANY` behaviour corresponding to `AND` of all tests and `OR` of all tests, respectively.
- Predicate Mask bits, which combine in effect with the CR being tested.
- Inversion of Predicate Masks (`~r3` instead of `r3`, or using `NE` rather than `EQ`) which results in an additional level of possible ANDing, ORing etc. that would otherwise need explicit instructions.

The most obviously useful combinations here are to set `BO[1]` to zero in order to turn `ALL` into Great-Big-NAND and `ANY` into Great-Big-NOR. Other Mode bits which perform behavioural inversion then have to work round the fact that the Condition Testing is NOR or NAND. The alternative to not having additional behavioural inversion (`SNZ`, `VSb`, `CTi`) would be to have a second (unconditional) branch directly after the first, which the first branch jumps over. This contrivance is avoided by the behavioural inversion bits.

Pseudocode and examples </>

Please see {SVP64 Appendix} regarding CR bit ordering and for the definition of `CR{n}`

For comparative purposes this is a copy of the v3.0B `bc` pseudocode

```
if (mode_is_64bit) then M <- 0
else M <- 32
if ¬BO[2] then CTR <- CTR - 1
ctr_ok <- BO[2] | ((CTR[M:63] != 0) ^ BO[3])
cond_ok <- BO[0] | ¬(CR[BI+32] ^ BO[1])
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
```

Simplified pseudocode including LRu and CTR skipping, which illustrates clearly that SVP64 Scalar Branches (VL=1) are **not** identical to v3.0B Scalar Branches. The key areas where differences occur are the inclusion of predication (which can still be used when VL=1), in when and why CTR is decremented (CTRtest Mode) and whether LR is updated (which is unconditional in v3.0B when LK=1, and conditional in SVP64 when LRu=1).

Inline comments highlight the fact that the Scalar Branch behaviour and pseudocode is still clearly visible and embedded within the Vectorized variant:

```
if (mode_is_64bit) then M <- 0
else M <- 32
# the bit of CR to test, if the predicate bit is zero,
# is overridden
testbit = CR[BI+32]
if ¬predicate_bit then testbit = SVRMmode.SNZ
# otherwise apart from the override ctr_ok and cond_ok
# are exactly the same
ctr_ok <- BO[2] | ((CTR[M:63] != 0) ^ BO[3])
cond_ok <- BO[0] | ¬(testbit ^ BO[1])
if ¬predicate_bit & ¬SVRMmode.sz then
```

```
  # this is entirely new: CTR-test mode still decrements CTR
  # even when predicate-bits are zero
  if ¬BO[2] & CTRtest & ¬CTi then
    CTR = CTR - 1
  # instruction finishes here
else
  # usual BO[2] CTR-mode now under CTR-test mode as well
  if ¬BO[2] & ¬(CTRtest & (cond_ok ^ CTi)) then CTR <- CTR - 1
  # new VLset mode, conditional test truncates VL
  if VLSET and VSb = (cond_ok & ctr_ok) then
    if SVRMmode.VLI then SVSTATE.VL = srcstep+1
    else                 SVSTATE.VL = srcstep
  # usual LR is now conditional, but also joined by SVLR
  lr_ok <- LK
  svlr_ok <- SVRMmode.SL
  if ctr_ok & cond_ok then
    if AA then NIA <-iea EXTS(BD || 0b00)
    else       NIA <-iea CIA + EXTS(BD || 0b00)
    if SVRMmode.LRu then lr_ok <- ¬lr_ok
    if SVRMmode.SLu then svlr_ok <- ¬svlr_ok
  if lr_ok   then LR   <-iea CIA + 4
  if svlr_ok then SVLR <- SVSTATE
```

Below is the pseudocode for SVP64 Branches, which is a little less obvious but identical to the above. The lack of obviousness is down to the early-exit opportunities.

Effective pseudocode for Horizontal-First Mode:

```
if (mode_is_64bit) then M <- 0
else M <- 32
cond_ok = not SVRMmode.ALL
for srcstep in range(VL):
    # select predicate bit or zero/one
    if predicate[srcstep]:
        # get SVP64 extended CR field 0..127
        SVCRf = SVP64EXTRA(BI>>2)
        CRbits = CR{SVCRf}
        testbit = CRbits[BI & 0b11]
        # testbit = CR[BI+32+srcstep*4]
    else if not SVRMmode.sz:
        # inverted CTR test skip mode
        if ¬BO[2] & CTRtest & ¬CTI then
          CTR = CTR - 1
        continue # skip to next element
    else
        testbit = SVRMmode.SNZ
    # actual element test here
    ctr_ok <- BO[2] | ((CTR[M:63] != 0) ^ BO[3])
    el_cond_ok <- BO[0] | ¬(testbit ^ BO[1])
    # check if CTR dec should occur
    ctrdec = ¬BO[2]
    if CTRtest & (el_cond_ok ^ CTi) then
       ctrdec = 0b0
    if ctrdec then CTR <- CTR - 1
    # merge in the test
    if SVRMmode.ALL:
        cond_ok &= (el_cond_ok & ctr_ok)
    else
        cond_ok |= (el_cond_ok & ctr_ok)
    # test for VL to be set (and exit)
    if VLSET and VSb = (el_cond_ok & ctr_ok) then
        if SVRMmode.VLI then SVSTATE.VL = srcstep+1
        else                 SVSTATE.VL = srcstep
        break
    # early exit?
    if SVRMmode.ALL != (el_cond_ok & ctr_ok):
         break
    # SVP64 rules about Scalar registers still apply!
    if SVCRf.scalar:
       break
# loop finally done, now test if branch (and update LR)
lr_ok <- LK
svlr_ok <- SVRMmode.SL
if cond_ok then
    if AA then NIA <-iea EXTS(BD || 0b00)
```

```
        else       NIA <-iea CIA + EXTS(BD || 0b00)
        if SVRMmode.LRu then lr_ok <- ¬lr_ok
        if SVRMmode.SLu then svlr_ok <- ¬svlr_ok
    if lr_ok then LR <-iea CIA + 4
    if svlr_ok then SVLR <- SVSTATE
```

Pseudocode for Vertical-First Mode:

```
    # get SVP64 extended CR field 0..127
    SVCRf = SVP64EXTRA(BI>>2)
    CRbits = CR{SVCRf}
    # select predicate bit or zero/one
    if predicate[srcstep]:
        if BRc = 1 then # CR0 vectorized
            CR{SVCRf+srcstep} = CRbits
        testbit = CRbits[BI & 0b11]
    else if not SVRMmode.sz:
        # inverted CTR test skip mode
        if ¬BO[2] & CTRtest & ¬CTI then
           CTR = CTR - 1
        SVSTATE.srcstep = new_srcstep
        exit # no branch testing
    else
        testbit = SVRMmode.SNZ
    # actual element test here
    cond_ok <- BO[0] | ¬(testbit ^ BO[1])
    # test for VL to be set (and exit)
    if VLSET and cond_ok = VSb then
        if SVRMmode.VLI
            SVSTATE.VL = new_srcstep+1
        else
            SVSTATE.VL = new_srcstep
```

Example Shader code </>

```
    // assume f() g() or h() modify a and/or b
    while(a > 2) {
        if(b < 5)
            f();
        else
            g();
        h();
    }
```

which compiles to something like:

```
    vec<i32> a, b;
    // ...
    pred loop_pred = a > 2;
    // loop continues while any of a elements greater than 2
    while(loop_pred.any()) {
        // vector of predicate bits
        pred if_pred = loop_pred & (b < 5);
        // only call f() if at least 1 bit set
        if(if_pred.any()) {
            f(if_pred);
        }
    label1:
        // loop mask ANDs with inverted if-test
        pred else_pred = loop_pred & ~if_pred;
        // only call g() if at least 1 bit set
        if(else_pred.any()) {
            g(else_pred);
        }
        h(loop_pred);
    }
```

which will end up as:

```
       # start from while loop test point
       b looptest
    while_loop:
       sv.cmpi CR80.v, b.v, 5     # vector compare b into CR64 Vector
       sv.bc/m=r30/~ALL/sz CR80.v.LT skip_f # skip when none
       # only calculate loop_pred & pred_b because needed in f()
       sv.crand CR80.v.SO, CR60.v.GT, CR80.V.LT # if = loop & pred_b
```

```
   f(CR80.v.SO)
skip_f:
   # illustrate inversion of pred_b. invert r30, test ALL
   # rather than SOME, but masked-out zero test would FAIL,
   # therefore masked-out instead is tested against 1 not 0
   sv.bc/m=~r30/ALL/SNZ CR80.v.LT skip_g
   # else = loop & ~pred_b, need this because used in g()
   sv.crternari(A&~B) CR80.v.SO, CR60.v.GT, CR80.V.LT
   g(CR80.v.SO)
skip_g:
   # conditionally call h(r30) if any loop pred set
   sv.bclr/m=r30/~ALL/sz BO[1]=1 h()
looptest:
   sv.cmpi CR60.v a.v, 2      # vector compare a into CR60 vector
   sv.crweird r30, CR60.GT # transfer GT vector to r30
   sv.bc/m=r30/~ALL/sz BO[1]=1 while_loop
end:
```

LRu example </>

show why LRu would be useful in a loop. Imagine the following c code:

```
for (int i = 0; i < 8; i++) {
    if (x < y) break;
}
```

Under these circumstances exiting from the loop is not only based on CTR it has become conditional on a CR result. Thus it is desirable that NIA *and* LR only be modified if the conditions are met

v3.0 pseudocode for `bclrl`:

```
if (mode_is_64bit) then M <- 0
else M <- 32
if ¬BO[2]  then CTR <- CTR - 1
ctr_ok <- BO[2] | ((CTR[M:63] != 0) ^ BO[3])
cond_ok <- BO[0] | ¬(CR[BI+32] ^  BO[1])
if ctr_ok & cond_ok then NIA <-iea LR[0:61] || 0b00
if LK then LR <-iea CIA + 4
```

the latter part for SVP64 `bclrl` becomes:

```
for i in 0 to VL-1:
    ...
    ...
    cond_ok <- BO[0] | ¬(CR[BI+32] ^  BO[1])
    lr_ok <- LK
    if ctr_ok & cond_ok then
       NIA <-iea LR[0:61] || 0b00
       if SVRMmode.LRu then lr_ok <- ¬lr_ok
    if lr_ok then LR <-iea CIA + 4
    # if NIA modified exit loop
```

The reason why should be clear from this being a Vector loop: unconditional destruction of LR when LK=1 makes `sv.bclrl` ineffective, because the intention going into the loop is that the branch should be to the copy of LR set at the *start* of the loop, not half way through it. However if the change to LR only occurs if the branch is taken then it becomes a useful instruction.

