RISC-V "V" Vector Extension

Version 0.7.0-draft-20190222

Table of Contents

Contributors include: Alon Amid, Krste Asanovic, Allen Baum, Alex Bradbury, Tony Brewer, Chris Celio, Silviu Chiricescu, Ken Dockser, Bob Dreyer, Roger Espasa, Sean Halle, John Hauser, David Horner, Bruce Hoult, Bill Huffman, Constantine Korikov, Ben Korpan, Robin Kruppe, Yunsup Lee, Guy Lemieux, Rich Newell, Albert Ou, David Patterson, Colin Schmidt, Alex Solomatnikov, Steve Wallach, Andrew Waterman, Jim Wilson.

Known issues with current version:

- encoding needs better formatting
- vector memory consistency model needs to be clarified
- interaction with privileged architectures

1. Introduction

This document describes the draft of the RISC-V base vector extension. The document describes all the individual features of the base vector extension.

Note

This is a draft of a stable proposal for the vector specification to be used for implementation and evaluation. Once the draft label is removed, version 0.7 is intended to be stable enough to begin developing toolchains, functional simulators, and initial implementations, though will continue to evolve with minor changes and updates.

The term *base vector extension* is used informally to describe the standard set of vector ISA components. This draft spec is intended to capture how a certain vector function will be implemented as vector instructions, but to not yet determine what set of vector instructions are mandatory for a given platform.

Note

Each actual platform profile will formally specify the mandatory components of any vector extension adopted by that platform. The base vector extension can expected to be close to that which will eventually be used in the standard Unix platform profile that supports vectors. Other platforms, including embedded platforms, may choose to implement subsets of these extensions. The exact set of mandatory supported instructions for an implementation to be compliant with a given profile is subject to change until each profile spec is ratified.

The base vector extension is designed to act as a base for additional vector extensions in various domains, including cryptography and machine learning.

2. Implementation-defined Constant Parameters

Each hart supporting the vector extension defines three parameters:

1. The maximum size of a single vector element in bits, *ELEN*, which must be a power of 2.

2. The number of bits in a vector register, *VLEN* ≥ *ELEN*, which must be a power of 2.

3. The striping distance in bits, *SLEN*, which must be VLEN ≥ SLEN ≥ 32, and which must be a power of 2.

Note

Platform profiles may set further constraints on these parameters, for example, requiring that ELEN ≥ max(XLEN,FLEN), or requiring a minimum VLEN value, or setting an SLEN value.

The ISA supports writing binary code that under certain constraints will execute portably on harts with different values for these parameters.

Note

Note

Code can be written that will expose differences in implementation parameters.

Thread contexts with active vector state cannot be migrated during execution between harts that have any difference in VLEN, ELEN, or SLEN parameters.

3. Vector Extension Programmer’s Model

The vector extension adds 32 vector registers, and five unprivileged CSRs (`vstart`, `vxsat`, `vxrm`, `vtype`, `vl`) to a base scalar RISC-V ISA. If the base scalar ISA does not include floating-point, then a `fcsr` register is also added to hold mirrors of the `vxsat` and `vxrm` CSRs as explained below.

Table 1. New vector CSRs

Address	Privilege	Name	Description
0x008	URW	vstart	Vector start position
0x009	URW	vxsat	Fixed-Point Saturate Flag
0x00A	URW	vxrm	Fixed-Point Rounding Mode
0x00B	URW	vtype	Vector data type register
0xC20	URO	vl	Vector length

3.1. Vector Registers

The vector extension adds 32 architectural vector registers, `v0`-`v31` to the base scalar RISC-V ISA.

Each vector register has a fixed VLEN bits of state.

Note: Zfinx ("F in X") is a new ISA option under consideration where floating-point instructions take their arguments from the integer register file. The 0.7 vector extension is also compatible with this option.

3.2. Vector Start Index CSR `vstart`

The `vstart` read-write CSR specifies the index of the first element to be executed by a vector instruction.

Normally, `vstart` is only written by hardware on a trap on a vector instruction, with the `vstart` value representing the element on which the trap was taken (either a synchronous exception or an asynchronous interrupt), and at which execution should resume after a resumable trap is handled.

All vector instructions are defined to begin execution with the element number given in the `vstart` CSR, leaving earlier elements in the destination vector undisturbed, and to reset the `vstart` CSR to zero at the end of execution.

If the value in the `vstart` register is greater than or equal to the vector length `vl` then no element operations are performed, though elements at the end of the destination vector past `vl` are zeroed, and the `vstart` register is reset to zero.

The `vstart` CSR is defined to have only enough writeable bits to hold the largest element index (one less than the maximum VLMAX) or lg2(VLEN) bits. The upper bits of the `vstart` CSR are hardwired to zero (reads zero, writes ignored).

Note: The maximum vector length is obtained with the largest LMUL setting (8) and the smallest SEW setting (8), so VLMAX_max = 8*VLEN/8 = VLEN. For example, for VLEN=256, `vstart` would have 8 bits to represent indices from 0 through 255.

The `vstart` CSR is writable by unprivileged code, but non-zero `vstart` values may cause vector instructions to run substantially slower on some implementations, so `vstart` should not be used by application programmers. A few vector instructions can not be executed with a non-zero `vstart` value and will raise an illegal instruction exception as defined below.

3.3. Vector Fixed-Point Rounding Mode Register `vxrm`

The vector fixed-point rounding-mode register holds a two-bit read-write rounding-mode field. The vector fixed-point rounding-mode is given a separate CSR address to allow independent access, but is also reflected as a field in the upper bits of `fcsr`. Systems without floating-point must add `fcsr` when adding the vector extension.

Table 2. vxrm encoding

Bits [1:0]		Abbreviation	Rounding Mode
0	0	rnu	round-to-nearest-up (add +0.5 LSB)
0	1	rne	round-to-nearest-even
1	0	rdn	round-down (truncate)
1	1	rod	round-to-odd (OR bits into LSB, aka "jam")

Bits[XLEN-1:2] should be written as zeros.

Note: The rounding mode can be set with a single `csrwi` instruction.

3.4. Vector Fixed-Point Saturation Flag `vxsat`

The `vxsat` CSR holds a single read-write bit that indicates if a fixed-point instruction has had to saturate an output value to fit into a destination format.

The `vxsat` bit is mirrored in the upper bits of `fcsr`.

3.5. Vector Fixed-Point Fields in `fcsr`

The `vxrm` and `vxsat` separate CSRs can also be accessed via fields in the floating-point CSR, `fcsr`. The `fcsr` register must be added to systems without floating-point that add a vector extension.

Table 3. fcsr layout

Bits	Name	Description
10:9	vxrm	Fixed-point rounding mode
8	vxsat	Fixed-point accrued saturation flag
7:5	frm	Floating-point rounding mode
4:0	fflags	Floating-point accrued exception flags

Note: The fields are packed into `fcsr` to make context-save/restore faster.

3.6. Vector type register, `vtype`

The XLEN-wide *vector type* CSR, `vtype` provides the default type used to interpret the contents of the vector register file. The vector type also determines the organization of elements in each vector register, and how multiple vector registers are grouped.

In the base vector extension, the `type` register has two fields, `vsew[2:0]`, and `vlmul`.

vtype-format.adoc

Note: The smallest base implementation has only four bits in `vtype`, two bits for `vsew[1:0]` and two bits for `vlmul[1:0]`.

Note: The vediv[1:0] field is used by the EDIV extension described below.

Note: Further standard and custom extensions to the vector base will extend these fields to support a greater variety of data types.

Note: An extended instruction encoding length would allow these fields to be specified in the instruction encoding, though `vlmul` might want to be varied with AVL.

3.6.1. Vector standard element width `vsew`

The value in `vsew` sets the dynamic *standard element width* (SEW). By default, a vector register is viewed as being divided into VLEN / SEW standard-width elements. In the base vector extension, only SEW up to max(XLEN,FLEN) are required to be supported.

Table 4. vsew[2:0] (standard element width) encoding

vsew[2:0]			SEW
0	0	0	8
0	0	1	16
0	1	0	32
0	1	1	64
1	0	0	128
1	0	1	256
1	1	0	512
1	1	1	1024

Table 5. Example VLEN = 128 bits

SEW	Elements per vector register
64	2
32	4
16	8
8	16

3.6.2. Vector Register Grouping (`vlmul`)

Multiple vector registers can be grouped together to form a *vector register group*, so that a single vector instruction can operate on multiple vector registers. Vector register groups allow double-width or larger elements to be operated on with the same vector length as standard-width elements. Vector register groups also provide greater execution efficiency for longer application vectors.

The number of vector registers in a group, *LMUL*, is an integer power of two set by the `vlmul` field in `vtype` (LMUL = $2^{\texttt{vlmul[1:0]}}$). The maximum vector length possible in a single vector instruction, VLMAX, is then increased by a factor of LMUL.

vlmul		LMUL	#groups	VLMAX	Grouped registers
0	0	1	32	VLEN/SEW	vn (no group)
0	1	2	16	2*VLEN/SEW	vn, vn+1
1	0	4	8	4*VLEN/SEW	vn, ..., vn+3
1	1	8	4	8*VLEN/SEW	vn, ..., vn+7

When `vlmul=01`, then vector operations on register v *n* also operate on vector register v *n*+1, giving twice the vector length in bits. Instructions specifying a vector operand with an odd-numbered vector register will raise an illegal instruction exception.

Similarly, when `vlmul=10`, vector instructions operate on four vector registers at a time, and instructions specifying vector operands using vector register numbers that are not multiples of four will raise an illegal instruction exception. When `vlmul=11`, operations operate on eight vector registers at a time, and instructions specifying vector operands using register numbers that are not multiples of eight will raise an illegal instruction exception.

Note: This grouping pattern (LMUL=8 has groups `v0`,`v8`,`v16`,`v24`) was adopted in 0.6 initially to avoid issues with the floating-point calling convention when floating-point values were overlaid on the vector registers, whereas earlier versions kept the vector register group names contiguous (LMUL=8 has groups `v0`, v1, v2, v3). In v0.7, the floating-point registers are separate again.

Mask register instructions always operate on a single vector register, regardless of LMUL setting.

3.7. Vector Length Register `vl`

The *XLEN*-bit-wide read-only `vl` CSR can only be updated by the `vsetvli` and `vsetvl` instructions, and the *fault-only-first* vector load instruction variants.

The `vl` register holds an unsigned integer specifying the number of elements to be updated by a vector instruction. Elements in the destination vector with indices ≥ `vl` are zeroed during execution of a vector instruction. As a special case, when `vl`=0, no elements are updated in the destination vector.

Note: The number of bits implemented in `vl` depends on the implementation's maximum vector length of the smallest supported type. The smallest vector implementation, RV32IV, would need at least six bits in `vl` to hold the values 0-32 (with VLEN=32, LMUL=8 and SEW=8 results in VLMAX of 32).

4. Mapping of Vector Elements to Vector Register State

The following diagrams illustrate how different width elements are packed into the bytes of a vector register depending on the current SEW and LMUL settings, as well as implementation ELEN and VLEN. Elements are packed into each vector register with the least-significant byte in the lowest-numbered bits.

> Note: Previous RISC-V vector proposals (< 0.6) hid this mapping from software, whereas this proposal has a specific mapping for all configurations, which reduces implementation flexibility but removes need for zeroing on config changes. Making the mapping explicit also has the advantage of simplifying oblivious context save-restore code, as the code can save the configuration in `vl` and `vtype`, then reset `vtype` to a convenient value (e.g., four vector groups of LMUL=8, SEW=ELEN) before saving all vector register bits without needing to parse the configuration. The reverse process will restore the state.

4.1. Mapping with LMUL=1

When LMUL=1, elements are simply packed in order from the least-significant to most-significant bits of the vector register.

> Note: To increase readability, vector register layouts are drawn with bytes ordered from right to left with increasing byte address. Bits within an element are numbered in a little-endian format with increasing bit index from right to left corresponding to increasing magnitude.

The element index is given in hexadecimal and is shown placed at the least-significant byte of the stored element. ELEN ⇐128 and LMUL=1 throughout.

```
VLEN=32b

Byte         3 2 1 0

SEW=8b       3 2 1 0
SEW=16b        1   0
SEW=32b            0

VLEN=64b

Byte        7 6 5 4 3 2 1 0

SEW=8b      7 6 5 4 3 2 1 0
SEW=16b       3   2   1   0
SEW=32b           1       0
SEW=64b                   0

VLEN=128b

Byte        F E D C B A 9 8 7 6 5 4 3 2 1 0

SEW=8b      F E D C B A 9 8 7 6 5 4 3 2 1 0
SEW=16b       7   6   5   4   3   2   1   0
SEW=32b           3       2       1       0
SEW=64b                   1               0
SEW=128b                                  0

VLEN=256b

Byte     1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0

SEW=8b   1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0
SEW=16b     F   E   D   C   B   A   9   8   7   6   5   4   3   2   1   0
SEW=32b         7       6       5       4       3       2       1       0
SEW=64b                 3               2               1               0
SEW=128b                                1                               0
```

4.2. Mapping with LMUL > 1

When vector registers are grouped, the elements of the vector register group are striped across the constituent vector registers. The striping distance in bits, SLEN, sets how many bits are packed contiguously into one vector register before moving to the next in the group.

For example, when SLEN = 128, the striping pattern is repeated in multiples of 128 bits. The first 128/SEW elements are packed contiguously at the start of the first vector register in the group. The next 128/SEW elements are packed contiguously at the start of the next vector register in the group. After packing the first LMUL*128/SEW elements at the start of each of the LMUL vector registers in the group, the second LMUL*128/SEW group of elements are packed into the second 128b segment of each of the vector registers in the group, and so on.

```
Example 1: VLEN=32b, SEW=16b, LMUL=2

Byte         3 2 1 0
v2*n           1   0
v2*n+1         3   2

Example 2: VLEN=64b, SEW=32b, LMUL=2

Byte         7 6 5 4 3 2 1 0
v2*n               1       0
v2*n+1             3       2

Example 3: VLEN=128b, SEW=32b, LMUL=2

Byte        F E D C B A 9 8 7 6 5 4 3 2 1 0
v2*n              3       2       1       0
v2*n+1            7       6       5       4

Example 4: VLEN=256b, SEW=32b, LMUL=2

Byte     1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0
v2*n           B       A       9       8       3       2       1       0
v2*n+1         F       E       D       C       7       6       5       4
```

If SEW > SLEN, the striping pattern places one element in each vector register in the group before moving to the next vector register in the group. So, when LMUL=2, the even-numbered vector register contains the even-numbered elements of the vector and the odd-numbered vector register contains the odd-numbered elements of the vector.

Note: In most implementations, the striping distance SLEN ≥ ELEN.

```
Example: VLEN=256b, SEW=256b, LMUL=2

Byte     1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0
v2*n                                                                     0
v2*n+1                                                                   1
```

When LMUL = 4, four vector registers hold elements as shown:

```
Example 1: VLEN=32b, SLEN=32b, SEW=16b, LMUL=4,

Byte         3 2 1 0
v4*n           1   0
v4*n+1         3   2
v4*n+2         5   4
v4*n+3         7   6

Example 2: VLEN=64b, SLEN=64b, SEW=32b, LMUL=4

Byte         7 6 5 4 3 2 1 0
v4*n               1       0
v4*n+1             3       2
v4*n+2             5       4
v4*n+3             7       6


Example 3: VLEN=128b, SLEN=64b, SEW=32b, LMUL=4

Byte          F E D C B A 9 8 7 6 5 4 3 2 1 0
v4*n                9       8       1       0   32b elements
v4*n+1              B       A       3       2
v4*n+2              D       C       5       4
v4*n+3              F       E       7       6

Example 3: VLEN=128b, SLEN=128b, SEW=32b, LMUL=4

Byte          F E D C B A 9 8 7 6 5 4 3 2 1 0
v4*n                3       2       1       0   32b elements
v4*n+1              7       6       5       4
v4*n+2              B       A       9       8
v4*n+3              F       E       D       C

Example 4: VLEN=256b, SLEN=128b, SEW=32b, LMUL=4

Byte     1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0
v4*n           13      12      11      10       3       2       1       0
v4*n+1         17      16      15      14       7       6       5       4
v4*n+2         1B      1A      19      18       B       A       9       8
v4*n+3         1F      1E      1D      1C       F       E       D       C

Example 5: VLEN=256b, SLEN=128b, SEW=256b, LMUL=4

Byte     1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0
v4*n                                                                    0
v4*n+1                                                                  1
v4*n+2                                                                  2
v4*n+3                                                                  3
```

A similar pattern is followed for LMUL = 8.

```
Example: VLEN=256b, SLEN=128b, SEW=32b, LMUL=8

Byte   1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0
v8*n         23      22      21      20       3       2       1       0
v8*n+1       27      26      25      24       7       6       5       4
v8*n+2       2B      2A      29      28       B       A       9       8
v8*n+3       2F      2E      2D      2C       F       E       D       C
v8*n+4       33      32      31      30      13      12      11      10
v8*n+5       37      36      35      34      17      16      15      14
v8*n+6       3B      3A      39      38      1B      1A      19      18
v8*n+7       3F      3E      3D      3C      1F      1E      1D      1C
```

Different striping patterns are architecturally visible, but software can be written that produces the same results regardless of striping pattern. The primary constraint is to not change the LMUL used to access values held in a vector register group (i.e., do not read values with a different LMUL than used to write values to the group).

Note The striping length SLEN for an implementation is set to optimize the tradeoff between datapath wiring for mixed-width operations and buffering needed to corner-turn wide vector unit-stride memory accesses into parallel accesses for the vector register file.

Note The previous explicit configuration design allowed these tradeoffs to be managed at the microarchitectural level and optimized for each configuration.

4.3. Mapping across Mixed-Width Operations

The pattern used to map elements within a vector register group is designed to reduce datapath wiring when supporting operations across multiple element widths. The recommended software strategy in this case is to modify `vtype` dynamically to keep SEW/LMUL constant (and hence VLMAX constant).

The following example shows four different packed element widths (8b, 16b, 32b, 64b) in a VLEN=256b/SLEN=128b implementation. The vector register grouping factor (LMUL) is increased by the relative element size such that each group can hold the same number of vector elements (32 in this example) to simplify stripmining code. Any operation between elements with the same index only touches operand bits located within the same 128b portion of the datapath.

```
VLEN=256b, SLEN=128b
Byte     1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0

SEW=8b, LMUL=1, VLMAX=32

v1       1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0

SEW=16b, LMUL=2, VLMAX=32

v2*n       17  16  15  14  13  12  11  10   7   6   5   4   3   2   1   0
v2*n+1     1F  1E  1D  1C  1B  1A  19  18   F   E   D   C   B   A   9   8

SEW=32b, LMUL=4, VLMAX=32

v4*n           13      12      11      10       3       2       1       0
v4*n+1         17      16      15      14       7       6       5       4
v4*n+2         1B      1A      19      18       B       A       9       8
v4*n+3         1F      1E      1D      1C       F       E       D       C

SEW=64b, LMUL=8, VLMAX=32

v8*n                   11              10               1               0
v8*n+1                 13              12               3               2
v8*n+2                 15              14               5               4
v8*n+3                 17              16               7               6
v8*n+4                 19              18               9               8
v8*n+5                 1B              1A               B               A
v8*n+6                 1D              1C               D               C
v8*n+7                 1F              1E               F               E
```

Larger LMUL settings can also used to simply increase vector length to reduce instruction fetch and dispatch overheads, in cases where fewer logical vector registers are required.

The following table shows each possible constant SEW/LMUL operating point for loops with mixed-width operations.

```
      Numbers in columns are LMUL values, and each column
      represents constant SEW/LMUL operating point

 SEW/LMUL    1   2   4   8  16  32  64 128 256 512 1024

     SEW
       8     8   4   2   1
      16         8   4   2   1
      32             8   4   2   1
      64                 8   4   2   1
     128                     8   4   2   1
     256                         8   4   2   1
     512                             8   4   2   1
    1024                                 8   4   2   1
```

> Note: Larger LMUL values can cause lower datapath utilization for short vectors if SLEN is less than the spatial datapath width. In the example above with VLEN=256b, SLEN=128b, and LMUL=8, if the implementation is purely spatial with a 256b-wide vector datapath, then for an application vector length less than 17, only half of the datapath will be active. The `vsetvl` instructions below could have a facility added to dynamically select an appropriate LMUL according to the required application vector length (AVL) and range of element widths.

