RFC ls006 FPR <-> GPR Move/Conversion </>

- 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/int_fp_mv/
- https://libre-soc.org/openpower/sv/rfc/ls006.fpintmv/
- https://bugs.libre-soc.org/show_bug.cgi?id=1015
- https://git.openpower.foundation/isa/PowerISA/issues/todo

Severity: Major

Status: New

Date: 09 Feb 2024 v2

Target: v3.2B

Source: v3.1B

Books and Section affected: **UPDATE**

- Book I 4.6.5 Floating-Point Move Instructions
- Book I 4.6.7.2 Floating-Point Convert To/From Integer 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

Single-precision Instructions added:

- `mffprs` – Move From FPR Single
- `mtfprs` – Move To FPR Single
- `ctfprs` – Convert To FPR Single

Identical (except Double-precision) Instructions added:

- `mffpr` – Move From FPR
- `mtfpr` – Move To FPR
- `cffpr` – Convert From FPR
- `ctfpr` – Convert To FPR

Submitter: Luke Leighton (Libre-SOC)

Requester: Libre-SOC

Impact on processor:

- Addition of three new Single-Precision GPR-FPR-based instructions
- Addition of four new Double-Precision GPR-FPR-based instructions

Impact on software:

- Requires support for new instructions in assembler, debuggers, and related tools.

Keywords:

```
GPR, FPR, Move, Conversion, ECMAScript, Saturating
```

Motivation

CPUs without VSX/VMX lack a way to efficiently transfer data between FPRs and GPRs, they need to go through memory, this proposal adds more efficient data transfer (both bitwise copy and Integer <-> FP conversion) instructions that transfer directly between FPRs and GPRs without needing to go through memory.

IEEE 754 does not specify what results are obtained when converting a NaN or out-of-range floating-point value to integer: consequently, different programming languages and ISAs have made different choices, making binary portability very difficult. Below is an overview of the different variants, listing the languages and hardware that implements each variant.

Notes and Observations:

- These instructions are present in many other ISAs.
- ECMAScript rounding as one instruction saves 32 scalar instructions including seven branch instructions.
- Both sets are orthogonal (no difference except being Single/Double). This allows IBM to follow the pre-existing precedent of allocating separate Major Opcodes (PO) for Double-precision and Single-precision respectively.

Changes

Add the following entries to:

- Book I 4.6.5 Floating-Point Move Instructions
- Book I 4.6.7.2 Floating-Point Convert To/From Integer Instructions
- Book I 1.6.1 and 1.6.2

Floating-point to Integer Conversion Overview </>

IEEE 754 does not specify what results are obtained when converting a NaN or out-of-range floating-point value to integer, so different programming languages and ISAs have made different choices. The different conversion modes supported by the `cffpr` instruction are as follows:

- P-Type:

  Used by most other PowerISA instructions, as well as commonly used floating-point to integer conversions on x86.

- S-Type:

  Used for WebAssembly's `trunc_sat_u`[1] and `trunc_sat_s`[2] instructions, as well as several notable programming languages:

  - Java's conversion from `float`/`double` to `long`/`int`[3]
  - Rust's `as` operator[4]
  - LLVM's `llvm.fptosi.sat`[5] and `llvm.fptoui.sat`[6] intrinsics
  - SPIR-V's OpenCL dialect's `OpConvertFToU`[7] and `OpConvertFToS`[8] instructions when decorated with the `SaturatedConversion`[9] decorator.

- E-Type:

  Used for ECMAScript's `ToInt32` abstract operation[10]. Also implemented in ARMv8.3A as the `FJCVTZS` instruction[11].

