RISC-V "V" Vector Extension

Version 0.9

Table of Contents

Contributors include: Alon Amid, Krste Asanovic, Allen Baum, Alex Bradbury, Tony Brewer, Chris Celio, Aliaksei Chapyzhenka, Silviu Chiricescu, Ken Dockser, Bob Dreyer, Roger Espasa, Sean Halle, John Hauser, David Horner, Bruce Hoult, Bill Huffman, Constantine Korikov, Ben Korpan, Hanna Kruppe, Yunsup Lee, Guy Lemieux, Filip Moc, 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 vector extension.

> 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 0.9 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.

This draft spec is intended to capture how a certain vector function will be implemented as vector instructions, but is not intended to determine what set of vector instructions are mandatory for a given platform profile.

The term *base vector extension* is used informally to describe the standard set of vector ISA components that will be required for the single-letter "V" extension, which is intended for use in standard server and application-processor platform profiles. 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 will only be determined when each profile spec is ratified.

The document describes all the individual features to be included in the base vector extension, along with drafts of some initial extensions to the base.

> The set of instructions to be included or not in the base "V" extension, and the naming of all the vector instruction subsets and extensions is not yet determined.

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* ≥ 8, 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.

> 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 SLEN, or requiring SLEN=VLEN.

> There is a proposal to allow ELEN to vary with LMUL that would relax the constraint that *VLEN* ≥ *ELEN*.

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

> 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 seven unprivileged CSRs (`vstart`, `vxsat`, `vxrm`, `vcsr`, `vtype`, `vl`, `vlenb`) to a base scalar RISC-V ISA.

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
0x00F	URW	vcsr	Vector control and status register
0xC20	URO	vl	Vector length
0xC21	URO	vtype	Vector data type register
0xC22	URO	vlenb	VLEN/8 (vector register length in bytes)

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.

> Zfinx ("F in X") is a new ISA option under consideration where floating-point instructions take their arguments from the integer register file. The 0.9 vector extension is also compatible with this option.

3.2. Vector Context Status in `mstatus`

A vector context status field, VS, is added to `mstatus[10:9]` and shadowed in `sstatus[10:9]`. It is defined analogously to the floating-point context status field, FS.

Attempts to execute any vector instruction, or to access the vector CSRs, raise an illegal-instruction exception when the VS field is set to Off.

When the VS field is set to Initial or Clean, executing any instruction that changes vector state, including the vector CSRs, will change VS to Dirty.

> Implementations may also change VS field to Dirty at any time, even when there is no change in vector state. Accurate setting of the VS field is an optimization.

3.3. Vector type register, `vtype`

The read-only XLEN-wide *vector type* CSR, `vtype` provides the default type used to interpret the contents of the vector register file, and can only be updated by `vsetvl{i}` instructions. The vector type also determines the organization of elements in each vector register, and how multiple vector registers are grouped.

> Earlier drafts allowed the `vtype` register to be written using regular CSR writes. Allowing updates only via the `vsetvl{i}` instructions simplifies maintenance of the `vtype` register state.

In the base vector extension, the `vtype` register has five fields, `vill`, `vma`, `vta`, `vsew[2:0]`, and `vlmul[2:0]`.

Table 2. `vtype` register layout

Bits	Name	Description
XLEN-1	vill	Illegal value if set
XLEN-2:8		Reserved (write 0)
7	vma	Vector mask agnostic
6	vta	Vector tail agnostic
5	vlmul[2]	Vector register group multiplier (LMUL) setting (fractional)
4:2	vsew[2:0]	Standard element width (SEW) setting
1:0	vlmul[1:0]	Vector register group multiplier (LMUL) setting

> The smallest base implementation supporting ELEN=32 requires storage for only seven bits of storage in `vtype`, two bits for `ma` and `ta`, two bits for `vsew[1:0]` and three bits for `vlmul[2:0]`. The illegal value represented by `vill` can be encoded using the illegal 64-bit combination in `vsew[1:0]` without requiring an additional storage bit.

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

> It is anticipated that an extended 64-bit instruction encoding would allow these fields to be specified statically in the instruction encoding.

3.3.1. Vector standard element width `vsew[2:0]`

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 "V" extension, only SEW up to ELEN = max(XLEN,FLEN) are required to be supported. Other platforms may impose different constraints on ELEN.

Table 3. 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 4. Example VLEN = 128 bits

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

3.3.2. Vector Register Grouping (`vlmul[2:0]`)

Multiple vector registers can be grouped together, so that a single vector instruction can operate on multiple vector registers. The term *vector register group* is used herein to refer to one or more vector registers used as a single operand to a vector instruction. 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 vector length multiplier, *LMUL*, when greater than 1, represents the default number of vector registers that are combined to form a vector register group. LMUL can have integer values 1,2,4,8.

LMUL can also be a fractional value, reducing the number of bits used in a vector register. LMUL can have fractional values 1/2, 1/4, 1/8. Fractional LMUL is used to increase the number of usable architectural registers when operating on mixed-width values, by not requiring that larger-width vectors occupy multiple vector registers. Instead, wider values can occupy a single vector register and narrower values can occupy a fraction of a vector register.

Implementations must support fractional LMUL settings for LMUL ≥ SEW/ELEN, for the ELEN value at LMUL=1. An attempt to set an unsupported SEW and LMUL configuration sets the `vill` bit in `vtype`.

> Requiring LMUL ≥ SEW/ELEN allows software to only use a single vector register to hold the widest (ELEN) elements with fractional LMUL used to hold narrower elements when operating on mixed-width elements. If LMUL < SEW/ELEN there is no guarantee an implementation would have enough bits in the fractional vector register to store at least one element, as VLEN=ELEN is a valid implementation choice.

LMUL is set by the signed `vlmul` field in `vtype` (LMUL = $2^{\texttt{vlmul[2:0]}}$).

The derived value VLMAX = LMUL*VLEN/SEW represents the maximum number of elements that can be operated on with a single vector instruction given the current SEW and LMUL settings as shown in the table below.

vlmul			LMUL	#groups	VLMAX	Registers grouped with register *n*
1	0	0	-	-	-	reserved
1	0	1	1/8	32	VLEN/SEW/8	v *n* (single register in group)
1	1	0	1/4	32	VLEN/SEW/4	v *n* (single register in group)
1	1	1	1/2	32	VLEN/SEW/2	v *n* (single register in group)
0	0	0	1	32	VLEN/SEW	v *n* (single register in group)
0	0	1	2	16	2*VLEN/SEW	v *n*, v *n*+1
0	1	0	4	8	4*VLEN/SEW	v *n*, …, v *n*+3
0	1	1	8	4	8*VLEN/SEW	v *n*, …, v *n*+7

When LMUL=2, the vector register group contains vector register v *n* and vector register v *n*+1, providing twice the vector length in bits. Instructions specifying a vector register group with an odd-numbered vector register will raise an illegal instruction exception.

When LMUL=4, the vector register group contains four vector registers, and instructions specifying vector register groups using vector register numbers that are not multiples of four will raise an illegal instruction exception.

When LMUL=8, the vector register group contains eight vector registers, and instructions specifying vector register groups using register numbers that are not multiples of eight will raise an illegal instruction exception.

Mask registers are always contained in a single vector register, regardless of LMUL.

3.3.3. Vector Tail Agnostic and Vector Mask Agnostic `vta` and `vma`

These two bits modify the behavior of destination tail elements and destination inactive masked-off elements respectively during the execution of vector instructions. The tail and inactive sets contain element positions that are not receiving new results during a vector operation, as defined in Section Prestart, Active, Inactive, Body, and Tail Element Definitions.

All systems must support all four options:

vta	vma	Tail Elements	Inactive Elements
0	0	undisturbed	undisturbed
0	1	undisturbed	agnostic
1	0	agnostic	undisturbed
1	1	agnostic	agnostic

When a set is marked undisturbed, the corresponding set of destination elements in any vector or mask destination operand retain the value they previously held.

When a set is marked agnostic, the corresponding set of destination elements in any vector or mask destination operand can either retain the value they previously held, or are overwritten with 1s. Within a single vector instruction, each destination element can be either left undisturbed or overwritten with 1s, in any combination, and the pattern of undisturbed or overwritten with 1s is not required to be deterministic when the instruction is executed with the same inputs.

> The agnostic policy was added to accomodate machines with vector register renaming, and/or that have deeply temporal vector registers. With an undisturbed policy, all elements would have to be read from the old physical destination vector register to be copied into the new physical destination vector register. This causes an inefficiency when these inactive or tail values are not required for subsequent calculations.

> The intent is for software to select the option that reduces micorarchitectural work by selecting agnostic when the value in the respective set does not matter.

> The value of all 1s instead of all 0s was chosen for the overwrite value to discourage software developers from depending on the value written.

> A simple in-order implementation can ignore the setting and simply execute all vector instructions using the undisturbed policy. The `vta` and `vma` state bits must still be provided in `vtype` for compatibilty and to support thread migration.

> An out-of-order implementation can choose to implement tail-agnostic + mask-agnostic using tail-agnostic + mask-undisturbed to reduce implementation complexity.

> The definition of agnostic result policy is left loose to accomodate migrating application threads between harts on a small in-order core (which probably leaves agnostic regions undisturbed) and harts on a larger out-of-order core with register renaming (which probably overwrites agnostic elements with 1s). As it might be necessary to restart in the middle, we allow arbitrary mixing of agnostic policies within a single vector instruction. This allowed mixing of policies also enables implementations that might change policies for different granules of a vector register, for example, using undisturbed within a granule that is actively operated on but renaming to all 1s for granules in the tail.

The assembly syntax adds two flags to the `vsetvli` instruction:

```
ta   # Tail agnostic
tu   # Tail undisturbed
ma   # Mask agnostic
mu   # Mask undisturbed

vsetvli t0, a0, e32,m4,ta,ma   # Tail agnostic, mask agnostic
vsetvli t0, a0, e32,m4,tu,ma   # Tail undisturbed, mask agnostic
vsetvli t0, a0, e32,m4,ta,mu   # Tail agnostic, mask undisturbed
vsetvli t0, a0, e32,m4,tu,mu   # Tail undisturbed, mask undisturbed
```

3.3.4. Vector Type Illegal `vill`

The `vill` bit is used to encode that a previous `vsetvl{i}` instruction attempted to write an unsupported value to `vtype`.

> The `vill` bit is held in bit XLEN-1 of the CSR to support checking for illegal values with a branch on the sign bit.

If the `vill` bit is set, then any attempt to execute a vector instruction that depends upon `vtype` will raise an illegal-instruction exception.

> `vsetvl{i}` and whole-register loads, stores, and moves do not depend upon `vtype`.

When the `vill` bit is set, the other XLEN-1 bits in `vtype` shall be zero.

3.4. 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 any destination vector register group with indices ≥ `vl` are unmodified during execution of a vector instruction. When `vstart` ≥ `vl`, no elements are updated in any destination vector register group.

> As a consequence, when `vl`=0, no elements are updated in the destination vector register group, regardless of `vstart`.

> Instructions that write a scalar integer or floating-point register do so even when `vstart` ≥ `vl`.

> 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).

3.5. Vector Byte Length `vlenb`

The *XLEN*-bit-wide read-only CSR `vlenb` holds the value VLEN/8, i.e., the vector register length in bytes.

> The value in `vlenb` is a design-time constant in any implementation.

> Without this CSR, several instructions are needed to calculate VLEN in bytes, and the code has to disturb current `vl` and `vtype` settings which require them to be saved and restored.

3.6. 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.

> All vector instructions, including `vsetvl{i}`, reset the `vstart` CSR to zero.

`vstart` is not modified by vector instructions that raise illegal-instruction exceptions.

For instructions where the number of elements to be performed is set by `vl`, if the value in the `vstart` register is greater than or equal to the vector length `vl` then no element operations are performed. The `vstart` register is then reset to zero.

The `vstart` CSR is defined to have only enough writable 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).

> 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 cannot be executed with a non-zero `vstart` value and will raise an illegal instruction exception as defined below.

> Making `vstart` visible to unprivileged code supports user-level threading libraries.

Implementations are permitted to raise illegal instruction exceptions when attempting to execute a vector instruction with a value of `vstart` that the implementation can never produce when executing that same instruction with the same `vtype` setting.

> For example, some implementations will never take interrupts during execution of a vector arithmetic instruction, instead waiting until the instruction completes to take the interrupt. Such implementations are permitted to raise an illegal instruction exception when attempting to execute a vector arithmetic instruction when `vstart` is nonzero.

3.7. 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 `vcsr`.

The fixed-point rounding algorithm is specified as follows. Suppose the pre-rounding result is `v`, and `d` bits of that result are to be rounded off. Then the rounded result is `(v >> d) + r`, where `r` depends on the rounding mode as specified in the following table.

