RFC ls016 DCT / FFT Twin Butterfly instructions </>

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

- https://libre-soc.org/openpower/sv/rfc/ls016/
- https://git.openpower.foundation/isa/PowerISA/issues/129
- https://bugs.libre-soc.org/show_bug.cgi?id=1076

Severity: Major

Status: New

Date: 29 Apr 2023

Target: v3.2B

Source: v3.1B

Books and Section affected:

```
Book I Fixed-Point Instructions
Book I Floating-Point Instructions
Appendix E Power ISA sorted by opcode
Appendix F Power ISA sorted by version
Appendix G Power ISA sorted by Compliancy Subset
Appendix H Power ISA sorted by mnemonic
```

Summary

```
Instructions added: maddsubrs, fdmadd(s), ffmadd(s), ffadd(s), ffsub(s)
```

Submitter: Luke Leighton (Libre-SOC)

Requester: Libre-SOC

Impact on processor:

```
Addition of new Twin-Butterfly instructions, 3-in 2-out
```

Impact on software:

```
Requires support for new instructions in assembler, debuggers,
and related tools.  Greatly decreases instruction count in
Audio/Video, DSP, Scientific Computing extremely commonly used
algorithms (NTT, FFT, DFT, DCT)
```

Keywords:

```
DCT, FFT, NTT, DFT, Twin-Butterfly, Audio/Video, DSP, Radar,
Scientific Computing.
```

Motivation

The list of uses for DCT is enormous - well over a hundred. https://en.wikipedia.org/wiki/Discrete_cosine_transform#General_applications The number of uses for FFT, DFT, NTT is also equally known to be extremely high https://en.wikipedia.org/wiki/Fast_Fourier_transform#Applications ARM has already added `vqrdmulhq_s16/32` instructions as their inclusion in any ISA replaces **eight** equivalent non-Twin-Butterfly instructions, which are often loop-unrolled, resulting in L1 I-Cache stripmining as well as requiring far greater resources (double the number of intermediate Vector registers) or much more complex hardware to get efficient execution.

Notes and Observations:

1. Whilst it is easy to justify these high-value instructions they are sufficiently complex as to consider being optional SFFS.
2. Although they are 3-in 2-out the actual encoding is as a double-overwrite reducing the number of operands down to three (RT RA and RB) where RT is a Read-Modify-Write and an additional RS (normally RT+1) is implicit.
3. As with the biginteger set of 3-in 2-out instructions if Power ISA did not already have LD/ST-with-Update, Load/Store-Quad, and other RTp and RAp instructions, these instructions would not be proposed.
4. The read and write of two overlapping registers normally requires an intermediate register (similar to the justifcation for CAS - Compare-and-Swap). When Vectorized the situation becomes even worse: an entire *Vector* of intermediate temporaries is required. Thus *even if implemented inefficiently* requiring more cycles to complete (taking an extra cycle to write the second result) these instructions still save on resources.
5. Macro-op fusion equivalents of these instructions is *not possible* for exactly the same reason that the equivalent CAS sequence may not be macro-op fused. Full in-place Vectorized FFT and DCT algorithms *only* become possible due to these instructions atomically reading **both** Butterfly operands into internal Reservation Stations (exactly like CAS).
6. Although desirable (particularly to detect overflow) Rc=1 is hard to conceptualise. It is likely that instead, Simple-V "saturation" if enabled will create an Rc=1 CR.SO flag (including SVP64Single).
7. Saturated variants are **not** included: that is what SVP64 and SVP64Single provides (SVP64 provides a signed/unsigned saturation enhancement)
8. Unlike in ARM, (except FP Single), 8 16 and 32 bit variants are **not** included: that is what SVP64 and SVP64Single provides (SVP64 "redefines" "FP Single" to be "half of the register/element width").

Changes

Add the following entries to:

- the Appendices of Book I
- Book I 3.3.9.1 Fixed-Point Arithmetic DCT/FFT Twin-Butterfly Instructions
- Book I 4.6.6.3 Floating-Point DCT/FFT Twin-Butterfly Instructions
- Book I 1.6.1 and 1.6.2

Introduction </>

Although best used with SVP64 REMAP these instructions may be used in a Scalar-only context to save considerably on DCT, DFT and FFT processing. Whilst some hardware implementations may not necessarily implement them efficiently (slower Micro-coding) savings still come from the reduction in temporary registers as well as instruction count.