Floating-point to Integer Conversion Semantics Summary </>

Let `round` be the result of `bfp_ROUND_TO_INTEGER(rmode, input)`. Let `w` be the number of bits in the result's type. The result of Floating-point to Integer conversion is as follows:

Type	Result Sign	Category of rounding					
		NaN	+Inf	-Inf	> Max Possible Result	< Min Possible Result	Else
P	Unsigned	0	2^w - 1	0	2^w - 1	0	round
	Signed	-2^(w-1)	2^(w-1)-1	-2^(w-1)	2^(w-1)-1	-2^(w-1)	round
S	Unsigned	0	2^w - 1	0	2^w - 1	0	round
	Signed	0	2^(w-1)-1	-2^(w-1)	2^(w-1)-1	-2^(w-1)	round
E	Either	0			round & (2^w - 1)		

---

[1] WASM's `trunc_sat_u`: https://webassembly.github.io/spec/core/exec/numerics.html#op-trunc-sat-u
[2] WASM's `trunc_sat_s`: https://webassembly.github.io/spec/core/exec/numerics.html#op-trunc-sat-s
[3] Java `float`/`double` to `long`/`int` conversion: https://docs.oracle.com/javase/specs/jls/se16/html/jls-5.html#jls-5.1.3
[4] Rust's `as` operator: https://doc.rust-lang.org/1.70.0/reference/expressions/operator-expr.html#numeric-cast
[5] LLVM's `llvm.fptosi.sat` intrinsic: https://llvm.org/docs/LangRef.html#llvm-fptosi-sat-intrinsic
[6] LLVM's `llvm.fptoui.sat` intrinsic: https://llvm.org/docs/LangRef.html#llvm-fptoui-sat-intrinsic
[7] SPIR-V's `OpConvertFToU` instruction: https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.html#OpConvertFToU
[8] SPIR-V's `OpConvertFToS` instruction: https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.html#OpConvertFToS
[9] SPIR-V's `SaturatedConversion` decorator: https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.html#_a_id_decoration_a_decoration
[10] ECMAScript's `ToInt32` abstract operation: https://262.ecma-international.org/14.0/#sec-toint32
[11] ARM's `FJCVTZS` instruction: https://developer.arm.com/documentation/dui0801/g/hko1477562192868

Immediate Tables </>

Tables that are used by `mffpr[s][.]/mtfpr[s]/cffpr[o][.]/ctfpr[s][.]`:

`IT` – Integer Type </>

IT	Integer Type	Assembly Alias Mnemonic
0	Signed 32-bit	`<op>w`
1	Unsigned 32-bit	`<op>uw`
2	Signed 64-bit	`<op>d`
3	Unsigned 64-bit	`<op>ud`

`CVM` – Float to Integer Conversion Mode </>

CVM	rounding_mode	Semantics
000	from FPSCR	P-Type
001	Truncate	P-Type
010	from FPSCR	S-Type
011	Truncate	S-Type
100	from FPSCR	E-Type
101	Truncate	E-Type
rest	–	invalid

Move To/From Floating-Point Register Instructions </>

These instructions perform a copy from one register file to another, as if by using a GPR/FPR store, followed by a FPR/GPR load.

Move From Floating-Point Register </>

```
mffpr RT, FRB
mffpr. RT, FRB
```

0-5	6-10	11-15	16-20	21-30	31	Form
PO	RT	//	FRB	XO	Rc	X-Form

```
RT <- (FRB)
```

The contents of `FPR[FRB]` are placed into `GPR[RT]`.

Special Registers altered:

```
CR0     (if Rc=1)
```

Architecture Note:

`mffpr` is equivalent to the combination of `stfd` followed by `ld`.

Architecture Note:

`mffpr` is a separate instruction from `mfvsrd` because `mfvsrd` requires VSX which may not be available on simpler implementations. Additionally, SVP64 may treat VSX instructions differently than SFFS instructions in a future version of the architecture.

Move From Floating-Point Register Single </>

```
mffprs RT, FRB
mffprs. RT, FRB
```

0-5	6-10	11-15	16-20	21-30	31	Form
PO	RT	//	FRB	XO	Rc	X-Form

```
RT <- [0] * 32 || SINGLE((FRB))
```

The contents of `FPR[FRB]` are converted to BFP32 by using `SINGLE`, then zero-extended to 64-bits, and the result stored in `GPR[RT]`.

Special Registers altered:

```
CR0     (if Rc=1)
```

Architecture Note:

`mffprs` is equivalent to the combination of `stfs` followed by `lwz`.