Table 5. vxrm encoding

Bits [1:0]		Abbreviation	Rounding Mode	Rounding increment, `r`
0	0	rnu	round-to-nearest-up (add +0.5 LSB)	`v[d-1]`
0	1	rne	round-to-nearest-even	`v[d-1] & (v[d-2:0]≠0 | v[d])`
1	0	rdn	round-down (truncate)	`0`
1	1	rod	round-to-odd (OR bits into LSB, aka "jam")	`!v[d] & v[d-1:0]≠0`

The rounding functions:

```
roundoff_unsigned(v, d) = (unsigned(v) >> d) + r
roundoff_signed(v, d) = (signed(v) >> d) + r
```

are used to represent this operation in the instruction descriptions below.

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

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

3.8. 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 `vcsr`.

3.9. Vector Control and Status Register `vcsr`

The `vxrm` and `vxsat` separate CSRs can also be accessed via fields in the vector control and status CSR, `vcsr`.

Table 6. vcsr layout

Bits	Name	Description
2:1	vxrm[1:0]	Fixed-point rounding mode
0	vxsat	Fixed-point accrued saturation flag

3.10. State of Vector Extension at Reset

The vector extension must have a consistent state at reset. In particular, `vtype` and `vl` must have values that can be read and then restored with a single `vsetvl` instruction.

> It is recommended that at reset, `vtype.vill` is set, the remaining bits in `vtype` are zero, and `vl` is set to zero.

The `vstart`, `vxrm`, `vxsat` CSRs can have arbitrary values at reset.

> Any use of the vector unit will require an initial `vsetvl{i}`, which will reset `vstart`. The `vxrm` and `vxsat` fields should be reset explicitly in software before use.

The vector registers can have arbitrary values at reset.

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, VLEN, and SLEN. Elements are packed into each vector register with the least-significant byte in the lowest-numbered bits.

4.1. Mapping for VLEN=SLEN and LMUL ≤ 1

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

> 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.

```
LMUL=1 examples.

The element index is given in hexadecimal and is shown placed at the
least-significant byte of the stored element.


 VLEN=SLEN=32b

 Byte         3 2 1 0

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

 VLEN=SLEN=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=SLEN=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=SLEN=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
```

When LMUL < 1, only the first LMUL*VLEN/SEW elements in the vector register are used. The remaining space in the vector register is treated as part of the tail.

```
 Example, VLEN=SLEN=128b, LMUL=1/4

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

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

4.2. Mapping with SLEN=VLEN and LMUL > 1

When vector registers are grouped, the elements of the vector register group are striped across the constituent vector registers. When SLEN=VLEN, the elements are packed contiguously in element order in each vector register in the group, moving to the next highest-numbered vector register in the group once each vector register is filled.

```
LMUL examples for SLEN=VLEN

VLEN=SLEN=32b, SEW=8b, LMUL=2

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

VLEN=SLEN=32b, SEW=16b, LMUL=2

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

VLEN=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

VLEN=SLEN=32b, SEW=32b, LMUL=4

Byte         3 2 1 0
v4*n               0
v4*n+1             1
v4*n+2             2
v4*n+3             3

VLEN=SLEN=64b, SEW=32b, LMUL=2

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

VLEN=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

VLEN=SLEN=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

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

4.3. Mapping with SLEN < VLEN and LMUL ≤ 1

The striping distance in bits, SLEN, sets the maximum displacement between vector register bit positions that participate in the same elemental mixed-width arithmetic operation.

> The striping distance SLEN can be designed as less than VLEN to reduce cross-datapath wiring for mixed-width operations on implementations that have wide spatial vector datapaths. For such machines, SLEN is typically at least 128 bits and used for vector datapaths operating on at least 256 bits per cycle or wider. For datapath designs where cross-datapath wiring is not a constraint, SLEN will usually be set to VLEN.

> Compared to v0.8, the SLEN<VLEN mapping pattern has changed to support fractional LMUL in a straightforward way and also to provide greater throughput on shorter vectors when using a wide spatial datapath. The main complication is that the memory connections are somewhat more complex, if not greater in number.

When SLEN < VLEN, each vector register is divided into VLEN/SLEN sections each holding SLEN bits.

When LMUL=1, successive vector elements are mapped into successive sections, wrapping back around to the first section until the vector register is full.

```
LMUL=1 examples for SLEN < VLEN

VLEN=256b, SLEN=128b, SEW=8b, LMUL=1

Section                  1               |                0
Byte     1F1E1D1C1B1A191817161514131211110| F E D C B A 9 8 7 6 5 4 3 2 1 0
         1F1D1B1917151311 F D B 9 7 5 3 1|1E1C1A1816141210 E C A 8 6 4 2 0

VLEN=256b, SLEN=128b, SEW=16b, LMUL=1

Byte     1F1E1D1C1B1A191817161514131211110| F E D C B A 9 8 7 6 5 4 3 2 1 0
            F   D   B   9   7   5   3   1|   E   C   A   8   6   4   2   0

VLEN=256b, SLEN=128b, SEW=32b, LMUL=1

Byte     1F1E1D1C1B1A191817161514131211110| F E D C B A 9 8 7 6 5 4 3 2 1 0
                7       5       3       1|       6       4       2       0

VLEN=256b, SLEN=128b, SEW=64b, LMUL=1

Byte     1F1E1D1C1B1A191817161514131211110| F E D C B A 9 8 7 6 5 4 3 2 1 0
                        3               1|               2               0

VLEN=256b, SLEN=64b, SEW=16b, LMUL=1

Section          3       |        2       |        1       |        0
Byte     1F1E1D1C1B1A1918|1716151413121110| F E D C B A 9 8| 7 6 5 4 3 2 1 0
            F   B   7   3|   E   A   6   2|   D   9   5   1|   C   8   4   0

VLEN=256b, SLEN=64b, SEW=32b, LMUL=1

Byte     1F1E1D1C1B1A1918|1716151413121110| F E D C B A 9 8| 7 6 5 4 3 2 1 0
                7       3|       6       2|       5       1|       4       0

VLEN=256b, SLEN=64b, SEW=64b, LMUL=1

Byte     1F1E1D1C1B1A1918|1716151413121110| F E D C B A 9 8| 7 6 5 4 3 2 1 0
                        3|               2|               1|               0
```

When LMUL < 1, only the first LMUL*VLEN/SEW elements in the vector register are used, with these elements mapped to sections in the same way as when LMUL=1. The remaining space in the vector register is treated as part of the tail.

> As with SLEN=VLEN designs, SLEN<VLEN implementations can treat fractional LMUL simply as though the vector length was reduced with LMUL=1.

```
Example, VLEN=256b, SLEN=128b

SEW=8b, LMUL=1/4

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

SEW=16b, LMUL=1/4

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

SEW=32b, LMUL=1/4

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

> The SLEN < VLEN mapping pattern does require full-width cross-datapath connections in the vector memory system between memory bytes and vector register bytes, but this is unavoidable in general, and a small fraction of all operand wiring in a vector unit.

4.4. Mapping with SLEN < VLEN and LMUL > 1

When SLEN < VLEN and LMUL > 1, the first vector register is packed with the initial VLEN/SEW elements in the same way as for LMUL=1. The second vector register in the group is packed with the next VLEN/SEW elements following the same pattern.

```
LMUL examples for SLEN < VLEN

VLEN=256b, SLEN=128b, SEW=32b, LMUL=2

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

VLEN=256b, SLEN=128b, SEW=64b, LMUL=2

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

VLEN=256b, SLEN=128b, SEW=64b, LMUL=4

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

If SEW > SLEN, the packing operates as if SLEN was increased to SEW.

> In most implementations, the striping distance SLEN ≥ ELEN.

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 SEW used to access values held in a vector register group (i.e., do not read values with a different SEW than used to write values to the group).

4.5. Mapping across Mixed-Width Operations

The vector ISA is designed to support mixed-width operations without requiring a large number of explicit additional rearrangement instructions or requiring a large amount of additional datapath wiring. The recommended software strategy is to modify `vtype` dynamically to keep SEW/LMUL constant (and hence VLMAX constant) when operating on vectors of different precision values.

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 (VLMAX=16 in this example) to simplify stripmining code. Any operation between elements with the same index only touches operand bits located within the same SLEN=128b section of the datapath.

```
Examples VLEN=256b, SLEN=128b, with SEW/LMUL=16

Section                  1               |               0
Byte     1F1E1D1C1B1A19181716151413121110| F E D C B A 9 8 7 6 5 4 3 2 1 0
vn        - - - - - - - - F D B 9 7 5 3 1| - - - - - - - - E C A 8 6 4 2 0  SEW=8b, LMUL=1/2

Byte     1F1E1D1C1B1A19181716151413121110| F E D C B A 9 8 7 6 5 4 3 2 1 0
vn         F   D   B   9   7   5   3   1|   E   C   A   8   6   4   2   0  SEW=16b, LMUL=1

Byte     1F1E1D1C1B1A19181716151413121110| F E D C B A 9 8 7 6 5 4 3 2 1 0
v2*n           7       5       3       1|       6       4       2       0  SEW=32b, LMUL=2
v2*n+1         F       D       B       9|       E       C       A       8

Byte     1F1E1D1C1B1A19181716151413121110| F E D C B A 9 8 7 6 5 4 3 2 1 0  SEW=64b, LMUL=4
v4*n                   3               1|               2               0
v4*n+1                 7               5|               6               4
v4*n+2                 B               9|               A               8
v4*n+3                 F               D|               E               C
```

The following table shows each possible constant SEW/LMUL operating point for loops with mixed-width operations. Each column represents a constant SEW/LMUL operating point. Entries in table are the LMUL values that yield that column's SEW/LMUL value for the datawidth on that row. In each column, an LMUL setting for a datawidth indicates that it can be aligned with the other datawidths in the same column that also have an LMUL setting, such that all have the same VLMAX and that element-wise operations between data widths are contained within SLEN-wide sections of the datapath.

SEW/LMUL	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192
SEW= 8	8	4	2	1	1/2	1/4	1/8							
SEW= 16		8	4	2	1	1/2	1/4	1/8						
SEW= 32			8	4	2	1	1/2	1/4	1/8					
SEW= 64				8	4	2	1	1/2	1/4	1/8				
SEW= 128					8	4	2	1	1/2	1/4	1/8			
SEW= 256						8	4	2	1	1/2	1/4	1/8		
SEW= 512							8	4	2	1	1/2	1/4	1/8	
SEW=1024								8	4	2	1	1/2	1/4	1/8

Larger LMUL settings can also used to simply increase vector length to reduce instruction fetch and dispatch overheads in cases where fewer vector register groups are needed.

> The SEW/LMUL values of 2048 and greater are shown in the table for completeness but they do not add a useful operating point in the base architecture as they use less than the full register capacity and do not enable more architectural registers.

4.6. Mask Register Layout

A vector mask occupies only one vector register regardless of SEW and LMUL. Each element is allocated a single mask bit in a mask vector register.

> Earlier designs (pre-0.9) had a varying number of bits per mask value (MLEN). In the 0.9 design, MLEN=1.

4.6.1. Mask Element Locations for SLEN=VLEN

The mask bit for element *i* is located in bit *i* of the mask register, independent of SEW or LMUL.

```
SLEN=VLEN=32b

         Byte    3   2   1   0
LMUL=1,SEW=8b
                 3   2   1   0  Element
               [03][02][01][00] Mask bit position in decimal

LMUL=2,SEW=16b
                     1       0
                   [01]    [00]
                     3       2
                   [03]    [02]

LMUL=4,SEW=32b               0
                           [00]
                             1
                           [01]
                             2
                           [02]
                             3
                           [03]
```

```
LMUL=2,SEW=8b
                  3   2   1   0
                [03][02][01][00]
                  7   6   5   4
                [07][06][05][04]

LMUL=8,SEW=32b
                              0
                            [00]
                              1
                            [01]
                              2
                            [02]
                              3
                            [03]
                              4
                            [04]
                              5
                            [05]
                              6
                            [06]
                              7
                            [07]

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]
```

4.7. Mask Element Locations for SLEN < VLEN

In systems with SLEN < VLEN, the mask elements are striped across the VLEN/SLEN sections of the vector mask register. Mask element 0 is in the least-significant bit of section 0, and mask element 1 is in least-significant bit of section 1, etc. In general, the the mask bit for element *i* is located bit:

mask_bit_index(*i*) = (*i* % (VLEN / SLEN)) * SLEN + floor(*i* / (VLEN / SLEN))

```
Mask register element location examples for SLEN < VLEN
The bit position of the LSB of each mask element is in decimal inside [] braces.

VLEN=256b, SLEN=128b, SEW=32b, LMUL=2, MLEN=16b

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

v2*n             7       5       3       1|       6       4       2       0
             [131]   [130]   [129]   [128]      [3]     [2]     [1]     [0]

v2*n+1           F       D       B       9|       E       C       A       8
             [135]   [134]   [133]   [132]      [7]     [6]     [5]     [4]


VLEN=256b, SLEN=128b, SEW=32b, LMUL=1/2, MLEN=64b

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

vn               -       -       3       1|       -       -       2       0
                             [129]   [128]                      [1]     [0]

VLEN=256b, SLEN=128b, SEW=64b, LMUL=1, MLEN=64b

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

vn                       3               1|               2               0
                     [129]           [128]              [1]             [0]
```

TODO: More examples might be useful.