Rationale for Twin Butterfly Integer DCT Instruction(s) </>

The number of general-purpose uses for DCT is huge. The number of instructions needed instead of these Twin-Butterfly instructions is also huge (**eight**) and given that it is extremely common to explicitly loop-unroll them quantity hundreds to thousands of instructions are dismayingly common (for all ISAs).

The goal is to implement instructions that calculate the expression:

```
fdct_round_shift((a +/- b) * c)
```

For the single-coefficient butterfly instruction, and:

```
fdct_round_shift(a * c1  +/- b * c2)
```

For the double-coefficient butterfly instruction.

In a 32-bit context `fdct_round_shift` is defined as `ROUND_POWER_OF_TWO(x, 14)`

```
#define ROUND_POWER_OF_TWO(value, n) \
        (((value) + (1 << ((n)-1))) >> (n))
```

These instructions are at the core of **ALL** FDCT calculations in many major video codecs, including -but not limited to- VP8/VP9, AV1, etc. ARM includes special instructions to optimize these operations, although they are limited in precision: `vqrdmulhq_s16`/`vqrdmulhq_s32`.

The suggestion is to have a single instruction to calculate both values `((a + b) * c) >> N`, and `((a - b) * c) >> N`. The instruction will run in accumulate mode, so in order to calculate the 2-coeff version one would just have to call the same instruction with different order a, b and a different constant c.

Example taken from libvpx https://chromium.googlesource.com/webm/libvpx/+/refs/tags/v1.13.0/vpx_dsp/fwd_txfm.c#132:

```
#include <stdint.h>
#define ROUND_POWER_OF_TWO(value, n) \
        (((value) + (1 << ((n)-1))) >> (n))
void twin_int(int16_t *t, int16_t x0, int16_t x1, int16_t cospi_16_64) {
    t[0] = ROUND_POWER_OF_TWO((x0 + x1) * cospi_16_64, 14);
    t[1] = ROUND_POWER_OF_TWO((x0 - x1) * cospi_16_64, 14);
}
```

8 instructions are required - replaced by just the one (maddsubrs):

```
add 9,5,4
subf 5,5,4
mullw 9,9,6
mullw 5,5,6
addi 9,9,8192
addi 5,5,8192
srawi 9,9,14
srawi 5,5,14
```

Integer Butterfly Multiply Add/Sub FFT/DCT </>

Add the following to Book I Section 3.3.9.1

A-Form

```
|0     |6     |11      |16     |21      |26    |31 |
| PO   |  RT  |   RA   |   RB  |   SH   |   XO |Rc |
```

- maddsubrs RT,RA,RB,SH

Pseudo-code:

```
n <- SH
sum <- (RT[0] || RT) + (RA[0] || RA)
diff <- (RT[0] || RT) - (RA[0] || RA)
prod1 <- MULS(RB, sum)
prod2 <- MULS(RB, diff)
if n = 0 then
    prod1_lo <- prod1[XLEN+1:(XLEN*2)]
    prod2_lo <- prod2[XLEN+1:(XLEN*2)]
    RT <- prod1_lo
    RS <- prod2_lo
else
    round <- [0]*(XLEN*2 + 1)
    round[XLEN*2 - n + 1] <- 1
    prod1 <- prod1 + round
    prod2 <- prod2 + round
    res1 <- prod1[XLEN - n + 1:XLEN*2 - n]
    res2 <- prod2[XLEN - n + 1:XLEN*2 - n]
    RT <- res1
    RS <- res2
```

Similar to `RTp`, this instruction produces an implicit result, `RS`, which under Scalar circumstances is defined as `RT+1`. For SVP64 if `RT` is a Vector, `RS` begins immediately after the Vector `RT` where the length of `RT` is set by `SVSTATE.MAXVL` (Max Vector Length).