The following pseudocode should **not** be implemented because it violates the fundamental principle of SVP64 which is that SVP64 looping is a thin wrapper around Scalar Instructions. The pseducode below is more an actual Vector ISA Branch and as such is not at all appropriate:

```
for i in 0 to VL-1:
    ...
    ...
    cond_ok <- BO[0] | ¬(CR[BI+32] ^  BO[1])
    if ctr_ok & cond_ok then NIA <-iea LR[0:61] || 0b00
# only at the end of looping is LK checked.
# this completely violates the design principle of SVP64
# and would actually need to be a separate (scalar)
# instruction "set LR to CIA+4 but retrospectively"
# which is clearly impossible
if LK then LR <-iea CIA + 4
```

[[!tag standards]] # Condition Register SVP64 Operations </>

DRAFT STATUS

Links:

- https://bugs.libre-soc.org/show_bug.cgi?id=687
- https://bugs.libre-soc.org/show_bug.cgi?id=936 write on failfirst
- https://bugs.libre-soc.org/show_bug.cgi?id=1183 enable mapreduce with failfirst
- {SVP64 Chapter}
- {Branch Mode}
- {CR Weird ops}
- [[openpower/isa/sprset]]
- [[openpower/isa/condition]]
- [[openpower/isa/comparefixed]]

Condition Register Fields are only 4 bits wide: this presents some interesting conceptual challenges for SVP64, which was designed primarily for vectors of arithmetic and logical operations. However if predicates may be bits of CR Fields it makes sense to extend Simple-V to cover CR Operations, especially given that Vectorized Rc=1 may be processed by Vectorized CR Operations that usefully in turn may become Predicate Masks to yet more Vector operations, like so:

```
sv.cmpi/ew=8 *B,*ra,0    # compare bytes against zero
sv.cmpi/ew=8 *B2,*ra,13. # and against newline
sv.cror PM.EQ,B.EQ,B2.EQ # OR compares to create mask
sv.stb/sm=EQ    ...      # store only nonzero/newline
```

Element width however is clearly meaningless for a 4-bit collation of Conditions, EQ LT GE SO. Likewise, arithmetic saturation (an important part of Arithmetic SVP64) has no meaning. An alternative Mode Format is required, and given that elwidths are meaningless for CR Fields the bits in SVP64 `RM` may be used for other purposes.

This alternative mapping **only** applies to instructions that **only** reference a CR Field or CR bit as the sole exclusive result. This section **does not** apply to instructions which primarily produce arithmetic results that also, as an aside, produce a corresponding CR Field (such as when Rc=1). Instructions that involve Rc=1 are definitively arithmetic in nature, where the corresponding Condition Register Field can be considered to be a “co-result”. Such CR Field “co-result” arithmeric operations are firmly out of scope for this section, being covered fully by {Arithmetic Mode}.

- Examples of Vectorizeable Defined Word-instructions to which this section does apply is
  - `mfcr` and `cmpi` (3 bit operands) and
  - `crnor` and `crand` (5 bit operands).
- Examples to which this section does **not** apply include `fadds.` and `subf.` which both produce arithmetic results (and a CR Field co-result).
- `mtcr` is considered [[openpower/sv/normal]] because it refers to the entire 32-bit Condition Register rather than to CR Fields.

The CR Mode Format still applies to `sv.cmpi` because despite taking a GPR as input, the output from the Base Scalar v3.0B `cmpi` instruction is purely to a Condition Register Field.

Other modes are still applicable and include:

- **Data-dependent fail-first**. useful to truncate VL based on analysis of a Condition Register result bit.
- **Reduction**. Reduction is useful for analysing a Vector of Condition Register Fields and reducing it to one single Condition Register Field.

Special atrention should be paid on the difference between Data-Dependent Fail-First on CR operations and [[openpower/sv/normal]] regarding the seemingly-contradictory behaviour of `Rc=1,VLi=0`. Explained below.

Format </>

SVP64 RM `MODE` (includes `ELWIDTH_SRC` bits) for CR-based operations:

6	7	19:20	21	22:23	description
/	/	0 0	RG	dz sz	simple mode
/	/	1 0	RG	dz sz	scalar reduce mode (mapreduce)
zz	SNZ	VLI 1	inv	CR-bit	Ffirst 3-bit mode
/	SNZ	VLI 1	inv	dz sz	Ffirst 5-bit mode (implies CR-bit from result)

Fields:

- **sz / dz** if predication is enabled will put zeros into the dest (or as src in the case of twin pred) when the predicate bit is zero. otherwise the element is ignored or skipped, depending on context.
- **zz** set both sz and dz equal to this flag
- **SNZ** In fail-first mode, on the bit being tested, when sz=1 and SNZ=1 a value “1” is put in place of “0”.
- **inv CR-bit** just as in branches (BO) these bits allow testing of a CR bit and whether it is set (inv=0) or unset (inv=1)
- **RG** Reverse-Gear: inverts the Vector Loop order (VL-1 downto 0) rather than the normal 0 upto VL-1

- **VLi** VL inclusive: in fail-first mode, the truncation of VL *includes* the current element at the failure point rather than excludes it from the count.

Data-dependent fail-first on CR operations </>

The principle of data-dependent fail-first is that if, during the course of sequentially evaluating an element's Condition Test, one such test is encountered which fails, then VL (Vector Length) is truncated (set) at that point. In the case of Arithmetic SVP64 Operations the Condition Register Field generated from Rc=1 is used as the basis for the truncation decision. However with CR-based operations that CR Field result to be tested is provided *by the operation itself*.

Data-dependent SVP64 Vectorized Operations involving the creation or modification of a CR can require an extra two bits, which are not available in the compact space of the SVP64 RM `MODE` Field. With the concept of element width overrides being meaningless for CR Fields it is possible to use the `ELWIDTH` field for alternative purposes.

Condition Register based operations such as `sv.mfcr` and `sv.crand` can thus be made more flexible. However the rules that apply in this section also apply to future CR-based instructions.

There are two primary different types of CR operations:

- Those which have a 3-bit operand field (referring to a CR Field)
- Those which have a 5-bit operand (referring to a bit within the whole 32-bit CR)

Examining these two types it is observed that the difference may be considered to be that the 5-bit variant *already* provides the prerequisite information about which CR Field bit (EQ, GE, LT, SO) is to be operated on by the instruction. Thus, logically, we may set the following rule:

- When a 5-bit CR Result field is used in an instruction, the 5-bit variant of Data-Dependent Fail-First must be used. i.e. the bit of the CR field to be tested is the one that has just been modified (created) by the operation.
- When a 3-bit CR Result field is used the 3-bit variant must be used, providing as it does the missing `CRbit` field in order to select which CR Field bit of the result shall be tested (EQ, LE, GE, SO)

The reason why the 3-bit CR variant needs the additional CR-bit field should be obvious from the fact that the 3-bit CR Field from the base Power ISA v3.0B operation clearly does not contain and is missing the two CR Field Selector bits. Thus, these two bits (to select EQ, LE, GE or SO) must be provided in another way.

Examples of the former type:

- crand, cror, crnor. These all are 5-bit destination (BT). The bit to be tested against `inv` is the one selected by `BT`
- mcrf. This has only 3-bit destiation (BF). In order to select the bit to be tested, the alternative encoding must be used. With `CRbit` coming from the SVP64 RM bits 22-23 the bit of BF to be tested is identified.

Just as with SVP64 {Branch Mode} there is the option to truncate VL to include the element being tested (`VLi=1`) and to exclude it (`VLi=0`).

Also exactly as with {Arithmetic Mode} fail-first, VL cannot, unlike {Load/Store Mode}, be set to an arbitrary value. Deterministic behaviour is *required*.

Apparent contradictory behaviour compared to Rc=1,VLi=0

In [[openpower/sv/normal]] mode when Rc=1 and VLi=0 the Vector of co-results appears to ignore VLi=0 because the last CR Field co-result element tested is written out regardless of the setting of VLi. This is because when Rc=1 the CR Fields are co-results *not* actual results.