5. Vector Instruction Formats

The instructions in the vector extension fit under three 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).

```
Format for Vector Load Instructions under LOAD-FP major opcode
31 29  28  27 26  25  24      20 19       15 14   12 11      7 6     0
 nf   | mew| mop | vm |  lumop   |     rs1     | width |    vd   |0000111| VL*  unit-stride
 nf   | mew| mop | vm |   rs2    |     rs1     | width |    vd   |0000111| VLS* strided
 nf   | mew| mop | vm |   vs2    |     rs1     | width |    vd   |0000111| VLX* indexed
  3      1     2     1      5            5          3          5       7

Format for Vector Store Instructions under STORE-FP major opcode
31 29  28  27 26  25  24      20 19       15 14   12 11      7 6     0
 nf   | mew| mop | vm |  sumop   |     rs1     | width |   vs3   |0100111| VS*  unit-stride
 nf   | mew| mop | vm |   rs2    |     rs1     | width |   vs3   |0100111| VSS* strided
 nf   | mew| mop | vm |   vs2    |     rs1     | width |   vs3   |0100111| VSX* indexed
  3      1     2     1      5            5          3          5        7
```

```
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/vd  |0101111| VAMO*
   5       1   1       5            5          3         5        7
```

```
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    | 0 0 0 |    vd   |1010111| OP-V (OPIVV)
  funct6   | vm  |   vs2    |    vs1    | 0 0 1 |  vd/rd  |1010111| OP-V (OPFVV)
  funct6   | vm  |   vs2    |    vs1    | 0 1 0 |  vd/rd  |1010111| OP-V (OPMVV)
  funct6   | vm  |   vs2    |   simm5   | 0 1 1 |    vd   |1010111| OP-V (OPIVI)
  funct6   | vm  |   vs2    |    rs1    | 1 0 0 |    vd   |1010111| OP-V (OPIVX)
  funct6   | vm  |   vs2    |    rs1    | 1 0 1 |    vd   |1010111| OP-V (OPFVF)
  funct6   | vm  |   vs2    |    rs1    | 1 1 0 |  vd/rd  |1010111| OP-V (OPMVX)
     6         1        5          5         3         5        7
```

```
Formats for Vector Configuration Instructions under OP-V major opcode

 31 30         25 24      20 19       15 14   12 11      7 6     0
 0 |        zimm[10:0]      |    rs1    | 1 1 1 |    rd   |1010111| vsetvli
 1 |   000000    |   rs2    |    rs1    | 1 1 1 |    rd   |1010111| vsetvl
 1        6             5          5         3         5        7
```

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.

5.1. 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.

> 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.

5.2. Vector Operands

Each vector operand has an *effective element width* (EEW) and an *effective* LMUL (EMUL) that is used to determine the size and location of all the elements within a vector register group. By default, for most operands of most instructions, EEW=SEW and EMUL=LMUL.

Some vector instructions have source and destination vector operands with the same number of elements but different widths, so that EEW and EMUL differ from SEW and LMUL respectively but EEW/EMUL = SEW/LMUL. For example, most widening arithmetic instructions have a source group with EEW=SEW and EMUL=LMUL but destination group with EEW=2*SEW and EMUL=2*LMUL. Narrowing instructions have a source operand that has EEW=2*SEW and EMUL=2*LMUL but destination where EEW=SEW and EMUL=LMUL.

Vector operands or results may occupy one or more vector registers depending on EMUL, 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.

The largest vector register group used by an instruction can not be greater than 8 vector registers (i.e., EMUL≤}8), and if a vector instruction would require greater than 8 vector registers in a group, an illegal instruction exception is raised. For example, attempting a widening operation producing a widened vector register group result when LMUL=8 will raise an illegal instruction exception as this would imply a result EMUL=16.

Widened scalar values, e.g., results from widening reduction operations, are held in the first element of a vector register and have EMUL=1.

5.3. Vector Masking

Masking is supported on many vector instructions. Element operations that are masked off (inactive) never generate exceptions. The destination vector register elements corresponding to masked-off elements are handled with either a mask-undisturbed or mask-agnostic policy depending on the setting of the `vma` bit in `vtype` (Section Vector Tail Agnostic and Vector Mask Agnostic `vta` and `vma`).

In the base vector extension, the mask value used to control execution of a masked vector instruction is always supplied by vector register `v0`.

> 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 cannot overlap the source mask register (`v0`), unless the destination vector register is being written with a mask value (e.g., comparisons) or the scalar result of a reduction. Otherwise, an illegal instruction exception is raised.

> 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.

When a mask is written with a compare result, destination mask bits past the end of the current vector length are handled according to the tail policy (undisturbed or agnostic) set by the `vta` bit in `vtype (Section Vector Tail Agnostic and Vector Mask Agnostic `vta` and `vma`).

5.3.1. Mask Encoding

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

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

> 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.mask[i]` 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.mask[i]=1, m=0
    vop.v*    v1, v2, v3        # unmasked vector operation, m=1
```

> 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.

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

The element indices 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 any destination vector register group unless masked agnostic is specified (`vtype.vma`=1), in which case inactive elements may be overwritten with 1s.
- 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, and do not update any destination vector register group unless tail agnostic is specified (`vtype.vta`=1), in which case tail elements may be overwritten with 1s. When LMUL < 1, the tail includes the elements past VLMAX that are held in the same vector register.
- 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 < max(VLMAX,VLEN/SEW))
```

6. Configuration-Setting Instructions

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

6.1. vsetvli/vsetvl instructions

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

```
vsetvli rd, rs1, vtypei # rd = new vl, rs1 = AVL, vtypei = new vtype setting
vsetvl  rd, rs1, rs2    # rd = new vl, rs1 = AVL, rs2 = new vtype value
```

The new vtype setting is encoded in the immediate fields of vsetvli and in the rs2 register for vsetvl. The new vector length setting is based on the requested application vector length (AVL), which is encoded in the rs1 and rd fields as follows:

Table 7. AVL used in vsetvli and vsetvl instructions

rd	rs1	AVL value	Description/Usage
0	0	Value in vl register	Change vtype keeping existing vl
!0	0	~0	Set vl to VLMAX
-	!0	Value in x[rs1]	Normal stripmining

When rs1 is not x0, the AVL is an unsigned integer held in the x register specified by rs1, and the new vl value is also written to the x register specified by rd.

When rs1=x0 but rd!=x0, the maximum unsigned integer value (~0) is used as the AVL, and the resulting VLMAX is written to vl and also to the x register specified by rd.

When rs1=x0 and rd=x0, the current vector length in vl is used as the AVL, and the resulting value is only written to vl.

> This form of the instruction allows the vtype register to be changed while maintaining the current vl, provided VLMAX is not reduced. The vl value can be reduced by this instruction if the SEW/LMUL ratio changes causes VLMAX to shrink. This design was chosen to ensure vl would always hold a legal value for current vtype setting. The current vl value can be read from the vl CSR.

```
Formats for Vector Configuration Instructions under OP-V major opcode

 31 30         25 24      20 19      15 14    12 11      7 6     0
 0 |        zimm[10:0]      |    rs1    | 1 1 1 |    rd   |1010111| vsetvli
 1 |   000000    |   rs2    |    rs1    | 1 1 1 |    rd   |1010111| vsetvl
 1        6            5          5        3         5        7
```

Table 8. `vtype` register layout

Bits	Name	Description
XLEN-1	vill	Illegal value if set
XLEN-2:8		Reserved (write 0)
7	vma	Vector mask agnostic
6	vta	Vector tail agnostic
5	vlmul[2]	Vector register group multiplier (LMUL) setting (fractional)
4:2	vsew[2:0]	Standard element width (SEW) setting
1:0	vlmul[1:0]	Vector register group multiplier (LMUL) setting

```
 Suggested assembler names used for vsetvli immediate

 e8    # SEW=8b
 e16   # SEW=16b
 e32   # SEW=32b
 e64   # SEW=64b
 e128  # SEW=128b
 e256  # SEW=256b
 e512  # SEW=512b
 e1024 # SEW=1024b

 mf8  # LMUL=1/8
 mf4  # LMUL=1/4
 mf2  # LMUL=1/2
 m1   # LMUL=1, assumed if m setting absent
 m2   # LMUL=2
 m4   # LMUL=4
 m8   # LMUL=8

Examples:
    vsetvli t0, a0, e8          # SEW= 8, LMUL=1
    vsetvli t0, a0, e8,m2       # SEW= 8, LMUL=2
    vsetvli t0, a0, e32,mf2     # SEW=32, LMUL=1/2
```

If the `vtype` setting is not supported by the implementation, then the `vill` bit is set in `vtype`, the remaining bits in `vtype` are set to zero, and the `vl` register is also set to zero.

> Earlier drafts required a trap when setting `vtype` to an illegal value. However, this would have added the first data-dependent trap on a CSR write to the ISA. The current scheme also supports light-weight runtime interrogation of the supported vector unit configurations by checking if `vill` is clear for a given setting.

6.2. Constraints on Setting `vl`

The `vsetvl{i}` instructions first set VLMAX according to the `vtype` argument, then set `vl` obeying the following constraints:

1. `vl = AVL` if `AVL ≤ VLMAX`
2. `ceil(AVL / 2) ≤ vl ≤ VLMAX` if `AVL < (2 * VLMAX)`
3. `vl = VLMAX` if `AVL ≥ (2 * VLMAX)`
4. Deterministic on any given implementation for same input AVL and VLMAX 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 the AVL argument to `vsetvl{i}` results in the same value in `vl`, provided the resultant VLMAX equals the value of VLMAX at the time that `vl` was read

> 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.

> 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:
    vsetvli a3, a0, e16,m4  # vtype = 16-bit integer vectors
    vle16.v v4, (a1)        # Get 16b vector
      slli t1, a3, 1        # Multiply length by two bytes/element
      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
    vse32.v v8, (a2)        # Store vector of 32b elements
      slli t1, a3, 2        # Multiply length by four bytes/element
      add a2, a2, t1        # Bump pointer
      sub a0, a0, a3        # Decrement count
      bnez a0, loop         # Any more?
```

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 only update active 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).

```
Format for Vector Load Instructions under LOAD-FP major opcode
31 29  28  27 26  25  24      20 19       15 14   12 11      7 6     0
 nf   | mew| mop | vm |  lumop    |     rs1     | width |    vd    |0000111| VL*  unit-stride
 nf   | mew| mop | vm |   rs2     |     rs1     | width |    vd    |0000111| VLS* strided
 nf   | mew| mop | vm |   vs2     |     rs1     | width |    vd    |0000111| VLX* indexed
  3      1     2     1       5           5           3          5        7