Special Registers Altered:

```
None
```

[DRAFT] Integer Butterfly Multiply Add and Round Shift FFT/DCT </>

A-Form

- maddrs RT,RA,RB,SH

Pseudo-code:

```
n <- SH
prod <- MULS(RB, RA)
if n = 0 then
    prod_lo <- prod[XLEN:(XLEN*2) - 1]
    RT <- (RT) + prod_lo
else
    res[0:XLEN*2-1] <- (EXTSXL((RT)[0], 1) || (RT)) + prod
    round <- [0]*XLEN*2
    round[XLEN*2 - n] <- 1
    res <- res + round
    RT <- res[XLEN - n:XLEN*2 - n -1]
```

Special Registers Altered:

```
None
```

[DRAFT] Integer Butterfly Multiply Sub and Round Shift FFT/DCT </>

A-Form

- msubrs RT,RA,RB,SH

Pseudo-code:

```
n <- SH
prod <- MULS(RB, RA)
if n = 0 then
    prod_lo <- prod[XLEN:(XLEN*2) - 1]
    RT <- (RT) - prod_lo
else
    res[0:XLEN*2-1] <- (EXTSXL((RT)[0], 1) || (RT)) - prod
    round <- [0]*XLEN*2
```

```
round[XLEN*2 - n] <- 1
res <- res + round
RT <- res[XLEN - n:XLEN*2 - n -1]
```

Special Registers Altered:

None

This pair of instructions is supposed to be used in complement to the maddsubrs to produce the double-coefficient butterfly instruction. In order for that to work, instead of passing c2 as coefficient, we have to pass c2-c1 instead.

In essence, we are calculating the quantity `a * c1 +/- b * c1` first, with `maddsubrs` *without* shifting (so `SH=0`) and then we add/sub `b * (c2-c1)` from the previous `RT`, and *then* do the shifting.

In the following example, assume `a` in `R1`, `b` in `R10`, `c1` in `R11` and `c2 - c1` in `R12`. The first instruction will put `a * c1 + b * c1` in `R1` (`RT`), `a * c1 - b * c1` in `RS` (here, `RS = RT +1`, so `R2`). Then, `maddrs` will add `b * (c2 - c1)` to `R1` (`RT`), and `msubrs` will subtract it from `R2` (`RS`), and then round shift right both quantities 14 bits:

```
maddsubrs 1,10,0,11
maddrs 1,10,12,14
msubrs 2,10,12,14
```

In scalar code, that would take ~16 instructions for both operations.

Twin Butterfly Floating-Point DCT and FFT Instruction(s) </>

Add the following to Book I Section 4.6.6.3

Floating-Point Twin Multiply-Add DCT [Single] </>

X-Form

```
|0     |6     |11      |16     |21      |31 |
| PO   |  FRT |  FRA   |  FRB  |   XO   |Rc |
```

- fdmadds FRT,FRA,FRB (Rc=0)

Pseudo-code:

```
FRS <- FPADD32(FRT, FRB)
sub <- FPSUB32(FRT, FRB)
FRT <- FPMUL32(FRA, sub)
```

The two IEEE754-FP32 operations

```
FRS <- [(FRT) + (FRB)]
FRT <- [(FRT) - (FRB)] * (FRA)
```

are simultaneously performed.

The Floating-Point operand in register FRT is added to the floating-point operand in register FRB and the result stored in FRS.

Using the exact same operand input register values from FRT and FRB that were used to create FRS, the Floating-Point operand in register FRB is subtracted from the floating-point operand in register FRT and the result then rounded before being multiplied by FRA to create an intermediate result that is stored in FRT.

The add into FRS is treated exactly as `fadds`. The creation of the result FRT is **not** the same as that of `fmsubs`, but is instead as if `fsubs` were performed first followed by `fmuls`. The creation of FRS and FRT are treated as parallel independent operations which occur at the same time.

Note that if Rc=1 an Illegal Instruction is raised. Rc=1 is `RESERVED`

Similar to `FRTp`, this instruction produces an implicit result, `FRS`, which under Scalar circumstances is defined as `FRT+1`. For SVP64 if `FRT` is a Vector, `FRS` begins immediately after the Vector `FRT` where the length of `FRT` is set by `SVSTATE.MAXVL` (Max Vector Length).

