The smallFloat Extensions for RISC-V

`Xf16`, `Xf16alt`, `Xf8`, `Xfvec` and `Xfaux` Extensions

Document Version 0.5

Stefan Mach
Integrated Systems Laboratory, ETH Zrich
`smach@iis.ee.ethz.ch`

September 6, 2018

Contributors to all versions of this spec in alphabetical order (please contact author to suggest corrections): Eric Flamand, Fabian Schuiki, Giuseppe Tagliavini.

Preface

This document is closely based on [1], extending the floating point capabilities of the ISA and originally intended for implementation in the PULP project[1]. The goal is providing “smallFloat”s, a set of minifloat formats, both in scalar and vectorial (packed-SIMD) form.

The scalar extensions “Xf16”, “Xf16alt” and “Xf8” are a repurposed version of the “F” standard extension as most of what they do is introduce the very same instructions on new floating-point formats. Hence the same encoding space is used and partly reused.

The vectorial extension “Xfvec” is encoded in its own encoding space, and provides all instructions from the scalar extensions on vectored (packed SIMD) data. The set of supported formats is extended to single-precision floating-point in case the D and Xfvec extensions are supported at the same time.

The auxiliary scalar and vectorial floating-point operation extension “Xfaux” is encoded within the encoding space of either the scalar or vectorial smallFloat extensions and provides, among others, expanding floating-point instructions for all floating-point formats supported in a given implementation.

This specification was originally created for inclusion in the PULP Project and **PULP-specific notes are printed in boldface.**

The rationale behind extending the ISA was to reuse as much of the standard extensions as possible - namely the floating-point extensions, adding only the needed vectorial instructions in a new encoding space.

For light-weight cores, the specification allows for the implementation of smallFloat as stand-alone extensions without the need of implementing the F standard extension.

This specification aims at defining the extensions in a general way. Implementations that make use of non-conforming design decisions when it comes to floating-point support, such as mapping the floating-point register file to the general purpose registers, can easily adapt the same design choices for the extensions herein.

[1]https://pulp-platform.org/

Changelog

Version 0.5

- Fix incorrect rounding mode for quarter-precision casts
- Add *VFCVTU*.vfmt.vfmt instructions for casting upper portions of vectors.
- Remove *VFCVT*.vfmt.vfmt instruction for the combination of *S* and *B* formats.
- Bump "Xf8" extension version number to *v0.2*

Version 0.4

- Split scalar extension into "Xf16","Xf16alt", "Xf8" extensions.
- Rename vectorial extension to "Xfvec".
- Adapt vectorial extension to act as an add-on to already supported scalar formats.
- Extract non-standard operations from previous extensions into its own "Xfaux" extension.
- Add explicit specification on how to move vectorial floating-point formats from/to memory.
- Change behavior of *VFCVT*.vfmt.vfmt to operate on the lowest entries instead of even entries.
- Add inverted sign bit to classification block resulting from *VFCLASS*.
- Change cast-and-pack operations to be more general, add double-precision support.
- Add list of pseudo-instructions to listings.
- Synchronize extensions with the floating-point changes in [1].
- Adapt title page and preface layout.
- Add CC-BY 4.0 license.

Version 0.3

- Add `binary16alt` format.
- Add encoding and mnemonic *AH* for `binary16alt` format.
- Add scalar expanding operations *FMULEX.S* and *FMACEX.S*.
- Add dot product and expanding dot product operations *VFDOTP* and *VFDOTPEX.vfmt.S*.
- Add cast and pack operations from float32 *VFCPKLO.vfmt.S* and *VFCPKHI.vfmt.S*.

Contents

Chapter 1

“Xf16” Non-Standard Extension for Half-Precision Floating-Point, Version 0.1

This section describes the non-standard half-precision instruction-set extension, which is named “Xf16” and adds half-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard. The half-precision extension is based on the base single-precision instruction subset F and the base double-precision instruction subset D.

The half-precision extension explicitly does not require any other floating-point extension to facilitate inclusion in light-weight cores without the need for hardware single-precision floating-point.

1.1 Data Format

Even though the definition herein lists an explicitly bit-addressed representation of the supported type, the internal register representation remains undefined unless used for interchange where explicitly noted.

The 16-bit half-precision floating-point representation is specified as `binary16` in IEEE 754-2008. It is defined having 1 sign bit, 5 exponent bits and 10 mantissa bits, cf. Figure 1.1. The exponent is biased by $2^{expbits-1} - 1 = 15$, the mantissa is normalized and does not store the implicit bit. The canonical NaN as specified in the F standard extension for this format is the bit pattern `0x7e00`.

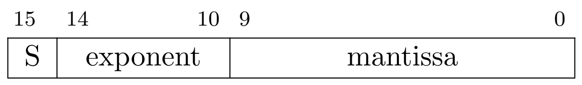


Figure 1.1: The IEEE 768-2008 half-precision format `binary16`.

1.2 Xf16 Register State

The half-precision extension operates on the 32 floating-point registers, `f0-f31`. In case of the F standard extension not being supported, the half-precision extension narrows the floating-point registers to 16 bits (FLEN=16 in Figure 8.1 in [1]). In case any floating-point standard extensions are supported, the `f` registers can now also hold 16-bit floating-point values as described in Section 9.2 of [1].

Light-weight cores might map the floating-point registers onto the general-purpose register file to save area, especially in cases where only the half-precision extension is supported.

The RI5CY PULP core implements direct mapping of a subset of floating-point registers to the integer register file for sub-32-bit floating-point formats.

1.3 Floating-Point Control and Status Register

The floating-point control and status register `fcsr` is used as defined in the F standard extension.

1.4 NaN Boxing of Narrower Values

The NaN-boxing scheme described in Section 9.2 of [1] *need not* be enforced for half-precision input operands in case the Xf16alt non-standard extension (see Chapter 2) is supported and *must not* be performed for any floating-point input operands if the Xfvec non-standard extension (see Chapter 4) is supported.

NaN-boxing loses its usefulness when the alternative half-precision floating-point format is supported alongside with either half-precision (from Xf16) or quarter-precision (from Xf8) floating-point formats. The goal of preventing register contents in narrower floating-point formats to be read as valid operands for wider floating-point instructions breaks down in these cases On one hand, since `binary16` and `binary16alt` values have the same width, they can represent valid operands in the respective other format. On the other hand, NaN-boxed `binary8` values represent valid `binary16alt` values since the exponent field in `binary16alt` overlaps the center point of its 16-bit container.

Checking input operands for proper NaN-boxing when also supporting vectored floating-point formats causes loss of efficiency when trying to perform scalar operations on the first entry of vectors. Such cases would require masking other entries such that the first entry becomes properly NaN-boxed again. Hence we decided against enforcing NaN-boxing when the Xfvec extension is active.

1.5 Subnormal Arithmetic

Operations on subnormal numbers are handled in accordance with the IEEE 754-2008 standard.

In the parlance of the IEEE standard, tininess is detected after rounding.

1.6 Half-Precision Load and Store Instructions

The FLH instruction loads a half-precision floating-point value from memory into floating-point register *rd*. FSH stores a half-precision value from the floating-point registers to memory.

The half-precision value may be a NaN-boxed quarter-precision value, if "Xf8" is supported as well.

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	width	rd	opcode
12	5	3	5	7
offset[11:0]	base	H	dest	LOAD-FP

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	width	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	H	offset[4:0]	STORE-FP

FLH and FSH are only guaranteed to execute atomically if the effective address is naturally aligned and FLEN$\geq$16.

Note that if the floating-point registers are directly mapped onto the integer register file, the FLH instruction is equivalent to the LH instruction and the FSH instruction is equivalent to the SH instructions if NaN-boxing is not performed.

1.7 Half-Precision Floating-Point Computational Instructions

A new supported format is added to the format field *fmt* of most instructions, as shown in Table 1.1. It is set to *H* (10) for instructions in the Xf16 extension. The change does not collide with other floating-point extensions.

Table 1.1: Format field encoding. **Bold** marks changes made from the F standard extension.

fmt field	Mnemonic	Meaning
00	S	32-bit single-precision
01	D	64-bit double-precision
10	**H**	**16-bit half-precision**
11	Q	128-bit quad-precision

The implementation of a custom floating-point format is very straight-forward if the present standard extensions are reused as much as possible. The previously reserved option was chosen to implement half-precision floating-point instructions.

The half-precision floating-point computational instructions are defined analogously to their single-precision counterparts, but operate on half-precision operands and produce half-precision results.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FADD/FSUB	H	src2	src1	RM	dest	OP-FP
FMUL/FDIV	H	src2	src1	RM	dest	OP-FP
FMIN-MAX	H	src2	src1	MIN/MAX	dest	OP-FP
FSQRT	H	0	src	RM	dest	OP-FP

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
src3	H	src2	src1	RM	dest	F[N]MADD/F[N]MSUB

1.8 Half-Precision Floating-Point Conversion and Move Instructions

Floating-point-to-integer and integer-to-floating-point conversion instructions are encoded in the OP-FP major opcode space. FCVT.W.H or FCVT.L.H converts a half-precision floating-point number in floating-point register *rs1* to a signed 32-bit or 64-bit integer, respectively, in integer register *rd*. FCVT.H.W or FCVT.H.L converts a 32-bit or 64-bit signed integer, respectively, in integer register *rs1* into a half-precision floating-point number in floating-point register *rd*. FCVT.WU.H, FCVT.LU.H, FCVT.H.WU, and FCVT.H.LU variants convert to or from unsigned integer values. FCVT.L[U].H and FCVT.H.L[U] are illegal in RV32. If the rounded result is not representable in the destination format, it is clipped to the nearest value and the invalid flag is set. The range of valid inputs for FCVT.*int*.H and the behavior for invalid inputs are given in Table 1.2.

Note that unlike for FCVT.*int*.S, the range of valid inputs is bounded by the range of the floating-point types instead of the integer types in some cases.

Table 1.2: Domains of half-precision-to-integer conversions and behavior for invalid inputs.

	FCVT.W.H	FCVT.WU.H	FCVT.L.H	FCVT.LU.H
Minimum valid input (after rounding)	$-65'504^*$	0	$-65'504^*$	0
Maximum valid input (after rounding)	$65'504$	$65'504$	$65'504$	$65'504$
Output for out-of-range negative input	n/a	0	n/a	0
Output for $-\infty$	-2^{31}	0	-2^{63}	0
Output for out-of-range positive input	n/a	n/a	n/a	n/a
Output for $+\infty$ or NaN	$2^{31}-1$	$2^{32}-1$	$2^{63}-1$	$2^{63}-1$

* The largest normal number is given as $\pm\left(2-2^{-10}\right)\cdot 2^{15} = \pm 65'504$ for half-precision floats.

All floating-point to integer and integer to floating-point conversion instructions round according to the *rm* field. In case that Xf16 is the widest floating-point format supported, a floating-point register can be initialized to floating-point positive zero using FCVT.H.W *rd, x0*, which will never raise any exceptions.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.*int*.H	H	W[U]/L[U]	src	RM	dest	OP-FP
FCVT.H.*int*	H	W[U]/L[U]	src	RM	dest	OP-FP

If the F standard extension is supported, the half-precision to single-precision and single-precision to half-precision conversion instructions, FCVT.S.H and FCVT.H.S, are encoded in the OP-FP major opcode space and both the source and destination are floating-point registers. If the D standard extension is supported, the half-precision to double-precision and double-precision to half-precision conversion instructions, FCVT.D.H and FCVT.H.D, are encoded in the OP-FP major opcode space and both the source and destination are floating-point registers. The *rs2* field encodes the datatype of the source, and the *fmt* field encodes the datatype of the destination.

FCVT.H.S and FCVT.H.D round according to the *rm* field; FCVT.S.H and FCVT.D.H will never round.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.S.H	S	H	src	000	dest	OP-FP
FCVT.H.S	H	S	src	RM	dest	OP-FP
FCVT.D.H	D	H	src	000	dest	OP-FP
FCVT.H.D	H	D	src	RM	dest	OP-FP

Floating-point to floating-point sign-injection instructions, FSGNJ.H, FSGNJN.H, and FSGNJX.H are defined analogously to the single-precision sign-injection instructions.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FSGNJ	H	src2	src1	J[N]/JX	dest	OP-FP

Instructions are provided to move bit patterns between the floating-point and integer registers. FMV.X.H moves the half-precision value in floating-point register *rs1* to a representation in IEEE 754-2008 standard encoding to the lower 16 bits of integer register *rd*. The higher bits of the destination register are filled with copies of the floating-point number’s sign bit. FMV.H.X moves the half-precision value encoded in IEEE 754-2008 standard encoding from the lower 16 bits of integer register *rs1* to the floating-point register *rd*. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FMV.X.H	H	0	src	000	dest	OP-FP
FMV.H.X	H	0	src	000	dest	OP-FP

1.9 Half-Precision Floating-Point Compare Instructions

The half-precision floating-point compare instructions are defined analogously to their single-precision counterparts, but operate on half-precision operands.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCMP	H	src2	src1	EQ/LT/LE	dest	OP-FP

1.10 Half-Precision Floating-Point Classify Instruction

The half-precision floating-point classify instruction, FCLASS.H, is defined analogously to its single-precision counterpart, but operates on half-precision operands.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCLASS	H	0	src	001	dest	OP-FP

Chapter 2

“Xf16alt” Non-Standard Extension for Alternative Half-Precision Floating-Point, Version 0.1

This section describes the non-standard alternative half-precision instruction-set extension, which is named “Xf16alt” and adds alternative half-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard. The alternative half-precision extension is based on non-standard instruction subset Xf16.

The alternative half-precision extension explicitly does not require any other floating-point extension to facilitate inclusion in light-weight cores without the need for hardware single-precision floating-point.

2.1 Data Format

Even though the definition herein lists an explicitly bit-addressed representation of the supported type, the internal register representation remains undefined unless used for interchange where explicitly noted.

