RFC ls006 FPR <-> GPR Move/Conversion

URLs:

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

Severity: Major

Status: New

Date: 20 Oct 2022

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:

- `fmvtgs` – Single-Precision Floating Move To GPR
- `fmvfgs` – Single-Precision Floating Move From GPR
- `fcvttgs` – Single-Precision Floating Convert To Integer In GPR
- `fcvtfgs` – Single-Precision Floating Convert From Integer In GPR

Identical (except Double-precision) Instructions added:

- `fmvtg` – Double-Precision Floating Move To GPR
- `fmvfg` – Double-Precision Floating Move From GPR
- `fcvttg` – Double-Precision Floating Convert To Integer In GPR
- `fcvtfg` – Double-Precision Floating Convert From Integer In GPR

Submitter: Luke Leighton (Libre-SOC)

Requester: Libre-SOC

Impact on processor:

- Addition of four 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, JavaScript
```

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 doesn't 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. 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.
- JavaScript 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

Immediate Tables

Tables that are used by `fmvtg[s][.]/fmvfg[s][.]/fcvt[s]tg[o][.]/fcvtfg[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`	OpenPower semantics
001	Truncate	OpenPower semantics
010	from `FPSCR`	Java/Saturating semantics
011	Truncate	Java/Saturating semantics
100	from `FPSCR`	JavaScript semantics
101	Truncate	JavaScript semantics
rest	–	illegal instruction trap for now

Moves

These instructions perform a straight unaltered bit-level copy from one Register File to another.

Floating Move To GPR

```
fmvtg RT, FRB
fmvtg. RT, FRB
```

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

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

Move a 64-bit float from a FPR to a GPR, just copying bits of the IEEE 754 representation directly. This is equivalent to `stfd` followed by `ld`. As `fmvtg` is just copying bits, `FPSCR` is not affected in any way.

Rc=1 tests RT and sets CR0, exactly like all other Scalar Fixed-Point operations.

Special Registers altered:

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

Floating Move To GPR Single

```
fmvtgs RT, FRB
fmvtgs. 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))  # SINGLE since that's what stfs uses
```

Move a BFP32 from a FPR to a GPR, by using `SINGLE` to extract the standard `BFP32` form from FRB and zero-extending the result to 64-bits and storing to RT. This is equivalent to `stfs` followed by `lwz`. As `fmvtgs` is just copying the BFP32 form, `FPSCR` is not affected in any way.

Rc=1 tests RT and sets CR0, exactly like all other Scalar Fixed-Point operations.

Special Registers altered:

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

Double-Precision Floating Move From GPR

```
fmvfg FRT, RB
fmvfg. FRT, RB
```

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

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

move a 64-bit float from a GPR to a FPR, just copying bits of the IEEE 754 representation directly. This is equivalent to `std` followed by `lfd`. As `fmvfg` is just copying bits, `FPSCR` is not affected in any way.

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

Special Registers altered:

```
CR1     (if Rc=1)
```

Floating Move From GPR Single

```
fmvfgs FRT, RB
fmvfgs. FRT, RB
```

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

```
FRT <- DOUBLE((RB)[32:63])  # DOUBLE since that's what lfs uses
```

Move a BFP32 from a GPR to a FPR, by using `DOUBLE` on the least significant 32-bits of RB to do the standard BFP32 in BFP64 trick and store the result in FRT. This is equivalent to `stw` followed by `lfs`. As `fmvfgs` is just copying the BFP32 form, `FPSCR` is not affected in any way.

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

Special Registers altered:

```
CR1     (if Rc=1)
```

Conversions

Unlike the move instructions these instructions perform conversions between Integer and Floating Point. Truncation can therefore occur, as well as exceptions.

Double-Precision Floating Convert From Integer In GPR