Move To Floating-Point Register </>

```
mtfpr FRT, RB
```

0-5	6-10	11-15	16-20	21-30	31	Form
PO	FRT	//	RB	XO	//	X-Form

```
FRT <- (RB)
```

The contents of `GPR[RB]` are placed into `FPR[FRT]`.

Special Registers altered:

```
None
```

Architecture Note:

`mtfpr` is equivalent to the combination of `std` followed by `lfd`.

Architecture Note:

`mtfpr` is a separate instruction from `mtvsrd` because `mtvsrd` requires VSX which may not be available on simpler implementations. Additionally, SVP64 may treat VSX instructions differently than SFFS instructions in a future version of the architecture.

Move To Floating-Point Register Single </>

```
mtfprs FRT, RB
```

0-5	6-10	11-15	16-20	21-30	31	Form
PO	FRT	//	RB	XO	//	X-Form

```
FRT <- DOUBLE((RB)[32:63])
```

The contents of bits 32:63 of `GPR[RB]` are converted to BFP64 by using `DOUBLE`, then the result is stored in `GPR[RT]`.

Special Registers altered:

```
None
```

Architecture Note:

`mtfprs` is equivalent to the combination of `stw` followed by `lfs`.

Conversion To/From Floating-Point Register Instructions </>

Convert To Floating-Point Register </>

```
ctfpr FRT, RB, IT
ctfpr. FRT, RB, IT
```

0-5	6-10	11-12	13-15	16-20	21-30	31	Form
PO	FRT	IT	//	RB	XO	Rc	X-Form

```
if IT[0] = 0 then  # 32-bit int -> 64-bit float
    # rounding never necessary, so don't touch FPSCR
    # based off xvcvsxwdp
    if IT = 0 then  # Signed 32-bit
        src <- bfp_CONVERT_FROM_SI32((RB)[32:63])
    else  # IT = 1 -- Unsigned 32-bit
        src <- bfp_CONVERT_FROM_UI32((RB)[32:63])
    FRT <- bfp64_CONVERT_FROM_BFP(src)
else
    # rounding may be necessary. based off xscvuxdsp
    reset_xflags()
    switch(IT)
        case(0):  # Signed 32-bit
            src <- bfp_CONVERT_FROM_SI32((RB)[32:63])
        case(1):  # Unsigned 32-bit
            src <- bfp_CONVERT_FROM_UI32((RB)[32:63])
        case(2):  # Signed 64-bit
            src <- bfp_CONVERT_FROM_SI64((RB))
        default:  # Unsigned 64-bit
            src <- bfp_CONVERT_FROM_UI64((RB))
    rnd <- bfp_ROUND_TO_BFP64(0b0, FPSCR.RN, src)
    result <- bfp64_CONVERT_FROM_BFP(rnd)
    cls <- fprf_CLASS_BFP64(result)

    if xx_flag = 1 then SetFX(FPSCR.XX)

    FRT <- result
    FPSCR.FPRF <- cls
    FPSCR.FR <- inc_flag
    FPSCR.FI <- xx_flag
```

Convert from a unsigned/signed 32/64-bit integer in RB to a 64-bit float in FRT.

If converting from a unsigned/signed 32-bit integer to a 64-bit float, rounding is never necessary, so `FPSCR` is unmodified and exceptions are never raised. Otherwise, `FPSCR` is modified and exceptions are raised as usual.

Rc=1 tests FRT and sets CR1, exactly like all other Scalar Floating-Point operations.