> Note: Narrower machines will set SLEN to be at least as large as the datapath spatial width, so there is no need to reduce LMUL. Wider machines might set SLEN lower than the spatial datapath width to reduce wiring for mixed-width operations (e.g., width=1024, ELEN=32, SLEN=128), in which case optimizing LMUL will be important.

4.4. Mask Register Layout

A vector mask occupies only one vector register regardless of SEW and LMUL. The mask bits that are used for each vector operation depends on the current SEW and LMUL setting.

The maximum number of elements in a vector operand is:

```
               VLMAX = LMUL * VLEN/SEW
```

A mask is allocated for each element by dividing the mask register into VLEN/VLMAX fields. The size of each mask element in bits, *MLEN*, is:

```
                MLEN = VLEN/VLMAX
                     = VLEN/(LMUL * VLEN/SEW)
                     = SEW/LMUL
```

The size of MLEN varies from ELEN (SEW=ELEN, LMUL=1) down to 1 (SEW=8b,LMUL=8), and hence a single vector register can always hold the entire mask register.

The mask bits for element *i* are located in bits [MLEN**i*+(MLEN-1) : MLEN**i*] of the mask register. When a mask element is written by a compare instruction, the low bit in the mask element is written with the compare result and the upper bits of the mask element are zeroed. When a value is read as a mask, only the least-significant bit of the mask element is used to control masking and the upper bits are ignored. Mask elements past the end of the current vector length are zeroed.

The pattern is such that for constant SEW/LMUL values, the effective predicate bits are located in the same bit of the mask vector register, which simplifies use of masking in loops with mixed-width elements.

```
VLEN=32b

         Byte    3   2   1   0
LMUL=1,SEW=8b
                 3   2   1   0  Element
               [24][16][08][00] Mask bit position in decimal

LMUL=2,SEW=16b
                     1       0
                   [08]    [00]
                     3       2
                   [24]    [16]

LMUL=4,SEW=32b               0
                           [00]
                             1
                           [08]
                             2
                           [16]
                             3
                           [24]
```

```
LMUL=2,SEW=8b
                 3   2   1   0
               [12][08][04][00]
                 7   6   5   4
               [28][24][20][16]

LMUL=8,SEW=32b
                             0
                           [00]
                             1
                           [04]
                             2
                           [08]
                             3
                           [12]
                             4
                           [16]
                             5
                           [20]
                             6
                           [24]
                             7
                           [28]

LMUL=8,SEW=8b
                 3   2   1   0
               [03][02][01][00]
                 7   6   5   4
               [07][06][05][04]
                 B   A   9   8
               [11][10][09][08]
                 F   E   D   C
               [15][14][13][12]
                13  12  11  10
               [19][18][17][16]
                17  16  15  14
               [23][22][21][20]
                1B  1A  19  18
               [27][26][25][24]
                1F  1E  1D  1C
               [31][30][29][28]
```

```
VLEN=256b, SLEN=128b
Byte     1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0

SEW=8b, LMUL=1, VLMAX=32

v1       1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0
       [248]          ...           [128] ...[96] ...[64] ...[32] ... [0]

Mask bit positions in decimal

SEW=16b, LMUL=2, VLMAX=32

v2*n       17  16  15  14  13  12  11  10   7   6   5   4   3   2   1   0
         [184]         ...          [128]   ...    [32]    ...       [0]
v2*n+1     1F  1E  1D  1C  1B  1A  19  18   F   E   D   C   B   A   9   8
         [248]         ...          [196]   ...    [96]    ...      [64]

SEW=32b, LMUL=4, VLMAX=32

v4*n           13      12      11      10       3       2       1       0
             [152]       ...        [128]    [24]      ...            [0]
v4*n+1         17      16      15      14       7       6       5       4
             [184]       ...        [160]    [56]      ...           [32]
v4*n+2         1B      1A      19      18       B       A       9       8
             [116]       ...        [192]    [88]      ...           [64]
v4*n+3         1F      1E      1D      1C       F       E       D       C
             [248]       ...        [224]   [120]      ...           [96]

SEW=64b, LMUL=8, VLMAX=32

v8*n                   11              10               1               0
                     [136]           [128]             [8]             [0]
v8*n+1                 13              12               3               2
                     [152]           [144]            [24]            [16]
v8*n+2                 15              14               5               4
                     [168]           [160]            [40]            [32]
v8*n+3                 17              16               7               6
                     [184]           [176]            [56]            [48]
v8*n+4                 19              18               9               8
                     [200]           [192]            [72]            [64]
v8*n+5                 1B              1A               B               A
                     [216]           [208]            [88]            [80]
v8*n+6                 1D              1C               D               C
                     [232]           [224]           [104]            [96]
v8*n+7                 1F              1E               F               E
                     [248]           [240]           [120]           [112]
```

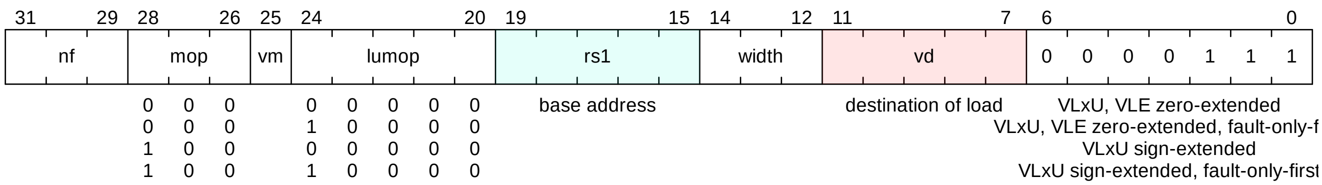
5. Vector Instruction Formats

The instructions in the vector extension fit under four existing major opcodes (LOAD-FP, STORE-FP, AMO) and one new major opcode (OP-V).

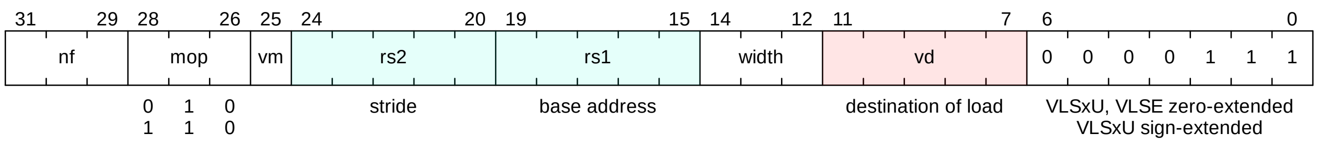
Vector loads and stores are encoding within the scalar floating-point load and store major opcodes (LOAD-FP/STORE-FP). The vector load and store encodings repurpose a portion of the standard scalar floating-point load/store 12-bit immediate field to provide further vector instruction encoding, with bit 25 holding the standard vector mask bit (see Mask Encoding).

5.1. Format for Vector Load Instructions under LOAD-FP major opcode

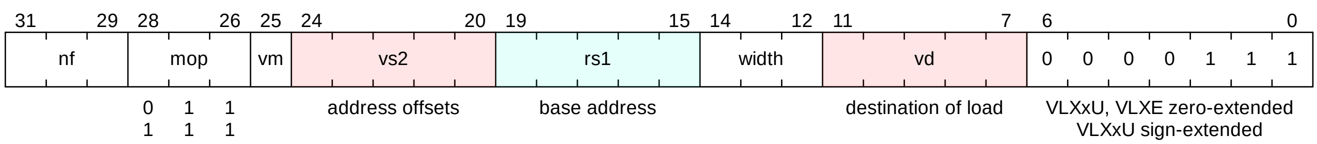
unit-stride



strided

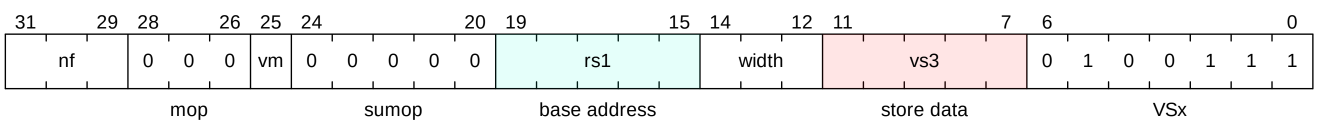


indexed

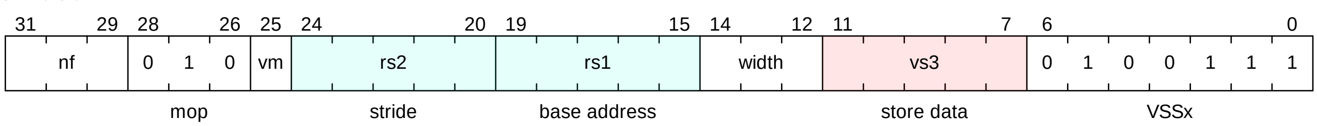


5.2. Format for Vector Store Instructions under STORE-FP major opcode

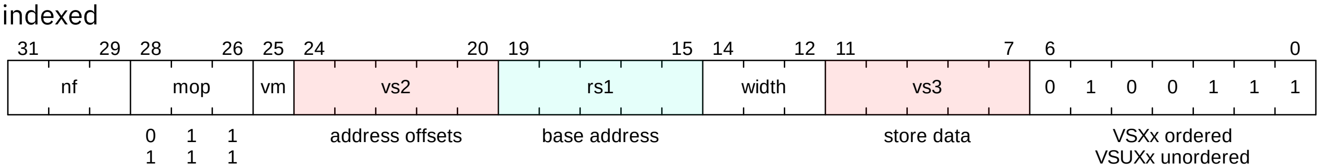
unit-stride



strided



indexed



vmem-format.adoc

5.3. Format for Vector AMO Instructions under AMO major opcode

31–27	26	25	24–20	19–15	14–12	11–7	6–0
amoop	wd	vm	vs2	rs1	width	vs3 / vvd	0 1 0 1 1 1 1
			address	base		source / destination	

amoop	
0 0 0 0 1	vamoSWAP
0 0 0 0 0	vamoADD
0 0 1 0 0	vamoXOR
0 1 1 0 0	vamoAND
0 1 0 0 0	vamoOR
1 0 0 0 0	vamoMIN
1 0 1 0 0	vamoMAX
1 1 0 0 0	vamoMINU
1 1 1 0 0	vamoMAXU

vamo-format.adoc

5.4. Formats for Vector Arithmetic Instructions under OP-V major opcode

31–26	25	24–20	19–15	14–12	11–7	6–0	
funct6	vm	vs2	vs1	funct3	vd	1 0 1 0 1 1 1	
				0 0 0			OPIVV
				0 0 1			OPFVV

31–26	25	24–20	19–15	14–12	11–7	6–0	
funct6	vm	vs2	vs1	0 1 0	vd / rd	1 0 1 0 1 1 1	OPMVV

31–26	25	24–20	19–15	14–12	11–7	6–0	
funct6	vm	vs2	simm5	0 1 1	vd	1 0 1 0 1 1 1	OPIVI

31–26	25	24–20	19–15	14–12	11–7	6–0	
funct6	vm	vs2	rs1	funct3	vd	1 0 1 0 1 1 1	
				1 0 0			OPIVX
				1 0 1			OPFVF

31–26	25	24–20	19–15	14–12	11–7	6–0	
funct6	vm	vs2	rs1	1 1 0	vd / rd	1 0 1 0 1 1 1	OPMVX

valu-format.adoc

5.5. Formats for Vector Configuration Instructions under OP-V major opcode

31	30–20	19–15	14–12	11–7	6–0	
0	zimm[1:0]	rs1	1 1 1	rd	1 0 1 0 1 1 1	vsetvli

31–25	24–20	19–15	14–12	11–7	6–0	
1 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	1 0 1 0 1 1 1	vsetvl

vcfg-format.adoc

Vector instructions can have scalar or vector source operands and produce scalar or vector results, and most vector instructions can be performed either unconditionally or conditionally under a mask.

Vector loads and stores move bit patterns between vector register elements and memory. Vector arithmetic instructions operate on values held in vector register elements.

5.6. Scalar operands

Scalar operands can be immediates, or taken from the `x` registers, the `f` registers, or element 0 of a vector register. Scalar results are written to an `x` or `f` register or to element 0 of a vector register. Any vector register can be used to hold a scalar regardless of the current LMUL setting.

Note: In a change from v0.6, the floating-point registers no longer overlay the vector registers and scalars can now come from the integer or floating-point registers. Not overlaying the `f` registers reduces vector register pressure, avoids interactions with the standard calling convention, simplifies high-performance scalar floating-point design, and provides compatibility with the Zfinx ISA option. Overlaying `f` with `v` would provide the advantage of lowering the number of state bits in some implementations, but complicates high-performance designs and would prevent compatibility with the Zfinx ISA option.

5.7. Vector Operands

Vector operands or results may occupy one or more vector registers depending on LMUL, but are always specified using the lowest-numbered vector register in the group. Using other than the lowest-numbered vector register to specify a vector register group will result in an illegal instruction exception.

Some vector instructions consume and produce wider-width elements and so operate on a larger vector register group than that specified in `vlmul`. The largest vector register group used by an instruction can not be greater than 8 vector registers, and if an vector instruction would require greater than 8 vector registers in a group, an illegal instruction exception is raised. For example, attempting a widening operation with LMUL=8 will raise an illegal instruction exception.

5.8. Vector Masking

Masking is supported on many vector instructions. Element operations that are masked off do not modify the destination vector register element and never generate exceptions.

In the base vector extension, the mask value used to control execution of a masked vector instruction is always supplied by vector register `v0`. Only the least-significant bit of each element of the mask vector is used to control execution.

Note: Future vector extensions may provide longer instruction encodings with space for a full mask register specifier.

The destination vector register group for a masked vector instruction can only overlap the source mask register (`v0`) when LMUL=1. Otherwise, an illegal vector instruction exception is raised.

Note: This constraint supports restart with a non-zero `vstart` value.

Other vector registers can be used to hold working mask values, and mask vector logical operations are provided to perform predicate calculations.

5.8.1. Mask Encoding

Where available, masking is encoded in a single-bit `vm` field in the instruction (`inst[25]`).

vm	Description

vm	Description
0	vector result, only where v0[i].LSB = 1
1	unmasked

Note: In earlier proposals, `vm` was a two-bit field `vm[1:0]` that provided both true and complement masking using `v0` as well as encoding scalar operations.

Vector masking is represented in assembler code as another vector operand, with `.t` indicating if operation occurs when `v0[i].LSB` is 1. If no masking operand is specified, unmasked vector execution (`vm=1`) is assumed.

```
vop.v* v1, v2, v3, v0.t # enabled where v0[i].LSB=1, m=0
vop.v* v1, v2, v3       # unmasked vector operation, m=1
```

Note: Even though the base only supports one vector mask register `v0` and only the true form of predication, the assembly syntax writes it out in full to be compatible with future extensions that might add a mask register specifier and supporting both true and complement masking. The `.t` suffix on the masking operand also helps to visually encode the use of a mask.

5.9. Prestart, Active, Inactive, Body, and Tail Element Definitions

The elements operated on during a vector instruction's execution can be divided into four disjoint subsets.

- The *prestart* elements are those whose element index is less than the initial value in the `vstart` register. The prestart elements do not raise exceptions and do not update the destination vector register.
- The *active* elements during a vector instruction's execution are the elements within the current vector length setting and where the current mask is enabled at that element position. The active elements can raise exceptions and update the destination vector register group.
- The *inactive* elements are the elements within the current vector length setting but where the current mask is disabled at that element position. The inactive elements do not raise exceptions and do not update the destination vector register.
- The *tail* elements during a vector instruction's execution are the elements past the current vector length setting. The tail elements do not raise exceptions, but do zero the results in the destination vector register group.
- In addition, another term, *body*, is used for the set of elements that are either active or inactive, i.e., after prestart but before the tail.

```
// for element index x
prestart    = (0 <= x < vstart)
mask(x)     = unmasked || v0[x].LSB == 1
active(x)   = (vstart <= x < vl) && mask(x)
inactive(x) = (vstart <= x < vl) && !mask(x)
body(x)     = active(x) || inactive(x)
tail(x)     = (vl <= x < VLMAX)
```

All regular vector instructions place zeros in the tail elements of the destination vector register group. Some vector arithmetic instructions are not maskable, so have no inactive elements, but still zero the tail elements.

Note

The inactive and tail update rules were designed to provide an efficient compromise between requirements of implementations with and without vector register ECC and/or renaming.

Note

Not zeroing past `vl` would penalize renamed implementations that would have to copy all elements past VL on every instruction execution, whereas it's a small penalty for non-renamed implementations to implement the tail zeroing. While a renamed machine could avoid copying for whole vector registers in a group by not renaming, operations on individual registers may be deep enough that requiring full occupancy for any vector length would be problematic. Zeroing values past `vl` does not impact most software, except for a small cost in some reduction cases.

Note

For zeroing tail updates, implementations with temporally long vector registers, either with or without register renaming, will be motivated to add microarchitectural state to avoid actually writing zeros to all tail elements, but this is a relatively simple microarchitectural optimization. For example, one bit per element group or a quantized VL can be used to track the extent of zeroing. An element group is the set of elements comprising the smallest atomic unit of execution in the microarchitecture (often equivalent to the width of the physical datapath in the machine). The microarchitectural state for an element group indicates that zero should be returned for the element group on a read, and that zero should be substituted in for any masked-off elements in the group on the first write to that element group (after which the element group zero bit can be cleared).

Note

Providing merging predication instead of zeroing inactive elements on a masked operation reduces code path length for many code blocks, and reduces register pressure by allowing different code paths to use disjoint sets of elements in the same vector register. Implementations with vector register ECC or renaming will have to perform read-update-write on the destination register value to preserve inactive elements on arithmetic instructions, so would appear to need an extra vector register read port. However, the arithmetic instructions are designed such that the largest read-port requirement is for fused multiply-add instructions that are destructive and overwrite one source, and hence do not need an extra read port to preserve inactive elements. Given that linear algebra is one of the more important applications for vector units, and that fused multiply-add is the dominant operation in linear algebra routines, microarchitectures will be optimized for fused multiply-add operations and so should be able to preserve inactive elements on other arithmetic operations without large additional cost. However, masked vector load instructions incur the cost of an additional read port on their destination register. The need to support resumable vector loads with non-zero `vstart` values also drives the need to preserve vector load destination register values. The AMOs have been defined to be destructive in their source operand to reduce the maximum read port requirement for the memory pipe. An option that was considered was to have loads behave differently from arithmetic instructions and to zero any masked-off elements. However, this would require additional instructions and increase register pressure, and vector loads must in any case still cope with non-zero `vstart` values through some mechanism.

6. Configuration-Setting Instructions

A set of instructions are provided to allow rapid configuration of the values in `vl` and `vtype` to match application needs.

6.1. `vsetvli`/`vsetvl` instructions

```
vsetvli rd, rs1, vtypei # rd = new vl, rs1 = AVL, vtypei = new vtype setting
                        # if rs1 = x0, then use maximum vector length
vsetvl  rd, rs1, rs2    # rd = new vl, rs1 = AVL, rs2 = new vtype value
                        # if rs1 = x0, then use maximum vector length
```

The `vsetvli` instruction sets the `vtype` and `vl` CSRs based on its arguments, and writes the new value of `vl` into `rd`.

The new `vtype` setting is encoded in the immediate fields of `vsetvli` and in the `rs2` register for `vsetvl`.

vcfg-format.adoc

vtype-format.adoc

```
 Suggested assembler names used for vtypei setting

 e8    #   8b elements
 e16   #  16b elements
 e32   #  32b elements
 e64   #  64b elements
 e128  # 128b elements

 m1   # Vlmul x1, assumed if m setting absent
 m2   # Vlmul x2
 m4   # Vlmul x4
 m8   # Vlmul x8

 d1   # EDIV 1, assumed if d setting absent
 d2   # EDIV 2
 d4   # EDIV 4
 d8   # EDIV 8

Examples:
    vsetvli t0, a0, e8          # SEW= 8, LMUL=1, EDIV=1
    vsetvli t0, a0, e8,m2       # SEW= 8, LMUL=2, EDIV=1
    vsetvli t0, a0, e32,m2,d4   # SEW=32, LMUL=2, EDIV=4
```

The `vtype` setting must be supported by the implementation, and the `vsetvl{i}` instructions will raise an illegal instruction exception if the setting is not supported.