Format for Vector Store Instructions under STORE-FP major opcode
31 29  28  27 26  25  24      20 19       15 14   12 11      7 6     0
 nf   | mew| mop | vm |  sumop    |     rs1     | width |   vs3    |0100111| VS*  unit-stride
 nf   | mew| mop | vm |   rs2     |     rs1     | width |   vs3    |0100111| VSS* strided
 nf   | mew| mop | vm |   vs2     |     rs1     | width |   vs3    |0100111| VSX* indexed
  3      1     2     1       5           5           3          5         7
```

Field	Description
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
mew	extended memory element size
mop[1:0]	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

Vector memory operations directly encode EEW of the data to be transferred statically in the instruction to reduce the number of `vtype` changes when accessing memory in a mixed-width routine. Indexed operations use the explicit EEW encoding in the instruction to set the size of the indices used, and use SEW/LMUL to specify the data width.

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 data vector register group has EEW=SEW, EMUL=LMUL, while the offset vector register group has EEW encoding in the instruction and EMUL=(EEW/SEW)*LMUL.

The vector offset operand is treated as a vector of byte-address offsets. If the vector offset elements are narrower than XLEN, they are zero-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.

The vector addressing modes are encoded using the 2-bit `mop[1:0]` field.

Table 9. encoding for loads

mop [1:0]		**Description**	**Opcodes**
0	0	unit-stride	VLE<EEW>
0	1	reserved	-
1	0	strided	VLSE<EEW>
1	1	indexed	VLXEI<EEW>

Table 10. encoding for stores

mop [1:0]		**Description**	**Opcodes**
0	0	unit-stride	VSE<EEW>
0	1	indexed-unordered	VSUXEI<EEW>
1	0	strided	VSSE<EEW>
1	1	indexed-ordered	VSXEI<EEW>

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

> 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. For implementations with precise vector traps, exceptions on indexed-unordered stores are precise.

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 11. lumop

lumop[4:0]					**Description**
0	0	0	0	0	unit-stride
0	0	x	x	x	reserved, x !=0
0	1	0	0	0	unit-stride, whole registers
0	1	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 12. sumop

sumop[4:0]					Description
0	0	0	0	0	unit-stride
0	0	x	x	x	reserved, x !=0
0	1	0	0	0	unit-stride, whole registers
0	1	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`).

> 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.

The `nf[2:0]` field also encodes the number of whole vector registers to transfer for the whole vector register load/store instructions.

7.3. Vector Load/Store Width Encoding

Vector loads and stores have the EEW encoded directly in the instruction. EMUL is calculated as EMUL = (EEW/SEW)*LMUL. If the EMUL would be out of range (EMUL>8 or EMUL<1/8), an illegal instruction exception is raised. The vector register groups must have legal register specifiers for the selected EMUL, else an illegal instruction is raised.

Vector loads and stores are encoded using width values that are not claimed by the standard scalar floating-point loads and stores. The mew bit (`inst[28]`) encodes expanded memory sizes of 128 bits and above.

Vector loads and stores up to EEW=ELEN must be supported in an implementation. Using a vector load/store with an unsupported EEW raises an illegal instruction exception.

	mew	width [2:0]			Mem bits	Reg bits	Opcodes
Standard scalar FP	x	0	0	1	16	FLEN	FLH/FSH
Standard scalar FP	x	0	1	0	32	FLEN	FLW/FSW
Standard scalar FP	x	0	1	1	64	FLEN	FLD/FSD
Standard scalar FP	x	1	0	0	128	FLEN	FLQ/FSQ
Vector 8b element	0	0	0	0	8	8	VLxE8/VSxE8
Vector 16b element	0	1	0	1	16	16	VLxE16/VSxE16
Vector 32b element	0	1	1	0	32	32	VLxE32/VSxE32
Vector 64b element	0	1	1	1	64	64	VLxE64/VSxE64
Vector 128b element	1	0	0	0	128	128	VLxE128/VSx-E128
Vector 256b element	1	1	0	1	256	256	VLxE256/VSx-E256
Vector 512b element	1	1	1	0	512	512	VLxE512/VSx-E512
Vector 1024b element	1	1	1	1	1024	1024	VLxE1024/VSx-E1024

Mem bits is the size of each element accessed in memory

Reg bits is the size of each element accessed in register

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>)
    vle32.v  vd, (rs1), vm # 32-bit loads

    # vs3 store data, rs1 base address, vm is mask encoding (v0.t or <missing>)
    vse64.v  vs3, (rs1), vm  # 64-bit stores
```

7.5. Vector Strided Instructions

```
    # Vector strided loads and stores

    # vd destination, rs1 base address, rs2 byte stride
    vlse8.v  vd, (rs1), rs2, vm  # Load bytes separated by stride

    # vs3 store data, rs1 base address, rs2 byte stride
    vsse128.v vs3, (rs1), rs2, vm  # Store 128b values separated by stride.
```

> Negative and zero strides are supported.

7.6. Vector Indexed Instructions

```
    # Vector indexed loads and stores

    # vd destination, rs1 base address, vs2 indices
    vlxei16.v  vd, (rs1), vs2, vm  # vs2 data EEW = SEW, indices EEW = 16b

    # Vector ordered-indexed store instructions
    # vs3 store data, rs1 base address, vs2 indices
    vsxei32.v vs3, (rs1), vs2, vm # SEW data, 32b indices

    # Vector unordered-indexed store instructions
    vsuxei64.v vs3, (rs1), vs2, vm # SEW data, 64b indices
```

7.7. Unit-stride Fault-Only-First Loads

The unit-stride fault-only-first load instruction is 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.

```
    vle8ff.v  vd, (rs1), vm
```

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

link:example/strlen.s[]
```

> 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.

Even when an exception is not raised, implementations are permitted to process fewer than `vl` elements and reduce `vl` accordingly, but if `vstart`=0 and `vl`>0, then at least one element must be processed.

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

> This set of instructions is intended to be included in the base "V" extension.

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

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

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

The EMUL setting must be such that EMUL * NFIELDS ≤ 8, otherwise an illegal instruction exception is raised.

> The product EMUL * NFIELDS represents the number of underlying vector registers that will be touched by a segmented load or store instruction. This constraint makes this total no larger than 1/4 of the architectural register file, and the same as for regular operations with EMUL=8. This constraint could be weakened in a future draft.

Each field will be held in successively numbered vector register groups. When EMUL>1, each field will occupy a vector register group held in multiple successively numbered vector registers, and the vector register group for each field must follow the usual vector register alignment constraints (e.g., when EMUL=2 and NFIELDS=4, each field's vector register group must start at an even vector register, but does not have to start at a multiple of 8 vector register number).

> 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.

If the vector register numbers accessed by the segment load or store would increment past 31, then an illegal instruction exception is raised.

> This constraint is to help provide forward-compatibility with a future longer instruction encoding that has more addressable vector registers.

The `vl` register gives the number of structures to move, which is equal to the number of elements transferred to each vector register group. Masking is also applied at the level of whole structures.

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

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.

> 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>e<eew>.v vd, (rs1), vm       # Unit-stride segment load template
vsseg<nf>e<eew>.v vs3, (rs1), vm       # Unit-stride segment store template

# Examples
vlseg8e8.v vd, (rs1), vm   # Load eight vector registers with eight byte fields.

vsseg3e32.v vs3, (rs1), vm  # Store packed vector of 3*4-byte segments from vs3,vs3+1,vs
```

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)
vsetvli a1, t0, e8
vlseg3e8.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
vsetvli a1, t0, e32
vlseg2e32.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>e<eew>ff.v vd, (rs1),  vm          # Unit-stride 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.

> Negative and zero strides are supported.

```
# Format
vlsseg<nf>e<eew>.v vd, (rs1), rs2, vm          # Strided segment loads
vssseg<nf>e<eew>.v vs3, (rs1), rs2, vm         # Strided segment stores

# Examples
vsetvli a1, t0, e8
vlsseg3e8.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
vsetvli a1, t0, e32
vssseg2e32.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
```

For strided segment stores where the byte stride is such that segments could overlap in memory, the segments must appear to be written in element order.

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`.

The data vector register group has EEW=SEW, EMUL=LMUL, while the index vector register group has EEW encoded in the instruction with EMUL=(EEW/SEW)*LMUL.

```
# Format
vlxseg<nf>ei<eew>.v vd, (rs1), vs2, vm          # Indexed segment loads
vsxseg<nf>ei<eew>.v vs3, (rs1), vs2, vm         # Indexed segment stores

# Examples
vsetvli a1, t0, e8
vlxseg3ei32.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
vsetvli a1, t0, e32
vsxseg2e32.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
```

For vector indexed segment loads, the destination vector register groups cannot overlap the source vector register group (specified by `vs2`), else an illegal instruction exception is raised.

> This constraint supports restart of indexed segment loads that raise exceptions partway through loading a structure.

Only ordered indexed segment stores are provided. The segments must appear to be written in element order.

7.9. Vector Load/Store Whole Register Instructions

> These instructions are still under early consideration for inclusion.

These instructions load and store whole vector registers (i.e., VLEN bits). The instructions operate with an EEW=8 and effective vector length `evl`=VLEN/8, regardless of current settings in `vtype` and `vl`. No elements are transferred if `vstart` ≥ VLEN/8. The usual property that no elements are written if `vstart` ≥ `vl` does not apply to these instructions.

> These instructions are intended to be used to save and restore vector registers when the type and length of the current contents of the vector register is not known, or where modifying `vl` and `vtype` would be costly. Examples include compiler register spills, vector function calls where values are passed in vector registers, interrupt handlers, and OS context switches. Software can determine the number of bytes transferred by reading the `vlenb` register.

```
Format for Vector Load Whole Register Instructions under LOAD-FP major opcode
31 29 28 26  25  24      20 19       15 14   12 11      7 6     0
 nf  | 000 | 1 |   01000   |    rs1    |  000  |    vd   |0000111| VL<nf>R

Format for Vector Store Whole Register Instructions under STORE-FP major opcode
31 29 28 26  25  24      20 19       15 14   12 11      7 6     0
 nf  | 000 | 1 |   01000   |    rs1    |  000  |   vs3   |0100111| VS<nf>R
```

The instructions operate similarly to unmasked unit-stride load and store instructions of elements, with the base address passed in the scalar `x` register specified by `rs1`.

The instructions transfer a single vector register specified by `vd` for loads and `vs3` for stores. The registers are transferred to and from memory with EEW=8 and EMUL=1.

> The vector whole register load instructions are encoded similar to unmasked zero-extended unit-stride loads of elements, with the `nf` field encoding how many vector registers to load and store. The vector whole register store instructions are encoded similar to unmasked unit-stride store of elements. The current base specification mandates that only `nf`=0 is supported, with other values of `nf` reserved. In a future extension, when multiple registers are transferred, the vector register contents are mapped to contiguous bytes in memory as if LMUL=1, with the lowest-numbered vector register held in the lowest-numbered memory addresses. The `nf` field encodes the number of vector registers to transfer, numbered successively after the base. The base register plus the `nf` value cannot exceed 31, else an illegal instruction exception is raised.

```
   # Format
   vl1r.v v3, (a0)      # Load v3 with VLEN/8 bytes held at address in a0

   vs1r.v v3, (a1)      # Store v3 to address in a1
```

8. Vector AMO Operations (Zvamo)

> Profiles will dictate whether vector AMO operations are supported. The expectation is that the base "V" extension used for 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.

```
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/vd  |0101111| VAMO*
   5      1   1      5           5         3        5        7
```

```
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 index 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)
```

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.

AMOs have the same index EEW scheme as indexed operations, except without the `mew` bit, which is is assumed to be zero, so offsets can have EEW=8,16,32,64 only. A vector of byte offsets in register `vs2` is added to the scalar base register in `rs1` to give the addresses of the AMO operations.

The data register `vs3` used dynamic SEW and MUL setting.

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.

> 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.

> 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 13. Vector AMO width encoding

	Width [2:0]			Index EEW	Mem data bits	Reg data 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	0	0	0	8	SEW	SEW	VAMO*EI8.V
Vector AMO	1	0	1	16	SEW	SEW	VAMO*EI16.V
Vector AMO	1	1	0	32	SEW	SEW	VAMO*EI32.V
Vector AMO	1	1	1	64	SEW	SEW	VAMO*EI64.V

Index bits is the EEW of the offsets.

Mem bits is the size of element accessed in memory

Reg bits is the size of element accessed in register

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

Vector AMO instructions are only supported for the memory data element widths supported by AMOs in the implementation’s scalar architecture. Other element widths raise an illegal instruction exception.

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 14. 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

The assembly syntax uses `x0` in the destination register position to indicate the return value is not required (`wd=0`).

```
# Vector AMOs for index EEW=32
vamoswapei32.v vd, (rs1), v2, vd,  v0.t # Write original value to register, wd=1
vamoswapei32.v x0, (rs1), v2, vs3, v0.t # Do not write original value to register, wd=0

vamoaddei32.v vd, (rs1), v2, vd,  v0.t # Write original value to register, wd=1
vamoaddei32.v x0, (rs1), v2, vs3, v0.t # Do not write original value to register, wd=0

vamoxorei32.v vd, (rs1), v2, vd,  v0.t # Write original value to register, wd=1
vamoxorei32.v x0, (rs1), v2, vs3, v0.t # Do not write original value to register, wd=0

vamoandei32.v vd, (rs1), v2, vd,  v0.t # Write original value to register, wd=1
vamoandei32.v x0, (rs1), v2, vs3, v0.t # Do not write original value to register, wd=0

vamoorei32.v vd, (rs1), v2, vd,  v0.t # Write original value to register, wd=1
vamoorei32.v x0, (rs1), v2, vs3, v0.t # Do not write original value to register, wd=0

vamominei32.v vd, (rs1), v2, vd,  v0.t # Write original value to register, wd=1
vamominei32.v x0, (rs1), v2, vs3, v0.t # Do not write original value to register, wd=0

vamomaxei32.v vd, (rs1), v2, vd,  v0.t # Write original value to register, wd=1
vamomaxei32.v x0, (rs1), v2, vs3, v0.t # Do not write original value to register, wd=0

vamominuei32.v vd, (rs1), v2, vd,  v0.t # Write original value to register, wd=1
vamominuei32.v x0, (rs1), v2, vs3, v0.t # Do not write original value to register, wd=0

vamomaxuei32.v vd, (rs1), v2, vd,  v0.t # Write original value to register, wd=1
vamomaxuei32.v x0, (rs1), v2, vs3, v0.t # Do not write original value to register, wd=0
```

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.