When looking at the *actual* number of results written (arithmetic results on arithmetic operations vs CR-Field results on *CR-Field* operations), and ignoring the Rc=1 co-results entirely, the totals (the behaviours) are consistent whether VLi=0 or VLi=1.

*Programmer's Note: Data-dependent fail-first stores an updated VL in the SVSTATE SPR, not in any GPR. If needed VL may be obtained by using the alias `getvl`.

Reduction and Iteration </>

Bearing in mind as described in the {SVP64 Appendix} SVP64 Horizontal Reduction is a deterministic schedule on top of base Scalar v3.0 operations, the same rules apply to CR Operations, i.e. that programmers must follow certain conventions in order for an *end result* of a reduction to be achieved. Unlike other Vector ISAs *there are no explicit reduction opcodes* in SVP64: Schedules however achieve the same effect.

Due to these conventions only reduction on operations such as `crand` and `cror` are meaningful because these have Condition Register Fields as both input and output. Meaningless operations are not prohibited because the cost in hardware of doing so is prohibitive, but neither are they `UNDEFINED`. Implementations are still required to execute them but are at liberty to optimise out any operations that would ultimately be overwritten, as long as Strict Program Order is still obvservable by the programmer.

Also bear in mind that 'Reverse Gear' may be enabled, which can be used in combination with overlapping CR operations to iteratively accumulate results. Issuing a `sv.crand` operation for example with `BA` differing from `BB` by one Condition Register Field would result in a cascade effect, where the first-encountered CR Field would set the result to zero, and also all subsequent CR Field elements thereafter:

```
# sv.crand/mr/rg CR4.ge.v, CR5.ge.v, CR4.ge.v
for i in VL-1 downto 0 # reverse gear
     CR.field[4+i].ge &= CR.field[5+i].ge
```

`sv.crxor` with reduction would be particularly useful for parity calculation for example, although there are many ways in which the same calculation could be carried out (`parityw`) after transferring a vector of CR Fields to a GPR using crweird operations.

Implementations are free and clear to optimise these reductions in any way they see fit, as long as the end-result is compatible with Strict Program Order being observed, and Interrupt latency is not adversely impacted. Good examples include `sv.cror/mr` which is a cumulative ORing of a Vector of CR Field bits, and consequently an easy target for parallelising.

Unusual and quirky CR operations </>

cmp and other compare ops

`cmp` and `cmpi` etc take GPRs as sources and create a CR Field as a result.

```
cmpli BF,L,RA,UI
cmpeqb BF,RA,RB
```

With `ELWIDTH` applying to the source GPR operands this is perfectly fine.

crweird operations

There are 4 weird CR-GPR operations and one reasonable one in the {CR Weird ops} set:

- crrweird
- mtcrweird
- crweirder
- crweird
- mcrfm - reasonably normal and referring to CR Fields for src and dest.

The “weird” operations have a non-standard behaviour, being able to treat *individual bits* of a GPR effectively as elements. They are expected to be Micro-coded by most Hardware implementations.

Effectively-separate Vector and Scalar Condition Register file </>

As mentioned in the introduction on {SVP64 Chapter} some prohibitions are made on instructions involving Condition Registers that allow implementors to actually consider the Scalar CR (fields CR0-CR7) as a completely separate register file from the Vector CRs (fields CR8-CR127).

The complications arise for existing Hardware implementations due to Power ISA not having had “Conditional Execution” added. Adding entirely new pipelines and a new Vector CR Register file is a much easier proposition to consider.

The prohibitions utilise the CR Field numbers implicitly to split out Vectorized CR operations to be considered completely separare and distinct from Scalar CR operations *even though they both use the same binary encoding*. This does in turn mean that at the Decode Phase it becomes necessary to examine not only the operation (`sv.crand`, `sv.cmp`) but also the CR Field numbers as well as whether, in the EXTRA2/3 Mode bits, the operands are Vectorized.

A future version of Power ISA, where SVP64Single is proposed, would in fact introduce “Conditional Execution”, including for VSX. At which point this prohibition becomes moot as Predication would be required to be added into the existing Scalar (and PackedSIMD VSX) side of existing Power ISA implementations.

[[!tag standards]]

Appendix </>

- https://bugs.libre-soc.org/show_bug.cgi?id=574 Saturation
- https://bugs.libre-soc.org/show_bug.cgi?id=558#c47 Parallel Prefix
- https://bugs.libre-soc.org/show_bug.cgi?id=697 Reduce Modes
- https://bugs.libre-soc.org/show_bug.cgi?id=864 parallel prefix simulator
- https://bugs.libre-soc.org/show_bug.cgi?id=809 OV sv.addex discussion
- ARM SVE Fault-first https://alastairreid.github.io/papers/sve-ieee-micro-2017.pdf

This is the appendix to {SVP64 Chapter}, providing explanations of modes etc. leaving the main svp64 page's primary purpose as outlining the instruction format.

Table of contents:

[[!toc]]

Partial Implementations </>

It is perfectly legal to implement subsets of SVP64 as long as illegal instruction traps are always raised on unimplemented features, so that soft-emulation is possible, even for future revisions of SVP64. With SVP64 being partly controlled through contextual SPRs, a little care has to be taken.

All SPRs not implemented including reserved ones for future use must raise an illegal instruction trap if read or written. This allows software the opportunity to emulate the context created by the given SPR.

See {Compliancy Levels} for full details.

XER, SO and other global flags </>

Vector systems are expected to be high performance. This is achieved through parallelism, which requires that elements in the vector be independent. XER SO/OV and other global "accumulation" flags (CR.SO) cause Read-Write Hazards on single-bit global resources, having a significant detrimental effect.

Consequently in SV, XER.SO behaviour is disregarded (including in `cmp` instructions). XER.SO is not read, but XER.OV may be written, breaking the Read-Modify-Write Hazard Chain that complicates microarchitectural implementations. This includes when `scalar identity behaviour` occurs. If precise OpenPOWER v3.0/1 scalar behaviour is desired then OpenPOWER v3.0/1 instructions should be used without an SV Prefix.

TODO jacob add about OV https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf

Of note here is that XER.SO and OV may already be disregarded in the Power ISA v3.0/1 SFFS (Scalar Fixed and Floating) Compliancy Subset. SVP64 simply makes it mandatory to disregard XER.SO even for other Subsets, but only for SVP64 Prefixed Operations.

XER.CA/CA32 on the other hand is expected and required to be implemented according to standard Power ISA Scalar behaviour. Interestingly, due to SVP64 being in effect a hardware for-loop around Scalar instructions executing in precise Program Order, a little thought shows that a Vectorized Carry-In-Out add is in effect a Big Integer Add, taking a single bit Carry In and producing, at the end, a single bit Carry out. High performance implementations may exploit this observation to deploy efficient Parallel Carry Lookahead.

```
# assume VL=4, this results in 4 sequential ops (below)
sv.adde r0.v, r4.v, r8.v

# instructions that get executed in backend hardware:
adde r0, r4, r8 # takes carry-in, produces carry-out
adde r1, r5, r9 # takes carry from previous
...
adde r3, r7, r11 # likewise
```

It can clearly be seen that the carry chains from one 64 bit add to the next, the end result being that a 256-bit "Big Integer Add with Carry" has been performed, and that CA contains the 257th bit. A one-instruction 512-bit Add-with-Carry may be performed by setting VL=8, and a one-instruction 1024-bit Add-with-Carry by setting VL=16, and so on. More on this in [[openpower/sv/biginteger]]

EXTRA Field Mapping </>

The purpose of the 9-bit EXTRA field mapping is to mark individual registers (RT, RA, BFA) as either scalar or vector, and to extend their numbering from 0..31 in Power ISA v3.0 to 0..127 in SVP64. Three of the 9 bits may also be used up for a 2nd Predicate (Twin Predication) leaving a mere 6 bits for qualifying registers. As can be seen there is significant pressure on these (and in fact all) SVP64 bits.

In Power ISA v3.1 prefixing there are bits which describe and classify the prefix in a fashion that is independent of the suffix. MLSS for example. For SVP64 there is insufficient space to make the SVP64 Prefix "self-describing", and consequently every single Scalar instruction had to be individually analysed, by rote, to craft an EXTRA Field Mapping. This process was semi-automated and is described in this section. The final results, which are part of the SVP64 Specification, are here: [[openpower/opcode_regs_deduped]]

- Firstly, every instruction's mnemonic (`add RT, RA, RB`) was analysed from reading the markdown formatted version of the Scalar pseudocode which is machine-readable and found in [[openpower/isatables]]. The analysis gives, by instruction, a "Register Profile". `add RT, RA, RB` for example is given a designation `RM-2R-1W` because it requires two GPR reads and one GPR write.
- Secondly, the total number of registers was added up (2R-1W is 3 registers) and if less than or equal to three then that instruction could be given an EXTRA3 designation. Four or more is given an EXTRA2 designation because there are only 9 bits available.
- Thirdly, the instruction was analysed to see if Twin or Single Predication was suitable. As a general rule this was if there was only a single operand and a single result (`extw` and LD/ST) however it was found that some 2 or 3 operand instructions also qualify. Given that 3 of the 9 bits of EXTRA had to be sacrificed for use in Twin Predication, some compromises were made, here. LDST is Twin but also has 3 operands in some operations, so only EXTRA2 can be used.
- Fourthly, a packing format was decided: for 2R-1W an EXTRA3 indexing could have been decided that RA would be indexed 0 (EXTRA bits 0-2), RB indexed 1 (EXTRA bits 3-5) and RT indexed 2 (EXTRA bits 6-8). In some cases (LD/ST with update) RA-as-a-source is given a **different** EXTRA index from RA-as-a-result (because it is possible to do, and perceived to be useful). Rc=1 co-results (CR0, CR1) are always given the same EXTRA index as their main result (RT, FRT).
- Fifthly, in an automated process the results of the analysis were outputted in CSV Format for use in machine-readable form by sv_analysis.py https://git.libre-soc.org/?p=openpower-isa.git;a=blob;f=src/openpower/sv/sv_analysis.py;hb=HEAD

This process was laborious but logical, and, crucially, once a decision is made (and ratified) cannot be reversed. Qualifying future Power ISA Scalar instructions for SVP64 is **strongly** advised to utilise this same process and the same sv_analysis.py program as a canonical method of maintaining the relationships. Alterations to that same program which change the Designation is **prohibited** once finalised (ratified through the Power ISA WG Process). It would be similar to deciding that `add` should be changed from X-Form to D-Form.