Note: Specifing that `vtype` is WARL is problematic as that would hide errors. The current spec is problematic in that it requires a trap based on a data value in a CSR write. It would simplify pipelines if `vtype` value errors were flagged at use not write, but somehow need to catch errant code without requiring full XLEN bits in `vtype` when only a few bits are actually used. One alternative is to allow substitution of a fixed illegal value in `vtype`, e.g., all 1s, if an attempt is made to write an unsupported value. This would then cause a trap on use.

The requested application vector length (AVL) is passed in `rs1` as an unsigned integer. Using `x0` as the `rs1` register specifier, encodes an infinite AVL, and so requests the maximum possible vector length.

Note A `vsetvl` with AVL=`x0` can be used to read current VLMAX, though this does overwrite `vl`.

6.2. Constraints on Setting `vl`

The resulting `vl` setting must satisfy the following constraints:

1. `vl = AVL` if `AVL ≤ VLMAX`
2. `vl ≥ ceil(AVL / 2)` if `AVL < (2 * VLMAX)`
3. `vl = VLMAX` if `AVL ≥ (2 * VLMAX)`
4. Deterministic on any given implementation for same input AVL and `vtype` values
5. These specific properties follow from the prior rules:
   a. `vl = 0` if `AVL = 0`
   b. `vl > 0` if `AVL > 0`
   c. `vl ≤ VLMAX`
   d. `vl ≤ AVL`
   e. a value read from `vl` when used as AVL arg to `vsetvl{i}` results in same value in `vl`

Note The `vl` setting rules are designed to be sufficiently strict to preserve `vl` behavior across register spills and context swaps for `AVL ≤ VLMAX`, yet flexible enough to enable implementations to improve vector lane utilization for `AVL > VLMAX`.

For example, this permits an implementation to set `vl = ceil(AVL / 2)` for `VLMAX < AVL < 2*VLMAX` in order to evenly distribute work over the last two iterations of a stripmine loop. Requirement 2 ensures that the first stripmine iteration of reduction loops uses the largest vector length of all iterations, even in the case of `AVL < 2*VLMAX`. This allows software to avoid needing to explicitly calculate a running maximum of vector lengths observed during a stripmined loop.

6.3. `vsetvl` Instruction

The `vsetvl` variant operates similarly to `vsetvli` except that it takes a `vtype` value from `rs2` and can be used for context restore, and when the `vtypei` field is too small to hold the desired setting.

Note Several active complex types can be held in different x registers and swapped in as needed using `vsetvl`.

6.4. Examples

The SEW and LMUL settings can be changed dynamically to provide high throughput on mixed-width operations in a single loop.

Example: Load 16-bit values, widen multiply to 32b, shift 32b result right by 3, store 32b values.

```
# Loop using only widest elements:
loop:
    vsetvli   a3, a0, e32,m8    # Use only 32-bit elements
    vlh.v     v8, (a1)          # Sign-extend 16b load values to 32b elements
      sll     t1, a3, 1
      add     a1, a1, t1        # Bump pointer
    vmul.vx   v8, v8, x10       # 32b multiply result
    vsrl.vi   v8, v8, 3         # Shift elements
    vsw.v     v8, (a2)          # Store vector of 32b results
      sll     t1, a3, 2
      add     a2, a2, t1        # Bump pointer
      sub     a0, a0, a3        # Decrement count
      bnez    a0, loop          # Any more?

# Alternative loop that switches element widths.
loop:
    vsetvli   a3, a0, e16,m4    # vtype = 16-bit integer vectors
    vlh.v     v4, (a1)          # Get 16b vector
      slli    t1, a3, 1
      add     a1, a1, t1        # Bump pointer
    vwmul.vx  v8, v4, x10       # 32b in <v8--v15>

    vsetvli   x0, a0, e32,m8    # Operate on 32b values
    vsrl.vi   v8, v8, 3
    vsw.v     v8, (a2)          # Store vector of 32b
      slli    t1, t1, 2
      add     a2, a2, t1        # Bump pointer
      sub     a0, a0, a3        # Decrement count
      bnez    a0, loop          # Any more?
```

The second loop is more complex but will have greater performance on machines where 16b widening multiplies are faster than 32b integer multiplies, and where 16b vector load can run faster due to the narrower writes to the vector regfile.

7. Vector Loads and Stores

Vector loads and stores move values between vector registers and memory. Vector loads and stores are masked and do not raise exceptions on inactive elements. Masked vector loads do not update inactive elements in the destination vector register group. Masked vector stores do not update inactive memory elements.

7.1. Vector Load/Store Instruction Encoding

Vector loads and stores are encoded within the scalar floating-point load and store major opcodes (LOAD-FP/STORE-FP). The vector load and store encodings repurpose a portion of the standard scalar floating-point load/store 12-bit immediate field to provide further vector instruction encoding, with bit 25 holding the standard vector mask bit (see Mask Encoding).

vmem-format.adoc

```
rs1[4:0] specifies x register holding base address
rs2[4:0] specifies x register holding stride
vs2[4:0] specifies v register holding address offsets
vs3[4:0] specifies v register holding store data
vd[4:0]  specifies v register destination of load

vm specifies vector mask
width[2:0] specifies size of memory elements, and distinguishes from FP scalar
mop[2:0] specifies memory addressing mode
nf[2:0] specifies the number of fields in each segment, for segment load/stores

lumop[4:0]/sumop[4:0] are additional fields encoding variants of unit-stride instructions
```

7.2. Vector Load/Store Addressing Modes

The base vector extension supports unit-stride, strided, and indexed (scatter/gather) addressing modes. Vector load/store base registers and strides are taken from the GPR x registers.

The base effective address for all vector accesses is given by the contents of the x register named in `rs1`.

Vector unit-stride operations access elements stored contiguously in memory starting from the base effective address.

Vector strided operations access the first memory element at the base effective address, and then access subsequent elements at address increments given by the byte offset contained in the x register specified by `rs2`.

Vector indexed operations add the contents of each element of the vector offset operand specified by `vs2` to the base effective address to give the effective address of each element. The vector offset operand is treated as a vector of byte offsets. If the vector offset elements are narrower than XLEN, they are sign-extended to XLEN before adding to the base effective address. If the vector offset elements are wider than XLEN, the least-significant XLEN bits are used in the address calculation.

Note

Current PoR for vector indexed instructions requires that vector byte offset (vs2) and vector read/write data (vs3/vd) are of same width. One question is whether and how to allow for two sizes of vector operand in a vector indexed instruction? For example, for scatter/gather of byte values in a 64-bit address space without requiring bytes use 64b of space in a vector register.

The vector addressing modes are encoded using the 3-bit `mop[2:0]` field.

Table 6. encoding for loads

mop [2:0]			Description	Opcodes
0	0	0	zero-extended unit-stride	VLxU,VLE
0	0	1	reserved	
0	1	0	zero-extended strided	VLSxU, VLSE
0	1	1	zero-extended indexed	VLXxU, VLXE
1	0	0	sign-extended unit-stride	VLx (x!=E)
1	0	1	reserved	
1	1	0	sign-extended strided	VLSx (x!=E)
1	1	1	sign-extended indexed	VLXx (x!=E)

Table 7. encoding for stores

mop [2:0]			Description	Opcodes
0	0	0	unit-stride	VSx
0	0	1	reserved	
0	1	0	strided	VSSx
0	1	1	indexed-ordered	VSXx
1	0	0	reserved	
1	0	1	reserved	
1	1	0	reserved	
1	1	1	indexed-unordered	VSUXx

The vector indexed memory operations have two forms, ordered and unordered. The indexed-unordered stores do not preserve element ordering on stores.

Note: The indexed-unordered variant is provided as a potential implementation optimization. Implementations are free to ignore the optimization and implement indexed-unordered identically to indexed-ordered.

Additional unit-stride vector addressing modes are encoded using the 5-bit `lumop` and `sumop` fields in the unit-stride load and store instruction encodings respectively.

Table 8. lumop

lumop[4:0]					Description
0	0	0	0	0	unit-stride
0	x	x	x	x	reserved, x!=0
1	0	0	0	0	unit-stride fault-only-first
1	x	x	x	x	reserved, x!=0

Table 9. sumop

sumop[4:0]					Description
0	0	0	0	0	unit-stride
0	x	x	x	x	reserved, x!=0
1	x	x	x	x	reserved

The `nf[2:0]` field encodes the number of fields in each segment. For regular vector loads and stores, `nf`=0, indicating that a single value is moved between a vector register group and memory at each element position. Larger values in the `nf` field are used to access multiple contiguous fields within a segment as described below in Section Vector Load/Store Segment Instructions (`Zvlsseg`).

Note: The `nf` field for segment load/stores has replaced the use of the same bits for an address offset field. The offset can be replaced with a single scalar integer calculation, while segment load/stores add more powerful primitives to move items to and from memory.

7.3. Vector Load/Store Width Encoding

The vector loads and stores are encoded using the width values that are not claimed by the standard scalar floating-point loads and stores. Three of the width types encode vector loads and stores that move fixed-size memory elements of 8 bits, 16 bits, or 32 bits, while the fourth encoding moves SEW-bit memory elements.

	Width [2:0]			Mem bits	Reg bits	Opcode
Standard scalar FP	0	0	1	16	FLEN	FLH/FSH
Standard scalar FP	0	1	0	32	FLEN	FLW/FSW
Standard scalar FP	0	1	1	64	FLEN	FLD/FSD
Standard scalar FP	1	0	0	128	FLEN	FLQ/FSQ
Vector byte	0	0	0	vl*8	vl*SEW	VxB
Vector halfword	1	0	1	vl*16	vl*SEW	VxH
Vector word	1	1	0	vl*32	vl*SEW	VxW
Vector element	1	1	1	vl*SEW	vl*SEW	VxE

Mem bits is the size of element accessed in memory

Reg bits is the size of element accessed in register

Fixed-sized vector loads can optionally sign or zero-extend their memory element into the destination register element if the register element is wider than the memory element. A fixed-size vector load raises an illegal instruction exception if the destination register element is narrower than the memory element. The variable-sized load is encoded as if a zero-extended load, with what would be the sign-extended encoding of a variable-sized load currently reserved.

Fixed-size vector stores take their operand from the least-significant bits of the register element if the register element if wider than the memory element. Fixed-sized vector stores raise an illegal instruction exception if the memory element is wider than the register element.

7.4. Vector Unit-Stride Instructions

Vector unit-stride loads and stores

```
# vd destination, rs1 base address, vm is mask encoding (v0.t or <missing>)
vlb.v         vd, (rs1), vm     # 8b signed
vlh.v         vd, (rs1), vm     # 16b signed
vlw.v         vd, (rs1), vm     # 32b signed

vlbu.v        vd, (rs1), vm     # 8b unsigned
vlhu.v        vd, (rs1), vm     # 16b unsigned
vlwu.v        vd, (rs1), vm     # 32b unsigned

vle.v         vd, (rs1), vm     # SEW

# vs3 store data, rs1 base address, vm is mask encoding (v0.t or <missing>)
vsb.v         vs3, (rs1), vm    # 8b store
vsh.v         vs3, (rs1), vm    # 16b store
vsw.v         vs3, (rs1), vm    # 32b store
vse.v         vs3, (rs1), vm    # SEW store
```

7.5. Vector Strided Instructions

Vector strided loads and stores

```
# vd destination, rs1 base address, rs2 byte stride
vlsb.v        vd, (rs1), rs2, vm  # 8b
vlsh.v        vd, (rs1), rs2, vm  # 16b
vlsw.v        vd, (rs1), rs2, vm  # 32b

vlsbu.v       vd, (rs1), rs2, vm  # unsigned 8b
vlshu.v       vd, (rs1), rs2, vm  # unsigned 16b
vlswu.v       vd, (rs1), rs2, vm  # unsigned 32b

vlse.v        vd, (rs1), rs2, vm  # SEW

# vs3 store data, rs1 base address, rs2 byte stride
vssb.v        vs3, (rs1), rs2, vm # 8b
vssh.v        vs3, (rs1), rs2, vm # 16b
vssw.v        vs3, (rs1), rs2, vm # 32b
vsse.v        vs3, (rs1), rs2, vm # SEW
```

7.6. Vector Indexed Instructions

```
# vd destination, rs1 base address, vs2 indices
vlxb.v       vd, (rs1), vs2, vm  # 8b
vlxh.v       vd, (rs1), vs2, vm  # 16b
vlxw.v       vd, (rs1), vs2, vm  # 32b

vlxbu.v      vd, (rs1), vs2, vm  # 8b unsigned
vlxhu.v      vd, (rs1), vs2, vm  # 16b unsigned
vlxwu.v      vd, (rs1), vs2, vm  # 32b unsigned

vlxe.v       vd, (rs1), vs2, vm  # SEW

# Vector ordered-indexed store instructions
# vs3 store data, rs1 base address, vs2 indices
vsxb.v       vs3, (rs1), vs2, vm # 8b
vsxh.v       vs3, (rs1), vs2, vm # 16b
vsxw.v       vs3, (rs1), vs2, vm # 32b
vsxe.v       vs3, (rs1), vs2, vm # SEW

# Vector unordered-indexed store instructions
vsuxb.v      vs3, (rs1), vs2, vm # 8b
vsuxh.v      vs3, (rs1), vs2, vm # 16b
vsuxw.v      vs3, (rs1), vs2, vm # 32b
vsuxe.v      vs3, (rs1), vs2, vm # SEW
```

7.7. Unit-stride Fault-Only-First Loads

The unit-stride fault-only-first load instructions are used to vectorize loops with data-dependent exit conditions (while loops). These instructions execute as a regular load except that they will only take a trap on element 0. If an element > 0 raises an exception, that element and all following elements in the destination vector register are not modified, and the vector length `vl` is reduced to the number of elements processed without a trap.

```
vlbff.v      vd, (rs1), vm     # 8b
vlhff.v      vd, (rs1), vm     # 16b
vlwff.v      vd, (rs1), vm     # 32b

vlbuff.v     vd, (rs1), vm     # unsigned 8b
vlhuff.v     vd, (rs1), vm     # unsigned 16b
vlwuff.v     vd, (rs1), vm     # unsigned 32b

vleff.v      vd, (rs1), vm     # SEW
```

strlen example using unit-stride fault-only-first instruction

```
# size_t strlen(const char *str)
# a0 holds *str

strlen:
    mv        a3, a0            # Save start
loop:
    vsetvli   a1, x0, e8        # Vector of bytes of maximum length
    vlbff.v   v1, (a3)          # Load bytes
    csrr      a1, vl            # Get bytes read
    vseq.vi   v0, v1, 0         # Set v0[i] where v1[i] = 0
    vmfirst.m a2, v0            # Find first set bit
    add       a3, a3, a1        # Bump pointer
    bltz      a2, loop          # Not found?

    add       a0, a0, a1        # Sum start + bump
    add       a3, a3, a2        # Add index
    sub       a0, a3, a0        # Subtract start address+bump

    ret
```

Note: Strided and scatter/gather fault-only-first instructions are not provided as they represent a large security hole, allowing software to check multiple random pages for accessibility without experiencing a trap. The unit-stride versions only allow probing a region immediately contiguous to a known region, and so do not appreciably impact security. It is possible that security mitigations can be implemented to allow fault-only-first variants of non-contiguous accesses in future vector extensions.

7.8. Vector Load/Store Segment Instructions (`Zvlsseg`)

Note: This is being written as an extension but could be mandated in some profiles.

The vector load/store segment instructions move multiple contiguous fields in memory to and from consecutively numbered vector registers.

Note: These operations support operations on "array-of-structures" datatypes by unpacking each field in a structure into separate vector registers.

As for regular vector loads and stores, the width encoding gives the size of the memory elements, which are homogeneous in size, while SEW encodes the size of the register elements.

The three-bit `nf` field in the vector instruction encoding is an unsigned integer that contains one less than the number of fields per segment, *NFIELDS*.

nf[2:0]			NFIELDS
0	0	0	1
0	0	1	2
0	1	0	3
0	1	1	4
1	0	0	5
1	0	1	6
1	1	0	7
1	1	1	8

LMUL must be set to 1 for a segment load or store (NFIELDS > 1), and each field will be held in successively numbered vector registers. If LMUL > 1, an illegal instruction exception will be raised.

The `vl` register gives the number of structures to move, which is equal to the number of elements transferred to each vector register.

If a trap is taken, `vstart` is in units of structures.

Note: An earlier version imposed a vector register number constraint, but this decreased ability to make use of all registers when NFIELDS was not a power of 2.

7.8.1. Vector Unit-Stride Segment Loads and Stores

The vector unit-stride load and store segment instructions move packed contiguous segments ("array-of-structures") into multiple destination vector register groups.

Note: For segments with heterogeneous-sized fields, software can later unpack fields using additional instructions after the segment load brings the values into the separate vector registers.

The assembler prefixes `vlseg`/`vsseg` are used for unit-stride segment loads and stores respectively.

```
# Format
vlseg<nf>{b,h,w}.v vd, (rs1),  vm    # Unit-stride signed segment load template
vlseg<nf>e.v vd, (rs1), vm           # Unit-stride segment load template
vlseg<nf>{b,h,w}u.v vd, (rs1), vm    # Unit-stride unsigned segment load template
vsseg<nf>{b,h,w,e}.v vs3, (rs1), vm  # Unit-stride segment store template

# Examples
vlseg2b.v  vd, (rs1), vm # Load vector of signed 2*1-byte segments into vd, vd+1
vlseg3bu.v vd, (rs1), vm # Load vector of unsigned 3*1-byte segments into vd,vd+1,vd+2
vlseg7w.v  vd, (rs1), vm # Load vector of 7*4-byte segments into vd, vd+1, ... vd+6
vlseg8e.v  vd, (rs1), vm # Load vector of 8*SEW-byte segments into vd, vd+1, .. vd+7

vsseg3b.v vs3, (rs1), vm # Store packed vector of 3*1-byte segments
                         # from vs3,vs3+1,vs3+2 to memory
```

For loads, the `vd` register will hold the first field loaded from the segment. For stores, the `vs3` register is read to provide the first field to be stored in each segment.

```
# Example 1
# Memory structure holds packed RGB pixels (24-bit data structure, 8bpp)
vlseg3bu.v v8, (a0), vm
# v8 holds the red pixels
# v9 holds the green pixels
# v10 holds the blue pixels

# Example 2
# Memory structure holds complex values, 32b for real and 32b for imaginary
vlseg2w.v v8, (a0), vm
# v8 holds real
# v9 holds imaginary
```

There are also fault-only-first versions of the unit-stride instructions.

```
# Template for vector fault-only-first unit-stride segment loads and stores.
vlseg<nf>{b,h,w}ff.v vd, (rs1), vm # Unit-stride signed fault-only-first segment loads
vlseg<nf>eff.v vd, (rs1),  vm      # Unit-stride fault-only-first segment loads
vlseg<nf>{b,h,w}uff.v vd,(rs1),vm # Unit-stride unsigned fault-only-first segment loads
```

7.8.2. Vector Strided Segment Loads and Stores

Vector strided segment loads and stores move contiguous segments where each segment is separated by the byte stride offset given in the `rs2` GPR argument.

```
# Format
vlsseg<nf>{b,h,w}.v vd, (rs1), rs2, vm    # Strided signed segment loads
vlsseg<nf>e.v vd, (rs1), rs2, vm          # Strided segment loads
vlsseg<nf>{b,h,w}u.v vd, (rs1), rs2, vm   # Strided unsigned segment loads
vssseg<nf>{b,h,w,e}.v vs3, (rs1), rs2, vm # Strided segment stores

# Examples
vlsseg3b.v v4, (x5), x6   # Load bytes at addresses x5+i*x6   into v4[i],
                          #  and bytes at addresses x5+i*x6+1 into v5[i],
                          #  and bytes at addresses x5+i*x6+2 into v6[i].

# Examples
vssseg2w.v v2, (x5), x6   # Store words from v2[i] to address x5+i*x6
                          #   and words from v3[i] to address x5+i*x6+4
```

7.8.3. Vector Indexed Segment Loads and Stores

Vector indexed segment loads and stores move contiguous segments where each segment is located at an address given by adding the scalar base address in the `rs1` field to byte offsets in vector register `vs2`.

```
# Format
vlxseg<nf>{b,h,w}.v vd, (rs1), vs2, vm    # Indexed signed segment loads
vlxseg<nf>e.v vd, (rs1), vs2, vm          # Indexed segment loads
vlxseg<nf>{b,h,w}u.v vd, (rs1), vs2, vm   # Indexed unsigned segment loads
vsxseg<nf>{b,h,w,e}.v vs3, (rs1), vs2, vm # Indexed segment stores

# Examples
vlxseg3bu.v v4, (x5), v3  # Load bytes at addresses x5+v3[i]   into v4[i],
                          #  and bytes at addresses x5+v3[i]+1 into v5[i],
                          #  and bytes at addresses x5+v3[i]+2 into v6[i].

# Examples
vsxseg2w.v v2, (x5), v5   # Store words from v2[i] to address x5+v5[i]
                          #   and words from v3[i] to address x5+v5[i]+4
```

8. Vector AMO Operations (Zvamo)

Note: Profiles will dictate whether vector AMO operations are supported. The expectation is that the Unix profile will require vector AMO operations.