> 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.

```
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    | 0 0 0 |    vd   |1010111| OP-V (OPIVV)
  funct6    | vm  |   vs2     |    vs1    | 0 0 1 |  vd/rd  |1010111| OP-V (OPFVV)
  funct6    | vm  |   vs2     |    vs1    | 0 1 0 |  vd/rd  |1010111| OP-V (OPMVV)
  funct6    | vm  |   vs2     |   simm5   | 0 1 1 |    vd   |1010111| OP-V (OPIVI)
  funct6    | vm  |   vs2     |    rs1    | 1 0 0 |    vd   |1010111| OP-V (OPIVX)
  funct6    | vm  |   vs2     |    rs1    | 1 0 1 |    vd   |1010111| OP-V (OPFVF)
  funct6    | vm  |   vs2     |    rs1    | 1 1 0 |  vd/rd  |1010111| OP-V (OPMVX)
     6         1        5          5        3        5        7
```

11.1. Vector Arithmetic Instruction encoding

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

Table 15. 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 SEW 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 operations, the scalar can be taken from a scalar f register. If FLEN>SEW, the value in the f registers is checked for a valid NaN-boxed value, in which case the least-significant SEW bits of the `f` register are used, else the canonical NaN value is used. If FLEN<SEW, the value is NaN-boxed (one-extended) to SEW.

> The 5-bit immediate is unsigned when either providing a register index in vrgather or a count for shift, clip, or slide. In all other cases it is signed and sign extended to SEW bits, even for bitwise and unsigned instructions, notably compare and add.

> 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]
```

> 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 sum input
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 product input
vop.vv vd, vs1, vs2, vm  # vd[i] = vs1[i] * vd[i] + vs2[i]
vop.vx vd, rs1, vs2, vm  # vd[i] = x[rs1] * vd[i] + vs2[i]

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

> 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 have EEW=2*SEW and EMUL=2*LMUL.

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.

```
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
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]
```

> 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.

> 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*`.

> For integer multiply-add, another possible widening option increases the size of the accumulator to EEW=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., EEW=2*SEW, EMUL=2*LMUL).

For all widening instructions, the destination EEW and EMUL values must be a supported configuration, 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 EMUL, otherwise an illegal instruction exception is raised.

The destination vector register group cannot overlap a source vector register group of a different EEW, otherwise an illegal instruction exception is raised.

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

> For the `vw<op>.wv vd, vs2, vs1` format instructions, it is legal for vd to equal vs2.

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 with EEW/EMUL=2*SEW/2*LMUL to a vector register group with the current LMUL/SEW vectors/elements.

If EEW > ELEN or EMUL > 8, an illegal instruction exception is raised.

> 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 EMUL values respectively, otherwise an illegal instruction exception is raised.

Where there is a second source vector register group (specified by `vs1`), this has the same (narrower) width as the result (i.e., EEW=SEW).

The destination vector register group cannot overlap the first source vector register group (specified by `vs2`), otherwise an illegal instruction exception is raised.

> It is safe to overwrite a second source vector register group with the same LMUL and element 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. The double-width source vector register group is signified by a w in the source operand suffix (e.g., `vnsra.wv`)

> 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 is 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
```

> 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 Extension

The vector integer extension instructions zero- or sign-extend a source vector integer operand with EEW less than SEW to fill SEW-sized elements in the destination. The EEW of the source is 1/2, 1/4, or 1/8 of the destination, while EMUL of the source is (EEW/SEW)*LMUL.

```
vzext.vf2 vd, vs2, vm  # Zero-extend SEW/2 source to SEW destination
vsext.vf2 vd, vs2, vm  # Sign-extend SEW/2 source to SEW destination
vzext.vf4 vd, vs2, vm  # Zero-extend SEW/4 source to SEW destination
vsext.vf4 vd, vs2, vm  # Sign-extend SEW/4 source to SEW destination
vzext.vf8 vd, vs2, vm  # Zero-extend SEW/8 source to SEW destination
vsext.vf8 vd, vs2, vm  # Sign-extend SEW/8 source to SEW destination
```

If the source EEW is not a supported width or the source EMUL is not a supported LMUL, an illegal instruction exception is raised.

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

To support multi-word integer arithmetic, instructions that operate on a carry bit are provided. For each operation (add or subtract), two instructions are provided: one to provide the result (SEW width), and the second to generate the carry output (single bit encoded as a mask boolean).

The carry inputs and outputs are represented using the mask register layout as described in Section Mask Register Layout. Due to encoding constraints, the carry input must come from the implicit `v0` register, but carry outputs can be written to any vector register that respects the source/destination overlap restrictions below.

`vadc` and `vsbc` add or subtract the source operands and the carry-in or borrow-in, and write the result to vector register `vd`. These instructions are encoded as masked instructions (`vm=0`), but they operate on and write back all body elements. Encodings corresponding to the unmasked versions (`vm=1`) are reserved.

`vmadc` and `vmsbc` add or subtract the source operands, optionally add the carry-in or subtract the borrow-in if masked (`vm=0`), and write the result back to mask register `vd`. If unmasked (`vm=1`), there is no carry-in or borrow-in. These instructions operate on and write back all body elements, even if masked.

```
# Produce sum with carry.

# vd[i] = vs2[i] + vs1[i] + v0[i].LSB
vadc.vvm   vd, vs2, vs1, v0  # Vector-vector

# vd[i] = vs2[i] + x[rs1] + v0[i].LSB
vadc.vxm   vd, vs2, rs1, v0  # Vector-scalar

# vd[i] = vs2[i] + imm + v0[i].LSB
vadc.vim   vd, vs2, imm, v0  # Vector-immediate

# Produce carry out in mask register format

# vd[i] = carry_out(vs2[i] + vs1[i] + v0[i].LSB)
vmadc.vvm   vd, vs2, vs1, v0  # Vector-vector

# vd[i] = carry_out(vs2[i] + x[rs1] + v0[i].LSB)
vmadc.vxm   vd, vs2, rs1, v0  # Vector-scalar

# vd[i] = carry_out(vs2[i] + imm + v0[i].LSB)
vmadc.vim   vd, vs2, imm, v0  # Vector-immediate

# vd[i] = carry_out(vs2[i] + vs1[i])
vmadc.vv    vd, vs2, vs1      # Vector-vector, no carry-in

# vd[i] = carry_out(vs2[i] + x[rs1])
vmadc.vx    vd, vs2, rs1      # Vector-scalar, no carry-in

# vd[i] = carry_out(vs2[i] + imm)
vmadc.vi    vd, vs2, imm      # Vector-immediate, no carry-in
```

Because implementing a carry propagation requires executing two instructions with unchanged inputs, destructive accumulations will require an additional move to obtain correct results.

```
 # Example multi-word arithmetic sequence, accumulating into v4
 vmadc.vvm v1, v4, v8, v0  # Get carry into temp register v1
 vadc.vvm v4, v4, v8, v0   # Calc new sum
 vmcpy.m v0, v1             # Move temp carry into v0 for next word
```

The subtract with borrow instruction `vsbc` performs the equivalent function to support long word arithmetic for subtraction. There are no subtract with immediate instructions.

```
# Produce difference with borrow.

# vd[i] = vs2[i] - vs1[i] - v0[i].LSB
vsbc.vvm   vd, vs2, vs1, v0  # Vector-vector

# vd[i] = vs2[i] - x[rs1] - v0[i].LSB
vsbc.vxm   vd, vs2, rs1, v0  # Vector-scalar

# Produce borrow out in mask register format

# vd[i] = borrow_out(vs2[i] - vs1[i] - v0[i].LSB)
vmsbc.vvm   vd, vs2, vs1, v0  # Vector-vector

# vd[i] = borrow_out(vs2[i] - x[rs1] - v0[i].LSB)
vmsbc.vxm   vd, vs2, rs1, v0  # Vector-scalar

# vd[i] = borrow_out(vs2[i] - vs1[i])
vmsbc.vv    vd, vs2, vs1      # Vector-vector, no borrow-in

# vd[i] = borrow_out(vs2[i] - x[rs1])
vmsbc.vx    vd, vs2, rs1      # Vector-scalar, no borrow-in
```

For `vmsbc`, the borrow is defined to be `1` iff the difference, prior to truncation, is negative.

For `vadc` and `vsbc`, an illegal instruction exception is raised if the destination vector register is `v0`.

> This constraint corresponds to the constraint on masked vector operations that overwrite the mask register.

For `vmadc` and `vmsbc`, an illegal instruction exception is raised if the destination vector register overlaps a source vector register group.

12.5. 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
```

> 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`.

12.6. 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, uimm, vm   # vector-immediate

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

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

Only the low lg2(SEW) bits are read to obtain the shift amount from a register value.

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

12.7. 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 unsigned immediate form supports shift amounts up to 31 only.

```
# Narrowing shift right logical, SEW = (2*SEW) >> SEW
vnsrl.wv vd, vs2, vs1, vm   # vector-vector
vnsrl.wx vd, vs2, rs1, vm   # vector-scalar
vnsrl.wi vd, vs2, uimm, vm   # vector-immediate

# Narrowing shift right arithmetic, SEW = (2*SEW) >> SEW
vnsra.wv vd, vs2, vs1, vm   # vector-vector
vnsra.wx vd, vs2, rs1, vm   # vector-scalar
vnsra.wi vd, vs2, uimm, vm   # vector-immediate
```

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

12.8. 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. The destination mask vector register may be the same as the source vector mask register (`v0`).

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

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

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

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

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

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

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

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

# Following two instructions are not provided directly
# Set if greater than or equal, unsigned
# vmsgeu.vx vd, vs2, rs1, vm    # Vector-scalar
# Set if greater than or equal, signed
# vmsge.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         vmslt{u}.vv vd, va, vb, vm
va <= vb        vmsle{u}.vv vd, va, vb, vm
va > vb         vmslt{u}.vv vd, vb, va, vm    vmsgt{u}.vv vd, va, vb, vm
va >= vb        vmsle{u}.vv vd, vb, va, vm    vmsge{u}.vv vd, va, vb, vm

va < x          vmslt{u}.vx vd, va, x, vm
va <= x         vmsle{u}.vx vd, va, x, vm
va > x          vmsgt{u}.vx vd, va, x, vm
va >= x         see below

va < i          vmsle{u}.vi vd, va, i-1, vm    vmslt{u}.vi vd, va, i, vm
va <= i         vmsle{u}.vi vd, va, i, vm
va > i          vmsgt{u}.vi vd, va, i, vm
va >= i         vmsgt{u}.vi vd, va, i-1, vm    vmsge{u}.vi vd, va, i, vm

va, vb vector register groups
x      scalar integer register
i      immediate
```

> The immediate forms of `vmslt{u}.vi` are not provided as the immediate value can be decreased by 1 and the `vmsle{u}.vi` variants used instead. The `vmsle.vi` range is -16 to 15, resulting in an effective `vmslt.vi` range of -15 to 16. The `vmsleu.vi` range is 0 to 15 (and `(~0)-15` to `~0`), giving an effective `vmsltu.vi` range of 1 to 16 (Note, `vmsltu.vi` with immediate 0 is not useful as it is always false). Similarly, `vmsge{u}.vi` is not provided and the comparison is implemented using `vmsgt{u}.vi` with the immediate decremented by one. The resulting effective `vmsge.vi` range is -15 to 16, and the resulting effective `vmsgeu.vi` range is 1 to 16 (Note, `vmsgeu.vi` with immediate 0 is not useful as it is always true).

> The `vmsgt` 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 `vmsge{u}.vx` form is not directly provided, and so the `va ≥ x` case requires special treatment.

> The `vmsge{u}.vx` could potentially be encoded in a non-orthogonal way under the unused OPIVI variant of `vmslt{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 `vmsge{u}.vx` operation can be synthesized by reducing the value of `x` by 1 and using the `vmsgt{u}.vx` instruction, when it is known that this will not underflow the representation in `x`.

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

va >= x,  x > minimum

   addi t0, x, -1; vmsgt{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: vmsge{u}.vx vd, va, x
  expansion: vmslt{u}.vx vd, va, x; vmnand.mm vd, vd, vd

masked va >= x, vd != v0

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

masked va >= x, vd == v0

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

masked va >= x, any vd

  pseudoinstruction: vmsge{u}.vx vd, va, x, v0.t, vt
  expansion: vmslt{u}.vx vt, va, x;  vmandnot.mm vt, v0, vt;  vmandnot.mm vd, vd, v0;  vmor.

  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
    vmslt.vv    v0, va, vb        # All body elements written
    vmslt.vv    v0, vb, vc, v0.t  # Only update at set mask
```

12.9. Vector Integer Min/Max Instructions

Signed and unsigned integer minimum 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
```

12.10. 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
```

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

> 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.

12.11. 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
```

> 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.

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

12.12. 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
```

12.13. 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`, `vnmsac`) and one that overwrites the first multiplicand (`vmadd`, `vnmsub`).

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

> "sac" is intended to be read as "subtract from accumulator". The opcode is "vnmsac" to match the (unfortunately counterintuitive) floating-point `fnmsub` instruction definition. Similarly for the "vnmsub" opcode.

```
# 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
vnmsac.vv vd, vs1, vs2, vm    # vd[i] = -(vs1[i] * vs2[i]) + vd[i]
vnmsac.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] = (vs1[i] * vd[i]) + vs2[i]
vmadd.vx vd, rs1, vs2, vm    # vd[i] = (x[rs1] * vd[i]) + vs2[i]

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

12.14. Vector Widening Integer Multiply-Add Instructions

The widening integer multiply-add instructions add a SEW-bit*SEW-bit multiply result to (from) a 2*SEW-bit value and produce a 2*SEW-bit result. All combinations of signed and unsigned multiply operands are supported.