Single Predication </>

This is a standard mode normally found in Vector ISAs. every element in every source Vector and in the destination uses the same bit of one single predicate mask.

In SVSTATE, for Single-predication, implementors MUST increment both srcstep and dststep, but depending on whether sz and/or dz are set, srcstep and dststep can still potentially become different indices. Only when sz=dz is srcstep guaranteed to equal dststep at all times.

Note that in some Mode Formats there is only one flag (zz). This indicates that *both* sz *and* dz are set to the same.

Example 1:

- VL=4
- mask=0b1101
- sz=0, dz=1

The following schedule for srcstep and dststep will occur:

srcstep	dststep	comment
0	0	both mask[src=0] and mask[dst=0] are 1
1	2	sz=1 but dz=0: dst skips mask[1], src soes not
2	3	mask[src=2] and mask[dst=3] are 1
3	end	loop has ended because dst reached VL-1

Example 2:

- VL=4
- mask=0b1101
- sz=1, dz=0

The following schedule for srcstep and dststep will occur:

srcstep	dststep	comment
0	0	both mask[src=0] and mask[dst=0] are 1
2	1	sz=0 but dz=1: src skips mask[1], dst does not
3	2	mask[src=3] and mask[dst=2] are 1
end	3	loop has ended because src reached VL-1

In both these examples it is crucial to note that despite there being a single predicate mask, with sz and dz being different, srcstep and dststep are being requested to react differently.

Example 3:

- VL=4
- mask=0b1101
- sz=0, dz=0

The following schedule for srcstep and dststep will occur:

srcstep	dststep	comment
0	0	both mask[src=0] and mask[dst=0] are 1
2	2	sz=0 and dz=0: both src and dst skip mask[1]
3	3	mask[src=3] and mask[dst=3] are 1
end	end	loop has ended because src and dst reached VL-1

Here, both srcstep and dststep remain in lockstep because sz=dz=0

Twin Predication </>

This is a novel concept that allows predication to be applied to a single source and a single dest register. The following types of traditional Vector operations may be encoded with it, *without requiring explicit opcodes to do so*

- VSPLAT (a single scalar distributed across a vector)
- VEXTRACT (like LLVM IR `extractelement`)
- VINSERT (like LLVM IR `insertelement`)
- VCOMPRESS (like LLVM IR `llvm.masked.compressstore.*`)
- VEXPAND (like LLVM IR `llvm.masked.expandload.*`)

Those patterns (and more) may be applied to:

- mv (the usual way that V* ISA operations are created)
- exts* sign-extension
- rwlinm and other RS-RA shift operations (**note**: excluding those that take RA as both a src and dest. These are not 1-src 1-dest, they are 2-src, 1-dest)
- LD and ST (treating AGEN as one source)
- FP fclass, fsgn, fneg, fabs, fcvt, frecip, fsqrt etc.
- Condition Register ops mfcr, mtcr and other similar

This is a huge list that creates extremely powerful combinations, particularly given that one of the predicate options is `(1<<r3)`

Additional unusual capabilities of Twin Predication include a back-to-back version of VCOMPRESS-VEXPAND which is effectively the ability to do sequentially ordered multiple VINSERTs. The source predicate selects a sequentially ordered subset of elements to be inserted; the destination predicate specifies the sequentially ordered recipient locations. This is equivalent to `llvm.masked.compressstore.*` followed by `llvm.masked.expandload.*` with a single instruction, but abstracted out from Load/Store and applicable in general to any 2P instruction.

This extreme power and flexibility comes down to the fact that SVP64 is not actually a Vector ISA: it is a loop-abstraction-concept that is applied *in general* to Scalar operations, just like the x86 `REP` instruction (if put on steroids).

Pack/Unpack </>

The pack/unpack concept of VSX `vpack` is abstracted out as Sub-Vector reordering. Two bits in the `SVSHAPE` [[sv/spr]] enable either “packing” or “unpacking” on the subvectors vec2/3/4.

First, illustrating a “normal” SVP64 operation with `SUBVL!=1:` (assuming no elwidth overrides), note that the VL loop is outer and the SUBVL loop inner:

```
def index():
    for i in range(VL):
        for j in range(SUBVL):
            yield i*SUBVL+j

for idx in index():
    operation_on(RA+idx)
```

For pack/unpack (again, no elwidth overrides), note that now there is the option to swap the SUBVL and VL loop orders. In effect the Pack/Unpack performs a Transpose of the subvector elements. Illustrated this time with a GPR mv operation:

```
# yield an outer-SUBVL or inner VL loop with SUBVL
def index_p(outer):
    if outer:
        for j in range(SUBVL):   # subvl is outer
            for i in range(VL):  # vl is inner
                yield i+VL*j
    else:
        for i in range(VL):        # vl is outer
            for j in range(SUBVL): # subvl is inner
                yield i*SUBVL+j
```

```
# walk through both source and dest indices simultaneously
for src_idx, dst_idx in zip(index_p(PACK), index_p(UNPACK)):
    move_operation(RT+dst_idx, RA+src_idx)
```

“yield” from python is used here for simplicity and clarity. The two Finite State Machines for the generation of the source and destination element offsets progress incrementally in lock-step.

Example VL=2, SUBVL=3, PACK_en=1 - elements grouped by vec3 will be redistributed such that Sub-elements 0 are packed together, Sub-elements 1 are packed together, as are Sub-elements 2.

```
srcstep=0   srcstep=1
0   1   2   3   4   5

dststep=0  dststep=1  dststep=2
0   3      1   4      2   5
```

Setting of both `PACK` and `UNPACK` is neither prohibited nor `UNDEFINED` because the reordering is fully deterministic, and additional REMAP reordering may be applied. Combined with Matrix REMAP this would give potentially up to 4 Dimensions of reordering.

Pack/Unpack has quirky interactions on {Swizzle Move} because it can set a different subvector length for destination, and has a slightly different pseudocode algorithm for Vertical-First Mode.

Ordering is as follows:

- SVSHAPE srcstep, dststep, ssubstep and dsubstep are advanced sequentially depending on PACK/UNPACK.
- srcstep and dststep are pushed through REMAP to compute actual Element offsets.
- Swizzle is independently applied to ssubstep and dsubstep

Pack/Unpack is enabled (set up) through {svstep instruction}.

Reduce modes </>

Reduction in SVP64 is deterministic and somewhat of a misnomer. A normal Vector ISA would have explicit Reduce opcodes with defined characteristics per operation: in SX Aurora there is even an additional scalar argument containing the initial reduction value, and the default is either 0 or 1 depending on the specifics of the explicit opcode. SVP64 fundamentally has to utilise *existing* Scalar Power ISA v3.0B operations, which presents some unique challenges.

The solution turns out to be to simply define reduction as permitting deterministic element-based schedules to be issued using the base Scalar operations, and to rely on the underlying microarchitecture to resolve Register Hazards at the element level. This goes back to the fundamental principle that SV is nothing more than a Sub-Program-Counter sitting between Decode and Issue phases.

For Scalar Reduction, Microarchitectures *may* take opportunities to parallelise the reduction but only if in doing so they preserve strict Program Order at the Element Level. Opportunities where this is possible include an `OR` operation or a MIN/MAX operation: it may be possible to parallelise the reduction, but for Floating Point it is not permitted due to different results being obtained if the reduction is not executed in strict Program-Sequential Order.

In essence it becomes the programmer’s responsibility to leverage the pre-determined schedules to desired effect.

Scalar result reduction and iteration </>

Scalar Reduction per se does not exist, instead is implemented in SVP64 as a simple and natural relaxation of the usual restriction on the Vector Looping which would terminate if the destination was marked as a Scalar. Scalar Reduction by contrast *keeps issuing Vector Element Operations* even though the destination register is marked as scalar *and* the same register is used as a source register. Thus it is up to the programmer to be aware of this, observe some conventions, and thus end up achieving the desired outcome of scalar reduction.

It is also important to appreciate that there is no actual imposition or restriction on how this mode is utilised: there will therefore be several valuable uses (including Vector Iteration and “Reverse-Gear”) and it is up to the programmer to make best use of the (strictly deterministic) capability provided.