Special Registers altered:

```
CR1               (if Rc=1)
FPRF FR FI FX XX  (if IT[0]=1)
```

Assembly Aliases </>

Assembly Alias	Full Instruction
`ctfprw FRT, RB`	`ctfpr FRT, RB, 0`
`ctfprw. FRT, RB`	`ctfpr. FRT, RB, 0`
`ctfpruw FRT, RB`	`ctfpr FRT, RB, 1`
`ctfpruw. FRT, RB`	`ctfpr. FRT, RB, 1`
`ctfprd FRT, RB`	`ctfpr FRT, RB, 2`
`ctfprd. FRT, RB`	`ctfpr. FRT, RB, 2`
`ctfprud FRT, RB`	`ctfpr FRT, RB, 3`
`ctfprud. FRT, RB`	`ctfpr. FRT, RB, 3`

Convert To Floating-Point Register Single </>

```
ctfprs FRT, RB, IT
ctfprs. FRT, RB, IT
```

0-5	6-10	11-12	13-15	16-20	21-30	31	Form
PO	FRT	IT	//	RB	XO	Rc	X-Form

```
# rounding may be necessary. based off xscvuxdsp
reset_xflags()
switch(IT)
    case(0):  # Signed 32-bit
        src <- bfp_CONVERT_FROM_SI32((RB)[32:63])
    case(1):  # Unsigned 32-bit
        src <- bfp_CONVERT_FROM_UI32((RB)[32:63])
    case(2):  # Signed 64-bit
        src <- bfp_CONVERT_FROM_SI64((RB))
    default:  # Unsigned 64-bit
        src <- bfp_CONVERT_FROM_UI64((RB))
rnd <- bfp_ROUND_TO_BFP32(FPSCR.RN, src)
result32 <- bfp32_CONVERT_FROM_BFP(rnd)
cls <- fprf_CLASS_BFP32(result32)
result <- DOUBLE(result32)

if xx_flag = 1 then SetFX(FPSCR.XX)

FRT <- result
FPSCR.FPRF <- cls
FPSCR.FR <- inc_flag
FPSCR.FI <- xx_flag
```

Convert from a unsigned/signed 32/64-bit integer in RB to a 32-bit float in FRT, following the usual 32-bit float in 64-bit float format. FPSCR is modified and exceptions are raised as usual.

Rc=1 tests FRT and sets CR1, exactly like all other Scalar Floating-Point operations.

Special Registers altered:

```
CR1     (if Rc=1)
FPRF FR FI FX XX
```

Assembly Aliases </>

Assembly Alias	Full Instruction
ctfprws FRT, RB	ctfpr FRT, RB, 0
ctfprws. FRT, RB	ctfpr. FRT, RB, 0
ctfpruws FRT, RB	ctfpr FRT, RB, 1
ctfpruws. FRT, RB	ctfpr. FRT, RB, 1
ctfprds FRT, RB	ctfpr FRT, RB, 2
ctfprds. FRT, RB	ctfpr. FRT, RB, 2
ctfpruds FRT, RB	ctfpr FRT, RB, 3
ctfpruds. FRT, RB	ctfpr. FRT, RB, 3

Convert From Floating-Point Register </>

```
cffpr RT, FRB, CVM, IT
cffpr. RT, FRB, CVM, IT
cffpro RT, FRB, CVM, IT
cffpro. RT, FRB, CVM, IT
```