The 16-bit alternative half-precision floating-point representation is specified as `binary16alt` in this specification. It is defined having 1 sign bit, 8 exponent bits and 7 mantissa bits, cf. Figure 2.1. The exponent is biased by $2^{expbits-1} - 1 = 127$, the mantissa is normalized and does not store the implicit bit. The canonical NaN as specified in the F standard extension for this format is the bit pattern `0x7fc0`.

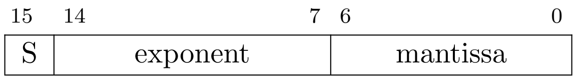


Figure 2.1: The custom alternative half-precision format `binary16alt`.

We reuse the exponent width of 8 bits for the alternative 16-bit floating-point format in order to retain the dynamic range of binary32 as well as enabling reuse in the data path.

2.2 Xf16alt Register State

The alternative half-precision extension operates on the 32 floating-point registers, `f0-f31`. In case of the F standard extension not being supported, the alternative half-precision extension narrows the floating-point registers to 16 bits (FLEN=16 in Figure 8.1 in [1]). In case any floating-point standard extensions are supported, the `f` registers can now also hold 16-bit alternative floating-point values as described in Section 9.2 of [1].

Light-weight cores might map the floating-point registers onto the general-purpose register file to save area, especially in cases where only the half-precision extension is supported.

The RI5CY PULP core implements direct mapping of a subset of floating-point registers to the integer register file for sub-32-bit floating-point formats.

2.3 Floating-Point Control and Status Register

The floating-point control and status register `fcsr` is used as defined in the F standard extension.

2.4 NaN Boxing of Narrower Values

The NaN-boxing scheme described in Section 9.2 of [1] *need not* be enforced for alternative half-precision input operands in case the Xf16 non-standard extension (see Chapter 1) or the Xf8 non-standard extension (see Chapter 3) is supported and *must not* be performed for any floating-point input operands if the Xfvec non-standard extension (see Chapter 4) is supported.

NaN-boxing loses its usefulness when the alternative half-precision floating-point format is supported alongside with either half-precision (from Xf16) or quarter-precision (from Xf8) floating-point formats. The goal of preventing register contents in narrower floating-point formats to be read as valid operands for wider floating-point instructions breaks down in these cases On one hand, since `binary16` and `binary16alt` values have the same width, they can represent valid operands in the respective other format. On the other hand, NaN-boxed `binary8` values represent valid `binary16alt` values since the exponent field in `binary16alt` overlaps the center point of its 16-bit container.

Checking input operands for proper NaN-boxing when also supporting vectored floating-point formats causes loss of efficiency when trying to perform scalar operations on the first entry of vectors. Such cases would require masking other entries such that the first entry becomes properly NaN-boxed again. Hence we decided against enforcing NaN-boxing when the Xfvec extension is active.

2.5 Subnormal Arithmetic

Operations on subnormal numbers are handled in accordance with the IEEE 754-2008 standard.

In the parlance of the IEEE standard, tininess is detected after rounding.

2.6 Alternative Half-Precision Load and Store Instructions

The FLAH pseudo-instruction is encoded as FLH and loads an alternative half-precision floating-point value from memory into floating-point register *rd*. FSAH pseudo-instruction is encoded as FSH and stores an alternative half-precision value from the floating-point registers to memory.

The alternative half-precision value may be a NaN-boxed quarter-precision value, if "Xf8" is supported as well.

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	width	rd	opcode
12	5	3	5	7
offset[11:0]	base	H	dest	LOAD-FP

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	width	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	H	offset[4:0]	STORE-FP

FLH and FSH are only guaranteed to execute atomically if the effective address is naturally aligned and FLEN$\geq$16.

Note that if the floating-point registers are directly mapped onto the integer register file, the FLH instruction is equivalent to the LH instruction and the FSH instruction is equivalent to the SH instructions if NaN-boxing is not performed.

Also note that the lack of dedicated AH load/store instructions will lead to errors when using a recoded FP architecture.

2.7 Alternative Half-Precision Floating-Point Computational Instructions

A new supported format is aliased onto the half-precision format defined in the Xf16 extension and used in the format field *fmt* of most instructions, as shown in Table 2.1. It is set to *AH* (10) for instructions in the Xf16alt extension and the two 16-bit floating-point formats are distinguished by other means, as shown below. The change does not collide with other floating-point extensions.

The implementation of a custom floating-point format is very straight-forward if the present standard extensions are reused as much as possible. The previously reserved option was chosen to implement half-precision floating-point instructions.

Table 2.1: Format field encoding. **Bold** marks changes made from the Xf16 extension.

fmt field	Mnemonic	Meaning
00	S	32-bit single-precision
01	D	64-bit double-precision
10	**AH**	**16-bit alternative half-precision**
11	Q	128-bit quad-precision

Table 2.2: Rounding mode encoding. **Bold** marks changes made from the F standard extension.

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude
101	**ALTF**	**In instruction's *rm* field, if *fmt*=10, operands are `binary16alt`; Otherwise, *Invalid. Reserved for future use.***
110		*Invalid. Reserved for future use.*
111		In instruction's *rm* field, selects dynamic rounding mode; In Rounding Mode register, *Invalid.*

In order to differentiate between the `binary16` and alternative `binary16alt` 16-bit floating-point formats, the rounding mode field *rm* is used as shown in Table 2.2.

Encoding of the format for floating-point instructions that *are affected by rounding modes* now works as follows:

- When the format field *fmt* is set to *[A]H* (10)
  - If the rounding mode field *rm* holds one of the valid patterns as per the F standard extension (000-100 or 111), half-precision (`binary16`) is selected.
  - If the rounding mode is set to *ALTF* (101), operations will interpret the operands as alternative half-precision (`binary16alt`) values. In this case, rounding mode is applied from the `frm` field in `fcsr`.
- Otherwise, *ALTF* (101) is treated as an invalid rounding mode as specified in the F standard extension.

Instructions that are *not* affected by rounding modes but use the *rm* or *rs2* field as part of their operation encoding (e.g. FMIN) are encoded as follows:

- When the format field *fmt* is set to *[A]H* (10)
  - Operations on `binary16` use the encoding in *rm/rs2* as defined in the Xf16 extension.

  - Operations on `binary16alt` use a new encoding in *rm/rs2* as shown in Chapter 6. This new encoding does not collide with other instructions.

- Otherwise, the instructions with the new encoding in *rm/rs2* are not defined/invalid.

The alternative half-precision floating-point computational instructions are defined analogously to their half-precision counterparts, but operate on alternative half-precision operands and produce half-precision results.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FADD/FSUB	AH	src2	src1	ALTF	dest	OP-FP
FMUL/FDIV	AH	src2	src1	ALTF	dest	OP-FP
FMIN-MAX	AH	src2	src1	MIN/MAX	dest	OP-FP
FSQRT	AH	0	src	ALTF	dest	OP-FP

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
src3	AH	src2	src1	ALTF	dest	F[N]MADD/F[N]MSUB

2.8 Alternative Half-Precision Floating-Point Conversion and Move Instructions

Floating-point-to-integer and integer-to-floating-point conversion instructions are encoded in the OP-FP major opcode space. FCVT.W.AH or FCVT.L.AH converts an alternative half-precision floating-point number in floating-point register *rs1* to a signed 32-bit or 64-bit integer, respectively, in integer register *rd*. FCVT.AH.W or FCVT.AH.L converts a 32-bit or 64-bit signed integer, respectively, in integer register *rs1* into an alternative half-precision floating-point number in floating-point register *rd*. FCVT.WU.AH, FCVT.LU.AH, FCVT.AH.WU, and FCVT.AH.LU variants convert to or from unsigned integer values. FCVT.L[U].AH and FCVT.AH.L[U] are illegal in RV32. If the rounded result is not representable in the destination format, it is clipped to the nearest value and the invalid flag is set. The range of valid inputs for FCVT.*int*.AH and the behavior for invalid inputs is the same as for FCVT.*int*.S.

All floating-point to integer and integer to floating-point conversion instructions round according to *frm*. In case that Xf16alt is the widest floating-point format supported, a floating-point register can be initialized to floating-point positive zero using FCVT.AH.W *rd, x0*, which will never raise any exceptions.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.*int*.AH	AH	W[U]/L[U]	src	ALTF	dest	OP-FP
FCVT.AH.*int*	AH	W[U]/L[U]	src	ALTF	dest	OP-FP

If the Xf16 standard extension is supported, the alternative half-precision to half-precision and half-precision to alternative half-precision conversion instructions, FCVT.H.AH and FCVT.AH.H, are added. If the F standard extension is supported, the half-precision to single-precision and single-precision to half-precision conversion instructions, FCVT.S.H and FCVT.H.S, are added. If the D standard extension is supported, the half-precision to double-precision and double-precision to half-precision conversion instructions, FCVT.D.H and FCVT.H.D, are added. The *rs2* field encodes the datatype of the source, and the *fmt* field encodes the datatype of the destination.

FCVT.H.AH rounds according to the *rm* field; FCVT.AH.H, FCVT.AH.S and FCVT.AH.D round according to *frm*; FCVT.S.AH and FCVT.D.AH will never round.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.H.AH	H	AH	src	RM	dest	OP-FP
FCVT.AH.H	AH	H	src	ALTF	dest	OP-FP
FCVT.S.AH	S	AH	src	000	dest	OP-FP
FCVT.AH.S	AH	S	src	ALTF	dest	OP-FP
FCVT.D.AH	D	AH	src	000	dest	OP-FP
FCVT.AH.D	AH	D	src	ALTF	dest	OP-FP

Floating-point to floating-point sign-injection instructions, FSGNJ.AH, FSGNJN.AH, and FSGNJX.AH are defined analogously to the single-precision sign-injection instructions.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FSGNJ	AH	src2	src1	J[N]/JX	dest	OP-FP

Instructions are provided to move bit patterns between the floating-point and integer registers. FMV.X.AH moves the alternative half-precision value in floating-point register *rs1* to a representation as specified in Figure 2.1 to the lower 16 bits of integer register *rd*. The higher bits of the destination register are filled with copies of the floating-point number's sign bit. FMV.AH.X moves the quarter-precision value encoded as specified in Figure 2.1 from the lower 16 bits of integer register *rs1* to the floating-point register *rd*. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FMV.X.AH	AH	0	src	100	dest	OP-FP
FMV.AH.X	AH	0	src	100	dest	OP-FP

2.9 Alternative Half-Precision Floating-Point Compare Instructions

The alternative half-precision floating-point compare instructions are defined analogously to their half-precision counterparts, but operate on alternative half-precision operands.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCMP	AH	src2	src1	EQ/LT/LE	dest	OP-FP

2.10 Alternative Half-Precision Floating-Point Classify Instruction

The alternative half-precision floating-point classify instruction, FCLASS.AH, is defined analogously to its half-precision counterpart, but operates on alternative half-precision operands.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCLASS	AH	0	src	101	dest	OP-FP

Chapter 3

"Xf8" Non-Standard Extension for Quarter-Precision Floating-Point, Version 0.2

This section describes the non-standard quarter-precision instruction-set extension, which is named "Xf8" and adds quarter-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard. The quarter-precision extension is based on the non-standard instruction subset Xf16.

The quarter-precision extension explicitly does not require any other floating-point extension to facilitate inclusion in light-weight cores without the need for hardware single-precision floating-point.

3.1 Data Format

Even though the definition herein lists an explicitly bit-addressed representation of the supported type, the internal register representation remains undefined unless used for interchange where explicitly noted.

The 8-bit quarter-precision floating-point representation is specified as `binary8` in this specification. We define it having 1 sign bit, 5 exponent bits and 2 mantissa bits, cf. Figure 3.1. The exponent is biased by $2^{expbits-1} - 1 = 15$, the mantissa is normalized and does not store the implicit bit. The canonical NaN as specified in the F standard extension for this format is the bit pattern `0x7e`.

When choosing a custom quarter-precision floating-point format, we decided to provide a large

7	6 2	1 0
S	exponent	mm

Figure 3.1: The custom quarter-precision format `binary8`.

exponent with a small mantissa. This is due to floating-point formats being used mostly for their high dynamic range. For best precision in the envelope of 8 bits, a fixed-point format would be better suited.

We chose to set the exponent width to 5 bits in order to provide reasonable dynamic range and to reuse parts of the hardware from the half-precision data path.

3.2 Xf8 Register State

The quarter-precision extension operates on the 32 floating-point registers, `f0-f31`. In case of wider floating-point extensions not being supported, the quarter-precision extension narrows the floating-point registers to 8 bits (FLEN=8 in Figure 8.1 in [1]). In case any wider floating-point extensions are supported, the `f` registers can now also hold 8-bit floating-point values as described in Section 9.2 of [1].

Light-weight cores might map the floating-point registers onto the general-purpose register file to save area, especially in cases where only the quarter-precision extension is supported.

The RI5CY PULP core implements direct mapping of a subset of floating-point registers to the integer register file for sub-32-bit floating-point formats.

3.3 Floating-Point Control and Status Register

The floating-point control and status register `fcsr` is used as defined in the F standard extension.

3.4 NaN Boxing of Narrower Values

The NaN-boxing scheme described in Section 9.2 of [1] *need not* be enforced for quarter-precision input operands in case the Xf16alt non-standard extension (see Chapter 2) is supported and *must not* be performed for any floating-point input operands if the Xfvec non-standard extension (see Chapter 4) is supported.

NaN-boxing loses its usefulness when the alternative half-precision floating-point format is supported alongside with either half-precision (from Xf16) or quarter-precision (from Xf8) floating-point formats. The goal of preventing register contents in narrower floating-point formats to be read as valid operands for wider floating-point instructions breaks down in these cases On one hand, since `binary16` and `binary16alt` values have the same width, they can represent valid operands in the respective other format. On the other hand, NaN-boxed `binary8` values represent valid `binary16alt` values since the exponent field in `binary16alt` overlaps the center point of its 16-bit container.