```
fcvtfg FRT, RB, IT
fcvtfg. 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(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
`fcvtfgw FRT, RB`	`fcvtfg FRT, RB, 0`
`fcvtfgw. FRT, RB`	`fcvtfg. FRT, RB, 0`
`fcvtfguw FRT, RB`	`fcvtfg FRT, RB, 1`
`fcvtfguw. FRT, RB`	`fcvtfg. FRT, RB, 1`
`fcvtfgd FRT, RB`	`fcvtfg FRT, RB, 2`
`fcvtfgd. FRT, RB`	`fcvtfg. FRT, RB, 2`
`fcvtfgud FRT, RB`	`fcvtfg FRT, RB, 3`
`fcvtfgud. FRT, RB`	`fcvtfg. FRT, RB, 3`

Floating Convert From Integer In GPR Single

```
fcvtfgs FRT, RB, IT
fcvtfgs. 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
`fcvtfgws FRT, RB`	`fcvtfg FRT, RB, 0`
`fcvtfgws. FRT, RB`	`fcvtfg. FRT, RB, 0`
`fcvtfguws FRT, RB`	`fcvtfg FRT, RB, 1`
`fcvtfguws. FRT, RB`	`fcvtfg. FRT, RB, 1`
`fcvtfgds FRT, RB`	`fcvtfg FRT, RB, 2`
`fcvtfgds. FRT, RB`	`fcvtfg. FRT, RB, 2`
`fcvtfguds FRT, RB`	`fcvtfg FRT, RB, 3`
`fcvtfguds. FRT, RB`	`fcvtfg. FRT, RB, 3`

Floating-point to Integer Conversion Overview

IEEE 754 doesn't 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. Below is an overview of the different variants, listing the languages and hardware that implements each variant.

For convenience, we will give those different conversion semantics names based on which common ISA or programming language uses them, since there may not be an established name for them:

Standard OpenPower conversion

This conversion performs "saturation with NaN converted to minimum valid integer". This is also exactly the same as the x86 ISA conversion semantics. OpenPOWER however has instructions for both:

- rounding mode read from FPSCR
- rounding mode always set to truncate

Java/Saturating conversion

For the sake of simplicity, the FP -> Integer conversion semantics generalized from those used by Java's semantics (and Rust's `as` operator) will be referred to as Java/Saturating conversion semantics.

Those same semantics are used in some way by all of the following languages (not necessarily for the default conversion method):

- Java's FP -> Integer conversion (only for long/int results)
- Rust's FP -> Integer conversion using the `as` operator
- LLVM's `llvm.fptosi.sat` and `llvm.fptoui.sat` intrinsics
- SPIR-V's OpenCL dialect's `OpConvertFToU` and `OpConvertFToS` instructions when decorated with the `SaturatedConversion` decorator.
- WebAssembly has also introduced trunc_sat_u and trunc_sat_s

JavaScript conversion

For the sake of simplicity, the FP -> Integer conversion semantics generalized from those used by JavaScripts's `ToInt32` abstract operation will be referred to as JavaScript conversion semantics.

This instruction is present in ARM assembler as FJCVTZS https://developer.arm.com/documentation/dui0801/g/hko1477562192868

Rc=1 and OE=1

All of these instructions have 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.

FP to Integer Conversion Simplified Pseudo-code

Key for pseudo-code:

term	result type	definition
`fp`	–	`f32` or `f64` (or other types from SimpleV)
`int`	–	`u32`/`u64`/`i32`/`i64` (or other types from SimpleV)
`uint`	–	the unsigned integer of the same bit-width as `int`
`int::BITS`	`int`	the bit-width of `int`
`uint::MIN_VALUE`	`uint`	the minimum value `uint` can store: `0`
`uint::MAX_VALUE`	`uint`	the maximum value `uint` can store: `2^int::BITS - 1`
`int::MIN_VALUE`	`int`	the minimum value `int` can store : `-2^(int::BITS-1)`
`int::MAX_VALUE`	`int`	the maximum value `int` can store : `2^(int::BITS-1) - 1`
`int::VALUE_COUNT`	Integer	the number of different values `int` can store (`2^int::BITS`). too big to fit in `int`.
`rint(fp, rounding_mode)`	`fp`	rounds the floating-point value `fp` to an integer according to rounding mode `rounding_mode`

OpenPower conversion semantics (section A.2 page 1009 (page 1035) of Power ISA v3.1B):