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
```

Floating-Point Multiply-Add FFT [Single] </>

X-Form

```
|0     |6     |11      |16     |21      |31 |
| PO   |  FRT |  FRA   |  FRB  |   XO   |Rc |
```

- ffmadds FRT,FRA,FRB (Rc=0)

Pseudo-code:

```
FRS <- FPMULADD32(FRT, FRA, FRB, -1, 1)
FRT <- FPMULADD32(FRT, FRA, FRB, 1, 1)
```

The two operations

```
FRS <- -([(FRT) * (FRA)] - (FRB))
FRT <-   [(FRT) * (FRA)] + (FRB)
```

are performed.

The floating-point operand in register FRT is multiplied by the floating-point operand in register FRA. The floating-point operand in register FRB is added to this intermediate result, and the intermediate stored in FRS.

Using the exact same values of FRT, FRT and FRB as used to create FRS, the floating-point operand in register FRT is multiplied by the floating-point operand in register FRA. The floating-point operand in register FRB is subtracted from this intermediate result, and the intermediate stored in FRT.

FRT is created as if a `fmadds` operation had been performed. FRS is created as if a `fnmsubs` operation had simultaneously been performed with the exact same register operands, in parallel, independently, at exactly the same time.

FRT is a Read-Modify-Write operation.

Note that if Rc=1 an Illegal Instruction is raised. Rc=1 is `RESERVED`

Similar to `FRTp`, this instruction produces an implicit result, `FRS`, which under Scalar circumstances is defined as `FRT+1`. For SVP64 if `FRT` is a Vector, `FRS` begins immediately after the Vector `FRT` where the length of `FRT` is set by `SVSTATE.MAXVL` (Max Vector Length).

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
```

Floating-Point Twin Multiply-Add DCT </>

X-Form

```
|0     |6     |11      |16     |21      |31 |
| PO   |  FRT |  FRA   |  FRB  |   XO   |Rc |
```

- fdmadd FRT,FRA,FRB (Rc=0)

Pseudo-code:

```
FRS <- FPADD64(FRT, FRB)
sub <- FPSUB64(FRT, FRB)
FRT <- FPMUL64(FRA, sub)
```

The two IEEE754-FP64 operations

```
FRS <- [(FRT) + (FRB)]
FRT <- [(FRT) - (FRB)] * (FRA)
```

are simultaneously performed.

The Floating-Point operand in register FRT is added to the floating-point operand in register FRB and the result stored in FRS.

Using the exact same operand input register values from FRT and FRB that were used to create FRS, the Floating-Point operand in register FRB is subtracted from the floating-point operand in register FRT and the result then rounded before being multiplied by FRA to create an intermediate result that is stored in FRT.

The add into FRS is treated exactly as `fadd`. The creation of the result FRT is **not** the same as that of `fmsub`, but is instead as if `fsub` were performed first followed by 'fmuls. The creation of FRS and FRT are treated as parallel independent operations which occur at the same time.

Note that if Rc=1 an Illegal Instruction is raised. Rc=1 is `RESERVED`

Similar to `FRTp`, this instruction produces an implicit result, `FRS`, which under Scalar circumstances is defined as `FRT+1`. For SVP64 if `FRT` is a Vector, `FRS` begins immediately after the Vector `FRT` where the length of `FRT` is set by `SVSTATE.MAXVL` (Max Vector Length).

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
```

Floating-Point Twin Multiply-Add FFT </>

X-Form

```
|0     |6     |11      |16     |21      |31 |
| PO   |  FRT |  FRA   |  FRB  |   XO   |Rc |
```

- ffmadd FRT,FRA,FRB (Rc=0)

Pseudo-code:

```
FRS <- FPMULADD64(FRT, FRA, FRB, -1, 1)
FRT <- FPMULADD64(FRT, FRA, FRB, 1, 1)
```

The two operations