Checking input operands for proper NaN-boxing when also supporting vectored floating-point formats causes loss of efficiency when trying to perform scalar operations on the first entry of vectors. Such cases would require masking other entries such that the first entry becomes properly NaN-boxed again. Hence we decided against enforcing NaN-boxing when the Xfvec extension is active.

3.5 Subnormal Arithmetic

Operations on subnormal numbers are handled in accordance with the IEEE 754-2008 standard. In the parlance of the IEEE standard, tininess is detected after rounding.

3.6 Quarter-Precision Load and Store Instructions

The FLB instruction loads a quarter-precision floating-point value from memory into floating-point register *rd*. FSB stores a quarter-precision value from the floating-point registers to memory.

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	width	rd	opcode
12	5	3	5	7
offset[11:0]	base	B	dest	LOAD-FP

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	width	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	B	offset[4:0]	STORE-FP

FLB and FSB are only guaranteed to execute atomically if the effective address is naturally aligned and FLEN$\geq$8.

Note that if the floating-point registers are directly mapped onto the integer register file, the FLB instruction is equivalent to the LB instruction and the FSB instruction is equivalent to the SB instructions if NaN-boxing is not performed.

3.7 Quarter-Precision Floating-Point Computational Instructions

A new supported format is added to the format field *fmt* of most instructions by changing its definition in the F standard extension as shown in Table 3.1. It is set to *B* (11) for instructions in the Xf8 extension. This change collides with the Q standard extension. The change does not collide with other floating-point extensions.

The implementation of a custom floating-point format is very straight-forward if the present standard extensions are reused as much as possible. The replacement of the Q *entry in the F extension's* fmt *field were the logical choice, extending the ISA towards smaller instead of larger formats.*

The quarter-precision floating-point computational instructions are defined analogously to their half-precision counterparts, but operate on quarter-precision operands and produce quarter-precision results.

Table 3.1: Format field encoding. **Bold** marks changes made from the Xf16 extension.

fmt field	Mnemonic	Meaning
00	S	32-bit single-precision
01	D	64-bit double-precision
10	H	16-bit half-precision
11	**B**	**8-bit quarter-precision**

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FADD/FSUB	B	src2	src1	RM	dest	OP-FP
FMUL/FDIV	B	src2	src1	RM	dest	OP-FP
FMIN-MAX	B	src2	src1	MIN/MAX	dest	OP-FP
FSQRT	B	0	src	RM	dest	OP-FP

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
src3	B	src2	src1	RM	dest	F[N]MADD/F[N]MSUB

3.8 Quarter-Precision Floating-Point Conversion and Move Instructions

Floating-point-to-integer and integer-to-floating-point conversion instructions are encoded in the OP-FP major opcode space. FCVT.W.B or FCVT.L.B converts a quarter-precision floating-point number in floating-point register *rs1* to a signed 32-bit or 64-bit integer, respectively, in integer register *rd*. FCVT.B.W or FCVT.B.L converts a 32-bit or 64-bit signed integer, respectively, in integer register *rs1* into a quarter-precision floating-point number in floating-point register *rd*. FCVT.WU.B, FCVT.LU.B, FCVT.B.WU, and FCVT.B.LU variants convert to or from unsigned integer values. FCVT.L[U].B and FCVT.B.L[U] are illegal in RV32. If the rounded result is not representable in the destination format, it is clipped to the nearest value and the invalid flag is set. The range of valid inputs for FCVT.*int*.B and the behavior for invalid inputs are given in Table 3.2.

Note that unlike for FCVT.*int*.S, the range of valid inputs is bounded by the range of the floating-point types instead of the integer types in some cases.

All floating-point to integer and integer to floating-point conversion instructions round according to the *rm* field. In case that Xf8 is the widest floating-point format supported, a floating-point register can be initialized to floating-point positive zero using FCVT.B.W *rd, x0*, which will never raise any exceptions.

Table 3.2: Domains of quarter-precision-to-integer conversions and behavior for invalid inputs.

	FCVT.W.B	FCVT.WU.B	FCVT.L.B	FCVT.LU.B
Minimum valid input (after rounding)	$-57'344^*$	0	$-57'344^*$	0
Maximum valid input (after rounding)	$57'344$	$57'344$	$57'344$	$57'344$
Output for out-of-range negative input	n/a	0	n/a	0
Output for $-\infty$	-2^{31}	0	-2^{31}	0
Output for out-of-range positive input	n/a	n/a	n/a	n/a
Output for $+\infty$ or NaN	$2^{31}-1$	$2^{32}-1$	$2^{31}-1$	$2^{32}-1$

* The largest normal number is given as $\pm\left(2-2^{-2}\right)\cdot 2^{15} = \pm 57'344$ for quarter-precision floats.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.*int*.B	B	W[U]/L[U]	src	RM	dest	OP-FP
FCVT.B.*int*	B	W[U]/L[U]	src	RM	dest	OP-FP

If the Xf16 extension is supported, the quarter-precision to half-precision and half-precision to quarter-precision conversion instructions FCVT.H.B and FCVT.B.H, are added. If the Xf16alt extension is supported, the quarter-precision to alternative half-precision and alternative half-precision to quarter-precision conversion instructions FCVT.AH.B and FCVT.B.AH, are added. If the F standard extension is supported, the quarter-precision to single-precision and single-precision to quarter-precision conversion instructions, FCVT.S.B and FCVT.B.S, are added. If the D standard extension is supported, the quarter-precision to double-precision and double-precision to quarter-precision conversion instructions, FCVT.D.B and FCVT.B.D, are added. The *rs2* field encodes the datatype of the source, and the *fmt* field encodes the datatype of the destination.

FCVT.B.H, FCVT.B.AH, FCVT.B.S and FCVT.B.D round according to the *rm* field; FCVT.H.B, FCVT.AH.B, FCVT.S.B and FCVT.D.B will never round.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.H.B	H	B	src	000	dest	OP-FP
FCVT.B.H	B	H	src	RM	dest	OP-FP
FCVT.AH.B	AH	B	src	ALTF	dest	OP-FP
FCVT.B.AH	B	AH	src	RM	dest	OP-FP
FCVT.S.B	S	B	src	000	dest	OP-FP
FCVT.B.S	B	S	src	RM	dest	OP-FP
FCVT.D.B	D	B	src	000	dest	OP-FP
FCVT.B.D	B	D	src	RM	dest	OP-FP

Floating-point to floating-point sign-injection instructions, FSGNJ.B, FSGNJN.B, and FSGNJX.B are defined analogously to the half-precision sign-injection instructions.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FSGNJ	B	src2	src1	J[N]/JX	dest	OP-FP

Instructions are provided to move bit patterns between the floating-point and integer registers. FMV.X.B moves the quarter-precision value in floating-point register *rs1* to a representation as specified in Figure 3.1 to the lower 8 bits of integer register *rd*. The higher bits of the destination register are filled with copies of the floating-point number's sign bit. FMV.B.X moves the quarter-precision value encoded as specified in Figure 3.1 from the lower 8 bits of integer register *rs1* to the floating-point register *rd*. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FMV.X.B	B	0	src	000	dest	OP-FP
FMV.B.X	B	0	src	000	dest	OP-FP

3.9 Quarter-Precision Floating-Point Compare Instructions

The quarter-precision floating-point compare instructions are defined analogously to their half-precision counterparts, but operate on quarter-precision operands.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCMP	B	src2	src1	EQ/LT/LE	dest	OP-FP

3.10 Quarter-Precision Floating-Point Classify Instruction

The quarter-precision floating-point classify instruction, FCLASS.B, is defined analogously to its half-precision counterpart, but operates on quarter-precision operands.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCLASS	B	0	src	001	dest	OP-FP

Chapter 4

“Xfvec” Non-Standard Extension for Vectorial (SIMD) Floating-Point, Version 0.2

This section describes the non-standard vectorial (packed-SIMD) floating-point instruction-set extension, which is named “Xfvec” and adds packed floating-point computational instructions. If Xfvec is supported, the vectorial floating-point instructions are added for *all* supported floating-point formats narrower than FLEN. Thus, the vectorial floating-point extension depends on some or all of the following extensions: the D standard extension, the F standard extension, the Xf16 extension, and the Xf16alt extension, as indicated in Table 4.1.

The extension does not collide with [1], but uses previously unused space in the OP base opcode. The formats supported herein are defined in the D, F, Xf16, Xf16alt, and Xf8 extensions.

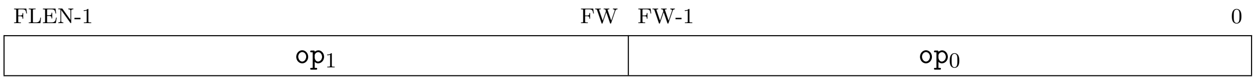
Table 4.1: FLEN, given by the widest supported floating-point format (rows) defines the vector dimension for all narrower supported formats (columns).

		Vector length `n` if supported			
		F	Xf16	Xf16alt	Xf8
FLEN	64	2	4	4	8
	32	$\times$	2	2	4
	16	$\times$	$\times$	$\times$	2

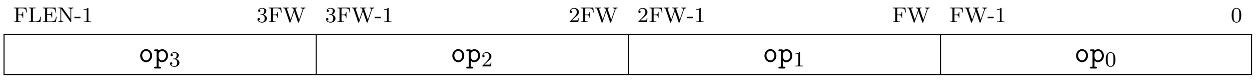
4.1 Xfvec Register State

The vectorial floating-point extension operates on 32 floating-point registers, `f0`-`f31`. The width of the floating-point registers FLEN is given by the widest scalar floating-point extension supported.

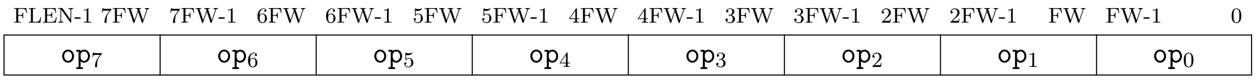
Vectors of narrower floating-point formats are packed inside the floating-point registers. These packed narrower values within one floating-point register are labelled as $\texttt{op}_{0..n-1}$ and henceforth called operand entries. The first entry, $\texttt{op}_0$ is always located at the lowest portion of the register,



(a) Packing of two FW=FLEN/2-wide values.



(b) Packing of four FW=FLEN/4-wide values.



(c) Packing of eight FW=FLEN/8-wide values.

Figure 4.1: Packing of floating-point values as entries inside a vectorial floating-point operand.

see Figure 4.1.

Mixed-format packing is unsupported and all operands are considered to have fully populated entries, i.e. they consist of `n`=FLEN/FormatWidth entries.

4.2 Floating-Point Control and Status Register

The floating-point control and status register `fcsr` is used as defined in the F standard extension. The accrued exception flags are collected from all the parallel sub-operations and logic-ORed into their positions in the status register. Rounding modes for all parallel operations are driven with the same rounding mode from `frm`.

As only the lowest eight bits in `fcsr` are used, we thought about duplicating the `fcsr` to fill the entire 32-bit CSR. The new vectorial status register, `vfcsr` would allow to collect exception flags from up to four parallel strands of operation (as in the case of vectorial quarter-precision operations). Also, each strand would have its own rounding mode.

We didn't see the need for the amount of control a vectorial status register would provide, as entries in a vector are usually all treated with the same requirements and rounding modes. Furthermore, the need for additional hardware resources such as control logic and register space and the resulting impacts in area and energy consumption are not worth the additional features.

As of [1], the currently unused part of `fcsr` is set to be used in the upcoming L standard extension, thus colliding with the considerations herein anyways.

4.3 NaN Generation, Propagation and NaN Boxing

The NaN generation and propagation scheme for vectorial formats is analogous to their counterparts defined in the respective scalar extensions and applies to all operand and result entries independently. Where an invalid operation exception would be raised in the scalar instructions, it is triggered if at least one operation on entries of the vectorial instruction triggers the invalid operation condition.

The NaN-boxing scheme described in Section 9.2 of [1] *must not* be performed for any floating-point input operands to scalar floating-point instructions.

Checking input operands for proper NaN-boxing when also supporting vectored floating-point formats causes loss of efficiency when trying to perform scalar operations on the first entry of vectors. Such cases would require masking other entries such that the first entry becomes properly NaN-boxed again. Hence we decided against enforcing NaN-boxing when the Xfvec extension is active.

4.4 Vectorial Floating-Point Load and Store Instructions

The FL*FLEN* and FS*FLEN* instructions are used to move vectorial floating-point operands between the floating-point register file and memory.

4.5 Vectorial Floating-Point Computational Instructions

Vectorial floating-point arithmetic instructions with one or two source operands use the new RVF-type format with the OP major opcode.

The RVF instruction format is a special case of the R-type format with some additional fields specified. First, the R-type's *funct7* field is divided into a 2-bit prefix *f2* and a 5-bit operation identifier *vecfltop*. Furthermore, the R-type's *funct3* field is divided into a replication bit *R* and the 2-bit vector format field *vfmt*.

The 2-bit prefix *f2* is set to *VF* (10) for all instructions in the Xfvec extension.

While a new major opcode could have been used for implementing the vectorial floating-point extension, we found this to be too wasteful. Instead, a completely unused portion of the existing OP opcode (except for one PULP-specific instruction that probably has never been executed to date) was identified, namely those starting with `10` in the MSB.

The 2-bit vectorial floating-point format field *vfmt* is encoded as shown in Table 4.2. It can be set to *S* (00), *AH* (01), *H* (10) or *B* (11) for instructions in the Xfvec extension.