If vector AMO instructions are supported, then the scalar Zaamo instructions (atomic operations from the standard A extension) must be present.

Vector AMO operations are encoded using the unused width encodings under the standard AMO major opcode. Each active element performs an atomic read-modify-write of a single memory location.

vamo-format.adoc

```
vs2[4:0] specifies v register holding address
vs3/vd[4:0] specifies v register holding source operand and destination

vm specifies vector mask
width[2:0] specifies size of memory elements, and distinguishes from scalar AMO
amoop[4:0] specifies the AMO operation
wd specifies whether the original memory value is written to vd (1=yes, 0=no)
```

AMOs have the same addressing mode as indexed operations except with no immediate offset. A vector of byte offsets in register `vs2` are added to the scalar base register in `rs1` to give the addresses of the AMO operations.

The `vs2` vector register supplies the byte offset of each element, while the `vs3` vector register supplies the source data for the atomic memory operation.

If the `wd` bit is set, the `vd` register is written with the initial value of the memory element. If the `wd` bit is clear, the `vd` register is not written.

Note: When `wd` is clear, the memory system does not need to return the original memory value, and the original values in `vd` will be preserved.

Note: The AMOs were defined to overwrite source data partly to reduce total memory pipeline read port count for implementations with register renaming. Also, to support the same addressing mode as vector indexed operations, and because vector AMOs are less likely to need results given that the primary use is parallel in-memory reductions.

Vector AMOs operate as if `aq` and `rl` bits were zero on each element with regard to ordering relative to other instructions in the same hart.

Vector AMOs provide no ordering guarantee between element operations in the same vector AMO instruction.

Table 10. Vector AMO width encoding

	Width [2:0]			Mem bits	Reg bits	Opcode
Standard scalar AMO	0	1	0	32	XLEN	AMO*.W
Standard scalar AMO	0	1	1	64	XLEN	AMO*.D
Standard scalar AMO	1	0	0	128	XLEN	AMO*.Q
Vector AMO	1	1	0	32	vl*SEW	VAMO*W.V
Vector AMO	1	1	1	64	vl*SEW	VAMO*D.V
Vector AMO	0	0	0	128	vl*SEW	VAMO*Q.V

Mem bits is the size of element accessed in memory

Reg bits is the size of element accessed in register

The vector AMO width encoding flips the high bit of the corresponding scalar AMO width encoding. SEW must be at least as wide as the AMO memory element size, otherwise an illegal instruction exception is raised. If the AMO memory element width is less than SEW, the value returned from memory is sign-extended to fill SEW.

If SEW is less than XLEN, then addresses in the vector `vs2` are sign-extended to XLEN. If SEW is greater than XLEN, an illegal instruction exception is raised.

Note, the AMO instruction encoding does not support arbitrary SEW-bit memory elements, only the standard 32-bit, 64-bit, 128-bit sizes required by the standard scalar base architecture.

The vector `amoop[4:0]` field uses the same encoding as the scalar 5-bit AMO instruction field, except that LR and SC are not supported.

Table 11. amoop

amoop					**opcode**
0	0	0	0	1	vamoswap
0	0	0	0	0	vamoadd
0	0	1	0	0	vamoxor
0	1	1	0	0	vamoand
0	1	0	0	0	vamoor
1	0	0	0	0	vamomin
1	0	1	0	0	vamomax
1	1	0	0	0	vamominu
1	1	1	0	0	vamomaxu

9. Vector Memory Alignment Constraints

If the elements accessed by a vector memory instruction are not naturally aligned to the memory element size, either an address misaligned exception is raised on that element or the element is transferred successfully.

Vector memory accesses follow the same rules for atomicity as scalar memory accesses.

10. Vector Memory Consistency Model

Vector memory instructions appear to execute in program order on the local hart. Vector memory instructions follow RVWMO at the instruction level, and element operations are ordered within the instruction as if performed by an element-ordered sequence of syntactically independent scalar instructions. Vector indexed-ordered stores write elements to memory in element order. Vector indexed-unordered stores do not preserve element order for writes within a single vector store instruction.

Note

Need to flesh out details.

11. Vector Arithmetic Instruction Formats

The vector arithmetic instructions use a new major opcode (OP-V = 1010111_2) which neighbors OP-FP. The three-bit `funct3` field is used to define sub-categories of vector instructions.

valu-format.adoc

11.1. Vector Arithmetic Instruction encoding

The `funct3` field encodes the operand type and source locations.

Table 12. funct3

funct3[2:0]			Operands	Source of scalar(s)	
0	0	0	OPIVV	vector-vector	-
0	0	1	OPFVV	vector-vector	-
0	1	0	OPMVV	vector-vector	-
0	1	1	OPIVI	vector-immediate	imm[4:0]
1	0	0	OPIVX	vector-scalar	GPR x register rs1
1	0	1	OPFVF	vector-scalar	FP f register rs1
1	1	0	OPMVX	vector-scalar	GPR x register rs1
1	1	1	OPCFG	scalars-imms	GPR x register rs1 & rs2/imm

Integer operations are performed using unsigned or two's-complement signed integer arithmetic depending on the opcode.

All standard vector floating-point arithmetic operations follow the IEEE-754/2008 standard. All vector floating-point operations use the dynamic rounding mode in the `frm` register.

Vector-vector operations take two vectors of operands from vector register groups specified by `vs2` and `vs1` respectively.

Vector-scalar operations can have three possible forms, but in all cases take one vector of operands from a vector register group specified by `vs2` and a second scalar source operand from one of three alternative sources.

1. For integer operations, the scalar can be a 5-bit immediate encoded in the `rs1` field. The value is sign- or zero-extended to SEW bits.

2. For integer operations, the scalar can be taken from the scalar `x` register specified by `rs1`. If XLEN>SEW, the least-significant bits of the `x` register are used. If XLEN<SEW, the value from the `x` register is sign-extended to SEW bits.

3. For floating-point operation, the scalar can be taken from a scalar `f` register. If FLEN>SEW, the least-significant bits of the `` `f` ``register are used. If FLEN<SEW, the value is NaN-boxed (one-extended) to SEW.

Note
The proposed Zfinx variants will take the floating-point scalar argument from the x registers.

Vector arithmetic instructions are masked under control of the `vm` field.

Assembly syntax pattern for vector binary arithmetic instructions

```
# Operations returning vector results, masked by vm (v0.t, <nothing>)
vop.vv       vd, vs2, vs1, vm  # integer vector-vector      vd[i] = vs2[i] op vs1[i]
vop.vx       vd, vs2, rs1, vm  # integer vector-scalar      vd[i] = vs2[i] op x[rs1]
vop.vi       vd, vs2, imm, vm  # integer vector-immediate   vd[i] = vs2[i] op imm

vfop.vv      vd, vs2, vs1, vm  # FP vector-vector operation vd[i] = vs2[i] fop vs1[i]
vfop.vf      vd, vs2, rs1, vm  # FP vector-scalar operation vd[i] = vs2[i] fop f[rs1]
```

Note: In the encoding, `vs2` is the first operand, while `rs1/simm5` is the second operand. This is the opposite to the standard scalar ordering. This arrangement retains the existing encoding conventions that instructions that read only one scalar register, read it from `rs1`, and that 5-bit immediates are sourced from the `rs1` field.

Assembly syntax pattern for vector ternary arithmetic instructions (multiply-add)

```
# Integer operations overwriting third source
vop.vv       vd, vs1, vs2, vm  # vd[i] = vs1[i]*vs2[i] + vd[i]
vop.vx       vd, rs1, vs2, vm  # vd[i] = x[rs1]*vs2[i] + vd[i]

# Integer operations overwriting first source
vop.vv       vd, vs1, vs2, vm  # vd[i] = vd[i]*vs1[i] + vs2[i]
vop.vx       vd, rs1, vs2, vm  # vd[i] = vd[i]*x[rs1] + vs2[i]

# Floating-point operations overwriting third source
vfop.vv      vd, vs1, vs2, vm  # vd[i] = vs1[i]*vs2[i] + vd[i]
vfop.vf      vd, rs1, vs2, vm  # vd[i] = f[rs1]*vs2[i] + vd[i]

# Floating-point operations overwriting first source
vfop.vv      vd, vs1, vs2, vm  # vd[i] = vd[i]*vs1[i] + vs2[i]
vfop.vf      vd, rs1, vs2, vm  # vd[i] = vd[i]*f[rs1] + vs2[i]
```

Note: For ternary multiply-add operations, the assembler syntax always places the destination vector register first, followed by either `rs1` or `vs1`, then `vs2`. This ordering provides a more natural reading of the assembler for these ternary operations, as the multiply operands are always next to each other.

11.2. Widening Vector Arithmetic Instructions

A few vector arithmetic instructions are defined to be *widening* operations where the destination elements are 2*SEW wide and are stored in a vector register group with twice the number of vector registers.

The first operand can be either single or double-width. These are generally written with a `vw*` prefix on the opcode or `vfw*` for vector floating-point operations.

Note: Originally, a `w` suffix was used on opcode, but this could be confused with the use of a `w` suffix to mean word-sized operations in doubleword integers, so the `w` was moved to prefix.

Note: The floating-point widening operations were changed to `vfw*` from `vwf*` to be more consistent with any scalar widening floating-point operations that will be written as `fw*`.

Note: For integer multiply-add, another possible widening option increases the size of the accumulator to 4*SEW (i.e., 4*SEW += SEW*SEW). These would be distinguished by a `vw4*` prefix on the opcode. These are not included at this time, but are a possible addition to spec.

The destination vector register group results are arranged as if both SEW and LMUL were at twice their current settings (i.e., the destination element width is 2*SEW, and the destination vector register group LMUL is 2*LMUL).

For all widening instructions, the destination element width must be a supported element width and the destination LMUL value must also be a supported LMUL value (≤8, i.e., current LMUL must be ≤4), otherwise an illegal instruction exception is raised.

The destination vector register group must be specified using a vector register number that is valid for the destination's LMUL value, otherwise an illegal instruction exception is raised.

The destination vector register group cannot overlap a source vector register group (including the mask register if masked), otherwise an illegal instruction exception is raised.

Note: This constraint is necessary to support restart with non-zero `vstart`.

Assembly syntax pattern for vector widening arithmetic instructions

```
# Double-width result, two single-width sources: 2*SEW = SEW op SEW
vwop.vv       vd, vs2, vs1, vm  # integer vector-vector      vd[i] = vs2[i] op vs1[i]
vwop.vx       vd, vs2, rs1, vm  # integer vector-scalar      vd[i] = vs2[i] op x[rs1]

# Double-width result, first source double-width,
# second source single-width: 2*SEW = 2*SEW op SEW
vwop.wv       vd, vs2, vs1, vm  # integer vector-vector      vd[i] = vs2[i] op vs1[i]
vwop.wx       vd, vs2, rs1, vm  # integer vector-scalar      vd[i] = vs2[i] op x[rs1]
```

11.3. Narrowing Vector Arithmetic Instructions

A few instructions are provided to convert double-width source vectors into single-width destination vectors. These instructions convert a vector register group organized as if LMUL and SEW were twice the current settings, and convert to a vector register group with the current LMUL/SEW vectors/elements.

If (2*LMUL > 8), or (2 * SEW) > ELEN, an illegal instruction exception is raised.

Note: An alternative design decision would have been to treat LMUL as defining the size of the source vector register group. The choice here is motivated by the belief the chosen approach will require fewer LMUL changes.

The source and destination vector register groups have to be specified with a vector register number that is legal for the source and destination LMUL value respectively, otherwise an illegal instruction exception is raised.

Where there is a second source vector, this has the same (narrower) width as the result.

A `vn*` prefix on the opcode is used to distinguish these instructions in the assembler, or a `vfn*` prefix for narrowing floating-point opcodes.

Note: Comparison operations that set a mask register are also implicitly a narrowing operation.

12. Vector Integer Arithmetic Instructions

A set of vector integer arithmetic instructions are provided.

12.1. Vector Single-Width Integer Add and Subtract

Vector integer add and subtract are provided. Reverse-subtract instructions are also provided for the vector-scalar forms.

```
# Integer adds.
vadd.vv       vd, vs2, vs1, vm  # Vector-vector
vadd.vx       vd, vs2, rs1, vm  # vector-scalar
vadd.vi       vd, vs2, imm, vm  # vector-immediate

# Integer subtract
vsub.vv       vd, vs2, vs1, vm  # Vector-vector
vsub.vx       vd, vs2, rs1, vm  # vector-scalar

# Integer reverse subtract
vrsub.vx      vd, vs2, rs1, vm  # vd[i] = rs1 - vs2[i]
vrsub.vi      vd, vs2, imm, vm  # vd[i] = imm - vs2[i]
```

12.2. Vector Widening Integer Add/Subtract

The widening add/subtract instructions are provided in both signed and unsigned variants, depending on whether the narrower source operands are first sign- or zero-extended before forming the double-width sum.

```
# Widening unsigned integer add/subtract, 2*SEW = SEW +/- SEW
vwaddu.vv     vd, vs2, vs1, vm  # vector-vector
vwaddu.vx     vd, vs2, rs1, vm  # vector-scalar
vwsubu.vv     vd, vs2, vs1, vm  # vector-vector
vwsubu.vx     vd, vs2, rs1, vm  # vector-scalar

# Widening signed integer add/subtract, 2*SEW = SEW +/- SEW
vwadd.vv      vd, vs2, vs1, vm  # vector-vector
vwadd.vx      vd, vs2, rs1, vm  # vector-scalar
vwsub.vv      vd, vs2, vs1, vm  # vector-vector
vwsub.vx      vd, vs2, rs1, vm  # vector-scalar

# Widening unsigned integer add/subtract, 2*SEW = 2*SEW +/- SEW
vwaddu.wv     vd, vs2, vs1, vm  # vector-vector
vwaddu.wx     vd, vs2, rs1, vm  # vector-scalar
vwsubu.wv     vd, vs2, vs1, vm  # vector-vector
vwsubu.wx     vd, vs2, rs1, vm  # vector-scalar

# Widening signed integer add/subtract, 2*SEW = 2*SEW +/- SEW
vwadd.wv      vd, vs2, vs1, vm  # vector-vector
vwadd.wx      vd, vs2, rs1, vm  # vector-scalar
vwsub.wv      vd, vs2, vs1, vm  # vector-vector
vwsub.wx      vd, vs2, rs1, vm  # vector-scalar
```

Note

An integer value can be doubled in width using the widening add instructions with a scalar operand of `x0`. Can define assembly pseudoinstructions `vwcvt.x.x.v vd,vs,vm = vwadd.vx vd,vs,x0,vm` and `vwcvtu.x.x.v vd,vs,vm = vwaddu.vx vd,vs,x0,vm`.

12.3. Vector Integer Add-with-Carry / Subtract-with-Borrow Instructions

To support multi-word arithmetic:

```
# vd[i] = vs1[i] + vs2[i] + v0[i].LSB
# v0[i] = carry(vs1[i] + vs2[i] + v0[i].LSB)

vadc.vv      vd, vs2, vs1      # Vector-vector
vadc.vx      vd, vs2, rs1      # Vector-scalar
vadc.vi      vd, vs2, imm      # Vector-scalar
```

This instruction is encoded as an unmasked instruction (vm=1) and operates on all body elements. The instruction adds the two source operands and also adds in the LSB of the implicit mask register `v0` as an implicit carry input. The sum is stored in the destination vector register group. The mask element in `v0` is written with the carry result in the LSB and the upper bits of the mask element are set to 0.

The subtract with carry instruction `vsbc` performs the equivalent function to support long word arithmetic for subtraction.

```
# vd[i] = vs1[i] - vs2[i] - v0[i].LSB
# v0[i] = carry(vs1[i] - vs2[i] - v0[i].LSB)

vsbc.vv      vd, vs2, vs1      # Vector-vector
vsbc.vx      vd, vs2, rs1      # Vector-scalar
```

The encodings corresponding to the masked versions (vm=0) of `vadc` and `vsbc` are reserved.

12.4. Vector Bitwise Logical Instructions

Bitwise logical operations.

```
vand.vv      vd, vs2, vs1, vm  # Vector-vector
vand.vx      vd, vs2, rs1, vm  # vector-scalar
vand.vi      vd, vs2, imm, vm  # vector-immediate

vor.vv       vd, vs2, vs1, vm  # Vector-vector
vor.vx       vd, vs2, rs1, vm  # vector-scalar
vor.vi       vd, vs2, imm, vm  # vector-immediate

vxor.vv      vd, vs2, vs1, vm  # Vector-vector
vxor.vx      vd, vs2, rs1, vm  # vector-scalar
vxor.vi      vd, vs2, imm, vm  # vector-immediate
```

Note: With an immediate of -1, scalar-immediate forms of the `vxor` instruction provide a bitwise NOT operation. This can be provided as an assembler pseudoinstruction `vnot.v`.

12.5. Vector Single-Width Bit Shift Instructions

A full complement of vector shift instructions are provided, including logical shift left, and logical (zero-extending) and arithmetic (sign-extending) shift right.

Bit shift operations

```
vsll.vv      vd, vs2, vs1, vm  # Vector-vector
vsll.vx      vd, vs2, rs1, vm  # vector-scalar
vsll.vi      vd, vs2, imm, vm  # vector-immediate

vsrl.vv      vd, vs2, vs1, vm  # Vector-vector
vsrl.vx      vd, vs2, rs1, vm  # vector-scalar
vsrl.vi      vd, vs2, imm, vm  # vector-immediate

vsra.vv      vd, vs2, vs1, vm  # Vector-vector
vsra.vx      vd, vs2, rs1, vm  # vector-scalar
vsra.vi      vd, vs2, imm, vm  # vector-immediate
```

Only the low lg2(SEW) bits are read to obtain the shift amount.

The immediate is treated as an unsigned shift amount, with a maximum shift amount of 31.

12.6. Vector Narrowing Integer Right Shift Instructions

The narrowing right shifts extract a smaller field from a wider operand and have both zero-extending (`srl`) and sign-extending (`sra`) forms. The shift amount can come from a vector or a scalar `x` register or a 5-bit immediate. The low lg2(2*SEW) bits of the vector or scalar shift amount value are used (e.g., the low 6 bits for a SEW=64-bit to SEW=32-bit narrowing operation). The immediate form supports shift amounts up to 31 only and is effectively zero-extended for SEW of > 32.

```
# Narrowing shift right logical, SEW = (2*SEW) >> SEW
vnsrl.vv      vd, vs2, vs1, vm  # vector-vector
vnsrl.vx      vd, vs2, rs1, vm  # vector-scalar
vnsrl.vi      vd, vs2, imm, vm  # vector-immediate

# Narrowing shift right arithmetic, SEW = (2*SEW) >> SEW
vnsra.vv      vd, vs2, vs1, vm  # vector-vector
vnsra.vx      vd, vs2, rs1, vm  # vector-scalar
vnsra.vi      vd, vs2, imm, vm  # vector-immediate
```

Note

It could be useful to add support for n4 variants, where the destination is 1/4 width of source.