In this mode, which is suited to operations involving carry or overflow, one register must be assigned, by convention by the programmer to be the “accumulator”. Scalar reduction is thus categorised by:

- One of the sources is a Vector
- the destination is a scalar
- optionally but most usefully when one source scalar register is also the scalar destination (which may be informally termed by convention the “accumulator”)
- That the source register type is the same as the destination register type identified as the “accumulator”. Scalar reduction on `cmp`, `setb` or `isel` makes no sense for example because of the mixture between CRs and GPRs.

Note that issuing instructions in Scalar reduce mode such as `setb` are neither `UNDEFINED` nor prohibited, despite them not making much sense at first glance. Scalar reduce is strictly defined behaviour, and the cost in hardware terms of prohibition of seemingly non-sensical operations is too great. Therefore it is permitted and

*required to be executed successfully. Implementors **MAY** choose to optimise such instructions in instances where their use results in "extraneous execution", i.e. where it is clear that the sequence of operations, comprising multiple overwrites to a scalar destination **without** cumulative, iterative, or reductive behaviour (no "accumulator"), may discard all but the last element operation. Identification of such is trivial to do for `setb` and `cmp`: the source register type is a completely different register file from the destination. Likewise Scalar reduction when the destination is a Vector is as if the Reduction Mode was not requested. However it would clearly be unacceptable to perform such optimisations on cache-inhibited LD/ST, so some considerable care needs to be taken.*

Typical applications include simple operations such as `ADD r3, r10.v, r3` where, clearly, r3 is being used to accumulate the addition of all elements of the vector starting at r10.

```
# add RT, RA,RB but when RT==RA
for i in range(VL):
     iregs[RA] += iregs[RB+i] # RT==RA
```

However, *unless* the operation is marked as "mapreduce" (`sv.add/mr`) SV ordinarily **terminates** at the first scalar operation. Only by marking the operation as "mapreduce" will it continue to issue multiple sub-looped (element) instructions in `Program Order`.

To perform the loop in reverse order, the `RG` (reverse gear) bit must be set. This may be useful in situations where the results may be different (floating-point) if executed in a different order. Given that there is no actual prohibition on Reduce Mode being applied when the destination is a Vector, the "Reverse Gear" bit turns out to be a way to apply Iterative or Cumulative Vector operations in reverse. `sv.add/rg r3.v, r4.v, r4.v` for example will start at the opposite end of the Vector and push a cumulative series of overlapping add operations into the Execution units of the underlying hardware.

Other examples include shift-mask operations where a Vector of inserts into a single destination register is required (see {Bitmanip ops}, bmset), as a way to construct a value quickly from multiple arbitrary bit-ranges and bit-offsets. Using the same register as both the source and destination, with Vectors of different offsets masks and values to be inserted has multiple applications including Video, cryptography and JIT compilation.

```
# assume VL=4:
# * Vector of shift-offsets contained in RC (r12.v)
# * Vector of masks contained in RB (r8.v)
# * Vector of values to be masked-in in RA (r4.v)
# * Scalar destination RT (r0) to receive all mask-offset values
sv.bmset/mr r0, r4.v, r8.v, r12.v
```

Due to the Deterministic Scheduling, Subtract and Divide are still permitted to be executed in this mode, although from an algorithmic perspective it is strongly discouraged. It would be better to use addition followed by one final subtract, or in the case of divide, to get better accuracy, to perform a multiply cascade followed by a final divide.

Note that single-operand or three-operand scalar-dest reduce is perfectly well permitted: the programmer may still declare one register, used as both a Vector source and Scalar destination, to be utilised as the "accumulator". In the case of `sv.fmadds` and `sv.maddhw` etc this naturally fits well with the normal expected usage of these operations.

If an interrupt or exception occurs in the middle of the scalar mapreduce, the scalar destination register **MUST** be updated with the current (intermediate) result, because this is how `Program Order` is preserved (Vector Loops are to be considered to be just another way of issuing instructions in Program Order). In this way, after return from interrupt, the scalar mapreduce may continue where it left off. This provides "precise" exception behaviour.

Note that hardware is perfectly permitted to perform multi-issue parallel optimisation of the scalar reduce operation: it's just that as far as the user is concerned, all exceptions and interrupts **MUST** be precise.

Fail-on-first </>

Data-dependent fail-on-first has two distinct variants: one for LD/ST (see {Load/Store Mode}, the other for arithmetic operations (actually, CR-driven) {Arithmetic Mode} and CR operations {Condition Register Fields Mode}. Note in each case the assumption is that vector elements are required appear to be executed in sequential Program Order, element 0 being the first.

- LD/ST ffirst (not to be confused with *Data-Dependent* LD/ST ffirst) treats the first LD/ST in a vector (element 0) as an ordinary one. Exceptions occur "as normal" on the first element. However for elements 1 and above, if an exception would occur, then VL is **truncated** to the previous element.
- Data-driven (CR-driven) fail-on-first activates when Rc=1 or other CR-creating operation produces a result (including cmp). Similar to branch, an analysis of the CR is performed and if the test fails, the vector operation terminates and discards all element operations above the current one (and the current one if VLi is not set), and VL is truncated to either the *previous* element or the current one, depending on whether VLi (VL "inclusive") is set.

Thus the new VL comprises a contiguous vector of results, all of which pass the testing criteria (equal to zero, less than zero).

The CR-based data-driven fail-on-first is new and not found in ARM SVE or RVV. At the same time it is also "old" because it is a generalisation of the Z80 Block compare instructions, especially CPIR which is based on

CP (compare) as the ultimate “element” (suffix) operation to which the repeat (prefix) is applied. It is extremely useful for reducing instruction count, however requires speculative execution involving modifications of VL to get high performance implementations. An additional mode (RC1=1) effectively turns what would otherwise be an arithmetic operation into a type of `cmp`. The CR is stored (and the CR.eq bit tested against the `inv` field). If the CR.eq bit is equal to `inv` then the Vector is truncated and the loop ends. Note that when RC1=1 the result elements are never stored, only the CRs.

VLi is only available as an option when `Rc=0` (or for instructions which do not have Rc). When set, the current element is always also included in the count (the new length that VL will be set to). This may be useful in combination with “inv” to truncate the Vector to *exclude* elements that fail a test, or, in the case of implementations of strncpy, to include the terminating zero.

In CR-based data-driven fail-on-first there is only the option to select and test one bit of each CR (just as with branch BO). For more complex tests this may be insufficient. If that is the case, a vectorized crops (crand, cror) may be used, and ffirst applied to the crop instead of to the arithmetic vector.

One extremely important aspect of ffirst is:

- LDST ffirst may never set VL equal to zero. This because on the first element an exception must be raised “as normal”.
- CR-based data-dependent ffirst on the other hand **can** set VL equal to zero. This is the only means in the entirety of SV that VL may be set to zero (with the exception of via the SV.STATE SPR). When VL is set zero due to the first element failing the CR bit-test, all subsequent vectorized operations are effectively `nops` which is *precisely the desired and intended behaviour*.

Another aspect is that for ffirst LD/STs, VL may be truncated arbitrarily to a nonzero value for any implementation-specific reason. For example: it is perfectly reasonable for implementations to alter VL when ffirst LD or ST operations are initiated on a nonaligned boundary, such that within a loop the subsequent iteration of that loop begins subsequent ffirst LD/ST operations on an aligned boundary. Likewise, to reduce workloads or balance resources.

CR-based data-dependent first on the other hand MUST not truncate VL arbitrarily to a length decided by the hardware: VL MUST only be truncated based explicitly on whether a test fails. This because it is a precise test on which algorithms will rely.

Note: there is no reverse-direction for Data-dependent Fail-First. REMAP will need to be activated to invert the ordering of element traversal.

Data-dependent fail-first on CR operations (crand etc) </>

Operations that actually produce or alter CR Field as a result do not also in turn have an Rc=1 mode. However it makes no sense to try to test the 4 bits of a CR Field for being equal or not equal to zero. Moreover, the result is already in the form that is desired: it is a CR field. Therefore, CR-based operations have their own SVP64 Mode, described in {Condition Register Fields Mode}

There are two primary different types of CR operations:

- Those which have a 3-bit operand field (referring to a CR Field)
- Those which have a 5-bit operand (referring to a bit within the whole 32-bit CR)

More details can be found in {Condition Register Fields Mode}.

CR Operations </>

CRs are slightly more involved than INT or FP registers due to the possibility for indexing individual bits (crops BA/BB/BT). Again however the access pattern needs to be understandable in relation to v3.0B / v3.1B numbering, with a clear linear relationship and mapping existing when SV is applied.

CR EXTRA mapping table and algorithm </>

Numbering relationships for CR fields are already complex due to being in BE format (*the relationship is not clearly explained in the v3.0B or v3.1 specification*). However with some care and consideration the exact same mapping used for INT and FP regfiles may be applied, just to the upper bits, as explained below. Firstly and most importantly a new notation `CR{field number}` is used to indicate access to a particular Condition Register Field (as opposed to the notation `CR[bit]` which accesses one bit of the 32 bit Power ISA v3.0B Condition Register).

`CR{n}` refers to `CR0` when `n=0` and consequently, for CR0-7, is defined, in v3.0B pseudocode, as:

```
CR{n} = CR[32+n*4:35+n*4]
```

For SVP64 the relationship for the sequential numbering of elements is to the CR **fields** within the CR Register, not to individual bits within the CR register.

The `CR{n}` notation is designed to give *linear sequential numbering* in the Vector domain on a straight sequential Vector Loop.