Table 4.2: Vectorial format field encoding.

vfmt field	Mnemonic	Meaning
00	S	Entries are 32-bit single-precision `binary32`
01	AH	Entries are 16-bit alternative half-precision (`binary16alt`)
10	H	Entries are 16-bit half-precision (`binary16`)
11	B	Entries are 8-bit quarter-precision (`binary8`)

VFADD.*vfmt*, VFSUB.*vfmt*, VFMUL.*vfmt*, VFDIV.*vfmt* perform floating-point addition, subtraction, multiplication, and division, respectively, between corresponding entries of *rs1* and *rs2*, writing the result to *rd*. VFMIN.*vfmt* and VFMAX.*vfmt* write, respectively, the smaller or larger corresponding entries of *rs1* and *rs2* to *rd*. VFSQRT.*vfmt* computes the square roots of the entries in *rs1* and writes the result to *rd*.

The replication bit *R* can be set for all instructions operating on at least two sources, *rs1* and *rs2*. If set, the first entry $\texttt{op}_0$ of *rs2* is used for all entries of *rs2*. This scalar replication of $\texttt{op}_0$ in *rs2* can be used to perform vector-scalar operations. No bits in the actual register at *rs2* are modified, unless it also acts as the destination *rd*. As such, VFADD.R.*vfmt*, VFSUB.R.*vfmt*, VFMUL.R.*vfmt*, VFDIV.R.*vfmt* perform floating-point addition, subtraction, multiplication, and division, respectively, between the entries of *rs1* and the first entry in *rs2*, writing the result to *rd*. VFMIN.R.*vfmt* and VFMAX.R.*vfmt* write, respectively, the smaller or larger entries of *rs1* and the first entry in *rs2* to *rd*.

All vectorial floating-point operations that perform rounding select the rounding mode using *frm* from the float control and status register `fcsr`. All entries are treated with the same rounding mode.

31 30	29 25	24 20	19 15	14	13 12	11 7	6 0
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode
2	5	5	5	1	2	5	7
VF	ADD/SUB	src2	src1	R	S/[A]H/B	dest	OP
VF	MUL/DIV	src2	src1	R	S/[A]H/B	dest	OP
VF	MIN/MAX	src2	src1	R	S/[A]H/B	dest	OP
VF	SQRT	0	src	0	S/[A]H/B	dest	OP

Vectorial fused-multiply-accumulate instructions are encoded in the OP major opcode space using the RVF-type instruction format. VFMAC.*vfmt*, VFMRE.*vfmt* perform floating-point fused multiply-accumulation or multiply-reduction between corresponding entries of *rs1*, *rs2* and *rd*, writing the result to *rd*. The operation performed is given as $rd = rd + rs1 * rs2$ or $rd = rd - rs1 * rs2$ for corresponding entries, respectively.

VFMAC.R.*vfmt*, VFMRE.R.*vfmt* perform floating-point fused multiply-accumulation or multiply-reduction between the entries of *rs1*, the first entry in *rs2* and the entries of *rd*, writing the result to *rd*.

31 30	29 25	24 20	19 15	14	13 12	11 7	6 0
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode
2	5	5	5	1	2	5	7
VF	MAC/MRE	src2	src1	R	S/[A]H/B	src3=dest	OP

---

*For now, vectorial fused multiply-add operations (VFMADD.*vfmt*, VFMSUB.*vfmt*) are not implemented. They might be implemented with a later revision of this document. Note that this will require one new major opcode.*

4.6 Vectorial Floating-Point Conversion and Move Instructions

For RV64 or for RV32 and FLEN$\leq$32 only, vectorial floating-point-to-integer and integer-to-floating-point conversion instructions are encoded in the OP major opcode space using the RVF-type instruction format. VFCVT.X.*vfmt* converts the floating-point numbers stored as entries of floating-point register *rs1* to signed *vfmt*-width integers, packed into integer register *rd*. VFCVT.*vfmt*.X converts signed *vfmt*-width integers stored as entries of integer register *rs1* into floating-point numbers, packed into floating-point register *rd*. VFCVT.XU.*vfmt* and VFCVT.*vfmt*.XU variants convert to or from unsigned integer values, respecitvely.

If the rounded result is not representable in the destination format, it is clipped to the nearest value and the invalid flag is set. The range of valid inputs for VFCVT.*int.vfmt* and the behavior for invalid inputs are given in Table 4.3.

Table 4.3: Domains of vectorial floating-point-to-integer conversions and behavior for invalid inputs.

FCVT.	X.S	XU.S	X.AH	XU.AH	X.H	XU.H	X.B	XU.B
Min. valid inp. (after rnd.)	-2^{31}	0	-2^{15}	0	-2^{15}	0	-128	0
Max. valid inp. (after rnd.)	$2^{31}-1$	$2^{32}-1$	$2^{15}-1$	$2^{16}-1$	$2^{15}-1$	$65'504^{*}$	127	255
Outp. for out-of-r. neg. inp.	-2^{31}	0	-2^{15}	0	-2^{15}	0	-128	0
Output for $-\infty$	-2^{31}	0	-2^{15}	0	-2^{15}	0	-128	0
Outp. for out-of-r. pos. inp.	$2^{31}-1$	$2^{32}-1$	$2^{15}-1$	$2^{16}-1$	$2^{15}-1$	n/a	127	255
Output for $+\infty$ or NaN	$2^{31}-1$	$2^{32}-1$	$2^{15}-1$	$2^{16}-1$	$2^{15}-1$	$2^{16}-1$	127	255

* The largest normal number is given as $\pm\left(2-2^{-10}\right)\cdot 2^{15} = \pm 65'504$ for half-precision floats.

All vectorial floating-point to integer and integer to floating-point conversion instructions round according to *frm* in `fcsr`. A floating-point register's entries can be initialized to floating-point positive zero using VFCVT.*vfmt*.X *rd, x0*, which will never raise any exceptions.

31 30	29 25	24 20	19 15	14	13 12	11 7	6 0
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode
2	5	5	5	1	2	5	7
VF	CVT	X.*vfmt*/*vfmt*.X	src	U	S/[A]H/B	dest	OP

For any two vectorial floating-point formats whose number of entries does not differ by a factor of more than two, vectorial floating-point to floating-point conversion instructions, VFCVT[U].*vfmt.vfmt*, are encoded in the OP major opcode space and both the source and destination are floating-point registers.

The limitation for vectorial floating-point casts to only work with formats within a width-difference of 2x means that VFCVT operations are not available for conversions between binary32 and binary8. The reason for this is, given FLEN=64, the 'lower' and 'upper' parts of the binary8 hold four entries each while the whole binary32 vector consists of only two entries.

In order to accommodate vectorial conversions between these formats, two additional instructions would be necessary which remain unspecified at this point. It is our intention to find a solution for this scenario in a later revision of this document.

The number of entries operated on is given by the vector length in the wider, less populous format, $\mathtt{n}_{wider}$. For VFCVT.*vfmt.vfmt*, the first $\mathtt{n}_{wider}$ entries of the more populous format are considered for the conversion. VFCVTU.*vfmt.vfmt* operates on the upper $\mathtt{n}_{wider}$ entries of the more populous vector. VFCVTU.*vfmt.vfmt* is illegal if source and destination formats are of same width. Conversions from a less populous format to a more populous one set the corresponding entries in the destination register, leaving other entries intact.

For example, given RV32 and FLEN=32, in a vectorial quarter-precision to half-precision conversion VFCVT.H.B, $\mathtt{op}_0$ and $\mathtt{op}_1$ of the quarter-precision source become $\mathtt{op}_0$ and $\mathtt{op}_1$ of the result, respectively. In VFCVTU.H.B $\mathtt{op}_2$ and $\mathtt{op}_3$ of the quarter-precision source become $\mathtt{op}_0$ and $\mathtt{op}_1$ of the result, respectively.

For all vectorial floating-point to floating-point conversion instructions, the *rs2* field encodes the datatype of the source, and the *vfmt* field encodes the datatype of the destination. VFCVT[U].AH.S, VFCVT[U].AH.H, VFCVT[U].H.S, VFCVT[U].H.AH, VFCVT[U].B.AH, and VFCVT[U].B.H round according to *frm*; VFCVT[U].S.AH, VFCVT[U].S.H, VFCVT[U].AH.B, and VFCVT[U].H.B will never round.

31 30	29 25	24 20	19 15	14	13 12	11 7	6 0
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode
2	5	5	5	1	2	5	7
VF	CVT[U]	*vfmt*.[A]H	src	U	S/B	dest	OP
VF	CVT[U]	*vfmt*.S/B	src	U	[A]H	dest	OP

Vectorial floating-point to fixed-point and fixed-point to floating-point conversion functions could also be added in this space, as there's enough room with this prefix.

Vectorial floating-point to floating-point sign-injection instructions, VFSGNJ.*vfmt*, VFSGNJN.*vfmt*, and VFSGNJX.*vfmt* are defined analogously to the scalar floating-point sign-injection instructions, applying the injections to all corresponding entries.

If the replication bit *R* is set, the sign of the first entry in *rs2* is used for injection. VFSGNJ.R.*vfmt*, VFSGNJN.R.*vfmt*, VFSGNJX.R.*vfmt* behave analogously, but use the replicated first entry of *rs2* for all entries of *rs2*.

31 30	29 25	24 20	19 15	14	13 12	11 7	6 0
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode
2	5	5	5	1	2	5	7
VF	SGNJ[N]	src2	src1	R	S/[A]H/B	dest	OP
VF	SGNJX	src2	src1	R	S/[A]H/B	dest	OP

For RV64, or RV32 and FLEN$\leq$32 only, instructions are provided to move vectorial bit patterns between the floating-point and integer registers. VFMV.X.*vfmt* moves the values packed in floating-point register *rs1* as a vector with entries represented in either IEEE 754-2008 standard encoding (for single- and half-precision or as specified in Figure 2.1 or Figure 3.1 (for alternative half-precision or quarter-precision, respectively) to the lower FLEN bits of integer register *rd*. For RV64 and FLEN<64, or RV32 and FLEN<32, the higher bits of the destination register are filled with copies of the highest entry's sign bit. VFMV.*vfmt*.X moves the vectorial floating-point values packed as entries encoded in IEEE 754-2008 standard encoding (for signle- and half-precision) or as in Figure 2.1 or Figure 3.1 (for alternative half-precision or quarter-precision, respectively) from the lower FLEN bits in integer register *rs1* to the floating-point register *rd*. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved.

This instructions are provided for the case when non-IEEE representation is used inside the floating-point and floating-point registers.

In the typical case, these move instructions can be mapped onto regular floating-point moves (FMV).

31 30	29 25	24 20	19 15	14	13 12	11 7	6 0
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode
2	5	5	5	1	2	5	7
VF	MV	0	src	dir	S/[A]H/B	dest	OP

4.7 Vectorial Floating-Point Packing Conversion Instructions

For FLEN≥32 only, packing floating-point conversion pseudo-instructions VFCPK.*vfmt*.*fmt*, *idx* allow casting two scalar floating-point values and packing the result as two adjacent entries into a vectorial operand at positions 2*idx* and 2*idx*+1.

For FLEN≥32, vectorial single-precision to floating-point packing conversion pseudo-instructions, VFCPK.*vfmt*.S, *idx* are encoded as VFCPK[A-D].*vfmt*.S in the OP major opcode space using the RVF-type instruction format. For FLEN=64, vectorial double-precision to floating-point packing conversion pseudo-instructions, VFCPK.*vfmt*.D, *idx* are encoded as VFCPK[A-D].*vfmt*.S in the OP major opcode space using the RVF-type instruction format.

Legal values for *fmt* and *idx* for a given target format are dependent on FLEN and given in Table 4.4. All other combinations are illegal instructions.

VFCPK.*vfmt*.S, *idx* will convert two single-precision floating-point values from floating-point registers *rs1* and *rs2* into the format specified in the *vfmt* field. VFCPK.*vfmt*.D, *idx* will convert two double-precision floating-point values from floating-point registers *rs1* and *rs2* into the format specified in the *vfmt* field. The resulting two values are packed and stored as entries $\texttt{op}_{2idx}$ and $\texttt{op}_{2idx+1}$ in floating-point register *rd*. Other entries in *rd* are preserved.

Table 4.4: Legal *fmt* and *idx* values for VFCPK.*vfmt*.S, *idx* for various FLEN and *vfmt* combinations.

		Target *vfmt*			
Legal [*fmt*/*idx*]		F	Xf16	Xf16alt	Xf8
FLEN	32	×	[S/0]	[S/0]	[S/0,1]
	64	[S,D/0]	[S,D/0,1]	[S,D/0,1]	[S,D/0-3]

31 30	29 25	24 20	19 15	14	13 12	11 7	6 0
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode
2	5	5	5	1	2	5	7
VF	CPK[A..D].*vfmt*.S	src2	src1	[A-D]	S/[A]H/B	dest	OP
VF	CPK[A..D].*vfmt*.D	src2	src1	[A-D]	S/[A]H/B	dest	OP

4.8 Vectorial Floating-Point Compare Instructions

Vectorial floating-point compare instructions perform the specified comparison (equal, not equal, less than, less than or equal, greater than, greater than or equal) between the entries of floating-point registers *rs1* and *rs2* and record the Boolean result in integer register *rd*. A Boolean `FALSE` corresponds to an entry with all bits cleared, while `TRUE` corresponds to all bits set.

VFLT.*vfmt*, VFLE.*vfmt*, VFGT.*vfmt* and VFGE.*vfmt* perform what the IEEE 754-2008 standard refers to as *signaling* comparisons: that is, an Invalid Operation exception is raised if either input is NaN. VFEQ.*vfmt* and VFNE.*vfmt* perform a *quiet* comparison: only signaling NaN inputs cause an Invalid Operation exception. For all six instructions, the result entry is 0 if either operand entry is NaN.