12.7. Vector Integer Comparison Instructions

The following integer compare instructions write 1 to the destination mask register element if the comparison evaluates to true, and 0 otherwise. The destination mask vector is always held in a single vector register, with a layout of elements as described in Section Mask Register Layout.

```
# Set if equal
vseq.vv       vd, vs2, vs1, vm  # Vector-vector
vseq.vx       vd, vs2, rs1, vm  # vector-scalar
vseq.vi       vd, vs2, imm, vm  # vector-immediate

# Set if not equal
vsne.vv       vd, vs2, vs1, vm  # Vector-vector
vsne.vx       vd, vs2, rs1, vm  # vector-scalar
vsne.vi       vd, vs2, imm, vm  # vector-immediate

# Set if less than, unsigned
vsltu.vv      vd, vs2, vs1, vm  # Vector-vector
vsltu.vx      vd, vs2, rs1, vm  # Vector-scalar

# Set if less than, signed
vslt.vv       vd, vs2, vs1, vm  # Vector-vector
vslt.vx       vd, vs2, rs1, vm  # vector-scalar

# Set if less than or equal, unsigned
vsleu.vv      vd, vs2, vs1, vm  # Vector-vector
vsleu.vx      vd, vs2, rs1, vm  # vector-scalar
vsleu.vi      vd, vs2, imm, vm  # Vector-immediate

# Set if less than or equal, signed
vsle.vv       vd, vs2, vs1, vm  # Vector-vector
vsle.vx       vd, vs2, rs1, vm  # vector-scalar
vsle.vi       vd, vs2, imm, vm  # vector-immediate

# Set if greater than, unsigned
vsgtu.vx      vd, vs2, rs1, vm  # Vector-scalar
vsgtu.vi      vd, vs2, imm, vm  # Vector-immediate

# Set if greater than, signed
vsgt.vx       vd, vs2, rs1, vm  # Vector-scalar
vsgt.vi       vd, vs2, imm, vm  # Vector-immediate

# Following two instructions are not provided directly
# Set if greater than or equal, unsigned
# vsgeu.vx    vd, vs2, rs1, vm  # Vector-scalar
# Set if greater than or equal, signed
# vsge.vx     vd, vs2, rs1, vm  # Vector-scalar
```

The following table indicates how all comparisons are implemented in native machine code.

Comparison	Assembler Mapping	Assembler Pseudoinstruction
va < vb	vslt{u}.vv vd, va, vb, vm	
va ⇐ vb	vsle{u}.vv vd, va, vb, vm	
va > vb	vslt{u}.vv vd, vb, va, vm	vsgt{u}.vv vd, va, vb, vm
va >= vb	vsle{u}.vv vd, vb, va, vm	vsge{u}.vv vd, va, vb, vm
va < x	vslt{u}.vx vd, va, x, vm	
va ⇐ x	vsle{u}.vx vd, va, x, vm	
va > x	vsgt{u}.vx vd, va, x, vm	
va >= x	see below	
va < i	vsle{u}.vi vd, va, i-1, vm	vslt{u}.vi vd, va, i, vm

Comparison	Assembler Mapping	Assembler Pseudoinstruction
va ⇐ i	vsle{u}.vi vd, va, i, vm	
va > i	vsgt{u}.vi vd, va, i, vm	
va >= i	vsgt{u}.vi vd, va, i-1, vm	vsge{u}.vi vd, va, i, vm

va, vb vector register groups

x scalar integer register

i immediate

Note: The immediate forms of `vslt{u}.vi` are not provided as the immediate value can be decreased by 1 and the `vsle{u}.vi` variants used instead. The `vsle.vi` range is -16 to 15, resulting in an effective `vslt.vi` range of -15 to 16. The `vsleu.vi` range is 0 to 15 (and `(~0)-15` to `~0`), giving an effective `vsltu.vi` range of 1 to 16 (Note, `vsltu.vi` with immediate 0 is not useful as it is always false). Similarly, `vsge{u}.vi` is not provided and the comparison is implemented using `vsgt{u}.vi` with the immediate decremented by one. The resulting effective `vsge.vi` range is -15 to 16, and the resulting effective `vsgeu.vi` range is 1 to 16 (Note, `vsgeu.vi` with immediate 0 is not useful as it is always true).

Note: The `vsgt` forms for register scalar and immediates are provided to allow a single comparison instruction to provide the correct polarity of mask value without using additional mask logical instructions.

To reduce encoding space, the `vsge{u}.vx` form is not directly provided, and so the `va ≥ x` case requires special treatment.

Note: The `vsge{u}.vx` could potentially be encoded in a non-orthogonal way under the unused OPIVI variant of `vslt{u}`. These would be the only instructions in OPIVI that use a scalar \`x\`register however. Alternatively, a further two funct6 encodings could be used, but these would have a different operand format (writes to mask register) than others in the same group of 8 funct6 encodings. The current PoR is to omit these instructions and to synthesize where needed as described below.

The `vsge{u}.vx` operation can be synthesized by reducing the value of `x` by `1` and using the `vsgt{u}.vx` instruction, when it is known that this will not underflow the representation in `x`.

```
Sequences to synthesize `vsge{u}.vx` instruction

va >= x,  x > minimum

   addi t0, x, -1; vsgt{u}.vx vd, va, t0, vm
```

The above sequence will usually be the most efficient implementation, but assembler pseudoinstructions can be provided for cases where the range of `x` is unknown.

```
unmasked va >= x

  pseudoinstruction: vsge{u}.vx vd, va, x
  expansion: vslt{u}.vx vd, va, x; vmnand.mm vd, vd, vd

masked va >= x, vd != v0

  pseudoinstruction: vsge{u}.vx vd, va, x, v0.t
  expansion: vslt{u}.vx vd, va, x, v0.t; vmxor.mm vd, vd, v0

masked va >= x, any vd

  pseudoinstruction: vsge{u}.vx vd, va, x, v0.t, vt
  expansion: vslt{u}.vx vt, va, x;  vmandnot.mm vd, vd, vt

  The vt argument to the pseudoinstruction must name a temporary vector register that is
  not same as vd and which will be clobbered by the pseudoinstruction
```

Comparisons effectively AND in the mask, e.g,

```
# (a < b) && (b < c) in two instructions
vslt.vv      v0, va, vb        # All body elements written
vslt.vv      v0, vb, vc, v0.t  # Only update at set mask
```

12.8. Vector Integer Min/Max Instructions

Signed and unsigned integer mininum and maximum instructions are supported.

```
# Unsigned minimum
vminu.vv      vd, vs2, vs1, vm  # Vector-vector
vminu.vx      vd, vs2, rs1, vm  # vector-scalar

# Signed minimum
vmin.vv       vd, vs2, vs1, vm  # Vector-vector
vmin.vx       vd, vs2, rs1, vm  # vector-scalar

# Unsigned maximum
vmaxu.vv      vd, vs2, vs1, vm  # Vector-vector
vmaxu.vx      vd, vs2, rs1, vm  # vector-scalar

# Signed maximum
vmax.vv       vd, vs2, vs1, vm  # Vector-vector
vmax.vx       vd, vs2, rs1, vm  # vector-scalar
```

12.9. Vector Single-Width Integer Multiply Instructions

The single-width multiply instructions perform a SEW-bit*SEW-bit multiply and return an SEW-bit-wide result. The **mulh** versions write the high word of the product to the destination register.

```
# Signed multiply, returning low bits of product
vmul.vv       vd, vs2, vs1, vm  # Vector-vector
vmul.vx       vd, vs2, rs1, vm  # vector-scalar

# Signed multiply, returning high bits of product
vmulh.vv      vd, vs2, vs1, vm  # Vector-vector
vmulh.vx      vd, vs2, rs1, vm  # vector-scalar

# Unsigned multiply, returning high bits of product
vmulhu.vv     vd, vs2, vs1, vm  # Vector-vector
vmulhu.vx     vd, vs2, rs1, vm  # vector-scalar

# Signed(vs2)-Unsigned multiply, returning high bits of product
vmulhsu.vv    vd, vs2, vs1, vm  # Vector-vector
vmulhsu.vx    vd, vs2, rs1, vm  # vector-scalar
```

Note: There is no `vmulhus` opcode to return high half of unsigned-vector * signed-scalar product.

Note: The current `vmulh*` opcodes perform simple fractional multiplies, but with no option to scale, round, and/or saturate the result. Can consider changing definition of `vmulh`, `vmulhu`, `vmulhsu` to use `vxrm` rounding mode when discarding low half of product. There is no possibility of overflow in this case.

12.10. Vector Widening Integer Multiply Instructions

The widening integer multiply instructions return the full 2*SEW-bit product from an SEW-bit*SEW-bit multiply.

```
# Widening signed-integer multiply
vwmul.vv      vd, vs2, vs1, vm  # vector-vector
vwmul.vx      vd, vs2, rs1, vm  # vector-scalar

# Widening unsigned-integer multiply
vwmulu.vv     vd, vs2, vs1, vm  # vector-vector
vwmulu.vx     vd, vs2, rs1, vm  # vector-scalar

# Widening signed-unsigned integer multiply
vwmulsu.vv    vd, vs2, vs1, vm  # vector-vector
vwmulsu.vx    vd, vs2, rs1, vm  # vector-scalar
```

12.11. Vector Single-Width Integer Multiply-Add Instructions

The integer multiply-add instructions are destructive and are provided in two forms, one that overwrites the addend or minuend (`vmacc`,`vmsac`) and one that overwrites the first multiplicand (`vmadd`,`vmsub`).

The low half of the product is added or subtracted from the third operand.

Note: "sac" is intended to be read as "subtract from accumulator".

```
# Integer multiply-add, overwrite addend
vmacc.vv      vd, vs1, vs2, vm  # vd[i] = +(vs1[i] * vs2[i]) + vd[i]
vmacc.vx      vd, rs1, vs2, vm  # vd[i] = +(x[rs1] * vs2[i]) + vd[i]

# Integer multiply-sub, overwrite minuend
vmsac.vv      vd, vs1, vs2, vm  # vd[i] = -(vs1[i] * vs2[i]) + vd[i]
vmsac.vx      vd, rs1, vs2, vm  # vd[i] = -(x[rs1] * vs2[i]) + vd[i]

# Integer multiply-add, overwrite multiplicand
vmadd.vv      vd, vs1, vs2, vm  # vd[i] = (vd[i] * vs1[i]) + vs2[i]
vmadd.vx      vd, rs1, vs2, vm  # vd[i] = (vd[i] * x[rs1]) + vs2[i]

# Integer multiply-sub, overwrite multiplicand
vmsub.vv      vd, vs1, vs2, vm  # vd[i] = (vd[i] * vs1[i]) - vs2[i]
vmsub.vx      vd, rs1, vs2, vm  # vd[i] = (vd[i] * x[rs1]) - vs2[i]
```

12.12. Vector Widening Integer Multiply-Add Instructions

The widening integer multiply-add instructions add (subtract) a SEW-bit*SEW-bit multiply result to (from) a 2*SEW-bit value and produce a 2*SEW-bit result.

```
# Widening unsigned-integer multiply-add, overwrite addend
vwmaccu.vv    vd, vs1, vs2, vm  # vd[i] = +(vs1[i] * vs2[i]) + vd[i]
vwmaccu.vx    vd, rs1, vs2, vm  # vd[i] = +(x[rs1] * vs2[i]) + vd[i]

# Widening signed-integer multiply-add, overwrite addend
vwmacc.vv     vd, vs1, vs2, vm  # vd[i] = +(vs1[i] * vs2[i]) + vd[i]
vwmacc.vx     vd, rs1, vs2, vm  # vd[i] = +(x[rs1] * vs2[i]) + vd[i]

# Widening unsigned-integer multiply-sub, overwrite minuend
vwmsacu.vv    vd, vs1, vs2, vm  # vd[i] = -(vs1[i] * vs2[i]) + vd[i]
vwmsacu.vx    vd, rs1, vs2, vm  # vd[i] = -(x[rs1] * vs2[i]) + vd[i]

# Widening signed-integer multiply-sub, overwrite minuend
vwmsac.vv     vd, vs1, vs2, vm  # vd[i] = -(vs1[i] * vs2[i]) + vd[i]
vwmsac.vx     vd, rs1, vs2, vm  # vd[i] = -(x[rs1] * vs2[i]) + vd[i]
```

Note

There is no signed-unsigned widening multiply-add instruction.

12.12.1. Vector Integer Divide Instructions

The divide and remainder instructions are equivalent to the RISC-V standard scalar integer multiply/divides, with the same results for extreme inputs.

```
# Unsigned divide.
vdivu.vv      vd, vs2, vs1, vm  # Vector-vector
vdivu.vx      vd, vs2, rs1, vm  # vector-scalar

# Signed divide
vdiv.vv       vd, vs2, vs1, vm  # Vector-vector
vdiv.vx       vd, vs2, rs1, vm  # vector-scalar

# Unsigned remainder
vremu.vv      vd, vs2, vs1, vm  # Vector-vector
vremu.vx      vd, vs2, rs1, vm  # vector-scalar

# Signed remainder
vrem.vv       vd, vs2, vs1, vm  # Vector-vector
vrem.vx       vd, vs2, rs1, vm  # vector-scalar
```

Note

The decision to include integer divide and remainder was contentious. The argument in favor is that without a standard instruction, software would have to pick some algorithm to perform the operation, which would likely perform poorly on some microarchitectures versus others.

Note

There is no instruction to perform a "scalar divide by vector" operation.

12.13. Vector Integer Merge Instruction

The vector integer merge instruction combines two source operands based on the mask field. Unlike regular arithmetic instructions, the merge operates on all body elements (i.e., the set of elements from `vstart` up to the current vector length in `vl`).

When the operation is masked (`vm=0`), the instructions combine two sources as follows. At elements where the mask value is zero, the first operand is copied to the destination element, otherwise the second operand is copied to the destination element. The first operand is always a vector register group specified by `vs2`. The second operand is a vector register group specified by `vs1` or a scalar `x` register specified by `rs1` or a 5-bit sign-extended immediate.

```
# Masked  operations, where vm=0
vmerge.vv     vd, vs2, vs1, v0.t  # vd[i] = v0[i].LSB ? vs1[i] : vs2[i]
vmerge.vx     vd, vs2, rs1, v0.t  # vd[i] = v0[i].LSB ? x[rs1] : vs2[i]
vmerge.vi     vd, vs2, imm, v0.t  # vd[i] = v0[i].LSB ? imm    : vs2[i]
```

When the operation is unmasked (`vm=1`), the instructions copies the second operand to the first `vl` locations of the destination vector. The first operand specifier (`vs2`) in the instruction encoding must contain `v0`, and any other vector register number in `vs2` is *reserved*.

Note

Microarchitectures can recognize this form to avoid unnecessary vector register file accesses from the first vector operand.

```
# Unmasked  operations, where vm=1
vmerge.vv     vd, v0, vs1       # vd[i] = vs1[i], pseudoinstruction vmv.v.v vd, vs1
vmerge.vx     vd, v0, rs1       # vd[i] = x[rs1], pseudoinstruction vmv.v.x vd, rs1
vmerge.vi     vd, v0, imm       # vd[i] = imm,    pseudoinstruction vmv.v.i vd, imm
```

An unmasked `vmerge.vv` instruction can be used to copy one vector register group to another. This is given a vector pseudoinstruction `vmv.v.v vdest, vsrc` (which expands to `vmerge.vv vdest, v0, vsrc`).

An unmasked `vmerge.vx` instruction can be used to *splat* a scalar `x` register value into all active elements of a vector. This is given a vector pseudoinstruction `vmv.v.x vd, rs1`, which expands to `vmerge.vx vd, v0, rs1`.

An unmasked `vmerge.vi` instruction can be used to initialize a vector register group with an immediate value. This is given a vector pseudoinstruction `vmv.v.i vd, imm`.

Note

Mask values can be widened into SEW-width elements using a sequence `vmv.v.i vd, 0; vmerge.vi vd, vd, 1, v0.t`.

13. Vector Fixed-Point Arithmetic Instructions

A set of vector arithmetic instructions are provided to support fixed-point arithmetic.

An N-bit element can hold two's-complement signed integers in the range $-2^{N-1} \ldots +2^{N-1}-1$, and unsigned integers in the range $0 \ldots +2^{N}-1$. The fixed-point instructions help preserve precision in narrow operands by supporting scaling and rounding, and can handle overflow by saturating results into the destination format range.

Note

The widening integer operations described above can also be used to remove the possibility of overflow.

13.1. Vector Single-Width Saturating Add and Subtract

Saturating forms of integer add and subtract are provided, for both signed and unsigned integers. If the result would overflow the destination, the result is replaced with the closest representable value, and the `vxsat` bit is set.

```
# Saturating adds of unsigned integers.
vsaddu.vv     vd, vs2, vs1, vm  # Vector-vector
vsaddu.vx     vd, vs2, rs1, vm  # vector-scalar
vsaddu.vi     vd, vs2, imm, vm  # vector-immediate

# Saturating adds of signed integers.
vsadd.vv      vd, vs2, vs1, vm  # Vector-vector
vsadd.vx      vd, vs2, rs1, vm  # vector-scalar
vsadd.vi      vd, vs2, imm, vm  # vector-immediate

# Saturating subtract of unsigned integers.
vssubu.vv     vd, vs2, vs1, vm  # Vector-vector
vssubu.vx     vd, vs2, rs1, vm  # vector-scalar

# Saturating subtract of signed integers.
vssub.vv      vd, vs2, vs1, vm  # Vector-vector
vssub.vx      vd, vs2, rs1, vm  # vector-scalar
```

13.2. Vector Single-Width Averaging Add and Subtract

The averaging add and subtract instructions right shift the result by one bit and round off the result according to the setting in `vxrm`. There can be no overflow in the result.

```
# For vrxm=rnu, round = 1

# Averaging add
# result = (src1 + src2 + round) >> 1;

# Averaging adds of integers.
vaadd.vv      vd, vs2, vs1, vm  # Vector-vector
vaadd.vx      vd, vs2, rs1, vm  # vector-scalar
vaadd.vi      vd, vs2, imm, vm  # vector-immediate

# Averaging subtract
# result = (src1 - src2 + round) >> 1;

# Averaging subtract of integers.
vasub.vv      vd, vs2, vs1, vm  # Vector-vector
vasub.vx      vd, vs2, rs1, vm  # vector-scalar
```

13.3. Vector Single-Width Fractional Multiply with Rounding and Saturation

The signed fractional multiply instruction produces a 2*SEW product of the two SEW inputs, then shifts the result right by SEW-1 bits, rounding these bits according to `vxrm`, then saturates the result to fit into SEW bits. If the result causes saturation, the `vxsat` bit is set.

Signed saturating and rounding fractional multiply

```
vsmul.vv      vd, vs2, vs1, vm  # vd[i] = clip((vs2[i]*vs1[i]+round)>>(SEW-1))
vsmul.vx      vd, vs2, rs1, vm  # vd[i] = clip((vs2[i]*x[rs1]+round)>>(SEW-1))
```

Note: When multiplying two N-bit signed numbers, the largest magnitude is obtained for $-2^{N-1} * -2^{N-1}$ producing a result $+2^{2N-2}$, which has a single (zero) sign bit when held in 2N bits. All other products have two sign bits in 2N bits. To retain greater precision in N result bits, the product is shifted right by one bit less than N, saturating the largest magnitude result but increasing result precision by one bit for all other products.

Note: Considering adding `vxrm`-controlled rounding to `vmulhu`, `vmulhsu`, and `vmulh` to further support fixed-point. These would not have saturation.