In OpenPOWER v3.0/1, BF/BT/BA/BB are all 5 bits. The top 3 bits (0:2) select one of the 8 CRs; the bottom 2 bits (3:4) select one of 4 bits *in* that CR (EQ/LT/GT/SO). The numbering was determined (after 4 months of analysis and research) to be as follows:

```
CR_index = (BA>>2)      # top 3 bits
bit_index = (BA & 0b11) # low 2 bits
CR_reg = CR{CR_index}     # get the CR
# finally get the bit from the CR.
CR_bit = (CR_reg & (1<<bit_index)) != 0
```

When it comes to applying SV, it is the *CR Field* number `CR_reg` to which SV EXTRA2/3 applies, **not** the `CR_bit` portion (bits 3-4):

```
if extra3_mode:
    spec = EXTRA3
elif EXTRA2[0]:  # vector mode
    spec = EXTRA2 << 1  # same as EXTRA3, shifted
else:            # scalar mode
    spec = (EXTRA2[0] << 2) | EXTRA2[1]
if spec[0]:
   # vector constructs "BA[0:2] spec[1:2] 00 BA[3:4]"
   return ((BA >> 2)<<6) | # hi 3 bits shifted up
          (spec[1:2]<<4) | # to make room for these
          (BA & 0b11)      # CR_bit on the end
else:
   # scalar constructs "00 spec[1:2] BA[0:4]"
   return (spec[1:2] << 5) | BA
```

Thus, for example, to access a given bit for a CR in SV mode, the v3.0B algorithm to determine CR_reg is modified to as follows:

```
CR_index = (BA>>2)      # top 3 bits
if spec[0]:
    # vector mode, 0-124 increments of 4
    CR_index = (CR_index<<4) | (spec[1:2] << 2)
else:
    # scalar mode, 0-32 increments of 1
    CR_index = (spec[1:2]<<3) | CR_index
# same as for v3.0/v3.1 from this point onwards
bit_index = (BA & 0b11) # low 2 bits
CR_reg = CR{CR_index}     # get the CR
# finally get the bit from the CR.
CR_bit = (CR_reg & (1<<bit_index)) != 0
```

Note here that the decoding pattern to determine CR_bit does not change.

Note: high-performance implementations may read/write Vectors of CRs in batches of aligned 32-bit chunks (CR0-7, CR7-15). This is to greatly simplify internal design. If instructions are issued where CR Vectors do not start on a 32-bit aligned boundary, performance may be affected.

CR fields as inputs/outputs of vector operations </>

CRs (or, the arithmetic operations associated with them) may be marked as Vectorized or Scalar. When Rc=1 in arithmetic operations that have no explicit EXTRA to cover the CR, the CR is Vectorized if the destination is Vectorized. Likewise if the destination is scalar then so is the CR.

When vectorized, the CR inputs/outputs are sequentially read/written to 4-bit CR fields. Vectorized Integer results, when Rc=1, will begin writing to CR8 (TBD evaluate) and increase sequentially from there. This is so that:

- implementations may rely on the Vector CRs being aligned to 8. This means that CRs may be read or written in aligned batches of 32 bits (8 CRs per batch), for high performance implementations.
- scalar Rc=1 operation (CR0, CR1) and callee-saved CRs (CR2-4) are not overwritten by vector Rc=1 operations except for very large VL
- CR-based predication, from CR32, is also not interfered with (except by large VL).

However when the SV result (destination) is marked as a scalar by the EXTRA field the *standard* v3.0B behaviour applies: the accompanying CR when Rc=1 is written to. This is CR0 for integer operations and CR1 for FP operations.

Note that yes, the CR Fields are genuinely Vectorized. Unlike in SIMD VSX which has a single CR (CR6) for a given SIMD result, SV Vectorized OpenPOWER v3.0B scalar operations produce a **tuple** of element results: the result of the operation as one part of that element *and a corresponding CR element*. Greatly simplified pseudocode:

```
for i in range(VL):
     # calculate the vector result of an add
     iregs[RT+i] = iregs[RA+i] + iregs[RB+i]
     # now calculate CR bits
     CRs{8+i}.eq = iregs[RT+i] == 0
     CRs{8+i}.gt = iregs[RT+i] > 0
     ... etc
```

If a “cumulated” CR based analysis of results is desired (a la VSX CR6) then a followup instruction must be performed, setting “reduce” mode on the Vector of CRs, using cr ops (crand, crnor) to do so. This provides far more flexibility in analysing vectors than standard Vector ISAs. Normal Vector ISAs are typically restricted to “were all results nonzero” and “were some results nonzero”. The application of mapreduce to Vectorized cr operations allows far more sophisticated analysis, particularly in conjunction with the new crweird operations see {CR Weird ops}.

Note in particular that the use of a separate instruction in this way ensures that high performance multi-issue OoO inplementations do not have the computation of the cumulative analysis CR as a bottleneck and hindrance, regardless of the length of VL.

Additionally, SVP64 {Branch Mode} may be used, even when the branch itself is to the following instruction. The combined side-effects of CTR reduction and VL truncation provide several benefits.

(see [[discussion]]. some alternative schemes are described there)

Rc=1 when SUBVL!=1 </>

sub-vectors are effectively a form of Packed SIMD (length 2 to 4). Only 1 bit of predicate is allocated per subvector; likewise only one CR is allocated per subvector.

This leaves a conundrum as to how to apply CR computation per subvector, when normally Rc=1 is exclusively applied to scalar elements. A solution is to perform a bitwise OR or AND of the subvector tests. Given that OE is ignored in SVP64, this field may (when available) be used to select OR or AND behavior.

Table of CR fields </> CRn is the notation used by the OpenPower spec to refer to CR field #i, so FP instructions with Rc=1 write to CR1 (n=1).

CRs are not stored in SPRs: they are registers in their own right. Therefore context-switching the full set of CRs involves a Vectorized mfcr or mtcr, using VL=8 to do so. This is exactly as how scalar OpenPOWER context-switches CRs: it is just that there are now more of them.

The 64 SV CRs are arranged similarly to the way the 128 integer registers are arranged. TODO a python program that auto-generates a CSV file which can be included in a table, which is in a new page (so as not to overwhelm this one). [[svp64/cr_names]]

Register Profiles </>

Instructions are broken down by Register Profiles as listed in the following auto-generated page: {SVP64 Augmentation Table}. These tables, despite being auto-generated, are part of the Specification.

SV pseudocode illustration </>

Single-predicated Instruction </>

illustration of normal mode add operation: zeroing not included, elwidth overrides not included. if there is no predicate, it is set to all 1s

```
function op_add(rd, rs1, rs2) # add not VADD!
  int i, id=0, irs1=0, irs2=0;
  predval = get_pred_val(FALSE, rd);
  for (i = 0; i < VL; i++)
    STATE.srcoffs = i # save context
    if (predval & 1<<i) # predication uses intregs
       ireg[rd+id] <= ireg[rs1+irs1] + ireg[rs2+irs2];
    if (!int_vec[rd].isvec) break;
    if (rd.isvec)  { id += 1; }
    if (rs1.isvec) { irs1 += 1; }
    if (rs2.isvec) { irs2 += 1; }
    if (id == VL or irs1 == VL or irs2 == VL) {
      # end VL hardware loop
      STATE.srcoffs = 0; # reset
      return;
    }
```

This has several modes:

- RT.v = RA.v RB.v
- RT.v = RA.v RB.s (and RA.s RB.v)
- RT.v = RA.s RB.s
- RT.s = RA.v RB.v
- RT.s = RA.v RB.s (and RA.s RB.v)
- RT.s = RA.s RB.s

All of these may be predicated. Vector-Vector is straightfoward. When one of source is a Vector and the other a Scalar, it is clear that each element of the Vector source should be added to the Scalar source, each result placed into the Vector (or, if the destination is a scalar, only the first nonpredicated result).

The one that is not obvious is RT=vector but both RA/RB=scalar. Here this acts as a “splat scalar result”, copying the same result into all nonpredicated result elements. If a fixed destination scalar was intended, then an all-Scalar operation should be used.

See https://bugs.libre-soc.org/show_bug.cgi?id=552

Assembly Annotation </>

Assembly code annotation is required for SV to be able to successfully mark instructions as “prefixed”.

A reasonable (prototype) starting point:

```
svp64 [field=value]*
```

Fields:

- ew=8/16/32 - element width
- sew=8/16/32 - source element width
- vec=2/3/4 - SUBVL
- mode=mr/satu/sats/crpred
- pred=1«3/r3/$_{r3/r10/}$r10/r30/~r30/lt/gt/le/ge/eq/ne

similar to x86 “rex” prefix.