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

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

12.15. Vector Quad-Widening Integer Multiply-Add Instructions (Extension `Zvqmac`)

The quad-widening integer multiply-add instructions add a SEW-bit*SEW-bit multiply result to (from) a 4*SEW-bit value and produce a 4*SEW-bit result. All combinations of signed and unsigned multiply operands are supported.

> These instructions are currently not planned to be part of the base V extension.

> On ELEN=32 machines, only 8b * 8b = 16b products accumulated in a 32b accumulator would be supported. Machines with ELEN=64 would also add 16b * 16b = 32b products accumulated in 64b.

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

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

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

# Quad-widening unsigned-signed-integer multiply-add, overwrite addend
vqmaccus.vx vd, rs1, vs2, vm    # vd[i] = +(unsigned(x[rs1]) * signed(vs2[i])) + vd[i]
```

12.16. Vector Integer Merge Instructions

The vector integer merge instructions combine two source operands based on a mask. 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`).

The `vmerge` instructions are always 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.

```
vmerge.vvm vd, vs2, vs1, v0  # vd[i] = v0.mask[i] ? vs1[i] : vs2[i]
vmerge.vxm vd, vs2, rs1, v0  # vd[i] = v0.mask[i] ? x[rs1] : vs2[i]
vmerge.vim vd, vs2, imm, v0  # vd[i] = v0.mask[i] ? imm    : vs2[i]
```

12.17. Vector Integer Move Instructions

The vector integer move instructions copy a source operand to a vector register group. The `vmv.v.v` variant copies a vector register group, whereas the `vmv.v.x` and `vmv.v.i` variants *splat* a scalar register or immediate to all active elements of the destination vector register group. These instructions are always unmasked (`vm=1`). The first operand specifier (`vs2`) must contain `v0`, and any other vector register number in `vs2` is *reserved*.

```
vmv.v.v vd, vs1 # vd[i] = vs1[i]
vmv.v.x vd, rs1 # vd[i] = rs1
vmv.v.i vd, imm # vd[i] = imm
```

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

> The vector integer move instructions share the encoding with the vector merge instructions, but with `vm=1` and `vs2=v0`.

13. Vector Fixed-Point Arithmetic Instructions

A set of vector arithmetic instructions is 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.

> 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`. Both unsigned and signed versions are provided. For `vaaddu`, `vaadd`, and `vasub`, there can be no overflow in the result. For `vasubu`, overflow is ignored.

```
# Averaging add

# Averaging adds of unsigned integers.
vaaddu.vv vd, vs2, vs1, vm   # roundoff_unsigned(vs2[i] + vs1[i], 1)
vaaddu.vx vd, vs2, rs1, vm   # roundoff_unsigned(vs2[i] + x[rs1], 1)

# Averaging adds of signed integers.
vaadd.vv vd, vs2, vs1, vm   # roundoff_signed(vs2[i] + vs1[i], 1)
vaadd.vx vd, vs2, rs1, vm   # roundoff_signed(vs2[i] + x[rs1], 1)

# Averaging subtract

# Averaging subtract of unsigned integers.
vasubu.vv vd, vs2, vs1, vm   # roundoff_unsigned(vs2[i] - vs1[i], 1)
vasubu.vx vd, vs2, rs1, vm   # roundoff_unsigned(vs2[i] - x[rs1], 1)

# Averaging subtract of signed integers.
vasub.vv vd, vs2, vs1, vm   # roundoff_signed(vs2[i] - vs1[i], 1)
vasub.vx vd, vs2, rs1, vm   # roundoff_signed(vs2[i] - x[rs1], 1)
```

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
# See vxrm  description for rounding calculation
vsmul.vv vd, vs2, vs1, vm  # vd[i] = clip(roundoff_signed(vs2[i]*vs1[i], SEW-1))
vsmul.vx vd, vs2, rs1, vm  # vd[i] = clip(roundoff_signed(vs2[i]*x[rs1], SEW-1))
```

> 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.

13.4. 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 low lg2(SEW) bits of the vector or scalar shift amount value are used. The immediate form supports shift amounts up to 31 only.

```
 # Scaling shift right logical
 vssrl.vv vd, vs2, vs1, vm   # vd[i] = roundoff_unsigned(vs2[i], vs1[i])
 vssrl.vx vd, vs2, rs1, vm   # vd[i] = roundoff_unsigned(vs2[i], x[rs1])
 vssrl.vi vd, vs2, uimm, vm   # vd[i] = roundoff_unsigned(vs2[i], uimm)

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

13.5. 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.

The second argument (vector element, scalar value, immediate value) gives the amount to right shift the source as in the narrowing shift instructions, which provides the scaling. 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.

```
# Narrowing unsigned clip
#                                SEW                              2*SEW   SEW
 vnclipu.wv vd, vs2, vs1, vm   # vd[i] = clip(roundoff_unsigned(vs2[i], vs1[i]))
 vnclipu.wx vd, vs2, rs1, vm   # vd[i] = clip(roundoff_unsigned(vs2[i], x[rs1]))
 vnclipu.wi vd, vs2, uimm, vm  # vd[i] = clip(roundoff_unsigned(vs2[i], uimm5))

# Narrowing signed clip
 vnclip.wv vd, vs2, vs1, vm   # vd[i] = clip(roundoff_signed(vs2[i], vs1[i]))
 vnclip.wx vd, vs2, rs1, vm   # vd[i] = clip(roundoff_signed(vs2[i], x[rs1]))
 vnclip.wi vd, vs2, uimm, vm  # vd[i] = clip(roundoff_signed(vs2[i], uimm5))
```

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 EEW of a vector floating-point operand does not correspond to a supported IEEE floating-point type, an illegal instruction exception is raised.

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

Vector floating-point instructions require the presence of base scalar floating-point extensions corresponding to the supported vector floating-point element widths.

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

If the floating-point unit status field `mstatus.FS` is `Off` then any attempt to execute a vector floating-point instruction will raise an illegal instruction exception. Any vector floating-point instruction that modifies any floating-point extension state (i.e., floating-point CSRs or `f` registers) must set `mstatus.FS` to `Dirty`.

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.

> 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 vd[i] = vs2[i] - f[rs1]
    vfrsub.vf vd, vs2, rs1, vm  # Scalar-vector vd[i] = f[rs1] - vs2[i]
```

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] = +(vs1[i] * vd[i]) + vs2[i]
vfmadd.vf vd, rs1, vs2, vm    # vd[i] = +(f[rs1] * vd[i]) + vs2[i]

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

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

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

> 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 destination mask vector register may be the same as the source vector mask register (`v0`).

The compare instructions follow the semantics of the scalar floating-point compare instructions. `vmfeq` and `vmfne` raise the invalid operation exception only on signaling NaN inputs. `vmflt`, `vmfle`, `vmfgt`, and `vmfge` raise the invalid operation exception on both signaling and quiet NaN inputs. `vmfne` writes 1 to the destination element when either operand is NaN, whereas the other comparisons write 0 when either operand is NaN.

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

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

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

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

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

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

```
Comparison      Assembler Mapping             Assembler pseudoinstruction

va < vb         vmflt.vv vd, va, vb, vm
va <= vb        vmfle.vv vd, va, vb, vm
va > vb         vmflt.vv vd, vb, va, vm    vmfgt.vv vd, va, vb, vm
va >= vb        vmfle.vv vd, vb, va, vm    vmfge.vv vd, va, vb, vm

va < f          vmflt.vf vd, va, f, vm
va <= f         vmfle.vf vd, va, f, vm
va > f          vmfgt.vf vd, va, f, vm
va >= f         vmfge.vf vd, va, f, vm

va, vb vector register groups
f      scalar floating-point register
```

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

> C99 floating-point quiet comparisons can be implemented by masking the signaling comparisons when either input is NaN, as follows. When the comparand is a non-NaN constant, the middle two instructions can be omitted.

```
    # Example of implementing isgreater()
    vmfeq.vv v0, va, va        # Only set where A is not NaN.
    vmfeq.vv v1, vb, vb        # Only set where B is not NaN.
    vmand.mm v0, v0, v1        # Only set where A and B are ordered,
    vmfgt.vv v0, va, vb, v0.t  #  so only set flags on ordered values.
```

> In the above sequence, it is tempting to mask the second `vmfeq` instruction and remove the `vmand` instruction, but this more efficient sequence incorrectly fails to raise the invalid exception when an element of `va` contains a quiet NaN and the corresponding element in `vb` contains a signaling NaN.

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 upper (SEW-10) bits of the result are filled with zeros. 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.

The `vfmerge.vfm` instruction is always 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.vfm vd, vs2, rs1, v0  # vd[i] = v0.mask[i] ? f[rs1] : vs2[i]
```

> Like the floating-point computational instructions, when FLEN > SEW, `vfmerge.vfm` substitutes a canonical NaN for `f[rs1]` if the latter is not properly NaN-boxed.

14.14. Vector Floating-Point Move Instruction

The vector floating-point move instruction *splats* a floating-point scalar operand to a vector register group. The instruction copies a scalar f register value to all active elements of a vector register group. This instruction is always unmasked (vm=1). The instruction must have the vs2 field set to v0, with all other values for vs2 reserved.

```
vfmv.v.f vd, rs1  # vd[i] = f[rs1]
```

> The vfmv.v.f instruction shares the encoding with the vfmerge.vfm instruction, but with vm=1 and vs2=v0.

> Like the floating-point computational instructions, when FLEN > SEW, vfmv.v.f substitutes a canonical NaN for f[rs1] if the latter is not properly NaN-boxed.

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

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 use the dynamic rounding mode in frm, except for the rtz variants, which round towards zero.

> The rtz variants are provided to accelerate truncating conversions from floating-point to integer, as is common in languages like C and Java.

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

A set of conversion instructions is 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.rtz.xu.f.v vd, vs2, vm   # Convert float to double-width unsigned integer, truncating
vfwcvt.rtz.x.f.v  vd, vs2, vm   # Convert float to double-width signed integer, truncating.

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.
```

These instructions have the same constraints on vector register overlap as other widening instructions (see Widening Vector Arithmetic Instructions).

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

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

> A full set of floating-point widening conversions is 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.

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

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

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

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

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

vfncvt.f.f.w vd, vs2, vm        # Convert double-width float to single-width float.
vfncvt.rod.f.f.w vd, vs2, vm    # Convert double-width float to single-width float,
                                #  rounding towards odd.
```

These instructions have the same constraints on vector register overlap as other narrowing instructions (see Narrowing Vector Arithmetic Instructions).

> A full set of floating-point widening conversions is not supported as single instructions. Conversions can be implemented in a sequence of halving steps. Results are equivalently rounded and the same exception flags are raised if all but the last halving step use round-towards-odd (`vfncvt.rod.f.f.w`). Only the final step should use the desired rounding mode.

> 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.

The destination vector register can overlap the mask register 'v0' for masked reductions.

> 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 from the source vector register group are excluded from the reduction, but the scalar operand is always included regardless of the mask values.

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

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( vs1[0] , vs2[*] )
vredmaxu.vs vd, vs2, vs1, vm   # vd[0] = maxu( vs1[0] , vs2[*] )
vredmax.vs  vd, vs2, vs1, vm   # vd[0] =  max( vs1[0] , vs2[*] )
vredminu.vs vd, vs2, vs1, vm   # vd[0] = minu( vs1[0] , vs2[*] )
vredmin.vs  vd, vs2, vs1, vm   # vd[0] =  min( vs1[0] , vs2[*] )
vredand.vs  vd, vs2, vs1, vm   # vd[0] =  and( vs1[0] , vs2[*] )
vredor.vs   vd, vs2, vs1, vm   # vd[0] =   or( vs1[0] , vs2[*] )
vredxor.vs  vd, vs2, vs1, vm   # vd[0] =  xor( vs1[0] , vs2[*] )
```

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 = 2*SEW + sum(zero-extend(SEW))

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

15.3.1. Vector Ordered Single-Width Floating-Point Sum Reduction

The `vfredosum` instruction must sum the floating-point values in element order, starting with the scalar in `vs1[0]`--that is, it performs the computation: `(((vs1[0] + vs2[0]) + vs2[1]) + …) vs2[vl-1]`, where each addition operates identically to the scalar floating-point instructions in terms of raising exception flags and generating or propagating special values.

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

When the operation is masked (`vm=0`), the masked-off elements do not affect the result or the exception flags.

> If no elements are active, no additions are performed, so the scalar in `vs1[0]` is simply copied to the destination register, without canonicalizing NaN values and without setting any exception flags. This behavior preserves the handling of NaNs, exceptions, and rounding when autovectorizing a scalar summation loop.

15.3.2. Vector Unordered Single-Width Floating-Point Sum Reduction

The unordered sum reduction instruction, `vfredsum`, provides an implementation more freedom in performing the reduction.