13.4. Vector Widening Saturating Scaled Multiply-Add

The widening saturating scaled multiply-add instructions perform an SEW-bit * SEW-bit multiply to yield a 2*SEW-bit product. The product is then right-shifted by SEW/2 bits with the shifted bits rounded off according to `vxrm`, and the rounded product is added to a 2*SEW-bit destination accumulator, with saturation if the result would overflow the destination accumulator. The `vxsat` bit is set if any overflow occurs.

SEW	Product Width	Rounded Product	Guard Accumulator	Bits
8	16	12	16	4
16	32	24	32	8
32	64	48	64	16

```
# Widening unsigned-integer scaled multiply-accumulate
vwsmaccu.vv   vd, vs1, vs2, vm  # vd[i] = clipu((+(vs1[i]*vs2[i]+round)>>SEW/2)+vd[i])
vwsmaccu.vx   vd, rs1, vs2, vm  # vd[i] = clipu((+(x[rs1]*vs2[i]+round)>>SEW/2)+vd[i])

# Widening signed-integer scaled multiply-accumulate
vwsmacc.vv    vd, vs1, vs2, vm  # vd[i] = clipu((+(vs1[i]*vs2[i]+round)>>SEW/2)+vd[i])
vwsmacc.vx    vd, rs1, vs2, vm  # vd[i] = clipu((+(x[rs1]*vs2[i]+round)>>SEW/2)+vd[i])

# Widening unsigned-integer scaled multiply-subtract
vwsmsacu.vv   vd, vs1, vs2, vm  # vd[i] = clipu(-((vs1[i]*vs2[i]+round)>>SEW/2)+vd[i])
vwsmsacu.vx   vd, rs1, vs2, vm  # vd[i] = clipu(-((x[rs1]*vs2[i]+round)>>SEW/2)+vd[i])

# Widening signed-integer scaled  multiply-subtract
vwsmsac.vv    vd, vs1, vs2, vm  # vd[i] = clipu(-((vs1[i]*vs2[i]+round)>>SEW/2)+vd[i])
vwsmsac.vx    vd, rs1, vs2, vm  # vd[i] = clipu(-((x[rs1]*vs2[i]+round)>>SEW/2)+vd[i])

# For vxrm=rnu, round = ( 1 << (SEW/2-1))
```

Note: An arbitrary scaling/shift amount would be more flexible but would require a fourth source operand.

13.5. Vector Single-Width Scaling Shift Instructions

These instructions shift the input value right, and round off the shifted out bits according to `vxrm`. The scaling right shifts have both zero-extending (`vssrl`) and sign-extending (`vssra`) forms. The shift amount can come from a vector or a scalar x register or a 5-bit immediate. The low lg2(2*SEW) bits of the vector or scalar shift amount value are used (e.g., the low 6 bits for a SEW=64-bit to SEW=32-bit narrowing operation). The immediate form supports shift amounts up to 31 only and is effectively zero-extended for SEW of > 32.

```
# For vxrm=rnu, round = 1 << (src2-1)
# Scaling shift right logical
vssrl.vv      vd, vs2, vs1, vm  # vd[i] = ((vs2[i] + round)>>vs1[i])
vssrl.vx      vd, vs2, rs1, vm  # vd[i] = ((vs2[i] + round)>>x[rs1])
vssrl.vi      vd, vs2, imm, vm  # vd[i] = ((vs2[i] + round)>>imm)

# Scaling shift right arithmetic
vssra.vv      vd, vs2, vs1, vm  # vd[i] = ((vs2[i] + round)>>vs1[i])
vssra.vx      vd, vs2, rs1, vm  # vd[i] = ((vs2[i] + round)>>x[rs1])
vssra.vi      vd, vs2, imm, vm  # vd[i] = ((vs2[i] + round)>>imm)
```

13.6. Vector Narrowing Fixed-Point Clip Instructions

The `vnclip` instructions are used to pack a fixed-point value into a narrower destination. The instructions support rounding, scaling, and saturation into the final destination format.

```
# Narrowing unsigned clip
vnclipu.vv    vd, vs2, vs1, vm  # vector-vector
vnclipu.vx    vd, vs2, rs1, vm  # vector-scalar
vnclipu.vi    vd, vs2, imm, vm  # vector-immediate

# Narrowing signed clip,  vd[i] = clip(round(vs2[i] + rnd) >> vs1[i])
#                          SEW           2*SEW                   SEW
vnclip.vv     vd, vs2, vs1, vm  # vector-vector
vnclip.vx     vd, vs2, rs1, vm  # vector-scalar
vnclip.vi     vd, vs2, imm, vm  # vector-immediate
```

For `vnclipu`/`vnclip`, the rounding mode is specified in the `vxrm` CSR. Rounding occurs around the least-significant bit of the destination and before saturation.

For `vnclipu`, the shifted rounded source value is treated as an unsigned integer and saturates if the result would overflow the destination viewed as an unsigned integer.

For `vnclip`, the shifted rounded source value is treated as a signed integer and saturates if the result would overflow the destination viewed as a signed integer.

If any destination element is saturated, the `vxsat` bit is set in the `vxsat` register.

14. Vector Floating-Point Instructions

The standard vector floating-point instructions treat 16-bit, 32-bit, 64-bit, and 128-bit elements as IEEE-754/2008-compatible values. If the current SEW does not correspond to a supported IEEE floating-point type, an illegal instruction exception is raised.

Note Note The floating-point element widths that are supported depend on the platform.

Platforms supporting 16-bit half-precision floating-point values will also have to implement scalar half-precision floating-point support in the `f` registers.

The vector floating-point instructions have the same behavior as the scalar floating-point instructions with regard to NaNs.

Scalar values for vector-scalar operations can be sourced from the standard scalar `f` registers.

Note Scalar floating-point values will be sourced from the integer x registers in the proposed Zfinx variant.

14.1. Vector Floating-Point Exception Flags

A vector floating-point exception at any active floating-point element sets the standard FP exception flags in the `fflags` register. Inactive elements do not set FP exception flags.

14.2. Vector Single-Width Floating-Point Add/Subtract Instructions

```
# Floating-point add
vfadd.vv      vd, vs2, vs1, vm  # Vector-vector
vfadd.vf      vd, vs2, rs1, vm  # vector-scalar

# Floating-point subtract
vfsub.vv      vd, vs2, vs1, vm  # Vector-vector
vfsub.vf      vd, vs2, rs1, vm  # vector-scalar
```

14.3. Vector Widening Floating-Point Add/Subtract Instructions

```
# Widening FP add/subtract, 2*SEW = SEW +/- SEW
vfwadd.vv     vd, vs2, vs1, vm  # vector-vector
vfwadd.vf     vd, vs2, rs1, vm  # vector-scalar
vfwsub.vv     vd, vs2, vs1, vm  # vector-vector
vfwsub.vf     vd, vs2, rs1, vm  # vector-scalar

# Widening FP add/subtract, 2*SEW = 2*SEW +/- SEW
vfwadd.wv     vd, vs2, vs1, vm  # vector-vector
vfwadd.wf     vd, vs2, rs1, vm  # vector-scalar
vfwsub.wv     vd, vs2, vs1, vm  # vector-vector
vfwsub.wf     vd, vs2, rs1, vm  # vector-scalar
```

14.4. Vector Single-Width Floating-Point Multiply/Divide Instructions

```
# Floating-point multiply
vfmul.vv    vd, vs2, vs1, vm    # Vector-vector
vfmul.vf    vd, vs2, rs1, vm    # vector-scalar

# Floating-point divide
vfdiv.vv    vd, vs2, vs1, vm    # Vector-vector
vfdiv.vf    vd, vs2, rs1, vm    # vector-scalar

# Reverse floating-point divide vector = scalar / vector
vfrdiv.vf   vd, vs2, rs1, vm    # scalar-vector, vd[i] = f[rs1]/vs2[i]
```

14.5. Vector Widening Floating-Point Multiply

Widening floating-point multiply

```
vfwmul.vv    vd, vs2, vs1, vm   # vector-vector
vfwmul.vf    vd, vs2, rs1, vm   # vector-scalar
```

14.6. Vector Single-Width Floating-Point Fused Multiply-Add Instructions

All four varieties of fused multiply-add are provided, and in two destructive forms that overwrite one of the operands, either the addend or the first multiplicand.

```
# FP multiply-accumulate, overwrites addend
vfmacc.vv     vd, vs1, vs2, vm  # vd[i] = +(vs1[i] * vs2[i]) + vd[i]
vfmacc.vf     vd, rs1, vs2, vm  # vd[i] = +(f[rs1] * vs2[i]) + vd[i]

# FP negate-(multiply-accumulate), overwrites subtrahend
vfnmacc.vv    vd, vs1, vs2, vm  # vd[i] = -(vs1[i] * vs2[i]) - vd[i]
vfnmacc.vf    vd, rs1, vs2, vm  # vd[i] = -(f[rs1] * vs2[i]) - vd[i]

# FP multiply-subtract-accumulator, overwrites subtrahend
vfmsac.vv     vd, vs1, vs2, vm  # vd[i] = +(vs1[i] * vs2[i]) - vd[i]
vfmsac.vf     vd, rs1, vs2, vm  # vd[i] = +(f[rs1] * vs2[i]) - vd[i]

# FP negate-(multiply-subtract-accumulator), overwrites minuend
vfnmsac.vv    vd, vs1, vs2, vm  # vd[i] = -(vs1[i] * vs2[i]) + vd[i]
vfnmsac.vf    vd, rs1, vs2, vm  # vd[i] = -(f[rs1] * vs2[i]) + vd[i]

# FP multiply-add, overwrites multiplicand
vfmadd.vv     vd, vs1, vs2, vm  # vd[i] = +(vd[i] * vs1[i]) + vs2[i]
vfmadd.vf     vd, rs1, vs2, vm  # vd[i] = +(vd[i] * f[rs1]) + vs2[i]

# FP negate-(multiply-add), overwrites multiplicand
vfnmadd.vv    vd, vs1, vs2, vm  # vd[i] = -(vd[i] * vs1[i]) - vs2[i]
vfnmadd.vf    vd, rs1, vs2, vm  # vd[i] = -(vd[i] * f[rs1]) - vs2[i]

# FP multiply-sub, overwrites multiplicand
vfmsub.vv     vd, vs1, vs2, vm  # vd[i] = +(vd[i] * vs1[i]) - vs2[i]
vfmsub.vf     vd, rs1, vs2, vm  # vd[i] = +(vd[i] * f[rs1]) - vs2[i]

# FP negate-(multiply-sub), overwrites multiplicand
vfnmsub.vv    vd, vs1, vs2, vm  # vd[i] = -(vd[i] * vs1[i]) + vs2[i]
vfnmsub.vf    vd, rs1, vs2, vm  # vd[i] = -(vd[i] * f[rs1]) + vs2[i]
```

Note

It would be possible to use the two unused rounding modes in the scalar FP FMA encoding to provide a few non-destructive FMAs. However, this would be the only maskable operation with three inputs and separate output.

14.7. Vector Widening Floating-Point Fused Multiply-Add Instructions

The widening floating-point fused multiply-add instructions all overwrite the wide addend with the result. The multiplier inputs are all SEW wide, while the addend and destination is 2*SEW bits wide.

```
# FP widening multiply-accumulate, overwrites addend
vfwmacc.vv    vd, vs1, vs2, vm  # vd[i] = +(vs1[i] * vs2[i]) + vd[i]
vfwmacc.vf    vd, rs1, vs2, vm  # vd[i] = +(f[rs1] * vs2[i]) + vd[i]

# FP widening negate-(multiply-accumulate), overwrites addend
vfwnmacc.vv   vd, vs1, vs2, vm  # vd[i] = -(vs1[i] * vs2[i]) - vd[i]
vfwnmacc.vf   vd, rs1, vs2, vm  # vd[i] = -(f[rs1] * vs2[i]) - vd[i]

# FP widening multiply-subtract-accumulator, overwrites addend
vfwmsac.vv    vd, vs1, vs2, vm  # vd[i] = +(vs1[i] * vs2[i]) - vd[i]
vfwmsac.vf    vd, rs1, vs2, vm  # vd[i] = +(f[rs1] * vs2[i]) - vd[i]

# FP widening negate-(multiply-subtract-accumulator), overwrites addend
vfwnmsac.vv   vd, vs1, vs2, vm  # vd[i] = -(vs1[i] * vs2[i]) + vd[i]
vfwnmsac.vf   vd, rs1, vs2, vm  # vd[i] = -(f[rs1] * vs2[i]) + vd[i]
```

14.8. Vector Floating-Point Square-Root Instruction

This is a unary vector-vector instruction.

Floating-point square root

```
vfsqrt.v      vd, vs2, vm   # Vector-vector square root
```

14.9. Vector Floating-Point MIN/MAX Instructions

The vector floating-point `vfmin` and `vfmax` instructions have the same behavior as the corresponding scalar floating-point instructions in version 2.2 of the RISC-V F/D/Q extension.

```
# Floating-point minimum
vfmin.vv      vd, vs2, vs1, vm  # Vector-vector
vfmin.vf      vd, vs2, rs1, vm  # vector-scalar

# Floating-point maximum
vfmax.vv      vd, vs2, vs1, vm  # Vector-vector
vfmax.vf      vd, vs2, rs1, vm  # vector-scalar
```

14.10. Vector Floating-Point Sign-Injection Instructions

Vector versions of the scalar sign-injection instructions. The result takes all bits except the sign bit from the vector `vs2` operands.

```
vfsgnj.vv     vd, vs2, vs1, vm  # Vector-vector
vfsgnj.vf     vd, vs2, rs1, vm  # vector-scalar

vfsgnjn.vv    vd, vs2, vs1, vm  # Vector-vector
vfsgnjn.vf    vd, vs2, rs1, vm  # vector-scalar

vfsgnjx.vv    vd, vs2, vs1, vm  # Vector-vector
vfsgnjx.vf    vd, vs2, rs1, vm  # vector-scalar
```

14.11. Vector Floating-Point Compare Instructions

These vector FP compare instructions compare two source operands and write the comparison result to a mask register. The destination mask vector is always held in a single vector register, with a layout of elements as described in Section Mask Register Layout.

The compare instructions follow the semantics of the scalar floating-point compare instructions.

```
# Compare equal
vfeq.vv       vd, vs2, vs1, vm  # Vector-vector
vfeq.vf       vd, vs2, rs1, vm  # vector-scalar

# Compare not equal
vfne.vv       vd, vs2, vs1, vm  # Vector-vector
vfne.vf       vd, vs2, rs1, vm  # vector-scalar

# Compare less than
vflt.vv       vd, vs2, vs1, vm  # Vector-vector
vflt.vf       vd, vs2, rs1, vm  # vector-scalar

# Compare less than or equal
vfle.vv       vd, vs2, vs1, vm  # Vector-vector
vfle.vf       vd, vs2, rs1, vm  # vector-scalar

# Compare greater than
vfgt.vf       vd, vs2, rs1, vm  # vector-scalar

# Compare greater than or equal
vfge.vf       vd, vs2, rs1, vm  # vector-scalar
```

Comparison	Assembler Mapping	Assembler pseudoinstruction
va < vb	vflt.vv vd, va, vb, vm	
va ⇐ vb	vfle.vv vd, va, vb, vm	
va > vb	vflt.vv vd, vb, va, vm	vfgt.vv vd, va, vb, vm
va >= vb	vfle.vv vd, vb, va, vm	vfge.vv vd, va, vb, vm
va < f	vflt.vf vd, va, f, vm	
va ⇐ f	vfle.vf vd, va, f, vm	
va > f	vfgt.vf vd, va, f, vm	
va >= f	vfge.vf vd, va, f, vm	

va, vb vector register groups

f scalar floating-point register

Note: Providing all forms is necessary to correctly handle unordered comparisons for NaNs.

To help implement the C99 floating-point comparison functions, a `vford` instruction is added that sets a mask register if the arguments are ordered (i.e., neither argument is NaN).

```
# Are args ordered?
vford.vv      vd, vs2, vs1, vm   # Vector-vector
vford.vf      vd, vs2, rs1, vm   # Vector-scalar
```

```
# Example of implementing isgreater()
vford.vv      v0, va, vb        # Only set where args are ordered,
vfgt.vv       v0, va, vb, v0.t  # so only set flags on ordered values.
```

Note: Detailed NaN signaling conventions to be added.

14.12. Vector Floating-Point Classify Instruction

This is a unary vector-vector instruction that operates in the same way as the scalar classify instruction.

```
vfclass.v     vd, vs2, vm   # Vector-vector
```

The 10-bit mask produced by this instruction is placed in the least-significant bits of the result elements. The instruction is only defined for SEW=16b and above, so the result will always fit in the destination elements.

14.13. Vector Floating-Point Merge Instruction

A vector-scalar floating-point merge instruction is provided, which operates on all body elements, from `vstart` up to the current vector length in `vl` regardless of mask value.

When the floating-point merge instruction is masked (`vm=0`), at elements where the mask value is zero, the first vector operand is copied to the destination element, otherwise a scalar floating-point register value is copied to the destination element.

```
vfmerge.vf    vd, vs2, rs1, v0.t  # vd[i] = v0[i].LSB ? f[rs1] : vs2[i]
```

The unmasked form (`vm=1`) can be used to *splat* a scalar `f` register value into all active elements of a vector. The instruction must have the `vs2` field set to `v0`, with all other values for `vs2` reserved.

```
vfmerge.vf    vd, v0, rs1  # vd[i] = f[rs1];
```

This is given a vector pseudoinstruction `vmv.v.f vd, rs1` which expands to `vfmerge.vf vd, v0, rs1`.

Note: In Zfinx systems, the instruction is identical to `vmerge.vx`.

14.14. Single-Width Floating-Point/Integer Type-Convert Instructions

Conversion operations are provided to convert to and from floating-point values and unsigned and signed integers, where both source and destination are SEW wide.

```
vfcvt.xu.f.v  vd, vs2, vm   # Convert float to unsigned integer.
vfcvt.x.f.v   vd, vs2, vm   # Convert float to signed integer.

vfcvt.f.xu.v  vd, vs2, vm   # Convert unsigned integer to float.
vfcvt.f.x.v   vd, vs2, vm   # Convert signed integer to float.
```

The conversions follow the same rules on exceptional conditions as the scalar conversion instructions. The conversions always use the dynamic rounding mode in `frm`.

14.15. Widening Floating-Point/Integer Type-Convert Instructions

A set of conversion instructions are provided to convert between narrower integer and floating-point datatypes to a type of twice the width.

```
vfwcvt.xu.f.v vd, vs2, vm   # Convert float to double-width unsigned integer.
vfwcvt.x.f.v  vd, vs2, vm   # Convert float to double-width signed integer.

vfwcvt.f.xu.v vd, vs2, vm   # Convert unsigned integer to double-width float.
vfwcvt.f.x.v  vd, vs2, vm   # Convert signed integer to double-width float.

vfwcvt.f.f.v  vd, vs2, vm   # Convert single-width float to double-width float.
```

Note A double-width IEEE floating-point value can always represent a single-width integer exactly.

Note A double-width IEEE floating-point value can always represent a single-width IEEE floating-point value exactly.

Note A full set of floating-point widening conversions are not supported as single instructions, but any widening conversion can be implemented as several doubling steps with equivalent results and no additional exception flags raised.

14.16. Narrowing Floating-Point/Integer Type-Convert Instructions

A set of conversion instructions are provided to convert wider integer and floating-point datatypes to a type of half the width.

```
vfncvt.xu.f.v vd, vs2, vm   # Convert double-width float to unsigned integer.
vfncvt.x.f.v  vd, vs2, vm   # Convert double-width float to signed integer.

vfncvt.f.xu.v vd, vs2, vm   # Convert double-width unsigned integer to float.
vfncvt.f.x.v  vd, vs2, vm   # Convert double-width signed integer to float.

vfncvt.f.f.v  vd, vs2, vm   # Convert double-width float to single-width float.
```

Note A full set of floating-point widening conversions are not supported as single instructions. Conversions can be implemented in several halving steps, with equivalently rounded results and with the same exception flags raised (possibly raised redundantly in multiple steps).