```
def fp_to_int_open_power<fp, int>(v: fp) -> int:
    if v is NaN:
        return int::MIN_VALUE
    if v >= int::MAX_VALUE:
        return int::MAX_VALUE
    if v <= int::MIN_VALUE:
        return int::MIN_VALUE
    return (int)rint(v, rounding_mode)
```

Java/Saturating conversion semantics (only for long/int results) (with adjustment to add non-truncate rounding modes):

```
def fp_to_int_java_saturating<fp, int>(v: fp) -> int:
    if v is NaN:
        return 0
    if v >= int::MAX_VALUE:
        return int::MAX_VALUE
    if v <= int::MIN_VALUE:
        return int::MIN_VALUE
    return (int)rint(v, rounding_mode)
```

Section 7.1 of the ECMAScript / JavaScript conversion semantics (with adjustment to add non-truncate rounding modes):

```
def fp_to_int_java_script<fp, int>(v: fp) -> int:
    if v is NaN or infinite:
        return 0
    v = rint(v, rounding_mode)  # assume no loss of precision in result
    v = v mod int::VALUE_COUNT  # 2^32 for i32, 2^64 for i64, result is non-negative
    bits = (uint)v
    return (int)bits
```

Double-Precision Floating Convert To Integer In GPR

```
fcvttg RT, FRB, CVM, IT
fcvttg. RT, FRB, CVM, IT
fcvttgo RT, FRB, CVM, IT
fcvttgo. 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 <- 0xFFFF_FFFF
    case(1):  # Unsigned 32-bit
        range_min <- bfp_CONVERT_FROM_UI32(0)
        range_max <- bfp_CONVERT_FROM_UI32(0xFFFF_FFFF)
        js_mask <- 0xFFFF_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):  # OpenPower semantics
        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(range_max)
        else  # Signed 32/64-bit
            result <- si64_CONVERT_FROM_BFP(range_max)
    case(2, 3):  # Java/Saturating semantics
        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(range_max)
        else  # Signed 32/64-bit
            result <- si64_CONVERT_FROM_BFP(range_max)
    default:  # JavaScript semantics
        # CVM = 6, 7 are illegal instructions
        # this works because the largest type we try to convert from has
        # 53 significand bits, and the largest type we try to 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)

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
    if IsNaN(src) | ¬bfp_COMPARE_EQ(src, result_bfp) then
        overflow <- 1  # signals SO only when OE = 1
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.

These instructions have 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.

Special Registers altered:

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

Assembly Aliases

Assembly Alias	Full Instruction
`fcvttgw RT, FRB, CVM`	`fcvttg RT, FRB, CVM, 0`
`fcvttgw. RT, FRB, CVM`	`fcvttg. RT, FRB, CVM, 0`
`fcvttgwo RT, FRB, CVM`	`fcvttgo RT, FRB, CVM, 0`
`fcvttgwo. RT, FRB, CVM`	`fcvttgo. RT, FRB, CVM, 0`
`fcvttguw RT, FRB, CVM`	`fcvttg RT, FRB, CVM, 1`
`fcvttguw. RT, FRB, CVM`	`fcvttg. RT, FRB, CVM, 1`
`fcvttguwo RT, FRB, CVM`	`fcvttgo RT, FRB, CVM, 1`
`fcvttguwo. RT, FRB, CVM`	`fcvttgo. RT, FRB, CVM, 1`
`fcvttgd RT, FRB, CVM`	`fcvttg RT, FRB, CVM, 2`
`fcvttgd. RT, FRB, CVM`	`fcvttg. RT, FRB, CVM, 2`
`fcvttgdo RT, FRB, CVM`	`fcvttgo RT, FRB, CVM, 2`
`fcvttgdo. RT, FRB, CVM`	`fcvttgo. RT, FRB, CVM, 2`
`fcvttgud RT, FRB, CVM`	`fcvttg RT, FRB, CVM, 3`
`fcvttgud. RT, FRB, CVM`	`fcvttg. RT, FRB, CVM, 3`
`fcvttgudo RT, FRB, CVM`	`fcvttgo RT, FRB, CVM, 3`
`fcvttgudo. RT, FRB, CVM`	`fcvttgo. RT, FRB, CVM, 3`

