

Chapter 30. "V" Standard Extension for Vector Operations, Version 1.0



The base vector extension is intended to provide general support for data-parallel execution within the 32-bit instruction encoding space, with later vector extensions supporting richer functionality for certain domains.

30.1. Introduction

This spec includes the complete set of currently frozen vector instructions. Other instructions that have been considered during development but are not present in this document are not included in the review and ratification process, and may be completely revised or abandoned. [Section 30.18](#) lists the standard vector extensions and which instructions and element widths are supported by each extension.

30.2. Implementation-defined Constant Parameters

Each hart supporting a vector extension defines two parameters:

1. The maximum size in bits of a vector element that any operation can produce or consume, $ELEN \geq 8$, which must be a power of 2.
2. The number of bits in a single vector register, $VLEN \geq ELEN$, which must be a power of 2, and must be no greater than 2^{16} .

Standard vector extensions ([Section 30.18](#)) and architecture profiles may set further constraints on $ELEN$ and $VLEN$.



Future extensions may allow $ELEN > VLEN$ by holding one element using bits from multiple vector registers, but this current proposal does not include this option.



The upper limit on $VLEN$ allows software to know that indices will fit into 16 bits (largest $VLMAX$ of 65,536 occurs for $LMUL=8$ and $SEW=8$ with $VLEN=65,536$). Any future extension beyond 64Kib per vector register will require new configuration instructions such that software using the old configuration instructions does not see greater vector lengths.

The vector extension supports writing binary code that under certain constraints will execute portably on harts with different values for the $VLEN$ parameter, provided the harts support the required element types and instructions.



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



In general, thread contexts with active vector state cannot be migrated during execution between harts that have any difference in $VLEN$ or $ELEN$ parameters.

30.3. 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 43. New vector CSRs

Address	Privilege	Name	Description
0x008	URW	vstart	Vector start element index
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)



The four CSR numbers 0x00B–0x00E are tentatively reserved for future vector CSRs, some of which may be mirrored into vcsr.

30.3.1. 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.

30.3.2. 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 mstatus.VS is set to Off.

When mstatus.VS is set to Initial or Clean, executing any instruction that changes vector state, including the vector CSRs, will change mstatus.VS to Dirty. Implementations may also change mstatus.VS from Initial or Clean to Dirty at any time, even when there is no change in vector state.



Accurate setting of mstatus.VS is an optimization. Software will typically use VS to reduce context-swap overhead.

If mstatus.VS is Dirty, mstatus.SD is 1; otherwise, mstatus.SD is set in accordance with existing specifications.

Implementations may have a writable misa.v field. Analogous to the way in which the floating-point unit is handled, the mstatus.VS field may exist even if misa.V is clear.



Allowing mstatus.VS to exist when misa.V is clear, enables vector emulation and simplifies handling of mstatus.VS in systems with writable misa.V.

30.3.3. Vector Context Status in vsstatus

When the hypervisor extension is present, a vector context status field, vs, is added to vsstatus[10:9]. It is defined analogously to the floating-point context status field, fs.

When $V=1$, both `vsstatus.VS` and `mstatus.VS` are in effect: attempts to execute any vector instruction, or to access the vector CSRs, raise an illegal-instruction exception when either field is set to Off.

When $V=1$ and neither `vsstatus.VS` nor `mstatus.VS` is set to Off, executing any instruction that changes vector state, including the vector CSRs, will change both `mstatus.VS` and `vsstatus.VS` to Dirty. Implementations may also change `mstatus.VS` or `vsstatus.VS` from Initial or Clean to Dirty at any time, even when there is no change in vector state.

If `vsstatus.VS` is Dirty, `vsstatus.SD` is 1; otherwise, `vsstatus.SD` is set in accordance with existing specifications.

If `mstatus.VS` is Dirty, `mstatus.SD` is 1; otherwise, `mstatus.SD` is set in accordance with existing specifications.

For implementations with a writable `misa.V` field, the `vsstatus.VS` field may exist even if `misa.V` is clear.