Note An integer value can be halved in width using the narrowing integer shift instructions with a shift amount of 0.

15. Vector Reduction Operations

Vector reduction operations take a vector register group of elements and a scalar held in element 0 of a vector register, and perform a reduction using some binary operator, to produce a scalar result in element 0 of a vector register. The scalar input and output operands are held in element 0 of a single vector register, not a vector register group, so any vector register can be the scalar source or destination of a vector reduction regardless of LMUL setting.

Note: Reductions read and write the scalar operand and result into element 0 of a vector register to avoid a loss of decoupling with the scalar processor, and to support future polymorphic use with future types not supported in the scalar unit.

Inactive elements are excluded from the reduction.

The other elements in the destination vector register (0 < index < VLEN/SEW) are zeroed.

If `vl`=0, no operation is performed and the destination register is not updated.

Traps on vector reduction instructions are always reported with a `vstart` of 0. Vector reduction operations raise an illegal instruction exception if `vstart` is non-zero.

The assembler syntax for a reduction operation is `vredop.vs`, where the `.vs` suffix denotes the first operand is a vector register group and the second operand is a scalar stored in element 0 of a vector register.

15.1. Vector Single-Width Integer Reduction Instructions

All operands and results of single-width reduction instructions have the same SEW width. Overflows wrap around on arithmetic sums.

```
# Simple reductions, where [*] denotes all active elements:
vredsum.vs    vd, vs2, vs1, vm   # vd[0] =  sum( vs2[*] , vs1[0] )
vredmaxu.vs   vd, vs2, vs1, vm   # vd[0] = maxu( vs2[*] , vs1[0] )
vredmax.vs    vd, vs2, vs1, vm   # vd[0] =  max( vs2[*] , vs1[0] )
vredminu.vs   vd, vs2, vs1, vm   # vd[0] = minu( vs2[*] , vs1[0] )
vredmin.vs    vd, vs2, vs1, vm   # vd[0] =  min( vs2[*] , vs1[0] )
vredand.vs    vd, vs2, vs1, vm   # vd[0] =  and( vs2[*] , vs1[0] )
vredor.vs     vd, vs2, vs1, vm   # vd[0] =   or( vs2[*] , vs1[0] )
vredxor.vs    vd, vs2, vs1, vm   # vd[0] =  xor( vs2[*] , vs1[0] )
```

15.2. Vector Widening Integer Reduction Instructions

The unsigned `vwredsumu.vs` instruction zero-extends the SEW-wide vector elements before summing them, then adds the 2*SEW-width scalar element, and stores the result in a 2*SEW-width scalar element.

The `vwredsum.vs` instruction sign-extends the SEW-wide vector elements before summing them.

```
# Unsigned sum reduction into double-width accumulator
vwredsumu.vs  vd, vs2, vs1, vm   # 2*SEW = sum(SEW) + 2*SEW

# Signed sum reduction into double-width accumulator
vwredsum.vs   vd, vs2, vs1, vm   # 2*SEW = sum(SEW) + 2*SEW
```

15.3. Vector Single-Width Floating-Point Reduction Instructions

Simple reductions.

```
vfredosum.vs  vd, vs2, vs1, vm # Ordered sum
vfredsum.vs   vd, vs2, vs1, vm # Unordered sum
vfredmax.vs   vd, vs2, vs1, vm # Maximum value
vfredmin.vs   vd, vs2, vs1, vm # Minimum value
```

The `vfredosum` instruction must sum the floating-point values in element order, while the `vfredsum` is allowed to perform the reduction in any order, provided the final result corresponds to some sequential ordering of `vl-1` floating-point add operations.

Note The ordered reduction supports compiler autovectorization, while the unordered FP sum allows for faster implementations.

Note Floating-point max and min reductions should return the same final value and exception flags regardless of operation order.

15.4. Vector Widening Floating-Point Reduction Instructions

Widening forms of the sum reductions are provided that read and write a double-width reduction result.

Simple reductions.

```
vfwredosum.vs vd, vs2, vs1, vm # Ordered reduce 2*SEW = sum(SEW) + 2*SEW
vfwredsum.vs  vd, vs2, vs1, vm  # Unordered reduce  2*SEW = sum(SEW) + 2*SEW
```

The reduction of the SEW-width elements is performed as in the single-width reduction case, with the final SEW-width result promoted to 2*SEW bits before adding to the 2*SEW-width accumulator.

For the `vfwredsum.vs` instruction, implementations may optionally perform the reduction by first promoting elements to the wider (2*SEW) format.

Note The `vfwredosum.vs` instruction must round as if performed sequentially in the original format.

16. Vector Mask Instructions

Several instructions are provided to help operate on mask values held in a vector register.

16.1. Vector Mask-Register Logical Instructions

Vector mask-register logical operations operate on mask registers. The size of one element in a mask register is SEW/LMUL, so these instructions all operate on single vector registers regardless of the setting of the `vlmul` field in `vtype`. They do not change the value of `vlmul`.

As with other vector instructions, the elements with indices less than `vstart` are unchanged, and `vstart` is reset to zero after execution. Vector mask logical instructions are always unmasked so there are no inactive elements. Mask elements past `vl`, the tail elements, are zeroed.

```
vmand.mm      vd, vs2, vs1  # vd =   vs2 &  vs1
vmnand.mm     vd, vs2, vs1  # vd = ~(vs2 &  vs1)
vmandnot.mm   vd, vs2, vs1  # vd =   vs2 & ~vs1
vmxor.mm      vd, vs2, vs1  # vd =   vs2 ^  vs1
vmor.mm       vd, vs2, vs1  # vd =   vs2 |  vs1
vmnor.mm      vd, vs2, vs1  # vd = ~(vs2 |  vs1)
vmornot.mm    vd, vs2, vs1  # vd =   vs2 | ~vs1
vmxnor.mm     vd, vs2, vs1  # vd = ~(vs2 ^  vs1)
```

Several assembler pseudoinstructions are defined as shorthand for common uses of mask logical operations:

```
vmmv.m vd, vs  => vmand.mm vd, vs, vs   # Move mask register
vmclr.m vd     => vmxor.mm vd, vd, vd   # Clear mask register
vmset.m vd     => vmxnor.mm vd, vd, vd  # Set mask register
vmnot.m vd, vs => vmnand.mm vd, vs, vs  # Invert bits
```

The set of eight mask logical instructions can generate any of the 16 possibly binary logical functions of the two input masks:

inputs					
0	0	1	1	src1	
0	1	0	1	src2	
output				**instruction**	**pseudoinstruction**
0	0	0	0	vmxor.mm vd, vd, vd	vmclr.m vd
1	0	0	0	vmnor.mm vd, src1, src2	
0	1	0	0	vmandnot.mm vd, src2, src1	
1	1	0	0	vmnand.mm vd, src1, src1	vmnot.m vd, src1
0	0	1	0	vmandnot.mm vd, src1, src2	
1	0	1	0	vmnand.mm vd, src2, src2	vmnot.m vd, src2
0	1	1	0	vmxor.mm vd, src1, src2	
1	1	1	0	vmnand.mm vd, src1, src2	
0	0	0	1	vmand.mm vd, src1, src2	

output				instruction	pseudoinstruction
1	0	0	1	vmxnor.mm vd, src1, src2	
0	1	0	1	vmand.mm vd, src2, src2	vmmv.m vd, src2
1	1	0	1	vmornot.mm vd, src2, src1	
0	0	1	1	vmand.mm vd, src1, src1	vmmv.m vd, src1
1	0	1	1	vmornot.mm vd, src1, src2	
1	1	1	1	vmxnor.mm vd, vd, vd	vmset.m vd

Note: The vector mask logical instructions are designed to be easily fused with a following masked vector operation to effectively expand the number of predicate registers by moving values into `v0` before use.

16.2. Vector mask population count vmpopc

```
vmpopc.m      rd, vs2, vm
```

The source operand is a single vector register holding mask register values as described in Section Mask Register Layout.

The `vmpopc.m` instruction counts the number of mask elements of the active elements of the vector source mask register that have their least-significant bit set, and writes the result to a scalar `x` register.

The operation can be performed under a mask, in which case only the masked elements are counted.

```
vmpopc.m      rd, vs2, v0.t # x[rd] = sum_i ( vs2[i].LSB && v0[i].LSB )
```

Traps on `vmpopc.m` are always reported with a `vstart` of 0. The `vmpopc` instruction will raise an illegal instruction exception if `vstart` is non-zero.

16.3. `vmfirst` find-first-set mask bit

```
vmfirst.m     rd, vs2, vm
```

The `vmfirst` instruction finds the lowest-numbered active element of the source mask vector that has its LSB set and writes that element's index to a GPR. If no element has an LSB set, -1 is written to the GPR.

Note: Software can assume that any negative value (highest bit set) corresponds to no element found, as vector lengths will never exceed 2^{31} on any implementation.

Traps on `vmfirst` are always reported with a `vstart` of 0. The `vmfirst` instruction will raise an illegal instruction exception if `vstart` is non-zero.

16.4. `vmsbf.m` set-before-first mask bit

```
    vmsbf.m vd, vs2, vm

 # Example

     7 6 5 4 3 2 1 0   Element number

     1 0 0 1 0 1 0 0   v3 contents
                       vmsbf.m v2, v3
     0 0 0 0 0 0 1 1   v2 contents

     1 0 0 1 0 1 0 1   v3 contents
                       vmsbf.m v2, v3
     0 0 0 0 0 0 0 0   v2

     0 0 0 0 0 0 0 0   v3 contents
                       vmsbf.m v2, v3
     1 1 1 1 1 1 1 1   v2

     1 1 0 0 0 0 1 1   v0 vcontents
     1 0 0 1 0 1 0 0   v3 contents
                       vmsbf.m v2, v3, v0.t
     0 1 x x x x 1 1   v2 contents
```

The `vmsbf.m` instruction takes a mask register as input and writes results to a mask register. The instruction writes a 1 to all active mask elements before the first source element that has a set LSB, then writes a zero to that element and all following active elements. If there is no set bit in the source vector, then all active elements in the destination are written with a 1.

The tail elements in the destination mask register are cleared.

16.5. `vmsif.m` set-including-first mask bit

The vector mask set-including-first instruction is similar to set-before-first, except it also includes the element with a set bit.

```
    vmsif.m vd, vs2, vm

 # Example

     7 6 5 4 3 2 1 0   Element number

     1 0 0 1 0 1 0 0   v3 contents
                       vmsif.m v2, v3
     0 0 0 0 0 1 1 1   v2 contents

     1 0 0 1 0 1 0 1   v3 contents
                       vmsif.m v2, v3
     0 0 0 0 0 0 0 1   v2

     1 1 0 0 0 0 1 1   v0 vcontents
     1 0 0 1 0 1 0 0   v3 contents
                       vmsif.m v2, v3, v0.t
     1 1 x x x x 1 1   v2 contents
```

The tail elements in the destination mask register are cleared.

16.6. `vmsof.m` set-only-first mask bit

The vector mask set-only-first instruction is similar to set-before-first, except it only sets the first element with a bit set, if any.

```
    vmsof.m vd, vs2, vm

 # Example

     7 6 5 4 3 2 1 0   Element number

     1 0 0 1 0 1 0 0   v3 contents
                       vmsof.m v2, v3
     0 0 0 0 0 1 0 0   v2 contents

     1 0 0 1 0 1 0 1   v3 contents
                       vmsof.m v2, v3
     0 0 0 0 0 0 0 1   v2

     1 1 0 0 0 0 1 1   v0 vcontents
     1 1 0 1 0 1 0 0   v3 contents
                       vmsof.m v2, v3, v0.t
     0 1 x x x x 0 0   v2 contents
```

The tail elements in the destination mask register are cleared.

16.7. Example using vector mask instructions

The following is an example of vectorizing a data-dependent exit loop.

```
# char* strcpy(char *dst, const char* src)
strcpy:
      mv      a2, a0          # Copy dst
loop:
    vsetvli   x0, x0, e8      # Max length vectors of bytes
    vlbuff.v  v1, (a1)        # Get src bytes
      csrr    t1, vl          # Get number of bytes fetched
    vseq.vi   v0, v1, 0       # Flag zero bytes
    vmfirst.m a3, v0          # Zero found?
      add a1, a1, t1          # Bump pointer
    vmsif.m   v0, v0          # Set mask up to and including zero byte.
    vsb.v     v1, (a2), v0.t  # Write out bytes
      add     a2, a2, t1      # Bump pointer
      bltz    a3, loop        # Zero byte not found, so loop

      ret

# char* strncpy(char *dst, const char* src, size_t n)
strncpy:
      mv      a3, a0          # Copy dst
loop:
    vsetvli   x0, a2, e8      # Vectors of bytes.
    vlbuff.v  v1, (a1)        # Get src bytes
    vseq.vi   v0, v1, 0       # Flag zero bytes
    vmfirst.m a4, v0          # Zero found?
    vmsif.m   v0, v0          # Set mask up to and including zero byte.
    vsb.v     v1, (a3), v0.t  # Write out bytes
      bgez    a4, exit        # Done
      csrr    t1, vl          # Get number of bytes fetched
      add     a1, a1, t1      # Bump pointer
      sub     a2, a2, t1      # Decrement count.
      add     a3, a3, t1      # Bump pointer
      bnez    a2, loop        # Anymore?

exit:
      ret
```

16.8. Vector Iota Instruction

The `vmiota.m` instruction reads a source vector mask register and writes to each element of the destination vector register group the sum of all the least-significant bits of elements in the mask register whose index is less than the element, e.g., a parallel prefix sum of the mask values.

This instruction can be masked, in which case only the enabled elements contribute to the sum and only the enabled elements are written.

```
vmiota.m vd, vs2, vm

# Example

    7 6 5 4 3 2 1 0   Element number

    1 0 0 1 0 0 0 1   v2 contents
                      vmiota.m v4, v2 # Unmasked
    2 2 2 1 1 1 1 0   v4 result

    1 1 1 0 1 0 1 1   v0 contents
    1 0 0 1 0 0 0 1   v2 contents
    2 3 4 5 6 7 8 9   v4 contents
                      vmiota.m v4, v2, v0.t # Masked
    1 1 1 5 1 7 1 0   v4 results
```

The result value is zero-extended to fill the destination element if SEW is wider than the result. If the result value would overflow the destination SEW, the least-significant SEW bits are retained.

Traps on `vmiota.m` are always reported with a `vstart` of 0, and execution is always restarted from the beginingg when resuming after a trap handler. An illegal instruction exception is raised if `vstart` is non-zero.

An illegal instruction exception is raised if the destination vector register group overlaps the source vector mask register. If the instruction is masked, an illegal instruction exception is issued if the destination vector register group overlaps `v0`.

Note: These constraints exist for two reasons. First, to simplify avoidance of WAR hazards in implementatons with temporally long vector registers and no vector register renaming. Second, to enable resuming execution after a trap simpler.

The `vmiota.m` instruction can be combined with memory scatter instructions (indexed stores) to perform vector compress functions.

```
// Compact non-zero elements from input memory array to output memory array
size_t compact_non_zero(size_t n, const int* in, int* out)
{
  size_t i;
  size_t count = 0;
  int *p = out;

  for (i=0; i<n; i++)
  {
    const int v = *in++;
    if (v != 0)
      *p++ = v;
  }

  return (size_t) (p - out);
}
```

```
# a0 = n
# a1 = &in
# a2 = &out

compact_non_zero:
    li        a6, 0                # Clear count of non-zero elements
loop:
    vsetvli   a5, a0, e32,m8       # 32-bit integers
    vlw.v     v8, (a1)             # Load input vector
      sub     a0, a0, a5           # Decrement number done
      slli    a5, a5, 2            # Multiply by four bytes
    vsne.vi   v0, v8, 0            # Locate non-zero values
      add     a1, a1, a5           # Bump input pointer
    vmpopc.m  a5, v0               # Count number of elements set in v0
    vmiota.m  v16, v0              # Get destination offsets of active elements
      add     a6, a6, a5           # Accumulate number of elements
    vsll.vi   v16, v16, 2, v0.t    # Multiply offsets by four bytes
      slli    a5, a5, 2            # Multiply number of non-zero elements by four bytes
    vsuxw.v   v8, (a2), v16, v0.t # Scatter using scaled vmiota results under mask
      add     a2, a2, a5           # Bump output pointer
      bnez    a0, loop             # Any more?

      mv      a0, a6               # Return count
      ret
```

16.9. Vector Element Index Instruction

The `vid.v` instruction writes each element's index to the destination vector register group, from 0 to `vl-1`.

```
vid.v        vd, vm  # Write element ID to destination.
```

The instruction can be masked.

An illegal instruction exception is generated if the destination vector register group overlaps the mask register and LMUL > 1.

Note: This constraint is to avoid WAR hazards in long vector implementations without register renaming, and to support restart.

Note: Microarchitectures can implement `vid.v` instruction using the same datapath as `vmiota.m` but with an implicit set mask source.

17. Vector Permutation Instructions

A range of permutation instructions are provided to move elements around within the vector registers.

17.1. Integer Extract Instruction

The integer extract operation transfers a single value between one element of a vector register and a GPR. This instruction ignores LMUL and vector register groups.

```
vext.x.v      rd, vs2, rs1  # rd = vs2[rs1]
```

The integer extract operation, `vext.x.v` reads one SEW-width element from a vector register group at the element index and writes it to GPR destination register rd. The GPR `rs1` register gives the element index, treated as an unsigned integer. If the index is out of range (i.e., x[rs1] ≥ VLEN/SEW), then zero is returned for the element value. If SEW > XLEN, the least-significant bits are copied to the destination and the upper SEW-XLEN bits are ignored. If SEW < XLEN, the value is zero-extended to XLEN.

An assembler pseudoinstruction `vmv.x.s rd, vs2` expanding to `vext.x.v rd, vs2, x0` is provided as a complement to the `vmv.s.x` instruction below.

17.2. Integer Scalar Move Instruction

The integer scalar move instruction transfers a single value from a scalar `x` register to element 0 of a vector register. The instructions ignore LMUL and vector register groups.

Note: In the base vector extension, this instruction can be used to initialize the input of a reduction instruction.

Note: Using scalar move instructions to access element 0 of other than the base register in a vector register group can expose differences in element layout between different RISC-V vector extension implementations.

```
vmv.s.x      vd, rs1  # vd[0] = rs1
```

The `vmv.s.x` instruction copies the scalar integer register to element 0 of the destination vector register. If SEW < XLEN, the least-significant bits are copied and the upper XLEN-SEW bits are ignored. If SEW > XLEN, the value is zero-extended to SEW bits.

The other elements in the destination vector register (0 < index < VLEN/SEW) are zeroed.

If `vl`=0, no operation is performed and the destination register is not updated.

Note: The complementary `vins.v.x` instruction, which allows a write to any element in a vector register, has been removed. This instruction would be the only instruction (apart from `vsetvl`) that requires two integer source operands, and also would be slow to execute in an implementation with vector register renaming, relegating its main use to debugger modifications to state. The alternative and more generally useful `vslide1up` and `vslide1down` instructions can be used to update vector register state in place over a debug link without accessing memory.

17.3. Floating-Point Scalar Move Instructions

The floating-point scalar read/write instructions transfer a single value between a scalar `f` register and element 0 of a vector register. The instructions ignore LMUL and vector register groups.

```
vfmv.f.s      rd, vs2  # rd = vs2[0] (rs1=0)
vfmv.s.f      vd, rs1  # vd[0] = rs1 (vs2=0)
```

The `vfmv.f.s` instruction copies a single SEW-wide element from index 0 of the source vector register to a destination scalar floating-point register. If SEW > FLEN, the least-significant FLEN bits are transferred and the upper SEW-FLEN bits are ignored. If SEW < FLEN, the value is NaN-boxed (1-extended) to FLEN bits.

The `vfmv.s.f` instruction copies the scalar register to element 0 of the destination vector register. If SEW < FLEN, the least-significant bits are copied and the upper XLEN-SEW bits are ignored. If SEW > XLEN, the value is NaN-boxed (1-extended) to SEW bits. The other elements in the destination vector register (0 < index < VLEN/SEW) are zeroed. If `vl`=0, no operation is performed and the destination register is not updated.