For actual assembler:

```
sv.asmcode/mode.vec{N}.ew=8,sw=16,m={pred},sm={pred} reg.v, src.s
```

Qualifiers:

- m={pred}: predicate mask mode
- sm={pred}: source-predicate mask mode (only allowed in Twin-predication)
- vec{N}: vec2 OR vec3 OR vec4 - sets SUBVL=2/3/4
- ew={N}: ew=8/16/32 - sets elwidth override
- sw={N}: sw=8/16/32 - sets source elwidth override
- ff={xx}: see fail-first mode
- sat{x}: satu / sats - see saturation mode
- mr: see map-reduce mode
- mrr: map-reduce, reverse-gear (VL-1 downto 0)
- mr.svm see map-reduce with sub-vector mode
- crm: see map-reduce CR mode
- crm.svm see map-reduce CR with sub-vector mode
- sz: predication with source-zeroing
- dz: predication with dest-zeroing

For modes:

- fail-first
  - ff=lt/gt/le/ge/eq/ne/so/ns
  - RC1 mode
- saturation:
  - sats
  - satu
- map-reduce:
  - mr OR crm: “normal” map-reduce mode or CR-mode.
  - mr.svm OR crm.svm: when vec2/3/4 set, sub-vector mapreduce is enabled

Parallel-reduction algorithm </>

The principle of SVP64 is that SVP64 is a fully-independent Abstraction of hardware-looping in between issue and execute phases that has no relation to the operation it issues. Additional state cannot be saved on context-switching beyond that of SVSTATE, making things slightly tricky.

Executable demo pseudocode, full version here

```
def preduce_yield(vl, vec, pred):
    step = 1
    ix = list(range(vl))
    while step < vl:
        step *= 2
        for i in range(0, vl, step):
            other = i + step // 2
            ci = ix[i]
            oi = ix[other] if other < vl else None
            other_pred = other < vl and pred[oi]
            if pred[ci] and other_pred:
                yield ci, oi
            elif other_pred:
                ix[i] = oi
```

```
def preduce_y(vl, vec, pred):
   for i, other in preduce_yield(vl, vec, pred):
       vec[i] += vec[other]
```

This algorithm works by noting when data remains in-place rather than being reduced, and referring to that alternative position on subsequent layers of reduction. It is re-entrant. If however interrupted and restored, some implementations may take longer to re-establish the context.

Its application by default is that:

- RA, FRA or BFA is the first register as the first operand (ci index offset in the above pseudocode)
- RB, FRB or BFB is the second (co index offset)
- RT (result) also uses ci **if RA==RT**

For more complex applications a REMAP Schedule must be used

Programmers's note: if passed a predicate mask with only one bit set, this algorithm takes no action, similar to when a predicate mask is all zero.

Implementor's Note: many SIMD-based Parallel Reduction Algorithms are implemented in hardware with MVs that ensure lane-crossing is minimised. The mistake which would be catastrophic to SVP64 to make is to then limit the Reduction Sequence for all implementors based solely and exclusively on what one specific internal microarchitecture does. In SIMD ISAs the internal SIMD Architectural design is exposed and imposed on the programmer. Cray-style Vector ISAs on the other hand provide convenient, compact and efficient encodings of abstract concepts. **It is the Implementor's responsibility to produce a design that complies with the above algorithm, utilising internal Micro-coding and other techniques to transparently insert micro-architectural lane-crossing Move operations if necessary or desired, to give the level of efficiency or performance required.**

Element-width overrides </> </>

Element-width overrides are best illustrated with a packed structure union in the c programming language. The following should be taken literally, and assume always a little-endian layout:

```
    #pragma pack
    typedef union {
        uint8_t  b[];
        uint16_t s[];
        uint32_t i[];
        uint64_t l[];
        uint8_t actual_bytes[8];
    } el_reg_t;

    elreg_t int_regfile[128];
```

Accessing (get and set) of registers given a value, register (in `elreg_t` form), and that all arithmetic, numbering and pseudo-Memory format is LE-endian and LSB0-numbered below:

```
    elreg_t& get_polymorphed_reg(elreg_t const& reg, bitwidth, offset):
        el_reg_t res; // result
        res.l = 0; // TODO: going to need sign-extending / zero-extending
        if !reg.isvec: // scalar access has no element offset
            offset = 0
        if bitwidth == 8:
            reg.b = int_regfile[reg].b[offset]
        elif bitwidth == 16:
            reg.s = int_regfile[reg].s[offset]
        elif bitwidth == 32:
            reg.i = int_regfile[reg].i[offset]
        elif bitwidth == 64:
            reg.l = int_regfile[reg].l[offset]
        return reg

    set_polymorphed_reg(elreg_t& reg, bitwidth, offset, val):
        if (!reg.isvec):
            # for safety mask out hi bits
            bytemask = (8 << bitwidth) - 1
            val &= bytemask
            # not a vector: first element only, overwrites high bits.
            # and with the *Architectural* definition being LE,
            # storing in the first DWORD works perfectly.
            int_regfile[reg].l[0] = val
        elif bitwidth == 8:
            int_regfile[reg].b[offset] = val
        elif bitwidth == 16:
            int_regfile[reg].s[offset] = val
        elif bitwidth == 32:
```

```
            int_regfile[reg].i[offset] = val
        elif bitwidth == 64:
            int_regfile[reg].l[offset] = val
```

In effect the GPR registers r0 to r127 (and corresponding FPRs fp0 to fp127) are reinterpreted to be "starting points" in a byte-addressable memory. Vectors - which become just a virtual naming construct - effectively overlap.

It is extremely important for implementors to note that the only circumstance where upper portions of an underlying 64-bit register are zero'd out is when the destination is a scalar. The ideal register file has byte-level write-enable lines, just like most SRAMs, in order to avoid READ-MODIFY-WRITE.

An example ADD operation with predication and element width overrides:

```
for (i = 0; i < VL; i++)
  if (predval & 1<<i) # predication
     src1 = get_polymorphed_reg(RA, srcwid, irs1)
     src2 = get_polymorphed_reg(RB, srcwid, irs2)
     result = src1 + src2 # actual add here
     set_polymorphed_reg(RT, destwid, ird, result)
     if (!RT.isvec) break
  if (RT.isvec)  { id += 1; }
  if (RA.isvec)  { irs1 += 1; }
  if (RB.isvec)  { irs2 += 1; }
```

Thus it can be clearly seen that elements are packed by their element width, and the packing starts from the source (or destination) specified by the instruction.

Twin (implicit) result operations </>

Some operations in the Power ISA already target two 64-bit scalar registers: `lq` for example, and LD with update. Some mathematical algorithms are more efficient when there are two outputs rather than one, providing feedback loops between elements (the most well-known being add with carry). 64-bit multiply for example actually internally produces a 128 bit result, which clearly cannot be stored in a single 64 bit register. Some ISAs recommend "macro op fusion": the practice of setting a convention whereby if two commonly used instructions (mullo, mulhi) use the same ALU but one selects the low part of an identical operation and the other selects the high part, then optimised micro-architectures may "fuse" those two instructions together, using Micro-coding techniques, internally.

The practice and convention of macro-op fusion however is not compatible with SVP64 Horizontal-First, because Horizontal Mode may only be applied to a single instruction at a time, and SVP64 is based on the principle of strict Program Order even at the element level. Thus it becomes necessary to add explicit more complex single instructions with more operands than would normally be seen in the average RISC ISA (3-in, 2-out, in some cases). If it was not for Power ISA already having LD/ST with update as well as Condition Codes and `lq` this would be hard to justify.

With limited space in the `EXTRA` Field, and Power ISA opcodes being only 32 bit, 5 operands is quite an ask. `lq` however sets a precedent: `RTp` stands for "RT pair". In other words the result is stored in RT and RT+1. For Scalar operations, following this precedent is perfectly reasonable. In Scalar mode, `maddedu` therefore stores the two halves of the 128-bit multiply into RT and RT+1.

What, then, of `sv.maddedu`? If the destination is hard-coded to RT and RT+1 the instruction is not useful when Vectorized because the output will be overwritten on the next element. To solve this is easy: define the destination registers as RT and RT+MAXVL respectively. This makes it easy for compilers to statically allocate registers even when VL changes dynamically.

Bear in mind that both RT and RT+MAXVL are starting points for Vectors, and bear in mind that element-width overrides still have to be taken into consideration, the starting point for the implicit destination is best illustrated in pseudocode:

```
# demo of maddedu
for (i = 0; i < VL; i++)
   if (predval & 1<<i) # predication
      src1 = get_polymorphed_reg(RA, srcwid, irs1)
      src2 = get_polymorphed_reg(RB, srcwid, irs2)
      src2 = get_polymorphed_reg(RC, srcwid, irs3)
      result = src1*src2 + src2
      destmask = (2<<destwid)-1
      # store two halves of result, both start from RT.
      set_polymorphed_reg(RT, destwid, ird      , result&destmask)
      set_polymorphed_reg(RT, destwid, ird+MAXVL, result>>destwid)
      if (!RT.isvec) break
   if (RT.isvec)  { id += 1; }
   if (RA.isvec)  { irs1 += 1; }
   if (RB.isvec)  { irs2 += 1; }
   if (RC.isvec)  { irs3 += 1; }
```

The significant part here is that the second half is stored starting not from RT+MAXVL at all: it is the *element* index that is offset by MAXVL, both halves actually starting from RT. If VL is 3, MAXVL is 5, RT is 1, and dest elwidth is 32 then the elements RT0 to RT2 are stored:

```
LSB0:  63:32     31:0
MSB0:  0:31      32:63
r0    unchanged unchanged
r1    RT1.lo    RT0.lo
r2    unchanged RT2.lo
r3    RT0.hi    unchanged
r4    RT2.hi    RT1.hi
r5    unchanged unchanged
```

Note that all of the LO halves start from r1, but that the HI halves start from half-way into r3. The reason is that with MAXVL bring 5 and elwidth being 32, this is the 5th element offset (in 32 bit quantities) counting from r1.