0-5	6-10	11-12	13-15	16-20	21	22-30	31	Form
PO	RT	IT	CVM	FRB	OE	XO	Rc	XO-Form

```
# based on xscvdpuxws
reset_xflags()
src <- bfp_CONVERT_FROM_BFP64((FRB))

switch(IT)
    case(0):  # Signed 32-bit
        range_min <- bfp_CONVERT_FROM_SI32(0x8000_0000)
        range_max <- bfp_CONVERT_FROM_SI32(0x7FFF_FFFF)
        js_mask <- 0x0000_0000_FFFF_FFFF
    case(1):  # Unsigned 32-bit
        range_min <- bfp_CONVERT_FROM_UI32(0)
        range_max <- bfp_CONVERT_FROM_UI32(0xFFFF_FFFF)
        js_mask <- 0x0000_0000_FFFF_FFFF
    case(2):  # Signed 64-bit
        range_min <- bfp_CONVERT_FROM_SI64(-0x8000_0000_0000_0000)
        range_max <- bfp_CONVERT_FROM_SI64(0x7FFF_FFFF_FFFF_FFFF)
        js_mask <- 0xFFFF_FFFF_FFFF_FFFF
    default:  # Unsigned 64-bit
        range_min <- bfp_CONVERT_FROM_UI64(0)
        range_max <- bfp_CONVERT_FROM_UI64(0xFFFF_FFFF_FFFF_FFFF)
        js_mask <- 0xFFFF_FFFF_FFFF_FFFF

if (CVM[2] = 1) | (FPSCR.RN = 0b01) then
    rnd <- bfp_ROUND_TO_INTEGER_TRUNC(src)
else if FPSCR.RN = 0b00 then
    rnd <- bfp_ROUND_TO_INTEGER_NEAR_EVEN(src)
else if FPSCR.RN = 0b10 then
    rnd <- bfp_ROUND_TO_INTEGER_CEIL(src)
else if FPSCR.RN = 0b11 then
    rnd <- bfp_ROUND_TO_INTEGER_FLOOR(src)

switch(CVM)
    case(0, 1):  # P-Type
        if IsNaN(rnd) then
            result <- si64_CONVERT_FROM_BFP(range_min)
        else if bfp_COMPARE_GT(rnd, range_max) then
            result <- ui64_CONVERT_FROM_BFP(range_max)
        else if bfp_COMPARE_LT(rnd, range_min) then
            result <- si64_CONVERT_FROM_BFP(range_min)
        else if IT[1] = 1 then  # Unsigned 32/64-bit
            result <- ui64_CONVERT_FROM_BFP(rnd)
        else  # Signed 32/64-bit
            result <- si64_CONVERT_FROM_BFP(rnd)
    case(2, 3):  # S-Type
        if IsNaN(rnd) then
            result <- [0] * 64
        else if bfp_COMPARE_GT(rnd, range_max) then
            result <- ui64_CONVERT_FROM_BFP(range_max)
        else if bfp_COMPARE_LT(rnd, range_min) then
            result <- si64_CONVERT_FROM_BFP(range_min)
        else if IT[1] = 1 then  # Unsigned 32/64-bit
            result <- ui64_CONVERT_FROM_BFP(rnd)
        else  # Signed 32/64-bit
            result <- si64_CONVERT_FROM_BFP(rnd)
    default:  # E-Type
        # CVM = 6, 7 are illegal instructions
        # using a 128-bit intermediate works here because the largest type
        # this instruction can convert from has 53 significand bits, and
        # the largest type this instruction can convert to has 64 bits,
        # and the sum of those is strictly less than the 128 bits of the
        # intermediate result.
        limit <- bfp_CONVERT_FROM_UI128([1] * 128)
        if IsInf(rnd) | IsNaN(rnd) then
            result <- [0] * 64
        else if bfp_COMPARE_GT(bfp_ABSOLUTE(rnd), limit) then
            result <- [0] * 64
        else
```

```
            result128 <- si128_CONVERT_FROM_BFP(rnd)
            result <- result128[64:127] & js_mask

    switch(IT)
        case(0):  # Signed 32-bit
            result <- EXTS64(result[32:63])
            result_bfp <- bfp_CONVERT_FROM_SI32(result[32:63])
        case(1):  # Unsigned 32-bit
            result <- EXTZ64(result[32:63])
            result_bfp <- bfp_CONVERT_FROM_UI32(result[32:63])
        case(2):  # Signed 64-bit
            result_bfp <- bfp_CONVERT_FROM_SI64(result)
        default:  # Unsigned 64-bit
            result_bfp <- bfp_CONVERT_FROM_UI64(result)

    overflow <- 0  # signals SO only when OE = 1
    if IsNaN(src) | ¬bfp_COMPARE_EQ(rnd, result_bfp) then
        overflow <- 1  # signals SO only when OE = 1
        vxcvi_flag <- 1
        xx_flag <- 0
        inc_flag <- 0
    else
        xx_flag <- ¬bfp_COMPARE_EQ(src, result_bfp)
        inc_flag <- bfp_COMPARE_GT(bfp_ABSOLUTE(result_bfp), bfp_ABSOLUTE(src))

    if vxsnan_flag = 1 then SetFX(FPSCR.VXSNAN)
    if vxcvi_flag = 1 then SetFX(FPSCR.VXCVI)
    if xx_flag = 1 then SetFX(FPSCR.XX)

    vx_flag <- vxsnan_flag | vxcvi_flag
    vex_flag <- FPSCR.VE & vx_flag
    if vex_flag = 0 then
        RT <- result
        FPSCR.FPRF <- undefined
        FPSCR.FR <- inc_flag
        FPSCR.FI <- xx_flag
    else
        FPSCR.FR <- 0
        FPSCR.FI <- 0
```