17.4. Vector Slide Instructions

The slide instructions move elements up and down a vector register group.

> Note: The slide operations can be implemented much more efficiently than using the arbitrary register gather instruction. Implementations may optimize certain OFFSET values for `vslideup` and `vslidedown`. In particular, power-of-2 offsets may operate substantially faster than other offsets.

For all of the `vslideup`, `vslidedown`, `vslide1up`, and `vslide1down` instructions, if `vl`=0, the instruction performs no operation and leaves the destination vector register unchanged.

17.4.1. Vector Slideup Instructions

```
vslideup.vx   vd, vs2, rs1, vm        # vd[i+rs1] = vs2[i]
vslideup.vi   vd, vs2, uimm[4:0], vm  # vd[i+imm] = vs2[i]
```

For `vslideup`, the value in `vl` specifies the number of destination elements that are written. The start index (*OFFSET*) for the destination can be either specified using an unsigned integer in the `x` register specified by `rs1`, or a 5-bit immediate treated as an unsigned 5-bit quantity.

```
   vslideup behavior for destination elements

   OFFSET is amount to slideup, either from x register or a 5-bit immediate

                    0 <  i < max(vstart, OFFSET)  Unchanged
  max(vstart, OFFSET) <= i < vl                   vd[i] = vs2[i-OFFSET] if mask enabled,
                                                   unchanged if not
                   vl <= i < VLMAX                Tail elements, vd[i] = 0
```

The destination vector register group for `vslideup` cannot overlap the vector register group of the source, and if operation is masked cannot overlap the vector mask register, otherwise an illegal instruction exception is raised.

> Note: The non-overlap constraints are to avoid WAR hazards on the input vectors during execution, and to enable restart with non-zero `vstart`.

17.4.2. Vector Slidedown Instructions

```
vslidedown.vx vd, vs2, rs1, vm       # vd[i] = vs2[i+rs1]
vslidedown.vi vd, vs2, uimm[4:0], vm # vd[i] = vs2[i+imm]
```

For `vslidedown`, the value in `vl` specifies the number of destination elements that are written.

The start index (*OFFSET*) for the source can be either specified using an unsigned integer in the `x` register specified by `rs1`, or a 5-bit immediate treated as an unsigned 5-bit quantity.

```
    vslidedown behavior for source elements for element i in slide
                     0 <= i+OFFSET < VLMAX   Read vs2[i+offset]
                 VLMAX <= i+OFFSET           Read as 0

    vslidedown behavior for destination element i in slide
                     0 <  i < vstart         Unchanged
                vstart <= i < vl             Updated if mask enabled, unchanged if not
                    vl <= i < VLMAX          Zeroed
```

Note: Microarchitectures can optimize zeros written to the end of a vector for large offsets by treating as effectively smaller vector length, and encoding using the same internal scheme as for regular vector instruction writes.

The destination vector register group for `vslidedown` cannot overlap the vector mask register if the instruction is masked, otherwise an illegal instruction exception is raised.

17.4.3. Vector Slide1up

Variants of slide are provided that only move by one element but which also allow a scalar integer value to be inserted at the vacated element position.

```
vslide1up.vx  vd, vs2, rs1, vm        # vd[0]=x[rs1], vd[i+1] = vs2[i]
```

The `vslide1up` instruction places the `x` register argument at location 0 of the destination vector register group, provided that element 0 is active, otherwise the destination element is unchanged. If XLEN < SEW, the value is zero-extended to SEW bits. If XLEN > SEW, the least-significant bits are copied over and the high SEW-XLEN bits are ignored.

The remaining active `vl-1` elements are copied over from index *i* in the source vector register group to index *i*+1 in the destination vector register group.

The `vl` register specifies how many of the destination vector register elements are written with source values, and all tail elements are zeroed.

```
   vslide1up behavior

                    i < vstart  unchanged
                0 = i = vstart  vd[i] = x[rs1] if mask enabled, unchanged if not
  max(vstart, 1) <= i < vl      vd[i] = vs2[i-1] if mask enabled, unchanged if not
              vl <= i < VLMAX   tail elements, vd[i] = 0
```

The `vslide1up` instruction requires that the destination vector register group does not overlap the source vector register group and the mask register if masked. Otherwise, an illegal instruction exception is raised.

17.4.4. Vector Slide1down Instruction

The `vslide1down` instruction copies the first `vl-1` active elements values from index *i*+1 in the source vector register group to index *i* in the destination vector register group.

The `vl` register specifies how many of the destination vector register elements are written with source values, and all tail elements are zeroed.

```
vslide1down.vx vd, vs2, rs1, vm      # vd[i] = vs2[i+1], vd[vl-1]=x[rs1]
```

The `vslide1down` instruction places the `x` register argument at location `vl-1` in the destination vector register, provided that element `vl-1` is active, otherwise the destination element is unchanged. If XLEN < SEW, the value is zero-extended to SEW bits. If XLEN > SEW, the least-significant bits are copied over and the high SEW-XLEN bits are ignored.

```
  vslide1down behavior

                    i < vstart  unchanged
          vstart <= i < vl-1    vd[i] = vs2[i+1] if mask enabled, unchanged if not
          vstart <= i = vl-1    vd[vl-1] = x[rs1] if mask enabled, unchanged if not
              vl <= i < VLMAX   tail elements, vd[i] = 0
```

The `vslide1down` instruction requires that the destination vector register group does not overlap the mask register if masked. Otherwise, an illegal instruction exception is raised.

Note: The `vslide1down` instruction can be used to load values into a vector register without using memory and without disturbing other vector registers. This provides a path for debuggers to modify the contents of a vector register, albeit slowly, with multiple repeated `vslide1down` invocations.

17.5. Vector Register Gather Instruction

The vector register gather instruction reads elements from a first source vector register group at locations given by a second source vector register group. The index values in the second vector are treated as unsigned integers. The source vector can be read at any index < VLMAX regardless of `vl`. The number of elements to write to the destination register is given by `vl`, and elements past `vl` in the destination are zeroed. The operation can be masked.

```
vrgather.vv   vd, vs2, vs1, vm # vd[i] = (vs1[i] >= VLMAX) ? 0 : vs2[vs1[i]];
```

If the element indices are out of range (`vs1[i]` ≥ VLMAX) then zero is returned for the element value.

Vector-scalar and vector-immediate forms of the register gather are also provided. These read one element from the source vector at the given index, and write this value to the `vl` elements at the start of the destination vector register.

Note: These forms allow any vector element to be "splatted" to an entire vector.

```
vrgather.vx   vd, vs2, rs1, vm # vd[i] = vs2[rs1]
vrgather.vi   vd, vs2, imm, vm # vd[i] = vs2[imm]
```

For any `vrgather` instruction, the destination vector register group cannot overlap with the source vector register groups, including the mask register if the operation is masked, otherwise an illegal instruction exception is raised.

17.6. Vector Compress Instruction

The vector compress instruction allows elements selected by a vector mask register from a source vector register group to be packed into contiguous elements at the start of the destination vector register group.

```
vcompress.vm  vd, vs2, vs1  # Compress into vd elements of vs2 where vs1 is enabled
```

The vector mask register specified by `vs1` indicates which of the first `vl` elements of vector register group `vs2` should be extracted and packed into contiguous elements at the beginning of vector register `vd`. Any remaining elements of `vd` are zeroed.

```
    Example use of vcompress instruction

        1 1 0 1 0 0 1 0 1   v0
        8 7 6 5 4 3 2 1 0   v1
                                vcompress.vm v2, v1, v0
        0 0 0 0 8 7 5 2 0   v2
```

The destination vector register group cannot overlap the source vector register group or the source vector mask register, otherwise an illegal instruction exception is raised.

A trap on a `vcompress` instruction is always reported with a `vstart` of 0. Executing a `vcompress` instruction with a non-zero `vstart` raises an illegal instruction exception.

Note: Although possible, `vcompress` is one of the more difficult instructions to restart with a non-zero `vstart`, so assumption is implementations will choose not do that but will instead restart from element 0. This does mean elements in destination register after `vstart` will already have been updated.

18. Exception Handling

On a trap during a vector instruction (caused by either a synchronous exception or an asynchronous interrupt), the existing `*epc` CSR is written with a pointer to the errant vector instruction, while the `vstart` CSR contains the element index that caused the trap to be taken.

> Note: We chose to add a `vstart` CSR to allow resumption of a partially executed vector instruction to reduce interrupt latencies and to simplify forward-progress guarantees. This is similar to the scheme in the IBM 3090 vector facility. To ensure forward progress without the `vstart` CSR, implementations would have to guarantee an entire vector instruction can always complete atomically without generating a trap. This is particularly difficult to ensure in the presence of strided or scatter/gather operations and demand-paged virtual memory.

18.1. Precise vector traps

Precise vector traps require that:

1. all instructions older than the trapping vector instruction have committed their results
2. no instructions newer than the trapping vector instruction have altered architectural state
3. any operations within the trapping vector instruction affecting result elements preceding the index in the `vstart` CSR have committed their results
4. no operations within the trapping vector instruction affecting elements at or following the `vstart` CSR have altered architectural state except if restarting and completing the affected vector instruction will recover the correct state.

We relax the last requirement to allow elements following `vstart` to have been updated at the time the trap is reported, provided that re-executing the instruction from the given `vstart` will correctly overwrite those elements.

> Note: We assume most supervisor-mode environments will require precise vector traps.

Except where noted above, vector instructions are allowed to overwrite their inputs, and so in most cases, the vector instruction restart must be from the `vstart` location. However, there are a number of cases where this overwrite is prohibited to enable execution of the the vector instructions to be idempotent and hence restartable from any location.

18.2. Imprecise vector traps

Imprecise vector traps are traps that are not precise. In particular, instructions newer than `*epc` may have committed results, and instructions older than `*epc` may have not completed execution. Imprecise traps are primarily intended to be used in situations where reporting an error and terminating execution is the appropriate response.

> Note: A platform might specify that interrupts are precise while other traps are imprecise. We assume many embedded platforms will only generate imprecise traps for vector instructions on fatal errors, so do not require resumable traps.

18.3. Selectable precise/imprecise traps

Some platforms may choose to provide a privileged mode bit to select between precise and imprecise vector traps. Imprecise mode would run at high-performance but possibly make it difficult to discern error causes, while precise mode would run more slowly, but support debugging of errors albeit with a possibility of not experiencing the same errors as in imprecise mode.

18.4. Swappable traps

Another trap mode can support swappable state in the vector unit, where on a trap, special instructions can save and restore the vector unit microarchitectural state, to allow execution to continue correctly around imprecise traps.

This mechanism is not defined in the base vector ISA.

19. Divided Element Extension ('Zvediv')

The divided element extension allows each element to be treated as a packed sub-vector of narrower elements. This provides efficient support for some forms of narrow-width and mixed-width arithmetic, and also to allow outer-loop vectorization of short vector and matrix operations. In addition to modifying the behavior of some existing instructions, a few new instructions are provided to operate on vectors when EDIV > 1.

Note: This is written as an extension for now, but could become part of mandatory base in Unix vector profile.

The divided element extension adds a two-bit field, `vediv[1:0]` to the `vtype` register.

vtype-format.adoc

The `vediv` field encodes the number of ways, *EDIV*, into which each SEW-bit element is subdivided into equal sub-elements. A vector register group is now considered to hold a vector of sub-vectors.

vediv [1:0]			**Division EDIV**
0	0	1	(undivided, as in base)
0	1	2	two equal sub-elements
1	0	4	four equal sub-elements
1	1	8	eight equal sub-elements

SEW	EDIV	Sub-element	Integer sum accumulator	Integer dot accumulator	FP sum/dot accumulator FLEN=32	FP sum/dot accumulator FLEN=64	FP sum/dot accumulator FLEN=128
8b	2	4b	8b	8b	-	-	-
8b	4	2b	8b	8b	-	-	-
8b	8	1b	8b	8b	-	-	-
16b	2	8b	16b	16b	-	-	-
16b	4	4b	8b	16b	-	-	-
16b	8	2b	8b	8b	-	-	-
32b	2	16b	32b	32b	32b	32b	32b
32b	4	8b	16b	32b	-	-	-
32b	8	4b	8b	16b	-	-	-
64b	2	32b	64b	64b	32b	64b	64b
64b	4	16b	32b	64b	32b	32b	32b
64b	8	8b	16b	32b	-	-	-
128b	2	64b	128b	128b	32b	64b	128b
128b	4	32b	64b	128b	32b	64b	64b
128b	8	16b	32b	64b	32b	32b	32b
256b	2	128b	256b	256b	32b	64b	128b
256b	4	64b	128b	256b	32b	64b	128b
256b	8	32b	64b	128b	32b	64b	64b

Each implementation defines a minimum size for a sub-element, *SELEN*, which must be at most 8 bits.

Note: While *SELEN* is a fourth implementation-specific parameter, values smaller than 8 would be considered an additional extension.

19.1. Instructions not affected by EDIV

The vector start register `vstart` and exception reporting continue to work as before.

The vector length `vl` control and vector masking continue to operate at the element level.

Vector masking continues to operate at the element level, so sub-elements cannot be individually masked.

Note: SEW can be changed dynamically to enabled per-element masking for sub-elements of 8 bits and greater.

Vector load/store and AMO instructions are unaffected by EDIV, and continue to move whole elements.

Vector mask logical operations are unchanged by EDIV setting, and continue to operate on vector registers containing element masks.

Vector mask population count (`vmpopc`), find-first and related instructions (`vmfirst`, `vmsbf`, `vmsif`, `vmsof`), iota (`vmiota`), and element index (`vid`) instructions are unaffected by EDIV.

Vector integer bit insert/extract, and integer and floating-point scalar move instruction are unaffected by EDIV.

Vector slide-up/slide-down are unaffected by EDIV.

Vector compress instructions are unaffected by EDIV.

19.2. Instructions Affected by EDIV

19.2.1. Regular Vector Arithmetic Instructions under EDIV

Most vector arithmetic operations are modified to operate on the individual sub-elements, so effective SEW is SEW/EDIV and effective vector length is `vl` * EDIV. For example, a vector add of 32-bit elements with a `vl` of 5 and EDIV of 4, operates identically to a vector add of 8-bit elements with a vector length of 20.

```
vsetvli t0, a0, e32,m1,d4  # Vectors of 32-bit elements, divided into byte sub-elements
vadd.vv v1,v2,v3           # Performs a vector of 4*vl 8-bit additions.
vsll.vx v1,v2,x1           # Performs a vector of 4*vl 8-bit shifts.
```

19.2.2. Vector Reduction Instructions under EDIV

Vector reduction instructions now operate independently on all elements in a vector, reducing sub-element values within an element to an element-wide result.

```
# Sum each sub-vector of four bytes into a 16-bit result.
vsetvli    t0, a0, e32,d4  # Vectors of 32-bit elements, divided into byte sub-elements
vredsum.vv v1, v2, v3      # v1[i][15:0] = v2[i][31:24] + v2[i][23:16]
                           #             + v2[i][15:8] + v2[i][7:0] + v3[i][31:0]

vredmax.vv v5, v6, v7      # v5[i][31:0] = max(v6[i][31:24], v6[i][23:16],
                           #             v6[i][15:8], v6[i][7:0], v7[i][31:0])
```

Integer sub-element non-sum reductions produce a final result that is max(8,SEW/EDIV) bits wide, sign- or zero-extended to full SEW if necessary.

Integer sub-element sum reductions produce a final result that is max(8,min(SEW,2*SEW/EDIV)) bits wide, sign- or zero-extended to full SEW if necessary.

Floating-point sub-element non-sum reductions produce a final result that is SEW/EDIV bits wide.

Floating-point sub-element sum reductions produce a final result that is min(2*SEW/EDIV,FLEN) bits wide, NaN-boxed to the full SEW width if necessary.

Widening vector reduction operations with non-zero EDIV are reserved. NOTE: While these could be defined, it is unclear they are needed and this reduces implementation complexity.

19.2.3. Vector Register Gather Instructions under EDIV

Vector register gather instructions under non-zero EDIV only gather sub-elements within the element. The source and index values are interpreted as relative to the enclosing element only. Index values ≥ EDIV write a zero value into the result sub-element.

```
       |       |       |  SEW = 32b, EDIV=4
        7 6 5 4 3 2 1 0  bytes
        d e a d b e e f  v1
        0 1 9 2 0 2 3 2  v2
                            vrgather.vv v3, v1, v2
        d a 0 e f e b e  v3
                            vrgather.vi v4, v1, 1
        a a a a e e e e  v4
```

Note Note

Vector register gathers with scalar or immediate arguments can "splat" values across sub-elements within an element.

Implementations can provide fast implementations of register gathers constrained within a single element width.

19.3. Vector Integer Dot-Product Instruction

The integer dot-product reduction `vdot.vv` performs an element-wise multiplication between the source sub-elements then accumulates the results into the destination vector element. Note the assembler syntax uses a `.vv` suffix since both inputs are vectors of elements.

Sub-element integer dot reductions produce a final result that is max(8,min(SEW,4*SEW/EDIV)) bits wide, sign- or zero-extended to full SEW if necessary.

```
# Unsigned dot-product
vdotu.vv      vd, vs2, vs1, vm  # Vector-vector

# Signed dot-product
vdot.vv       vd, vs2, vs1, vm   # Vector-vector
```

```
# Dot product, SEW=32, EDIV=1
vdot.vv       vd, vs2, vs1, vm  # vd[i][31:0] += vs2[i][31:0] * vs1[i][31:0]

# Dot product, SEW=32, EDIV=2
vdot.vv       vd, vs2, vs1, vm  # vd[i][31:0] += vs2[i][31:16] * vs1[i][31:16]
                                #              + vs2[i][15:0] * vs1[i][15:0]

# Dot product, SEW=32, EDIV=4
vdot.vv       vd, vs2, vs1, vm  # vd[i][31:0] += vs2[i][31:24] * vs1[i][31:24]
                                #              + vs2[i][23:16] * vs1[i][23:16]
                                #              + vs2[i][15:8] * vs1[i][15:8]
                                #              + vs2[i][7:0] * vs1[i][7:0]
```

19.4. Vector Floating-Point Dot Product Instruction

The floating-point dot-product reduction `vfdot.vv` performs an element-wise multiplication between the source sub-elements then accumulates the results into the destination vector element. Note the assembler syntax uses a `.vv` suffix since both inputs are vectors of elements.

Signed dot-product

```
vfdot.vv      vd, vs2, vs1, vm  # Vector-vector
```

```
# Dot product. SEW=32, EDIV=2
vfdot.v   vd, vs2, vs1, vm  # vd[i][31:0] += vs2[i][31:16] * vs1[i][31:16]
                            #             +  vs2[i][15:0] * vs1[i][15:0]

# Floating-point sub-vectors of two half-precision floats packed into 32-bit elements.
vsetvli   t0, a0, e32,m1,d2 # Vectors of 32-bit elements, divided into 16b sub-elements
vfdot.vv  v1, v2, v3        # v1[i][31:0] += v2[i][31:16]*v3[i][31:16]
                            #             +  v2[i][16:0]*v3[i][16:0]

# Floating-point sub-vectors of two half-precision floats packed into 32-bit elements.
vsetvli   t0, a0, e32,m1,d2 # Vectors of 32-bit elements, divided into 16b sub-elements
vfdot.vv  v1, v2, v3        # v1[i][31:0] += v2[i][31:16]*v3[i][31:16]
                            #             +  v2[i][16:0]*v3[i][16:0]

# Floating-point sub-vectors of four half-precision floats packed into 64-bit elements.
vsetvli   t0, a0, e64,m1,d4 # Vectors of 64-bit elements, divided into 16b sub-elements
vfdot.vv  v1, v2, v3
              # v1[i][31:0] +=  v2[i][31:16]*v3[i][31:16] + v2[i][16:0]*v3[i][16:0] +
              #                 v2[i][63:48]*v3[i][63:48] + v2[i][47:32]*v3[i][47:32];
              # v1[i][63:32] = ~0 (NaN boxing)
```

20. Vector Instruction Listing

inst-table.adoc

vector-examples.adoc