*While VFLT.*vfmt*, VFGT.*vfmt *and VFEQ.*vfmt *are equivalent to reversing operands of VFGE.*vfmt*, VFLE.*vfmt *and VFNE.*vfmt*, respectively, this property is not given when the replication bit is set. For that reason, all six instructions are explicitly encoded.*

Setting or clearing all destination bits of an entry was chosen so that masking and bit manipulation operations can be more easily performed on the resulting pattern.

VFLT.R.*vfmt*, VFGT.R.*vfmt*, VFEQ.R.*vfmt*, VFG.R.*vfmt*, VFLE.R.*vfmt* and VFNE.R.*vfmt* behave analogously, but use the replicated first entry of *rs2* for all entries of *rs2*.

31 30	29 25	24 20	19 15	14	13 12	11 7	6 0
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode
2	5	5	5	1	2	5	7
VF	EQ/NE	src2	src1	R	S/[A]H/B	dest	OP
VF	LT/LE	src2	src1	R	S/[A]H/B	dest	OP
VF	GT/GE	src2	src1	R	S/[A]H/B	dest	OP

4.9 Vectorial Floating-Point Classify Instruction

Unless RV32 and FLEN=64 and *vfmt*=B, the vectorial floating-point classify instructions, VF-CLASS.*vfmt*, examine the entries in floating-point register *rs1* and write to integer register *rd* a vector of 8-bit classification blocks that indicate the class of each floating-point entry using a bit mask. The format of the classification block is described in Figure 4.2.

The corresponding *mask* bit in an entry of *rd* will be set if the property is true and clear otherwise. All other *mask* bits in *rd* are cleared. Note that exactly one *mask* bit in each entry *rd* will be set. The sign bit is placed in the highest bit of the block.

The classification block corresponding to the first entry in *rs1* is placed in the lowest byte of *rd*, the second inside the second byte and so on. For less populous formats, the classification block of the highest entry is replicated to fill the entire destination register.

31 30	29 25	24 20	19 15	14	13 12	11 7	6 0
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode
2	5	5	5	1	2	5	7
VF	CLASS	00001	src	0	[A]H/B	dest	OP

7	6	5 0
sign	$\overline{sign}$	*mask*

(a) Format of the classification block.

sign	*mask* bit	Meaning
0	0	*rs1* is $+\infty$.
1	0	*rs1* is $-\infty$.
0	1	*rs1* is a positive normal number.
1	1	*rs1* is a negative normal number.
0	2	*rs1* is a positive subnormal number.
1	2	*rs1* is a negative subnormal number.
0	3	*rs1* is +0.
1	3	*rs1* is −0.
-	4	*rs1* is a signaling NaN.
-	5	*rs1* is a quiet NaN.

(b) Meaning of *class* mask bits.

31 24	23 16	15 8	7 0
$\texttt{class}_3$	$\texttt{class}_2$	$\texttt{class}_1$	$\texttt{class}_0$

(c) Classification blocks in *rd* in RV32 for a vectorial format with four entries, e.g. vectorial quarter-precision.

31 24	23 16	15 8	7 0
$\texttt{class}_1$	$\texttt{class}_1$	$\texttt{class}_1$	$\texttt{class}_0$

(d) Classification blocks in *rd* in RV32 for a vectorial format with two entries, e.g. vectorial half-precision.

Figure 4.2: Format of classification block returned by FCLASS instruction.

We chose to replicate the block for cases of only two vector entries so that checks on the second entry can be performed by addressing the second entry in either byte- or half-word mode.

Chapter 5

“Xfaux” Non-Standard Extension for Auxiliary Operations on Scalar and Vectorial Floating-Point, Version 0.2

This section describes the non-standard auxiliary operation floating-point instruction-set extension, which is named “Xfaux” and adds non-standard floating-point computational instructions. If Xfaux is supported, its scalar floating-point instructions are added for *all* applicable floating-point formats supported in the system. If Xfvec is also supported, the vectorial floating-point instructions within Xfaux are added for *all* applicable vectorial floating-point formats. Thus, the vectorial floating-point extension depends on some or all of the following extensions: the D standard extension, the F standard extension, the Xf16 extension, the Xf16alt extension, and the Xf8 extension.

The extension is does not collide with [1], but uses previously unused space in the OP and FP-OP base opcodes. The formats referred to herein are defined in the F, Xf16, Xf16alt, and Xf8 extensions.

5.1 Auxiliary Expanding Scalar Floating-Point Computational Instructions

When either Xf16, Xf16alt or Xf8 are supported, expanding computational instructions are encoded in the OP-FP major opcode space using the R-type instruction format. FMULEX.S.*fmt* performs floating-point multiplication between *rs1* and *rs2*, writing the result to *rd* in single-precision (IEEE `binary32`). The operation is performed as if done on infinite precision and subsequently rounded to `binary32`.

FMACEX.S.*fmt* performs floating-point multiply-accumulation between *rs1*, *rs2* and *rd*, storing the result in *rd* in single-precision (IEEE `binary32`). The operation is performed as if done on infinite precision and subsequently rounded to `binary32`.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FMULEX.S	[A]H/B	src2	src1	RM	dest	OP-FP
FMACEX.S	[A]H/B	src2	src1	RM	src3=dest	OP-FP

There is also (barely) room for MULEX and MACEX to both FP16 formats, but its unclear whether they would be useful.

Other options would be ading FADDEX.S or even double-precision versions of the instructions.

5.2 Auxiliary Vectorial Floating-Point Computational Instructions

If Xfvec is supported, vectorial floating-point arithmetic instructions use the RVF-type format from Xfvec with the OP major opcode.

VFAVG.*vfmt* performs floating-point averaging between corresponding entries of *rs1* and *rs2*, writing the result to *rd*.

VFAVG.R.*vfmt* performs floating-point averaging between the entries of *rs1* and the first entry in *rs2*, writing the result to *rd*.

31 30	29 25	24 20	19 15	14	13 12	11 7	6 0
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode
2	5	5	5	1	2	5	7
VF	AVG	src2	src1	R	S/[A]H/B	dest	OP

5.3 Auxiliary Vectorial Floating-Point Computational Reduction Instructions

If Xfvec is supported, vector-to-scalar reduction instructions are encoded in the OP major opcode space using the RVF-type instruction format.

VFDOTP.*vfmt* performs floating-point dot product between the vectors in *rs1* and *rs2*. Since the destination is a scalar, it will be stored into the lowest entry of *rd*, as per Section 9.2 of [1].

VFDOTP.R.*vfmt* performs floating-point dot product using the entries in *rs1* and the first entry in *rs2*, storing the result in the lowest entry of *rd*.

31 30	29 25	24 20	19 15	14	13 12	11 7	6 0
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode
2	5	5	5	1	2	5	7
VF	DOTP	src2	src1	R	S/[A]H/B	dest	OP

5.4 Auxiliary Expanding Vectorial Floating-Point Computational Reduction Instructions

If Xfvec is supported, expanding vector-to-scalar instructions are encoded in the OP major opcode space using the RVF-type instruction format.

VFDOTPEX.S.*vfmt* performs floating-point dot product between the vectors in *rs1* and *rs2*, writing the result to *rd* in single-precision (IEEE `binary32`). The operation is performed as if done on infinite precision and subsequently rounded to `binary32`.

VFDOTPEX.S.R.*vfmt* performs floating-point dot product between the vectors in *rs1* and *rs2*, writing the result to *rd* in single-precision (IEEE `binary32`). The operation is performed as if done on infinite precision and subsequently rounded to `binary32`.

This instruction is invalid for single-precision arguments (*vfmt*=S).

31 30	29 25	24 20	19 15	14	13 12	11 7	6 0
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode
2	5	5	5	1	2	5	7
VF	DOTPEX.S	src2	src1	R	[A]H/B	dest	OP

Expanding dot product to double-precision or the 16-bit FP formats would be possible too, and may be added at a later point in time.

Chapter 6

smallFloat Instruction Set Extension Listings

This chapter lists all instructions specified in the smallFloat instruction set extensions (Xf16, Xf16alt, Xf8, Xfvec, Xfaux).

Table 6.1 contains a listing of newly added RISC-V pseudoinstructions.

31 30 29 27	26 25	24 20	19 15	14 13 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
rs3	fc2	rs2	rs1	rm*	rd	opcode	R4-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV32Xf16 Half-Precision Floating-Point Extension, bit[26,25]=10 (binary16)

imm[11:0]			rs1	001	rd	0000111	FLH	load
imm[11:5]		rs2	rs1	001	imm[4:0]	0100111	FSH	store
rs3	10	rs2	rs1	rm*	rd	1000011	FMADD.H	$(rs1 * rs2) + rs3$
rs3	10	rs2	rs1	rm*	rd	1000111	FMSUB.H	$(rs1 * rs2) - rs3$
rs3	10	rs2	rs1	rm*	rd	1001011	FNMSUB.H	$-(rs1 * rs2) + rs3$
rs3	10	rs2	rs1	rm*	rd	1001111	FNMADD.H	$-(rs1 * rs2) - rs3$
0000010		rs2	rs1	rm*	rd	1010011	FADD.H	$rs1 + rs2$
0000110		rs2	rs1	rm*	rd	1010011	FSUB.H	$rs1 - rs2$
0001010		rs2	rs1	rm*	rd	1010011	FMUL.H	$rs1 * rs2$
0001110		rs2	rs1	rm*	rd	1010011	FDIV.H	$rs1/rs2$
0101110		00000	rs1	rm*	rd	1010011	FSQRT.H	$\sqrt{rs1}$
0010010		rs2	rs1	000	rd	1010011	FSGNJ.H	rs1, sign of rs2
0010010		rs2	rs1	001	rd	1010011	FSGNJN.H	rs1, inv. sign of rs2
0010010		rs2	rs1	010	rd	1010011	FSGNJX.H	rs1, sign rs1 $\oplus$ sign rs2
0010110		rs2	rs1	000	rd	1010011	FMIN.H	min
0010110		rs2	rs1	001	rd	1010011	FMAX.H	max
1010010		rs2	rs1	010	rd	1010011	FEQ.H	equal
1010010		rs2	rs1	001	rd	1010011	FLT.H	less than
1010010		rs2	rs1	000	rd	1010011	FLE.H	less than or equal
1100010		00000	rs1	rm*	rd	1010011	FCVT.W.H	to sgn. word (32bit)
1100010		00001	rs1	rm*	rd	1010011	FCVT.WU.H	to usgn. word (32bit)
1101010		00000	rs1	rm*	rd	1010011	FCVT.H.W	from sgn. word (32bit)
1101010		00001	rs1	rm*	rd	1010011	FCVT.H.WU	from usgn. word (32bit)
1110010		00000	rs1	000	rd	1010011	FMV.X.H	fp reg $\rightarrow$ int reg
1110010		00000	rs1	001	rd	1010011	FCLASS.H	classify
1111010		00000	rs1	000	rd	1010011	FMV.H.X	int reg $\rightarrow$ fp reg

RV64Xf16 Half-Precision Floating-Point Extension (in addition to RV32Xf16)

1100010	00010	rs1	rm*	rd	1010011	FCVT.L.H	to sgn. long (64bit)
1100010	00011	rs1	rm*	rd	1010011	FCVT.LU.H	to usgn. long (64bit)
1101010	00010	rs1	rm*	rd	1010011	FCVT.H.L	from sgn. long (64bit)
1101010	00011	rs1	rm*	rd	1010011	FCVT.H.LU	from usgn. long (64bit)

Conversions with F Standard Extension

0100000	00010	rs1	000	rd	1010011	FCVT.S.H	binary16 $\rightarrow$ binary32
0100010	00000	rs1	rm*	rd	1010011	FCVT.H.S	binary32 $\rightarrow$ binary16

Conversions with D Standard Extension (in addition to the above)

0100001	00010	rs1	000	rd	1010011	FCVT.D.H	binary16 $\rightarrow$ binary64
0100010	00001	rs1	rm*	rd	1010011	FCVT.H.D	binary64 $\rightarrow$ binary16

* Only valid rounding modes allowed (000-100, 111)

31 30 29 27	26 25	24 20	19 15	14 13 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
rs3	fc2	rs2	rs1	rm*	rd	opcode	R4-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV32Xf16alt Alt. Half-Prec. Floating-Point Ext., bit[26,25]=10 (binary16alt)

imm[11:0]			rs1	001	rd	0000111	@FLAH	load
imm[11:5]		rs2	rs1	001	imm[4:0]	0100111	@FSAH	store
rs3	10	rs2	rs1	101	rd	1000011	FMADD.AH	$(rs1 * rs2) + rs3$
rs3	10	rs2	rs1	101	rd	1000111	FMSUB.AH	$(rs1 * rs2) - rs3$
rs3	10	rs2	rs1	101	rd	1001011	FNMSUB.AH	$-(rs1 * rs2) + rs3$
rs3	10	rs2	rs1	101	rd	1001111	FNMADD.AH	$-(rs1 * rs2) - rs3$
0000010		rs2	rs1	101	rd	1010011	FADD.AH	$rs1 + rs2$
0000110		rs2	rs1	101	rd	1010011	FSUB.AH	$rs1 - rs2$
0001010		rs2	rs1	101	rd	1010011	FMUL.AH	$rs1 * rs2$
0001110		rs2	rs1	101	rd	1010011	FDIV.AH	$rs1/rs2$
0101110		00000	rs1	101	rd	1010011	FSQRT.AH	$\sqrt{rs1}$
0010010		rs2	rs1	100	rd	1010011	FSGNJ.AH	rs1, sign of rs2
0010010		rs2	rs1	101	rd	1010011	FSGNJN.AH	rs1, inv. sign of rs2
0010010		rs2	rs1	110	rd	1010011	FSGNJX.AH	rs1, sign rs1 $\oplus$ sign rs2
0010110		rs2	rs1	100	rd	1010011	FMIN.AH	min
0010110		rs2	rs1	101	rd	1010011	FMAX.AH	max
1010010		rs2	rs1	110	rd	1010011	FEQ.AH	equal
1010010		rs2	rs1	101	rd	1010011	FLT.AH	less than
1010010		rs2	rs1	100	rd	1010011	FLE.AH	less than or equal
1100010		00000	rs1	101	rd	1010011	FCVT.W.AH	to sgn. word (32bit)
1100010		00001	rs1	101	rd	1010011	FCVT.WU.AH	to usgn. word (32bit)
1101010		00000	rs1	101	rd	1010011	FCVT.AH.W	from sgn. word (32bit)
1101010		00001	rs1	101	rd	1010011	FCVT.AH.WU	from usgn. word (32bit)
1110010		00000	rs1	100	rd	1010011	FMV.X.AH	fp reg $\rightarrow$ int reg
1110010		00000	rs1	101	rd	1010011	FCLASS.AH	classify
1111010		00000	rs1	100	rd	1010011	FMV.AH.X	int reg $\rightarrow$ fp reg