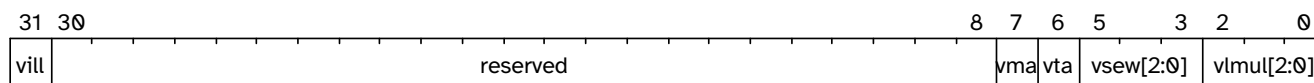
30.3.4. Vector Type (`vtype`) Register

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 `vset{i}vl{i}` instructions. The vector type determines the organization of elements in each vector register, and how multiple vector registers are grouped. The `vtype` register also indicates how masked-off elements and elements past the current vector length in a vector result are handled.



Allowing updates only via the `vset{i}vl{i}` instructions simplifies maintenance of the `vtype` register state.

The `vtype` register has five fields, `vill`, `vma`, `vta`, `vsew[2:0]`, and `vlmul[2:0]`. Bits `vtype[XLEN-2:8]` should be written with zero, and non-zero values in this field are reserved.



This diagram shows the layout for RV32 systems, whereas in general `vill` should be at bit `XLEN-1`.

Table 44. `vtype` register layout

Bits	Name	Description
XLEN-1	<code>vill</code>	Illegal value if set
XLEN-2:8	0	Reserved if non-zero
7	<code>vma</code>	Vector mask agnostic
6	<code>vta</code>	Vector tail agnostic
5:3	<code>vsew[2:0]</code>	Selected element width (SEW) setting
2:0	<code>vlmul[2:0]</code>	Vector register group multiplier (LMUL) setting



A small implementation supporting $ELEN=32$ requires only seven bits of state 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 internally encoded using the illegal 64-bit combination in `vsew[1:0]` without requiring an additional storage bit to hold `vill`.



Further standard and custom vector extensions may extend these fields to support a greater variety of data types.



The primary motivation for the `vtype` CSR is to allow the vector instruction set to fit into a 32-bit instruction encoding space. A separate `vset{i}vl{i}` instruction can be used to set `vl` and/or `vtype` fields before execution of a vector instruction, and implementations may choose to fuse these two instructions into a single internal vector microop. In many cases, the `vl` and `vtype` values can be reused across multiple instructions, reducing the static and dynamic instruction overhead from the `vset{i}vl{i}` instructions. It is anticipated that a future extended 64-bit instruction encoding would allow these fields to be specified statically in the instruction encoding.

30.3.4.1. Vector Selected Element Width (`vsew[2:0]`)

The value in `vsew` sets the dynamic selected element width (SEW). By default, a vector register is viewed as being divided into `VLEN/SEW` elements.

Table 45. `vsew[2:0]` (selected element width) encoding

<code>vsew[2:0]</code>			SEW
0	0	0	8
0	0	1	16
0	1	0	32
0	1	1	64
1	X	X	Reserved



While it is anticipated the larger `vsew[2:0]` encodings (100-111) will be used to encode larger SEW, the encodings are formally reserved at this point.

Table 46. Example `VLEN = 128 bits`

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

The supported element width may vary with LMUL.



The current set of standard vector extensions do not vary supported element width with LMUL. Some future extensions may support larger SEWs only when bits from multiple vector registers are combined using LMUL. In this case, software that relies on large SEW should attempt to use the largest LMUL, and hence the fewest vector register groups, to increase the number of implementations on which the code will run. The `vill` bit in `vtype` should be checked after setting `vtype` to see if the configuration is supported, and an alternate code path should be provided if it is not. Alternatively, a profile can mandate the minimum SEW at each LMUL setting.

30.3.4.2. 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 can be used to provide greater execution efficiency for longer application vectors, but the main reason for their inclusion is to allow double-width or larger elements to be operated on with the same vector length as single-width elements. 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. Implementations must support *LMUL* integer values of 1, 2, 4, and 8.



The vector architecture includes instructions that take multiple source and destination vector operands with different element widths, but the same number of elements. The effective LMUL (EMUL) of each vector operand is determined by the number of registers required to hold the elements. For example, for a widening add operation, such as add 32-bit values to produce 64-bit results, a double-width result requires twice the LMUL of the single-width inputs.

LMUL can also be a fractional value, reducing the number of bits used in a single vector register. Fractional *LMUL* is used to increase the number of effective usable vector register groups when operating on mixed-width values.