Convert from 64-bit float in FRB to a unsigned/signed 32/64-bit integer in RT, with the conversion overflow/rounding semantics following the chosen `CVM` value. `FPSCR` is modified and exceptions are raised as usual.

This instruction has an Rc=1 mode which sets CR0 in the normal way for any instructions producing a GPR result. Additionally, when OE=1, if the numerical value of the FP number is not 100% accurately preserved (due to truncation or saturation and including when the FP number was NaN) then this is considered to be an Integer Overflow condition, and CR0.SO, XER.SO and XER.OV are all set as normal for any GPR instructions that overflow. When `RT` is not written (`vex_flag = 1`), all CR0 bits except SO are undefined.

Special Registers altered:

```
CR0              (if Rc=1)
XER SO, OV, OV32 (if OE=1)
FPRF=0bUUUUU FR FI FX XX VXSNAN VXCV
```

Assembly Aliases </>

Assembly Alias	Full Instruction
`cffprw RT, FRB, CVM`	`cffpr RT, FRB, CVM, 0`
`cffprw. RT, FRB, CVM`	`cffpr. RT, FRB, CVM, 0`
`cffprwo RT, FRB, CVM`	`cffpro RT, FRB, CVM, 0`
`cffprwo. RT, FRB, CVM`	`cffpro. RT, FRB, CVM, 0`
`cffpruw RT, FRB, CVM`	`cffpr RT, FRB, CVM, 1`
`cffpruw. RT, FRB, CVM`	`cffpr. RT, FRB, CVM, 1`
`cffpruwo RT, FRB, CVM`	`cffpro RT, FRB, CVM, 1`
`cffpruwo. RT, FRB, CVM`	`cffpro. RT, FRB, CVM, 1`
`cffprd RT, FRB, CVM`	`cffpr RT, FRB, CVM, 2`
`cffprd. RT, FRB, CVM`	`cffpr. RT, FRB, CVM, 2`
`cffprdo RT, FRB, CVM`	`cffpro RT, FRB, CVM, 2`
`cffprdo. RT, FRB, CVM`	`cffpro. RT, FRB, CVM, 2`
`cffprud RT, FRB, CVM`	`cffpr RT, FRB, CVM, 3`
`cffprud. RT, FRB, CVM`	`cffpr. RT, FRB, CVM, 3`
`cffprudo RT, FRB, CVM`	`cffpro RT, FRB, CVM, 3`
`cffprudo. RT, FRB, CVM`	`cffpro. RT, FRB, CVM, 3`

Instruction Formats </>

Add the following entries to Book I 1.6.1.19 XO-FORM:

```
|0    |6   |11  |13   |16   |21  |22  |31  |
| PO  | RT | IT | CVM | FRB | OE | XO | Rc |
```

Add the following entries to Book I 1.6.1.15 X-FORM:

```
|0   |6    |11  |13  |16   |21  |31  |
| PO | FRT | IT | // | RB  | XO | Rc |
| PO | FRT | //      | RB  | XO | Rc |
| PO | RT  | //      | FRB | XO | Rc |
```

Instruction Fields </>

Add XO to FRB's Formats list in Book I 1.6.2 Word Instruction Fields.

Add XO to FRT's Formats list in Book I 1.6.2 Word Instruction Fields.

Add new fields:

```
IT (11:12)
    Field used to specify integer type for FPR <-> GPR conversions.

    Formats: X, XO

CVM (13:15)
    Field used to specify conversion mode for
    integer -> floating-point conversion.

    Formats: XO
```

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
VA	I	#	3.2B	todo	

[[!tag opf_rfc]]