The implementation can produce a result equivalent to a reduction tree composed of binary operator nodes, with the inputs being elements from the source vector register group (`vs2`) and the source scalar value (`vs1[0]`). Each operator in the tree accepts two inputs and produces one result. Each operator first computes an exact sum as a RISC-V scalar floating-point addition with infinite exponent range and precision, then converts this exact sum to a floating-point format with range and precision each at least as great as the element floating-point format indicated by SEW, rounding using the currently active floating-point dynamic rounding mode. A different floating-point range and precision may be chosen for the result of each operator. A node where one input is derived only from elements masked-off or beyond the active vector length may either treat that input as the additive identity of the appropriate EEW or simply copy the other input to its output. The rounded result from the root node in the tree is converted (rounded again, using the dynamic rounding mode) to the standard floating-point format indicated by SEW. An implementation is allowed to add an additional additive identity to the final result.

The additive identity is +0.0 when rounding down (towards -∞) or -0.0 for all other rounding modes.

The reduction tree structure must be deterministic for a given value in `vtype` and `vl`.

> As a consequence of this definition, implementations need not propagate NaN payloads through the reduction tree when no elements are active. In particular, if no elements are active and the scalar input is NaN, implementations are permitted to canonicalize the NaN and, if the NaN is signaling, set the invalid exception flag. Implementations are alternatively permitted to pass through the original NaN and set no exception flags, as with `vfredosum`.

> The `vfredosum` instruction is a valid implementation of the `vfredsum` instruction.

15.3.3. Vector Single-Width Floating Max and Min Reductions

> Floating-point max and min reductions should return the same final value and raise the same 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 sum
vfwredsum.vs vd, vs2, vs1, vm  # Unordered sum
```

The reduction of the SEW-width elements is performed as in the single-width reduction case, with the elements in vs2 promoted to 2*SEW bits before adding to the 2*SEW-bit accumulator.

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. Each element in a mask register is a single bit, 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`. The destination vector register may be the same as either source vector register.

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 handled according to the setting of `vta` in `vtype` (Section Vector Tail Agnostic and Vector Mask Agnostic `vta` and `vma`).

```
    vmand.mm vd, vs2, vs1     # vd[i] =   vs2.mask[i] &&  vs1.mask[i]
    vmnand.mm vd, vs2, vs1    # vd[i] = !(vs2.mask[i] &&  vs1.mask[i])
    vmandnot.mm vd, vs2, vs1  # vd[i] =   vs2.mask[i] && !vs1.mask[i]
    vmxor.mm  vd, vs2, vs1    # vd[i] =   vs2.mask[i] ^^  vs1.mask[i]
    vmor.mm  vd, vs2, vs1     # vd[i] =   vs2.mask[i] ||  vs1.mask[i]
    vmnor.mm  vd, vs2, vs1    # vd[i] = !(vs2.mask[i] ||  vs1.mask[i])
    vmornot.mm  vd, vs2, vs1  # vd[i] =   vs2.mask[i] || !vs1.mask[i]
    vmxnor.mm vd, vs2, vs1    # vd[i] = !(vs2.mask[i] ^^  vs1.mask[i])
```

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

```
    vmmv.m vd, vs  => vmand.mm vd, vs, vs  # Copy 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 vmmv.m instruction was previously called vmcpy.m, but with new layout it is more consistent to name as a "mv" because bits are copied without interpretation. The vmcpy.m assembler psuedo-instruction can be retained for compatibility.

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	
1	0	0	1	vmxnor.mm vd, src1, src2	
0	1	0	1	vmand.mm vd, src2, src2	vmcpy.m vd, src2
1	1	0	1	vmornot.mm vd, src2, src1	
0	0	1	1	vmand.mm vd, src1, src1	vmcpy.m vd, src1
1	0	1	1	vmornot.mm vd, src1, src2	
1	1	1	1	vmxnor.mm vd, vd, vd	vmset.m vd

> 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 vpopc

```
    vpopc.m rd, vs2, vm
```

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

The `vpopc.m` instruction counts the number of mask elements of the active elements of the vector source mask register that have the value 1 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.

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

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

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

```
    vfirst.m rd, vs2, vm
```

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

> Software can assume that any negative value (highest bit set) corresponds to no element found, as vector lengths will never exceed $2^{(XLEN-1)}$ on any implementation.

Traps on `vfirst` are always reported with a `vstart` of 0. The `vfirst` 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 is a `1`, then writes a `0` 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 handled according to the setting of the `vta` bit in `vtype` (Section Vector Tail Agnostic and Vector Mask Agnostic `vta` and `vma`).

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

The destination register cannot overlap either the source register or the mask register ('v0') if the instruction is masked.

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 handled according to the setting of the `vta` bit in `vtype` (Section Vector Tail Agnostic and Vector Mask Agnostic `vta` and `vma`).

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

The destination register cannot overlap either the source register or the mask register ('v0') if the instruction is masked.

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 handled according to the setting of the `vta` bit in `vtype` (Section Vector Tail Agnostic and Vector Mask Agnostic `vta` and `vma`).

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

The destination register cannot overlap either the source register or the mask register ('v0') if the instruction is masked.

16.7. Example using vector mask instructions

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

```
link:example/strcpy.s[]

link:example/strncpy.s[]
```

16.8. Vector Iota Instruction

The `viota.m` instruction reads a source vector mask register and writes to each element of the destination vector register group the sum of all the 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.

```
 viota.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
                       viota.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
                       viota.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 `viota.m` are always reported with a `vstart` of 0, and execution is always restarted from the beginning 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`.

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

The `viota.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
    vle32.v v8, (a1)               # Load input vector
      sub a0, a0, a5               # Decrement number done
      slli a5, a5, 2               # Multiply by four bytes
    vmsne.vi v0, v8, 0             # Locate non-zero values
      add a1, a1, a5               # Bump input pointer
    vpopc.m a5, v0                 # Count number of elements set in v0
    viota.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
    vsuxei32.v v8, (a2), v16, v0.t # Scatter using scaled viota 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.

The `vs2` field of the instruction must be set to `v0`, otherwise the encoding is *reserved*.

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.

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

> Microarchitectures can implement `vid.v` instruction using the same datapath as `viota.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 Scalar Move Instructions

The integer scalar read/write instructions transfer a single value between a scalar `x` register and element 0 of a vector register. The instructions ignore LMUL and vector register groups.

```
vmv.x.s rd, vs2  # x[rd] = vs2[0] (rs1=0)
vmv.s.x vd, rs1  # vd[0] = x[rs1] (vs2=0)
```

The `vmv.x.s` instruction copies a single SEW-wide element from index 0 of the source vector register to a destination integer register. If SEW > XLEN, the least-significant XLEN bits are transferred and the upper SEW-XLEN bits are ignored. If SEW < XLEN, the value is sign-extended to XLEN bits.

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 sign-extended to SEW bits. The other elements in the destination vector register (0 < index < VLEN/SEW) are unchanged. If `vstart` ≥ `vl`, no operation is performed and the destination register is not updated.

> As a consequence, when `vl`=0, no elements are updated in the destination vector register group, regardless of `vstart`.

The encodings corresponding to the masked versions (`vm=0`) of `vmv.x.s` and `vmv.s.x` are reserved.

17.2. 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  # f[rd] = vs2[0] (rs1=0)
vfmv.s.f vd, rs1  # vd[0] = f[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, `vfmv.f.s` substitutes an FLEN-bit canonical NaN if the element value is not correctly NaN-boxed for FLEN. If SEW < FLEN, the value is NaN-boxed (1-extended) to FLEN bits.

The `vfmv.s.f` instruction copies the scalar floating-point register to element 0 of the destination vector register. If SEW < FLEN and the value is not correctly NaN-boxed for SEW bits, an SEW-bit canonical NaN is substituted. If SEW > FLEN, the value is NaN-boxed (1-extended) to SEW bits. The other elements in the destination vector register (0 < index < VLEN/SEW) are unchanged. If `vstart` ≥ `vl`, no operation is performed and the destination register is not updated.

> As a consequence, when `vl`=0, no elements are updated in the destination vector register group, regardless of `vstart`.

The encodings corresponding to the masked versions (`vm=0`) of `vfmv.f.s` and `vfmv.s.f` are reserved.

17.3. Vector Slide Instructions

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

> 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`, `v[f]slide1up`, and `v[f]slide1down` instructions, if `vstart` ≥ `vl`, the instruction performs no operation and leaves the destination vector register unchanged.

> As a consequence, when `vl`=0, no elements are updated in the destination vector register group, regardless of `vstart`.

The slide instructions may be masked, with mask element *i* controlling whether *destination* element *i* is written.

17.3.1. Vector Slideup Instructions

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

For `vslideup`, the value in `vl` specifies the maximum 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. If XLEN > SEW, *OFFSET* is *not* truncated to SEW bits. Destination elements *OFFSET* through `vl-1` are written if unmasked and if *OFFSET* < `vl`.

```
  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[i] enabled,
                                                  unchanged if not
                  vl <= i < VLMAX                Tail elements, unchanged
```

The destination vector register group for `vslideup` cannot overlap the source vector register group, otherwise an illegal instruction exception is raised.

> The non-overlap constraint avoids WAR hazards on the input vectors during execution, and enables restart with non-zero `vstart`.

17.3.2. 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+uimm]
```

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. If XLEN > SEW, *OFFSET* is *not* truncated to SEW bits.

```
    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[i] enabled, unchanged if not
                    vl <= i < VLMAX          Unchanged
```

17.3.3. 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]
vfslide1up.vf vd, vs2, rs1, vm        # vd[0]=f[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 sign-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 unchanged.

```
   vslide1up behavior

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

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

The `vfslide1up` instruction is defined analogously, but sources its scalar argument from an `f` register. If SEW < FLEN and the value is not correctly NaN-boxed for SEW bits, an SEW-bit canonical NaN is substituted. If FLEN < SEW, the scalar value is NaN-boxed (one-extended) to SEW bits.

17.3.4. 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 unchanged.

```
vslide1down.vx  vd, vs2, rs1, vm      # vd[i] = vs2[i+1], vd[vl-1]=x[rs1]
vfslide1down.vf vd, vs2, rs1, vm      # vd[i] = vs2[i+1], vd[vl-1]=f[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 sign-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[i] enabled, unchanged if not
         vstart <= i = vl-1    vd[vl-1] = x[rs1] if mask[i] enabled, unchanged if not
             vl <= i < VLMAX   unchanged
```

The `vfslide1down` instruction is defined analogously, but sources its scalar argument from an `f` register. If SEW < FLEN and the value is not correctly NaN-boxed for SEW bits, an SEW-bit canonical NaN is substituted. If FLEN < SEW, the scalar value is NaN-boxed (one-extended) to SEW bits.

> 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.

17.4. 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 the remaining elements past `vl` are handled according to the current tail policy (Section Vector Tail Agnostic and Vector Mask Agnostic `vta` and `vma`). 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. The index value in the scalar register and the immediate are treated as unsigned integers. If XLEN > SEW, the index value is *not* truncated to SEW bits.

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

```
vrgather.vx vd, vs2, rs1, vm # vd[i] = (x[rs1] >= VLMAX) ? 0 : vs2[x[rs1]]
vrgather.vi vd, vs2, uimm, vm # vd[i] = (uimm >= VLMAX) ? 0 : vs2[uimm]
```

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

> When SEW=8, `vrgather.vv` can only reference vector elements 0-255.

17.5. 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`. The remaining elements of vd are treated as tail elements according to the current tail policy (Section Vector Tail Agnostic and Vector Mask Agnostic vta and vma).

```
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
    1 2 3 4 5 6 7 8 9   v2

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

vcompress is encoded as an unmasked instruction (vm=1). The equivalent masked instruction (vm=0) is reserved.

The destination vector register group cannot overlap the source vector register group or the source 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.

> 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.

17.6. Whole Vector Register Move

The vmv<nr>r.v instructions copy whole vector registers (i.e., all VLEN bits) and can copy whole vector register groups. The instructions operate as if EEW=8, EMUL = nr, effective length evl=VLEN/8 * EMUL, regardless of current settings in vtype and vl.

> These instructions are intended to aid compilers to shuffle vector registers without needing to know or change vl or vtype.

> The usual property that no elements are written if vstart ≥ vl does not apply to these instructions.

The instruction is encoded as an OPIVI instruction. The number of vector registers to copy is encoded in the low three bits of the simm field using the same encoding as the nf field for memory instructions, i.e., simm = nr-1. nr must be 1, 2, 4, or 8.

> A future extension may support other numbers of registers to be moved. Values of simm other than 0, 1, 3, and 7 are currently reserved.

> The instruction uses the same funct6 encoding as the vsmul instruction but with an immediate operand, and only the unmasked version (vm=1). This encoding is chosen as it is close to the related vmerge encoding, and it is unlikely the vsmul instruction would benefit from an immediate form.

```
vmv<nr>r.v vd, vs2  # General form

vmv1r.v v1, v2   #  Copy v1=v2
vmv2r.v v10, v12 #  Copy v10=v12; v11=v13
vmv4r.v v4, v8   #  Copy v4=v8; v5=v9; v6=v10; v7=v11
vmv8r.v v0, v8   #  Copy v0=v8; v1=v9; ...;  v7=v15
```

The source and destination vector register numbers must be aligned appropriately for the vector register group size.