RV64Xf16alt Alt. Half-Prec. Floating-Point Ext. (in addition to RV32Xf16alt)

1100010	00010	rs1	101	rd	1010011	FCVT.L.AH	to sgn. long (64bit)
1100010	00011	rs1	101	rd	1010011	FCVT.LU.AH	to usgn. long (64bit)
1101010	00010	rs1	101	rd	1010011	FCVT.AH.L	from sgn. long (64bit)
1101010	00011	rs1	101	rd	1010011	FCVT.AH.LU	from usgn. long (64bit)

Conversions with F Standard Extension

0100000	00110	rs1	000	rd	1010011	FCVT.S.AH	binary16alt $\rightarrow$ binary32
0100010	00000	rs1	101	rd	1010011	FCVT.AH.S	binary32 $\rightarrow$ binary16alt

Conversions with D Standard Extension (in addition to the above)

0100001	00110	rs1	000	rd	1010011	FCVT.D.AH	binary16alt $\rightarrow$ binary64
0100010	00001	rs1	101	rd	1010011	FCVT.AH.D	binary64 $\rightarrow$ binary16alt

Conversions with Xf16 Extension

0100010	00110	rs1	rm*	rd	1010011	FCVT.H.AH	binary16alt $\rightarrow$ binary16
0100010	00010	rs1	101	rd	1010011	FCVT.AH.H	binary16 $\rightarrow$ binary16alt

* Only valid rounding modes allowed (000-100, 111)

31 30 29 27	26 25	24 20	19 15	14 13 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
rs3	fc2	rs2	rs1	rm*	rd	opcode	R4-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV32Xf8 Quarter-Precision Floating-Point Extension, bit[26,25]=11 (binary8)

imm[11:0]			rs1	000	rd	0000111	FLB	load
imm[11:5]		rs2	rs1	000	imm[4:0]	0100111	FSB	store
rs3	11	rs2	rs1	rm*	rd	1000011	FMADD.B	$(rs1 * rs2) + rs3$
rs3	11	rs2	rs1	rm*	rd	1000111	FMSUB.B	$(rs1 * rs2) - rs3$
rs3	11	rs2	rs1	rm*	rd	1001011	FNMSUB.B	$-(rs1 * rs2) + rs3$
rs3	11	rs2	rs1	rm*	rd	1001111	FNMADD.B	$-(rs1 * rs2) - rs3$
0000011		rs2	rs1	rm*	rd	1010011	FADD.B	$rs1 + rs2$
0000111		rs2	rs1	rm*	rd	1010011	FSUB.B	$rs1 - rs2$
0001011		rs2	rs1	rm*	rd	1010011	FMUL.B	$rs1 * rs2$
0001111		rs2	rs1	rm*	rd	1010011	FDIV.B	$rs1/rs2$
0101111		00000	rs1	rm*	rd	1010011	FSQRT.B	$\sqrt{rs1}$
0010011		rs2	rs1	000	rd	1010011	FSGNJ.B	rs1, sign of rs2
0010011		rs2	rs1	001	rd	1010011	FSGNJN.B	rs1, inv. sign of rs2
0010011		rs2	rs1	010	rd	1010011	FSGNJX.B	rs1, sign rs1 $\oplus$ sign rs2
0010111		rs2	rs1	000	rd	1010011	FMIN.B	min
0010111		rs2	rs1	001	rd	1010011	FMAX.B	max
1010011		rs2	rs1	010	rd	1010011	FEQ.B	equal
1010011		rs2	rs1	001	rd	1010011	FLT.B	less than
1010011		rs2	rs1	000	rd	1010011	FLE.B	less than or equal
1100011		00000	rs1	rm*	rd	1010011	FCVT.W.B	to sgn. word (32bit)
1100011		00001	rs1	rm*	rd	1010011	FCVT.WU.B	to usgn. word (32bit)
1101011		00000	rs1	rm*	rd	1010011	FCVT.B.W	from sgn. word (32bit)
1101011		00001	rs1	rm*	rd	1010011	FCVT.B.WU	from usgn. word (32bit)
1110011		00000	rs1	000	rd	1010011	FMV.X.B	fp reg $\rightarrow$ int reg
1110011		00000	rs1	001	rd	1010011	FCLASS.B	classify
1111011		00000	rs1	000	rd	1010011	FMV.B.X	int reg $\rightarrow$ fp reg

RV64Xf8 Quarter-Precision Floating-Point Extension (in addition to RV32Xf8)

1100011		00010	rs1	rm*	rd	1010011	FCVT.L.B	to sgn. long (64bit)
1100011		00011	rs1	rm*	rd	1010011	FCVT.LU.B	to usgn. long (64bit)
1101011		00010	rs1	rm*	rd	1010011	FCVT.B.L	from sgn. long (64bit)
1101011		00011	rs1	rm*	rd	1010011	FCVT.B.LU	from usgn. long (64bit)

Conversions with F Standard Extension

0100000		00011	rs1	000	rd	1010011	FCVT.S.B	binary8 $\rightarrow$ binary32
0100011		00000	rs1	rm*	rd	1010011	FCVT.B.S	binary32 $\rightarrow$ binary8

Conversions with D Standard Extension (in addition to the above)

0100001		00011	rs1	000	rd	1010011	FCVT.D.B	binary8 $\rightarrow$ binary64
0100011		00001	rs1	rm*	rd	1010011	FCVT.B.D	binary64 $\rightarrow$ binary8

Conversions with Xf16 Extension

0100010		00011	rs1	000	rd	1010011	FCVT.H.B	binary8 $\rightarrow$ binary16
0100011		00010	rs1	rm*	rd	1010011	FCVT.B.H	binary16 $\rightarrow$ binary8

Conversions with Xf16alt Extension

0100010		00011	rs1	101	rd	1010011	FCVT.AH.B	binary8 $\rightarrow$ binary16alt
0100011		00110	rs1	rm*	rd	1010011	FCVT.B.AH	binary16alt $\rightarrow$ binary8

* Only valid rounding modes allowed (000-100, 111)

31 30	29 27	26 25 24 20	19 15	14	13 12	11 7	6 0	
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode	RVF-type

Xfvec Vectorial Floating-Point Ext. with F, FLEN=64, vfmt=00 (binary32)

10	00001	rs2	rs1	0	00	rd	0110011	VFADD.S	$rs1 + rs2$
10	00001	rs2	rs1	1	00	rd	0110011	VFADD.R.S	$rs1 + rs2$, R
10	00010	rs2	rs1	0	00	rd	0110011	VFSUB.S	$rs1 - rs2$
10	00010	rs2	rs1	1	00	rd	0110011	VFSUB.R.S	$rs1 - rs2$, R
10	00011	rs2	rs1	0	00	rd	0110011	VFMUL.S	$rs1 * rs2$
10	00011	rs2	rs1	1	00	rd	0110011	VFMUL.R.S	$rs1 * rs2$, R
10	00100	rs2	rs1	0	00	rd	0110011	VFDIV.S	$rs1/rs2$
10	00100	rs2	rs1	1	00	rd	0110011	VFDIV.R.S	$rs1/rs2$, R
10	00101	rs2	rs1	0	00	rd	0110011	VFMIN.S	min
10	00101	rs2	rs1	1	00	rd	0110011	VFMIN.R.S	min, R
10	00110	rs2	rs1	0	00	rd	0110011	VFMAX.S	max
10	00110	rs2	rs1	1	00	rd	0110011	VFMAX.R.S	max, R
10	00111	00000	rs1	0	00	rd	0110011	VFSQRT.S	$\sqrt{rs1}$
10	01000	rs2	rs1	0	00	rd	0110011	VFMAC.S	$(rs1 * rs2) + rd$
10	01000	rs2	rs1	1	00	rd	0110011	VFMAC.R.S	$(rs1 * rs2) + rd$, R
10	01001	rs2	rs1	0	00	rd	0110011	VFMRE.S	$(rs1 * rs2) - rd$
10	01001	rs2	rs1	1	00	rd	0110011	VFMRE.R.S	$(rs1 * rs2) - rd$, R
10	01100	00001	rs1	0	00	rd	0110011	VFCLASS.S	classify
10	01101	rs2	rs1	0	00	rd	0110011	VFSGNJ.S	rs1, sign of rs2
10	01101	rs2	rs1	1	00	rd	0110011	VFSGNJ.R.S	rs1, sign of rs2, R
10	01110	rs2	rs1	0	00	rd	0110011	VFSGNJN.S	rs1, inv. sign of rs2
10	01110	rs2	rs1	1	00	rd	0110011	VFSGNJN.R.S	rs1, inv. sign of rs2, R
10	01111	rs2	rs1	0	00	rd	0110011	VFSGNJX.S	rs1, sign rs1 $\oplus$ sign rs2
10	01111	rs2	rs1	1	00	rd	0110011	VFSGNJX.R.S	rs1, sign rs1 $\oplus$ sign rs2, R
10	10000	rs2	rs1	0	00	rd	0110011	VFEQ.S	equal
10	10000	rs2	rs1	1	00	rd	0110011	VFEQ.R.S	equal, R
10	10001	rs2	rs1	0	00	rd	0110011	VFNE.S	not equal
10	10001	rs2	rs1	1	00	rd	0110011	VFNE.R.S	not equal, R
10	10010	rs2	rs1	0	00	rd	0110011	VFLT.S	less than
10	10010	rs2	rs1	1	00	rd	0110011	VFLT.R.S	less than, R
10	10011	rs2	rs1	0	00	rd	0110011	VFGE.S	greater than or equal
10	10011	rs2	rs1	1	00	rd	0110011	VFGE.R.S	greater than or equal, R
10	10100	rs2	rs1	0	00	rd	0110011	VFLE.S	less than or equal
10	10100	rs2	rs1	1	00	rd	0110011	VFLE.R.S	less than or equal, R
10	10101	rs2	rs1	0	00	rd	0110011	VFGT.S	greater than
10	10101	rs2	rs1	1	00	rd	0110011	VFGT.R.S	greater than, R
10	11000	rs2	rs1	0	00	rd	0110011	VFCPKA.S.S	2xbinary32 $\rightarrow$ binary32 *op0,1*
10	11010	rs2	rs1	0	00	rd	0110011	VFCPKA.S.D	2xbinary64 $\rightarrow$ binary32 *op0,1*

* Only valid rounding modes allowed (000-100, 111)

31 30	29 27	26 25 24 20	19 15	14	13 12	11 7	6 0	
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode	RVF-type

Unless RV32D Supported

10	01100	00000	rs1	0	00	rd	0110011	VFMV.X.S	fp reg → int reg
10	01100	00000	rs1	1	00	rd	0110011	VFMV.S.X	int reg → fp reg
10	01100	00010	rs1	0	00	rd	0110011	VFCVT.X.S	to vector of sgn. words
10	01100	00010	rs1	1	00	rd	0110011	VFCVT.XU.S	to vector of usgn. words
10	01100	00011	rs1	0	00	rd	0110011	VFCVT.S.X	from vector of sgn. words
10	01100	00011	rs1	1	00	rd	0110011	VFCVT.S.XU	from vector of usgn. words

* Only valid rounding modes allowed (000-100, 111)

31 30	29 27	26 25	24 20	19 15	14	13 12	11 7	6 0	
f2	vecfltop		rs2	rs1	R	vfmt	rd	opcode	RVF-type

Xfvec Vectorial Floating-Point Ext. with Xf16, FLEN$\geq$32, vfmt=10 (binary16)