*With only integer LMUL values, a loop operating on a range of sizes would have to allocate at least one whole vector register (*LMUL*=1) for the narrowest data type and then would consume multiple vector registers (*LMUL*>1) to form a vector register group for each wider vector operand. This can limit the number of vector register groups available. With fractional LMUL, the widest values need occupy only a single vector register while narrower values can occupy a fraction of a single vector register, allowing all 32 architectural vector register names to be used for different values in a vector loop even when handling mixed-width values. Fractional LMUL implies portions of vector registers are unused, but in some cases, having more shorter register-resident vectors improves efficiency relative to fewer longer register-resident vectors.*

Implementations must provide fractional *LMUL* settings that allow the narrowest supported type to occupy a fraction of a vector register corresponding to the ratio of the narrowest supported type's width to that of the largest supported type's width. In general, the requirement is to support $LMUL \geq SEW_{MIN}/ELEN$, where SEW_{MIN} is the narrowest supported SEW value and *ELEN* is the widest supported SEW value. In the standard extensions, $SEW_{MIN}=8$. For standard vector extensions with *ELEN*=32, fractional *LMUL*s of 1/2 and 1/4 must be supported. For standard vector extensions with *ELEN*=64, fractional *LMUL*s of 1/2, 1/4, and 1/8 must be supported.



When $LMUL < SEW_{MIN}/ELEN$, 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. For example, with $VLEN=ELEN=32$, and $SEW_{MIN}=8$, an LMUL of 1/8 would only provide four bits of storage in a vector register.

For a given supported fractional *LMUL* setting, implementations must support *SEW* settings between SEW_{MIN} and $LMUL * ELEN$, inclusive.

The use of *vtype* encodings with $LMUL < SEW_{MIN}/ELEN$ is reserved, but implementations can set *vill* if they do not support these configurations.



Requiring all implementations to set `vll` in this case would prohibit future use of this case in an extension, so to allow for a future definition of $LMUL < SEW_{MIN}/ELEN$ behavior, we consider the use of this case to be reserved.



It is recommended that assemblers provide a warning (not an error) if a `vsetvli` instruction attempts to write an $LMUL < SEW_{MIN}/ELEN$.

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

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[2:0]			LMUL	#groups	VLMAX	Registers grouped with register <i>n</i>
1	0	0	-	-	-	reserved
1	0	1	1/8	32	$VLEN/SEW/8$	<i>v n</i> (single register in group)
1	1	0	1/4	32	$VLEN/SEW/4$	<i>v n</i> (single register in group)
1	1	1	1/2	32	$VLEN/SEW/2$	<i>v n</i> (single register in group)
0	0	0	1	32	$VLEN/SEW$	<i>v n</i> (single register in group)
0	0	1	2	16	$2 * VLEN/SEW$	<i>v n</i> , <i>v n+1</i>
0	1	0	4	8	$4 * VLEN/SEW$	<i>v n</i> , ..., <i>v n+3</i>
0	1	1	8	4	$8 * VLEN/SEW$	<i>v n</i> , ..., <i>v n+7</i>

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 an $LMUL=2$ vector register group with an odd-numbered vector register are reserved.

When $LMUL=4$, the vector register group contains four vector registers, and instructions specifying an $LMUL=4$ vector register group using vector register numbers that are not multiples of four are reserved.

When $LMUL=8$, the vector register group contains eight vector registers, and instructions specifying an $LMUL=8$ vector register group using register numbers that are not multiples of eight are reserved.

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

30.3.4.3. 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 30.5.4](#).

All systems must support all four options:

<code>vta</code>	<code>vma</code>	Tail Elements	Inactive Elements
0	0	undisturbed	undisturbed
0	1	undisturbed	agnostic

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

Mask destination tail elements are always treated as tail-agnostic, regardless of the setting of vta.

When a set is marked undisturbed, the corresponding set of destination elements in a vector register group retain the value they previously held.

When a set is marked agnostic, the corresponding set of destination elements in any vector 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 accommodate machines with vector register renaming. 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 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 settings and simply execute all vector instructions using the undisturbed policy. The vta and vma state bits must still be provided in vtype for compatibility 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 accommodate 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.