> A future extension may relax the vector register alignment restrictions.

> If vd is equal to vs2 the instruction is a NOP.

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.

> 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.

> 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.

> 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 EDIV extension is currently not planned to be part of the base "V" extension, and will change substantially from the current sketch.

> This section has not been updated to account for new mask format in v0.9.

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.

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

Table 16. `vtype` register layout

Bits	Name	Description
XLEN-1	vill	Illegal value if set
XLEN-2:10		Reserved (write 0)
9:8	vediv[1:0]	Used by EDIV extension
7	vma	Mask agnostic
6	vta	Tail agnostic
5	vlmul[2]	Vector register group multiplier (LMUL) setting (fractional)
4:2	vsew[2:0]	Standard element width (SEW) setting
1:0	vlmul[1:0]	Vector register group multiplier (LMUL) setting

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

The assembly syntax for vsetvli has additional options added to encode the EDIV options.

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

vsetvli t0, a0, e32,m2,d4   # SEW=32, LMUL=2, EDIV=4
```

SEW	EDIV	Sub-element	Integer accumulator		FP sum/dot accumulator		
			sum	dot	FLEN=32	FLEN=64	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.

> 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.

> 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 (`vpopc`), find-first and related instructions (`vfirst`, `vmsbf`, `vmsif`, `vmsof`), iota (`viota`), 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 Add with Carry/Subtract with Borrow Reserved under EDIV>1

For EDIV > 1, `vadc`, `vmadc`, `vsbc`, `vmsbc` are reserved.

19.2.3. Vector Reduction Instructions under EDIV

Vector single-width integer sum reduction instructions are reserved under EDIV>1. Other vector single-width reductions and vector widening integer sum reduction instructions now operate independently on all elements in a vector, reducing sub-element values within an element to an element-wide result.

The scalar input is taken from the least-significant bits of the second operand, with the number of bits equal to the number of significant result bits (i.e., for sum and dot reductions, the number of bits are given in table above, for non-sum and non-dot reductions, equal to the element size).

```
# 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
vwredsum.vs 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][15:0]

# Find maximum among sub-elements
vredmax.vs v5, v6, v7 # v5[i][7:0] = max(v6[i][31:24], v6[i][23:16],
                      #                   v6[i][15:8], v6[i][7:0], v7[i][7:0])
```

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 widening 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.

Single-width floating-point reductions produce a final result that is SEW/EDIV bits wide.

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

19.2.4. 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
```

> 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.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]

# 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

Integer					**Integer**				**FP**			
funct3					funct3				funct3			
OPIVV	V				OP-MVV	V			OPFVV	V		
OPIVX		X			OP-MVX		X		OPFVF		F	
OPIVI			I									

funct6					**funct6**				**funct6**			
000000	V	X	I	vadd	000000	V		vred-sum	000000	V	F	vfadd
000001					000001	V		vredand	000001	V		vfred-sum
000010	V	X		vsub	000010	V		vredor	000010	V	F	vfsub
000011		X	I	vrsub	000011	V		vredxor	000011	V		vfredo-sum
000100	V	X		vminu	000100	V		vred-minu	000100	V	F	vfmin
000101	V	X		vmin	000101	V		vredmin	000101	V		vfred-min
000110	V	X		vmaxu	000110	V		vred-maxu	000110	V	F	vfmax
000111	V	X		vmax	000111	V		vred-max	000111	V		vfred-max
001000					001000	V	X	vaaddu	001000	V	F	vfsgnj
001001	V	X	I	vand	001001	V	X	vaadd	001001	V	F	vfsgnjn
001010	V	X	I	vor	001010	V	X	vasubu	001010	V	F	vfsgnjx
001011	V	X	I	vxor	001011	V	X	vasub	001011			
001100	V	X	I	vrgath-er	001100				001100			
001101					001101				001101			
001110		X	I	vslide-up	001110		X	vs-lide1up	001110		F	vfs-lide1up
001111		X	I	vslide-down	001111		X	vslide1-down	001111		F	vfs-lide1-down

funct6					funct6				funct6			
010000	V	X	I	vadc	010000	V		VWXU-NARY0	010000	V		VWFUNARY0
					010000		X	VRXU-NARY0	010000		F	VRFUNARY0
010001	V	X	I	vmadc	010001				010001			
010010	V	X		vsbc	010010	V		VXU-NARY0	010010	V		VFUNARY0
010011	V	X		vmsbc	010011				010011	V		VFUNARY1
010100					010100	V		VMU-NARY0	010100			
010101					010101				010101			
010110					010110				010110			
010111	V	X	I	vmerge/vmv	010111	V		vcom-press	010111		F	vfmerge.vf/vfmv
011000	V	X	I	vmseq	011000	V		vmand-not	011000	V	F	vmfeq
011001	V	X	I	vmsne	011001	V		vmand	011001	V	F	vmfle
011010	V	X		vmsltu	011010	V		vmor	011010			
011011	V	X		vmslt	011011	V		vmxor	011011	V	F	vmflt
011100	V	X	I	vmsleu	011100	V		vmornot	011100	V	F	vmfne
011101	V	X	I	vmsle	011101	V		vmnand	011101		F	vmfgt
011110		X	I	vmsgtu	011110	V		vmnor	011110			
011111		X	I	vmsgt	011111	V		vmxnor	011111		F	vmfge

funct6					funct6				funct6			
100000	V	X	I	vsaddu	100000	V	X	vdivu	100000	V	F	vfdiv
100001	V	X	I	vsadd	100001	V	X	vdiv	100001		F	vfrdiv
100010	V	X		vssubu	100010	V	X	vremu	100010			
100011	V	X		vssub	100011	V	X	vrem	100011			
100100					100100	V	X	vmulhu	100100	V	F	vfmul
100101	V	X	I	vsll	100101	V	X	vmul	100101			
100110					100110	V	X	vmulhsu	100110			
100111	V	X		vsmul	100111	V	X	vmulh	100111		F	vfrsub
			I	vmv<nf>r								
101000	V	X	I	vsrl	101000				101000	V	F	vfmadd
101001	V	X	I	vsra	101001	V	X	vmadd	101001	V	F	vfnmadd
101010	V	X	I	vssrl	101010				101010	V	F	vfmsub
101011	V	X	I	vssra	101011	V	X	vnmsub	101011	V	F	vfnmsub
101100	V	X	I	vnsrl	101100				101100	V	F	vfmacc
101101	V	X	I	vnsra	101101	V	X	vmacc	101101	V	F	vfnmacc
101110	V	X	I	vnclipu	101110				101110	V	F	vfmsac
101111	V	X	I	vnclip	101111	V	X	vnmsac	101111	V	F	vfnmsac

funct6					funct6				funct6			
110000	V			vwredsumu	110000	V	X	vwaddu	110000	V	F	vfwadd
110001	V			vwredsum	110001	V	X	vwadd	110001	V		vfwredsu
110010					110010	V	X	vwsubu	110010	V	F	vfwsub
110011					110011	V	X	vwsub	110011	V		vfwredosum
110100					110100	V	X	vwaddu.w	110100	V	F	vfwadd.w
110101					110101	V	X	vwadd.w	110101			
110110					110110	V	X	vwsubu.w	110110	V	F	vfwsub.w
110111					110111	V	X	vwsub.w	110111			
111000	V			vdotu	111000	V	X	vwmulu	111000	V	F	vfwmul
111001	V			vdot	111001				111001	V		vfdot
111010					111010	V	X	vwmulsu	111010			
111011					111011	V	X	vwmul	111011			
111100	V	X		vqmaccu	111100	V	X	vwmaccu	111100	V	F	vfwmacc
111101	V	X		vqmacc	111101	V	X	vwmacc	111101	V	F	vfwnmacc
111110		X		vqmaccus	111110		X	vwmaccus	111110	V	F	vfwmsac
111111	V	X		vqmaccsu	111111	V	X	vwmaccsu	111111	V	F	vfwnmsac

Table 17. VRXUNARY0 encoding space

vs2	
00000	vmv.s.x

Table 18. VWXUNARY0 encoding space

vs1	
00000	vmv.x.s
10000	vpopc
10001	vfirst

Table 19. VXUNARY0 encoding space

vs1	
00010	vzext.vf8
00011	vsext.vf8
00100	vzext.vf4
00101	vsext.vf4
00110	vzext.vf2
00111	vsext.vf2

Table 20. VRFUNARY0 encoding space

vs2	
00000	vfmv.s.f

Table 21. VWFUNARY0 encoding space

vs1	
00000	vfmv.f.s

Table 22. VFUNARY0 encoding space

vs1	name
single-width converts	
00000	vfcvt.xu.f.v
00001	vfcvt.x.f.v
00010	vfcvt.f.xu.v
00011	vfcvt.f.x.v
00110	vfcvt.rtz.xu.f.v
00111	vfcvt.rtz.x.f.v
widening converts	
01000	vfwcvt.xu.f.v
01001	vfwcvt.x.f.v
01010	vfwcvt.f.xu.v
01011	vfwcvt.f.x.v
01100	vfwcvt.f.f.v
01110	vfwcvt.rtz.xu.f.v
01111	vfwcvt.rtz.x.f.v
narrowing converts	
10000	vfncvt.xu.f.w
10001	vfncvt.x.f.w
10010	vfncvt.f.xu.w
10011	vfncvt.f.x.w
10100	vfncvt.f.f.w
10101	vfncvt.rod.f.f.w
10110	vfncvt.rtz.xu.f.w
10111	vfncvt.rtz.x.f.w

Table 23. VFUNARY1 encoding space

vs1	name
00000	vfsqrt.v
10000	vfclass.v

Table 24. VMUNARY0 encoding space

vs1	
00001	vmsbf
00010	vmsof
00011	vmsif
10000	viota
10001	vid

Appendix A: Vector Assembly Code Examples

The following are provided as non-normative text to help explain the vector ISA.

A.1. Vector-vector add example

```
link:example/vvaddint32.s[]
```

A.2. Example with mixed-width mask and compute.

```
# Code using one width for predicate and different width for masked
# compute.
#   int8_t a[]; int32_t b[], c[];
#   for (i=0;  i<n; i++) { b[i] =  (a[i] < 5) ? c[i] : 1; }
#
# Mixed-width code that keeps SEW/LMUL=8
  loop:
    vsetvli a4, a0, e8,m1  # Byte vector for predicate calc
    vle8.v v1, (a1)               # Load a[i]
      add a1, a1, a4              # Bump pointer.
    vmslt.vi v0, v1, 5            # a[i] < 5?

    vsetvli x0, a0, e32,m4 # Vector of 32-bit values.
      sub a0, a0, a4              # Decrement count
    vmv.v.i v4, 1                 # Splat immediate to destination
    vle32.v v4, (a3), v0.t        # Load requested elements of C.
      sll t1, a4, 2
      add a3, a3, t1              # Bump pointer.
    vse32.v v4, (a2)              # Store b[i].
      add a2, a2, t1              # Bump pointer.
      bnez a0, loop               # Any more?
```

A.3. Memcpy example

```
link:example/memcpy.s[]
```

A.4. Conditional example

```
# (int16) z[i] = ((int8) x[i] < 5) ? (int16) a[i] : (int16) b[i];
#

loop:
    vsetvli t0, a0, e8,m1   # Use 8b elements.
    vle8.v v0, (a1)         # Get x[i]
      sub a0, a0, t0        # Decrement element count
      add a1, a1, t0        # x[i] Bump pointer
    vmslt.vi v0, v0, 5      # Set mask in v0
      slli t0, t0, 1        # Multiply by 2 bytes
    vsetvli t0, a0, e16,m2  # Use 16b elements.
    vle16.v v1, (a2), v0.t  # z[i] = a[i] case
    vmnot.m v0, v0          # Invert v0
      add a2, a2, t0        # a[i] bump pointer
    vle16.v v1, (a3), v0.t  # z[i] = b[i] case
      add a3, a3, t0        # b[i] bump pointer
    vse16.v v1, (a4)        # Store z
      add a4, a4, t0        # z[i] bump pointer
      bnez a0, loop
```

A.5. SAXPY example

```
link:example/saxpy.s[]
```

A.6. SGEMM example

```
link:example/sgemm.S[]
```

Appendix B: Calling Convention

In the RISC-V psABI, the vector registers `v0`-`v31` are all caller-saved. The `vstart`, `vl`, and `vtype` CSRs are also caller-saved.

The `vxrm` and `vxsat` fields have thread storage duration.

Executing a system call causes `v0`-`v31` to become unspecified.

> This scheme allows system calls that cause context switches to avoid saving and later restoring the vector registers.

> The values that `v0`-`v31` assume after a system call cannot expose information from other processes, so typically the registers will either remain intact or will be zeroed.