10	00001	rs2	rs1	0	10	rd	0110011	VFADD.H	$rs1 + rs2$
10	00001	rs2	rs1	1	10	rd	0110011	VFADD.R.H	$rs1 + rs2$, R
10	00010	rs2	rs1	0	10	rd	0110011	VFSUB.H	$rs1 - rs2$
10	00010	rs2	rs1	1	10	rd	0110011	VFSUB.R.H	$rs1 - rs2$, R
10	00011	rs2	rs1	0	10	rd	0110011	VFMUL.H	$rs1 * rs2$
10	00011	rs2	rs1	1	10	rd	0110011	VFMUL.R.H	$rs1 * rs2$, R
10	00100	rs2	rs1	0	10	rd	0110011	VFDIV.H	$rs1/rs2$
10	00100	rs2	rs1	1	10	rd	0110011	VFDIV.R.H	$rs1/rs2$, R
10	00101	rs2	rs1	0	10	rd	0110011	VFMIN.H	min
10	00101	rs2	rs1	1	10	rd	0110011	VFMIN.R.H	min, R
10	00110	rs2	rs1	0	10	rd	0110011	VFMAX.H	max
10	00110	rs2	rs1	1	10	rd	0110011	VFMAX.R.H	max, R
10	00111	00000	rs1	0	10	rd	0110011	VFSQRT.H	$\sqrt{rs1}$
10	01000	rs2	rs1	0	10	rd	0110011	VFMAC.H	$(rs1 * rs2) + rd$
10	01000	rs2	rs1	1	10	rd	0110011	VFMAC.R.H	$(rs1 * rs2) + rd$, R
10	01001	rs2	rs1	0	10	rd	0110011	VFMRE.H	$(rs1 * rs2) - rd$
10	01001	rs2	rs1	1	10	rd	0110011	VFMRE.R.H	$(rs1 * rs2) - rd$, R
10	01100	00001	rs1	0	10	rd	0110011	VFCLASS.H	classify
10	01101	rs2	rs1	0	10	rd	0110011	VFSGNJ.H	rs1, sign of rs2
10	01101	rs2	rs1	1	10	rd	0110011	VFSGNJ.R.H	rs1, sign of rs2, R
10	01110	rs2	rs1	0	10	rd	0110011	VFSGNJN.H	rs1, inv. sign of rs2
10	01110	rs2	rs1	1	10	rd	0110011	VFSGNJN.R.H	rs1, inv. sign of rs2, R
10	01111	rs2	rs1	0	10	rd	0110011	VFSGNJX.H	rs1, sign rs1 $\oplus$ sign rs2
10	01111	rs2	rs1	1	10	rd	0110011	VFSGNJX.R.H	rs1, sign rs1 $\oplus$ sign rs2, R
10	10000	rs2	rs1	0	10	rd	0110011	VFEQ.H	equal
10	10000	rs2	rs1	1	10	rd	0110011	VFEQ.R.H	equal, R
10	10001	rs2	rs1	0	10	rd	0110011	VFNE.H	not equal
10	10001	rs2	rs1	1	10	rd	0110011	VFNE.R.H	not equal, R
10	10010	rs2	rs1	0	10	rd	0110011	VFLT.H	less than
10	10010	rs2	rs1	1	10	rd	0110011	VFLT.R.H	less than, R
10	10011	rs2	rs1	0	10	rd	0110011	VFGE.H	greater than or equal
10	10011	rs2	rs1	1	10	rd	0110011	VFGE.R.H	greater than or equal, R
10	10100	rs2	rs1	0	10	rd	0110011	VFLE.H	less than or equal
10	10100	rs2	rs1	1	10	rd	0110011	VFLE.R.H	less than or equal, R
10	10101	rs2	rs1	0	10	rd	0110011	VFGT.H	greater than
10	10101	rs2	rs1	1	10	rd	0110011	VFGT.R.H	greater than, R
10	11000	rs2	rs1	0	10	rd	0110011	VFCPKA.H.S	2xbinary32 $\rightarrow$ binary16 *op0,1*

[*] Only valid rounding modes allowed (000-100, 111)

31 30	29 27 26 25	24 20	19 15	14	13 12	11 7	6 0	
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode	RVF-type

Unless RV32D Supported

10	01100	00000	rs1	0	10	rd	0110011	VFMV.X.H	fp reg $\rightarrow$ int reg
10	01100	00000	rs1	1	10	rd	0110011	VFMV.H.X	int reg $\rightarrow$ fp reg
10	01100	00010	rs1	0	10	rd	0110011	VFCVT.X.H	to vector of sgn. halfwords
10	01100	00010	rs1	1	10	rd	0110011	VFCVT.XU.H	to vector of usgn. halfwords
10	01100	00011	rs1	0	10	rd	0110011	VFCVT.H.X	from vector of sgn. halfwords
10	01100	00011	rs1	1	10	rd	0110011	VFCVT.H.XU	from vector of usgn. halfwords

Conversions when D Standard Extension Supported (in addition to the above)

10	01100	00110	rs1	0	00	rd	0110011	VFCVT.S.H	*op0,1* binary16 $\rightarrow$ binary32
10	01100	00110	rs1	1	00	rd	0110011	VFCVTU.S.H	*op2,3* binary16 $\rightarrow$ binary32
10	01100	00100	rs1	0	10	rd	0110011	VFCVT.H.S	binary32 $\rightarrow$ binary16 *op0,1*
10	01100	00100	rs1	1	10	rd	0110011	VFCVTU.H.S	binary32 $\rightarrow$ binary16 *op2,3*
10	11000	rs2	rs1	1	10	rd	0110011	VFCPKB.H.S	2xbinary32 $\rightarrow$ binary16 *op2,3*
10	11010	rs2	rs1	0	10	rd	0110011	VFCPKA.H.D	2xbinary64 $\rightarrow$ binary16 *op0,1*
10	11010	rs2	rs1	1	10	rd	0110011	VFCPKB.H.D	2xbinary64 $\rightarrow$ binary16 *op2,3*

* Only valid rounding modes allowed (000-100, 111)

31 30	29 27	26 25 24 20	19 15	14	13 12	11 7	6 0	
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode	RVF-type

Xfvec Vectorial Floating-Point Ext. with Xf16alt, FLEN≥32, vfmt=01 (binary16alt)

10	00001	rs2	rs1	0	01	rd	0110011	VFADD.AH	$rs1 + rs2$
10	00001	rs2	rs1	1	01	rd	0110011	VFADD.R.AH	$rs1 + rs2$, R
10	00010	rs2	rs1	0	01	rd	0110011	VFSUB.AH	$rs1 - rs2$
10	00010	rs2	rs1	1	01	rd	0110011	VFSUB.R.AH	$rs1 - rs2$, R
10	00011	rs2	rs1	0	01	rd	0110011	VFMUL.AH	$rs1 * rs2$
10	00011	rs2	rs1	1	01	rd	0110011	VFMUL.R.AH	$rs1 * rs2$, R
10	00100	rs2	rs1	0	01	rd	0110011	VFDIV.AH	$rs1/rs2$
10	00100	rs2	rs1	1	01	rd	0110011	VFDIV.R.AH	$rs1/rs2$, R
10	00101	rs2	rs1	0	01	rd	0110011	VFMIN.AH	min
10	00101	rs2	rs1	1	01	rd	0110011	VFMIN.R.AH	min, R
10	00110	rs2	rs1	0	01	rd	0110011	VFMAX.AH	max
10	00110	rs2	rs1	1	01	rd	0110011	VFMAX.R.AH	max, R
10	00111	00000	rs1	0	01	rd	0110011	VFSQRT.AH	$\sqrt{rs1}$
10	01000	rs2	rs1	0	01	rd	0110011	VFMAC.AH	$(rs1 * rs2) + rd$
10	01000	rs2	rs1	1	01	rd	0110011	VFMAC.R.AH	$(rs1 * rs2) + rd$, R
10	01001	rs2	rs1	0	01	rd	0110011	VFMRE.AH	$(rs1 * rs2) - rd$
10	01001	rs2	rs1	1	01	rd	0110011	VFMRE.R.AH	$(rs1 * rs2) - rd$, R
10	01100	00001	rs1	0	01	rd	0110011	VFCLASS.AH	classify
10	01101	rs2	rs1	0	01	rd	0110011	VFSGNJ.AH	rs1, sign of rs2
10	01101	rs2	rs1	1	01	rd	0110011	VFSGNJ.R.AH	rs1, sign of rs2, R
10	01110	rs2	rs1	0	01	rd	0110011	VFSGNJN.AH	rs1, inv. sign of rs2
10	01110	rs2	rs1	1	01	rd	0110011	VFSGNJN.R.AH	rs1, inv. sign of rs2, R
10	01111	rs2	rs1	0	01	rd	0110011	VFSGNJX.AH	rs1, sign rs1 ⊕ sign rs2
10	01111	rs2	rs1	1	01	rd	0110011	VFSGNJX.R.AH	rs1, sign rs1 ⊕ sign rs2, R
10	10000	rs2	rs1	0	01	rd	0110011	VFEQ.AH	equal
10	10000	rs2	rs1	1	01	rd	0110011	VFEQ.R.AH	equal, R
10	10001	rs2	rs1	0	01	rd	0110011	VFNE.AH	not equal
10	10001	rs2	rs1	1	01	rd	0110011	VFNE.R.AH	not equal, R
10	10010	rs2	rs1	0	01	rd	0110011	VFLT.AH	less than
10	10010	rs2	rs1	1	01	rd	0110011	VFLT.R.AH	less than, R
10	10011	rs2	rs1	0	01	rd	0110011	VFGE.AH	greater than or equal
10	10011	rs2	rs1	1	01	rd	0110011	VFGE.R.AH	greater than or equal, R
10	10100	rs2	rs1	0	01	rd	0110011	VFLE.AH	less than or equal
10	10100	rs2	rs1	1	01	rd	0110011	VFLE.R.AH	less than or equal, R
10	10101	rs2	rs1	0	01	rd	0110011	VFGT.AH	greater than
10	10101	rs2	rs1	1	01	rd	0110011	VFGT.R.AH	greater than, R
10	11000	rs2	rs1	0	01	rd	0110011	VFCPKA.AH.S	2xfp32 → binary16alt *op0,1*

* Only valid rounding modes allowed (000-100, 111)

31 30	29 27	26 25 24 20	19 15	14	13 12	11 7	6 0		
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode	RVF-type	

Unless RV32D Supported

10	01100	00000	rs1	0	01	rd	0110011	VFMV.X.AH	fp reg → int reg
10	01100	00000	rs1	1	01	rd	0110011	VFMV.AH.X	int reg → fp reg
10	01100	00010	rs1	0	01	rd	0110011	VFCVT.X.AH	to vec. of sgn. halfwords
10	01100	00010	rs1	1	01	rd	0110011	VFCVT.XU.AH	to vec. of usgn. halfwords
10	01100	00011	rs1	0	01	rd	0110011	VFCVT.AH.X	from vec. of sgn. halfwords
10	01100	00011	rs1	1	01	rd	0110011	VFCVT.AH.XU	from vec. of usgn. halfwords

Conversions when D Standard Extension Supported (in addition to the above)

10	01100	00101	rs1	0	00	rd	0110011	VFCVT.S.AH	*op0,1* binary16alt → binary32
10	01100	00101	rs1	1	00	rd	0110011	VFCVTU.S.AH	*op2,3* binary16alt → binary32
10	01100	00100	rs1	0	01	rd	0110011	VFCVT.AH.S	binary32 → binary16alt *op0,1*
10	01100	00100	rs1	1	01	rd	0110011	VFCVTU.AH.S	binary32 → binary16alt *op2,3*
10	11000	rs2	rs1	1	01	rd	0110011	VFCPKB.AH.S	2xfp32 → binary16 *op2,3*
10	11010	rs2	rs1	0	01	rd	0110011	VFCPKA.AH.D	2xfp64 → binary16 *op0,1*
10	11010	rs2	rs1	1	01	rd	0110011	VFCPKB.AH.D	2xfp64 → binary16 *op2,3*

Conversions when Xf16 Extension Supported

10	01100	00101	rs1	0	10	rd	0110011	VFCVT.H.AH	vec. binary16alt → binary16
10	01100	00101	rs1	1	10	rd	0110011	VFCVTU.H.AH	vec. binary16alt → binary16
10	01100	00110	rs1	0	01	rd	0110011	VFCVT.AH.H	vec. binary16 → binary16alt
10	01100	00110	rs1	1	01	rd	0110011	VFCVTU.AH.H	vec. binary16 → binary16alt

* Only valid rounding modes allowed (000-100, 111)

31 30	29 27 26 25	24 20	19 15	14	13 12	11 7	6 0	
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode	RVF-type

Xfvec Vectorial Floating-Point Ext. with Xf8, FLEN≥16, vfmt=11 (binary8)

10	00001	rs2	rs1	0	11	rd	0110011	VFADD.B	$rs1 + rs2$
10	00001	rs2	rs1	1	11	rd	0110011	VFADD.R.B	$rs1 + rs2$, R
10	00010	rs2	rs1	0	11	rd	0110011	VFSUB.B	$rs1 - rs2$
10	00010	rs2	rs1	1	11	rd	0110011	VFSUB.R.B	$rs1 - rs2$, R
10	00011	rs2	rs1	0	11	rd	0110011	VFMUL.B	$rs1 * rs2$
10	00011	rs2	rs1	1	11	rd	0110011	VFMUL.R.B	$rs1 * rs2$, R
10	00100	rs2	rs1	0	11	rd	0110011	VFDIV.B	$rs1/rs2$
10	00100	rs2	rs1	1	11	rd	0110011	VFDIV.R.B	$rs1/rs2$, R
10	00101	rs2	rs1	0	11	rd	0110011	VFMIN.B	min
10	00101	rs2	rs1	1	11	rd	0110011	VFMIN.R.B	min, R
10	00110	rs2	rs1	0	11	rd	0110011	VFMAX.B	max
10	00110	rs2	rs1	1	11	rd	0110011	VFMAX.R.B	max, R
10	10111	00000	rs1	0	11	rd	0110011	VFSQRT.B	$\sqrt{rs1}$
10	01000	rs2	rs1	0	11	rd	0110011	VFMAC.B	$(rs1 * rs2) + rd$
10	01000	rs2	rs1	1	11	rd	0110011	VFMAC.R.B	$(rs1 * rs2) + rd$, R
10	01001	rs2	rs1	0	11	rd	0110011	VFMRE.B	$(rs1 * rs2) - rd$
10	01001	rs2	rs1	1	11	rd	0110011	VFMRE.R.B	$(rs1 * rs2) - rd$, R
10	01101	rs2	rs1	0	11	rd	0110011	VFSGNJ.B	rs1, sign of rs2
10	01101	rs2	rs1	1	11	rd	0110011	VFSGNJ.R.B	rs1, sign of rs2, R
10	01110	rs2	rs1	0	11	rd	0110011	VFSGNJN.B	rs1, inv. sign of rs2
10	01110	rs2	rs1	1	11	rd	0110011	VFSGNJN.R.B	rs1, inv. sign of rs2, R
10	01111	rs2	rs1	1	11	rd	0110011	VFSGNJX.B	rs1, sign rs1 $\oplus$ sign rs2
10	01111	rs2	rs1	0	11	rd	0110011	VFSGNJX.R.B	rs1, sign rs1 $\oplus$ sign rs2, R
10	10000	rs2	rs1	0	11	rd	0110011	VFEQ.B	equal
10	10000	rs2	rs1	1	11	rd	0110011	VFEQ.R.B	equal, R
10	10001	rs2	rs1	0	11	rd	0110011	VFNE.B	not equal
10	10001	rs2	rs1	1	11	rd	0110011	VFNE.R.B	not equal, R
10	10010	rs2	rs1	0	11	rd	0110011	VFLT.B	less than
10	10010	rs2	rs1	1	11	rd	0110011	VFLT.R.B	less than, R
10	10011	rs2	rs1	0	11	rd	0110011	VFGE.B	greater than or equal
10	10011	rs2	rs1	1	11	rd	0110011	VFGE.R.B	greater than or equal, R
10	10100	rs2	rs1	0	11	rd	0110011	VFLE.B	less than or equal
10	10100	rs2	rs1	1	11	rd	0110011	VFLE.R.B	less than or equal, R
10	10101	rs2	rs1	0	11	rd	0110011	VFGT.B	greater than
10	10101	rs2	rs1	1	11	rd	0110011	VFGT.R.B	greater than, R