Programmer's note: accessing registers that have been placed starting on a non-contiguous boundary (half-way along a scalar register) can be inconvenient: REMAP can provide an offset but it requires extra instructions to set up. A simple solution is to ensure that MAXVL is rounded up such that the Vector ends cleanly on a contiguous register boundary. MAXVL=6 in the above example would achieve that

Additional DRAFT Scalar instructions in 3-in 2-out form with an implicit 2nd destination:

- {Fixed Point pseudocode}
- {Floating Point pseudocode}

[[!tag standards]]

Simple-V Compliancy Levels </>

The purpose of the Compliancy Levels is to provide a documented stable base for implementors to achieve software interoperability without requiring a high and unnecessary hardware cost unrelated to their needs. The bare minimum requirement, particularly suited for Ultra-embedded, requires just one instruction, reservation of SPRs, and the rest may entirely be Soft-emulated by raising Illegal Instruction traps. At the other end of the spectrum is the full REMAP Structure Packing suitable for traditional Vector Processing workloads and High-performance energy-efficient DSP workloads.

To achieve full soft-emulated interoperability, all implementations **must**, at the bare minimum, raise Illegal Instruction traps for all SPRs including all reserved SPRs, all SVP64-related Context instructions (REMAP), as well as for the entire SVP64 Prefix space.

Even if the Power ISA Scalar Specification states that a given Scalar instruction need not or must not raise an illegal instruction on UNDEFINED behaviour, unimiplemented parts of SVP64 MUST* raise an illegal instruction trap when (and only when) that same Scalar instruction is Prefixed*. It is absolutely critical to note that when not Prefixed, under no circumstances shall the Scalar instruction deviate from the Scalar Power ISA Specification.

Summary of Compliancy Levels, each Level includes all lower levels:

- **Zero-Level**: Simple-V is not implemented (at all) in hardware. This Level is required to be listed because all capabilities of Simple-V must be Soft-emulatable by way of Illegal Instruction Traps.
- **Ultra-embedded**: `setvl` instruction. Register Files as Standard Power ISA. `scalar identity behaviour` implemented.
- **Embedded**: `svstep` instruction, and support for Hardware for-looping in both Horizontal-First and Vertical-First Mode as well as Predication (Single and Twin) for the GPRs r3, r10 and r30. CR-Field-based Predicates do not need to be added.
- **Embedded DSP/AV**: 128 registers, element-width overrides, and Saturation and Mapreduce/Iteration Modes.
- **High-end DSP/AV**: Same as Embedded-DSP/AV except also including Indexed and Offset REMAP capability.
- **3D/Advanced/Supercomputing**: all SV Branch instructions; crweird and vector-assist instructions (`set-before-first` etc); Swizzle Move instructions; Matrix, DCT/FFT and Indexing REMAP capability; Fail-First and Predicate-Result Modes.

These requirements within each Level constitute the minimum mandatory capabilities. It is also permitted that any Level include any part of a higher Compliancy Level. For example: an Embedded Level is permitted to have 128 GPRs, FPRs and CR Fields, but the Compliance Tests for Embedded will only test for 32. DSP/VPU Level is permitted to implement the DCT REMAP capability, but will not be permitted to declare meeting the 3D/Advanced Level unless implementing *all* REMAP Capabilities.

Power ISA Compliancy Levels

The SV Compliancy Levels have nothing to do with the Power ISA Compliancy Levels (SFS, SFFS, Linux, AIX). They are separate and independent. It is perfectly fine to implement Ultra-Embedded on AIX, and perfectly fine to implement 3D/Advanced on SFS. **Compliance with SV Levels does not convey or remove the obligation of Compliance with SFS/SFFS/Linux/AIX Levels and vice-versa**.

Zero-Level </>

This level exists to indicate the critical importance of all and any features attempted to be executed on hardware that has no support at all for Simple-V being **required** to raise Illegal Exceptions. **This includes existing Power ISA Implementations:** IBM POWER being the most notable.

With parts of the Power ISA being “silent executed” (hints for example), it is absolutely critical to have all capabilities of Simple-V sit within full Illegal Instruction space of existing and future Hardware.

Ultra-Embedded Level </>

This level exists as an entry-level into SVP64, most suited to resource constrained soft cores, or Hardware implementations where unit cost is a much higher priority than execution speed.

This level sets the bare minimum requirements, where everything with the exception of `scalar identity` and the `setvl` instruction may be software-emulated through JIT Translation or Illegal Instruction traps. SVSTATE, as effectively a Sub-Program-Counter, joins MSR and PC (CIA, NIA) as direct peers and must be switched on any context-switch (Trap or Exception)

- PC is saved/restored to/from SRR0
- MSR is saved/restored to/from SRR1
- SVSTATE **must** also be saved/restored to/from SVSRR1

Any implementation that implements Hypervisor Mode must also correspondingly follow the Power ISA Spec guidelines for HSRR0 and HSRR1, and must save/restore SVSTATE to/from HSVSRR1 in all circumstances involving save/restore to/from HSRR0 and HSRR1.

Illegal Instruction Trap **must** be raised on:

- Any SV instructions not implemented

- any unimplemented SV Context SPRs read or written
- all unimplemented uses of the SVP64 Prefix
- non-scalar-identity SVP64 instructions

Implementors are free and clear to implement any other features of SVP64 however only by meeting all of the mandatory requirements above will Compliance with the Ultra-Embedded Level be achieved.

Note that `scalar identity` is defined as being when the execution of an SVP64 Prefixed instruction is identical in every respect to Scalar non-prefixed, i.e. as if the Prefix had not been present. Additionally all SV SPRs must be zero and the 24-bit `RM` field must be zero.

Embedded Level </>

This level is more suitable for Hardware implementations where performance and power saving begins to matter. A second instruction, `svstep`, used by Vertical-First Mode, is required, as is hardware-level looping in Horizontal-First Mode. Illegal Instruction trap may not be used to emulate `svstep`.

At the bare minimum, Twin and Single Predication must be supported for at least the GPRs r3, r10 and r30. CR Field Predication may also be supported in hardware but only by also increasing the number of CR Fields to the required total 128.

Another important aspect is that when Rc=1 is set, CR Field Vector co-results are produced. Should these exceed CR7 (CR8-CR127) and the number of CR Fields has not been increased to 128 then an Illegal Instruction Trap must be raised. In practical terms, to avoid this occurrence in Embedded software, MAXVL should not exceed 8 for Arithmetic or Logical operations with Rc=1.

Zeroing on source and destination for Predicates must also be supported (sz, dz) however all other Modes (Saturation, Fail-First, Predicate-Result, Iteration/Reduction) are entirely optional. Implementation of Element-Width Overrides is also optional.

One of the important side-benefits of this SV Compliancy Level is that it brings Hardware-level support for Scalar Predication (VL=MAXVL=1) to the entire Scalar Power ISA, completely without modifying the Scalar Power ISA. The cost in software is that Predicated instructions are Prefixed to 64-bit.

DSP / Audio / Video Level </>

This level is best suited to high-performance power-efficient but specialist Compute workloads. 128 GPRs, FPRs and CR Fields are all required, as is element-width overrides to allow data processing down to the 8-bit level. SUBVL support (Sub-Vector vec2/3/4) is also required, as is Pack/Unpack EXTRA format (helps with Pixel and Audio Stream Structured data)

All SVP64 Modes must be implemented in hardware: Saturation in particular is a necessity for Audio DSP work. Reduction as well to assist with Audio/Video.

It is not mandatory for this Level to have DCT/FFT REMAP Capability in hardware but due to the high prevalence of DCT and FFT in Audio, Video and DSP workloads it is strongly recommended. Matrix (Dimensional) REMAP and Swizzle may also be useful to help with 24-bit (3 byte) Structured Audio Streams and are also recommended but not mandatory.

High-end DSP </>

In this Compliancy Level the benefits of the Offset and Index REMAP subsystem becomes worth its hardware cost. In lower-performing DSP and A/V workloads it is not.

3D / Advanced / Supercomputing </>

This Compliancy Level is for highest performance and energy efficiency. All aspects of SVP64 must be entirely implemented, in full, in Hardware. How that is achieved is entirely at the discretion of the implementor: there are no hard requirements of any kind on the level of performance, just as there are none in the Vulkan(TM) Specification.

Throughout the SV Specification however there are hints to Micro-Architects: byte-level write-enable lines on Register Files is strongly recommended, for example, in order to avoid unnecessary Read-Modify-Write cycles and additional Register Hazard Dependencies on fine-grained (8/16/32-bit) operations. Just as with SRAMs multiple write-enable lines may be raised to update higher-width elements.

Examples </>

Assuming that hardware implements scalar operations only, and implements predication but not elwidth overrides:

```
setvli r0, 4            # sets VL equal to 4
sv.addi r5, r0, 1       # raises an 0x700 trap
setvli r0, 1            # sets VL equal to 1
sv.addi r5, r0, 1       # gets executed by hardware
sv.addi/ew=8 r5, r0, 1  # raises an 0x700 trap
sv.ori/sm=EQ r5, r0, 1  # executed by hardware
```

The first `sv.addi` raises an illegal instruction trap because VL has been set to 4, and this is not supported. Likewise elwidth overrides if requested always raise illegal instruction traps.

Such an implementation would qualify for the “Ultra-Embedded” SV Level. It would not qualify for the “Embedded” level because when VL=4 an Illegal Exception is raised, and the Embedded Level requires full VL Loop support in hardware.

[[!tag standards]]