Floating Convert Single To Integer In GPR

```
fcvtstg RT, FRB, CVM, IT
fcvtstg. RT, FRB, CVM, IT
fcvtstgo RT, FRB, CVM, IT
fcvtstgo. 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_BFP32(SINGLE((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 <- 0xFFFF_FFFF
    case(1):  # Unsigned 32-bit
        range_min <- bfp_CONVERT_FROM_UI32(0)
        range_max <- bfp_CONVERT_FROM_UI32(0xFFFF_FFFF)
        js_mask <- 0xFFFF_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):  # OpenPower semantics
        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(range_max)
        else  # Signed 32/64-bit
            result <- si64_CONVERT_FROM_BFP(range_max)
    case(2, 3):  # Java/Saturating semantics
        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(range_max)
        else  # Signed 32/64-bit
            result <- si64_CONVERT_FROM_BFP(range_max)
    default:  # JavaScript semantics
        # CVM = 6, 7 are illegal instructions
        # this works because the largest type we try to convert from has
        # 53 significand bits, and the largest type we try to 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)

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
    if IsNaN(src) | ¬bfp_COMPARE_EQ(src, result_bfp) then
        overflow <- 1  # signals SO only when OE = 1
else
    FPSCR.FR <- 0
    FPSCR.FI <- 0
```

Convert from 32-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, following the usual 32-bit float in 64-bit float format. `FPSCR` is modified and exceptions are raised as usual.

These instructions have 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.

Special Registers altered:

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

Assembly Aliases

Assembly Alias	Full Instruction
`fcvtstgw RT, FRB, CVM`	`fcvtstg RT, FRB, CVM, 0`
`fcvtstgw. RT, FRB, CVM`	`fcvtstg. RT, FRB, CVM, 0`
`fcvtstgwo RT, FRB, CVM`	`fcvtstgo RT, FRB, CVM, 0`
`fcvtstgwo. RT, FRB, CVM`	`fcvtstgo. RT, FRB, CVM, 0`
`fcvtstguw RT, FRB, CVM`	`fcvtstg RT, FRB, CVM, 1`
`fcvtstguw. RT, FRB, CVM`	`fcvtstg. RT, FRB, CVM, 1`
`fcvtstguwo RT, FRB, CVM`	`fcvtstgo RT, FRB, CVM, 1`
`fcvtstguwo. RT, FRB, CVM`	`fcvtstgo. RT, FRB, CVM, 1`
`fcvtstgd RT, FRB, CVM`	`fcvtstg RT, FRB, CVM, 2`
`fcvtstgd. RT, FRB, CVM`	`fcvtstg. RT, FRB, CVM, 2`
`fcvtstgdo RT, FRB, CVM`	`fcvtstgo RT, FRB, CVM, 2`
`fcvtstgdo. RT, FRB, CVM`	`fcvtstgo. RT, FRB, CVM, 2`
`fcvtstgud RT, FRB, CVM`	`fcvtstg RT, FRB, CVM, 3`
`fcvtstgud. RT, FRB, CVM`	`fcvtstg. RT, FRB, CVM, 3`
`fcvtstgudo RT, FRB, CVM`	`fcvtstgo RT, FRB, CVM, 3`
`fcvtstgudo. RT, FRB, CVM`	`fcvtstgo. 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]]