```
FRS <- -([(FRT) * (FRA)] - (FRB))
FRT <-   [(FRT) * (FRA)] + (FRB)
```

are performed.

The floating-point operand in register FRT is multiplied by the floating-point operand in register FRA. The float- ing-point operand in register FRB is added to this intermediate result, and the intermediate stored in FRS.

Using the exact same values of FRT, FRT and FRB as used to create FRS, the floating-point operand in register FRT is multiplied by the floating-point operand in register FRA. The float- ing-point operand in register FRB is subtracted from this intermediate result, and the intermediate stored in FRT.

FRT is created as if a `fmadd` operation had been performed. FRS is created as if a `fnmsub` operation had simultaneously been performed with the exact same register operands, in parallel, independently, at exactly the same time.

FRT is a Read-Modify-Write operation.

Note that if Rc=1 an Illegal Instruction is raised. Rc=1 is `RESERVED`

Similar to `FRTp`, this instruction produces an implicit result, `FRS`, which under Scalar circumstances is defined as `FRT+1`. For SVP64 if `FRT` is a Vector, `FRS` begins immediately after the Vector `FRT` where the length of `FRT` is set by `SVSTATE.MAXVL` (Max Vector Length).

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
```

Floating-Point Add FFT/DCT [Single] </>

A-Form

```
|0     |6     |11      |16     |21      |26    |31 |
| PO   | FRT  |  FRA   |  FRB  |     /  |   XO |Rc |
```

- ffadds FRT,FRA,FRB (Rc=0)

Pseudo-code:

```
FRT <- FPADD32(FRA, FRB)
FRS <- FPSUB32(FRB, FRA)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
```

Floating-Point Add FFT/DCT [Double] </>

A-Form

```
|0     |6     |11      |16     |21      |26    |31 |
| PO   | FRT  |  FRA   |  FRB  |     /  |   XO |Rc |
```

- ffadd FRT,FRA,FRB (Rc=0)

Pseudo-code:

```
FRT <- FPADD64(FRA, FRB)
FRS <- FPSUB64(FRB, FRA)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
```

Floating-Point Subtract FFT/DCT [Single] </>

A-Form

```
|0     |6     |11      |16     |21      |26    |31 |
| PO   | FRT  |  FRA   |  FRB  |     /  |   XO |Rc |
```

- ffsubs FRT,FRA,FRB (Rc=0)

Pseudo-code:

```
FRT <- FPSUB32(FRB, FRA)
FRS <- FPADD32(FRA, FRB)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
```

Floating-Point Subtract FFT/DCT [Double] </>

A-Form

```
|0     |6     |11      |16     |21      |26    |31 |
| PO   | FRT  |  FRA   |  FRB  |     /  |   XO |Rc |
```

- ffsub FRT,FRA,FRB (Rc=0)

Pseudo-code:

```
FRT <- FPSUB64(FRB, FRA)
FRS <- FPADD64(FRA, FRB)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
```

Instruction Formats </>

Add the following entries to Book I 1.6.1 Word Instruction Formats:

A-FORM </>

```
|0     |6     |11      |16     |21      |26    |31 |
| PO   |  RT  |   RA   |   RB  |   SH   |   XO |Rc |
```

Add the following new fields to Book I 1.6.2 Word Instruction Fields:

```
SH (21:25)
    Field used to specify a shift amount.
    Formats: A
```

Appendices </>

```
Appendix E Power ISA sorted by opcode
Appendix F Power ISA sorted by version
Appendix G Power ISA sorted by Compliancy Subset
Appendix H Power ISA sorted by mnemonic
```

Form	Book	Page	Version	Mnemonic	Description
A	I	#	3.2B	maddsubrs	Integer DCT/FFT Twin-Butterfly
X	I	#	3.2B	fdmadds	FP DCT Twin-Butterfly Single
X	I	#	3.2B	ffmadds	FP FFT Twin-Butterfly Single
X	I	#	3.2B	fdmadds	FP DCT Twin-Butterfly Double
X	I	#	3.2B	ffmadds	FP FFT Twin-Butterfly Double
X	I	#	3.2B	ffadds	FP FFT Twin-Butterfly Single
X	I	#	3.2B	ffadd	FP FFT Twin-Butterfly Double
X	I	#	3.2B	ffsubs	FP FFT Twin-Butterfly Single
X	I	#	3.2B	ffsub	FP FFT Twin-Butterfly Double

[[!tag opf_rfc]]