* Only valid rounding modes allowed (000-100, 111)

31 30	29 27 26 25	24 20	19 15	14	13 12	11 7	6 0	
f2	vecfltop	rs2	rs1	R	vfmt	rd	opcode	RVF-type

Unless RV32D Supported

10	01100	00000	rs1	0	11	rd	0110011	VFMV.X.B	fp reg → int reg
10	01100	00000	rs1	1	11	rd	0110011	VFMV.B.X	int reg → fp reg
10	01100	00001	rs1	0	11	rd	0110011	VFCLASS.B	classify
10	01100	00010	rs1	0	11	rd	0110011	VFCVT.X.B	to vector of sgn. bytes
10	01100	00010	rs1	1	11	rd	0110011	VFCVT.XU.B	to vector of usgn. bytes
10	01100	00011	rs1	0	11	rd	0110011	VFCVT.B.X	from vector of sgn. bytes
10	01100	00011	rs1	1	11	rd	0110011	VFCVT.B.XU	from vector of usgn. bytes

Conversions when F Standard Extension Supported

10	11000	rs2	rs1	0	11	rd	0110011	VFCPKA.B.S	2xbinary32 → binary8 *op0,1*
10	11000	rs2	rs1	1	11	rd	0110011	VFCPKB.B.S	2xbinary32 → binary8 *op2,3*

Conversions when D Standard Extension Supported (in addition to the above)

10	11001	rs2	rs1	0	11	rd	0110011	VFCPKC.B.S	2xbinary32 → binary8 *op4,5*
10	11001	rs2	rs1	1	11	rd	0110011	VFCPKD.B.S	2xbinary32 → binary8 *op6,7*
10	11010	rs2	rs1	0	11	rd	0110011	VFCPKA.B.D	2xbinary64 → binary8 *op0,1*
10	11010	rs2	rs1	1	11	rd	0110011	VFCPKB.B.D	2xbinary64 → binary8 *op2,3*
10	11011	rs2	rs1	0	11	rd	0110011	VFCPKC.B.D	2xbinary64 → binary8 *op4,5*
10	11011	rs2	rs1	1	11	rd	0110011	VFCPKD.B.D	2xbinary64 → binary8 *op6,7*

Conversions when Xf16 Extension Supported

10	01100	00111	rs1	0	10	rd	0110011	VFCVT.H.B	*op0,1* binary8 → binary16
10	01100	00111	rs1	1	10	rd	0110011	VFCVTU.H.B	*op2,3* binary8 → binary16
10	01100	00110	rs1	0	11	rd	0110011	VFCVT.B.H	binary16 → binary8 *op0,1*
10	01100	00110	rs1	1	11	rd	0110011	VFCVTU.B.H	binary16 → binary8 *op2,3*

Conversions when Xf16alt Extension Supported

10	01100	00111	rs1	0	01	rd	0110011	VFCVT.AH.B	*op0,1* binary8 → binary16alt
10	01100	00111	rs1	1	01	rd	0110011	VFCVTU.AH.B	*op2,3* binary8 → binary16alt
10	01100	00101	rs1	0	11	rd	0110011	VFCVT.B.AH	binary16alt → binary8 *op0,1*
10	01100	00101	rs1	1	11	rd	0110011	VFCVTU.B.AH	binary16alt → binary8 *op2,3*

* Only valid rounding modes allowed (000-100, 111)

31 30	29 27	26 25	24 20	19 15	14	13 12	11 7	6 0	
f2	vecfltop		rs2	rs1	R	vfmt	rd	opcode	RVF-type

Xfaux Auxiliary Floating-Point Extension with F, fmt=00 (binary32)

When Xfvec Extension Supported, FLEN=64, vfmt=00 (binary32)

10	01010	rs2	rs1	0	00	rd	0110011	VFDOTP.S	dotp(rs1,rs2)
10	01010	rs2	rs1	1	00	rd	0110011	VFDOTP.R.S	dotp(rs1,rs2), R
10	10110	rs2	rs1	0	00	rd	0110011	VFAVG.S	$(rs1 + rs2)/2$
10	10110	rs2	rs1	1	00	rd	0110011	VFAVG.R.S	$(rs1 + rs2)/2$, R

Xfaux Auxiliary Floating-Point Extension with Xf16, fmt=10 (binary16)

0100110	rs2	rs1	rm*	rd	1010011	FMULEX.S.H	fp32($rs1 * rs2$)
0101010	rs2	rs1	rm*	rd	1010011	FMACEX.S.H	fp32($(rs1 * rs2) + rd$)

When Xfvec Extension Supported, FLEN≥32, vfmt=10 (binary16)

10	01010	rs2	rs1	0	10	rd	0110011	VFDOTP.H	dotp(rs1,rs2)
10	01010	rs2	rs1	1	10	rd	0110011	VFDOTP.R.H	dotp(rs1,rs2), R
10	01011	rs2	rs1	0	10	rd	0110011	VFDOTPEX.S.H	fp32(dotp(rs1,rs2))
10	01011	rs2	rs1	1	10	rd	0110011	VFDOTPEX.S.R.H	fp32(dotp(rs1,rs2)), R
10	10110	rs2	rs1	0	10	rd	0110011	VFAVG.H	$(rs1 + rs2)/2$
10	10110	rs2	rs1	1	10	rd	0110011	VFAVG.R.H	$(rs1 + rs2)/2$, R

Xfaux Auxiliary Floating-Point Extension with Xf16alt, fmt=10 (binary16alt)

0100110	rs2	rs1	101	rd	1010011	FMULEX.S.AH	fp32($rs1 * rs2$)
0101010	rs2	rs1	101	rd	1010011	FMACEX.S.AH	fp32($(rs1 * rs2) + rd$)

When Xfvec Extension Supported, FLEN≥32, vfmt=01 (binary16alt)

10	01010	rs2	rs1	0	01	rd	0110011	VFDOTP.AH	dotp(rs1,rs2)
10	01010	rs2	rs1	1	01	rd	0110011	VFDOTP.R.AH	dotp(rs1,rs2), R
10	01011	rs2	rs1	0	01	rd	0110011	VFDOTPEX.S.AH	fp32(dotp(rs1,rs2))
10	01011	rs2	rs1	1	01	rd	0110011	VFDOTPEX.S.R.AH	fp32(dotp(rs1,rs2)), R
10	10110	rs2	rs1	0	01	rd	0110011	VFAVG.AH	$(rs1 + rs2)/2$
10	10110	rs2	rs1	1	01	rd	0110011	VFAVG.R.AH	$(rs1 + rs2)/2$, R

Xfaux Auxiliary Floating-Point Extension with Xf8, fmt=11 (binary8)

0100111	rs2	rs1	rm*	rd	1010011	FMULEX.S.B	fp32($rs1 * rs2$)
0101011	rs2	rs1	rm*	rd	1010011	FMACEX.S.B	fp32($(rs1 * rs2) + rd$)

When Xfvec Extension Supported, FLEN≥16, vfmt=11 (binary8)

10	01010	rs2	rs1	0	11	rd	0110011	VFDOTP.B	dotp(rs1,rs2)
10	01010	rs2	rs1	1	11	rd	0110011	VFDOTP.R.B	dotp(rs1,rs2), R
10	01011	rs2	rs1	0	11	rd	0110011	VFDOTPEX.S.B	fp32(dotp(rs1,rs2))
10	01011	rs2	rs1	1	11	rd	0110011	VFDOTPEX.S.R.B	fp32(dotp(rs1,rs2)), R
10	10110	rs2	rs1	0	11	rd	0110011	VFAVG.B	$(rs1 + rs2)/2$
10	10110	rs2	rs1	1	11	rd	0110011	VFAVG.R.B	$(rs1 + rs2)/2$, R

* Only valid rounding modes allowed (000-100, 111)

Table 6.1: smallFloat RISC-V pseudoinstructions.

Pseudoinstruction	Base Instruction(s)	Meaning
`flah rd, rt, offset`	`flh rd, rt, offset`	Alternative half-precision load
`fsah rd, rt, offset`	`fsh rd, rt, offset`	Alternative half-precision store

`fl{h|ah|b} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`fl{h|h|b} rd, symbol[11:0](rt)`
		Floating-point load global

`fs{h|ah|b} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`fs{h|h|b} rd, symbol[11:0](rt)`
		Floating-point store global

`fmv.h rd, rs`	`fsgnj.h rd, rs, rs`	Copy half-precision register
`fabs.h rd, rs`	`fsgnjx.h rd, rs, rs`	Half-precision absolute value
`fneg.h rd, rs`	`fsgnjn.h rd, rs, rs`	Half-precision negate
`fmv.ah rd, rs`	`fsgnj.ah rd, rs, rs`	Copy alt. half-precision register
`fabs.ah rd, rs`	`fsgnjx.ah rd, rs, rs`	Alt. half-precision absolute value
`fneg.ah rd, rs`	`fsgnjn.ah rd, rs, rs`	Alt. half-precision negate
`fmv.b rd, rs`	`fsgnj.b rd, rs, rs`	Copy quarter-precision register
`fabs.b rd, rs`	`fsgnjx.b rd, rs, rs`	Quarter-precision absolute value
`fneg.b rd, rs`	`fsgnjn.b rd, rs, rs`	Quarter-precision negate
`vfcpk.s.s, rd, rs1, rs2, 0`	`vfckpa.s.s, rd, rs1, rs2`	Cast 2x fp32 to fp32 vec. *op0,1*
`vfcpk.s.d, rd, rs1, rs2, 0`	`vfckpa.s.d, rd, rs1, rs2`	Cast 2x fp64 to fp32 vec. *op0,1*
`vfcpk.h.s, rd, rs1, rs2, 0`	`vfckpa.h.s, rd, rs1, rs2`	Cast 2x fp32 to fp16 vec. *op0,1*
`vfcpk.h.s, rd, rs1, rs2, 1`	`vfckpb.h.s, rd, rs1, rs2`	Cast 2x fp32 to fp16 vec. *op2,3*
`vfcpk.h.d, rd, rs1, rs2, 0`	`vfckpa.h.d, rd, rs1, rs2`	Cast 2x fp64 to fp16 vec. *op0,1*
`vfcpk.h.d, rd, rs1, rs2, 1`	`vfckpb.h.d, rd, rs1, rs2`	Cast 2x fp64 to fp16 vec. *op2,3*
`vfcpk.ah.s, rd, rs1, rs2, 0`	`vfckpa.ah.s, rd, rs1, rs2`	Cast 2x fp32 to fp16alt vec. *op0,1*
`vfcpk.ah.s, rd, rs1, rs2, 1`	`vfckpb.ah.s, rd, rs1, rs2`	Cast 2x fp32 to fp16alt vec. *op2,3*
`vfcpk.ah.d, rd, rs1, rs2, 0`	`vfckpa.ah.d, rd, rs1, rs2`	Cast 2x fp64 to fp16alt vec. *op0,1*
`vfcpk.ah.d, rd, rs1, rs2, 1`	`vfckpb.ah.d, rd, rs1, rs2`	Cast 2x fp64 to fp16alt vec. *op2,3*
`vfcpk.b.s, rd, rs1, rs2, 0`	`vfckpa.b.s, rd, rs1, rs2`	Cast 2x fp32 to fp8 vec. *op0,1*
`vfcpk.b.s, rd, rs1, rs2, 1`	`vfckpb.b.s, rd, rs1, rs2`	Cast 2x fp32 to fp8 vec. *op2,3*
`vfcpk.b.s, rd, rs1, rs2, 2`	`vfckpc.b.s, rd, rs1, rs2`	Cast 2x fp32 to fp8 vec. *op4,5*
`vfcpk.b.s, rd, rs1, rs2, 3`	`vfckpd.b.s, rd, rs1, rs2`	Cast 2x fp32 to fp8 vec. *op6,7*
`vfcpk.b.d, rd, rs1, rs2, 0`	`vfckpa.b.d, rd, rs1, rs2`	Cast 2x fp64 to fp8 vec. *op0,1*
`vfcpk.b.d, rd, rs1, rs2, 1`	`vfckpb.b.d, rd, rs1, rs2`	Cast 2x fp64 to fp8 vec. *op2,3*
`vfcpk.b.d, rd, rs1, rs2, 2`	`vfckpc.b.d, rd, rs1, rs2`	Cast 2x fp64 to fp8 vec. *op4,5*
`vfcpk.b.d, rd, rs1, rs2, 3`	`vfckpd.b.d, rd, rs1, rs2`	Cast 2x fp64 to fp8 vec. *op6,7*

Bibliography

[1] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: Base user-level isa version 2.2. 2017.