In addition, except for mask load instructions, any element in the tail of a mask result can also be written with the value the mask-producing operation would have calculated with vl=VLMAX. Furthermore, for mask-logical instructions and vmsbf.m, vmsif.m, vmsof.m mask-manipulation instructions, any element in the tail of the result can be written with the value the mask-producing operation would have calculated with vl=VLEN, SEW=8, and LMUL=8 (i.e., all bits of the mask register can be overwritten).



Mask tails are always treated as agnostic to reduce complexity of managing mask data, which can be written at bit granularity. There appears to be little software need to support tail-undisturbed for mask register values. Allowing mask-generating

instructions to write back the result of the instruction avoids the need for logic to mask out the tail, except mask loads cannot write memory values to destination mask tails as this would imply accessing memory past software intent.

The assembly syntax adds two mandatory 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
```



Prior to v0.9, when these flags were not specified on a `vsetvli`, they defaulted to mask-undisturbed/tail-undisturbed. The use of `vsetvli` without these flags is deprecated, however, and specifying a flag setting is now mandatory. The default should perhaps be tail-agnostic/mask-agnostic, so software has to specify when it cares about the non-participating elements, but given the historical meaning of the instruction prior to introduction of these flags, it was decided to always require them in future assembly code.

30.3.4.4. Vector Type Illegal (`vll`)

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



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



`vset{i}vl{i}` and whole register loads and stores do not depend upon `vtype`.

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

30.3.5. Vector Length (`vl`) Register

The `XLEN`-bit-wide read-only `vl` CSR can only be updated by the `vset{i}vl{i}` 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 with results from a vector instruction, as further detailed in [Section 30.5.4](#).



The number of bits implemented in `vl` depends on the implementation's maximum vector length of the smallest supported type. The smallest vector implementation with `VLEN=32` and supporting `SEW=8` would need at least six bits in `vl` to hold the

values 0-32 ($VLEN=32$, with $LMUL=8$ and $SEW=8$, yields $VLMAX=32$).

30.3.6. Vector Byte Length ($vlenb$) Register

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.

30.3.7. Vector Start Index ($vstart$) Register

The $XLEN$ -bit-wide read-write $vstart$ CSR specifies the index of the first element to be executed by a vector instruction, as described in [Section 30.5.4](#).

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 $vset\{i\}vl\{i\}$, reset the $vstart$ CSR to zero.

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

The $vstart$ CSR is defined to have only enough writable bits to hold the largest element index (one less than the maximum $VLMAX$).



*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 use of $vstart$ values greater than the largest element index for the current $vtype$ setting is reserved.



It is recommended that implementations trap if $vstart$ is out of bounds. It is not required to trap, as a possible future use of upper $vstart$ bits is to store imprecise trap information.

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.



When migrating a software thread between two harts with different microarchitectures, the `vstart` value might not be supported by the new hart microarchitecture. The runtime on the receiving hart might then have to emulate instruction execution up to the next supported `vstart` element position. Alternatively, migration events can be constrained to only occur at mutually supported `vstart` locations.

30.3.8. Vector Fixed-Point Rounding Mode (`vfrm`) Register

The vector fixed-point rounding-mode register holds a two-bit read-write rounding-mode field in the least-significant bits (`vfrm[1:0]`). The upper bits, `vfrm[XLEN-1:2]`, should be written as zeros.

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`.



A new rounding mode can be set while saving the original rounding mode using a single `csrwi` instruction.

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 $(v \gg d) + r$, where r depends on the rounding mode as specified in the following table.

Table 47. `vfrm` encoding

<code>vfrm[1:0]</code>		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	$v[d-1] \& (v[d-2:0] \neq 0 \mid v[d])$
1	0	rdn	round-down (truncate)	0
1	1	rod	round-to-odd (OR bits into LSB, aka "jam")	$!v[d] \& v[d-1:0] \neq 0$

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.

30.3.9. Vector Fixed-Point Saturation Flag (`vxsat`)

The `vxsat` CSR has a single read-write least-significant bit (`vxsat[0]`) that indicates if a fixed-point instruction has had to saturate an output value to fit into a destination format. Bits `vxsat[XLEN-1:1]` should be written as zeros.

The `vxsat` bit is mirrored in `vcsr`.

30.3.10. Vector Control and Status (`vcsr`) Register

The `vfrm` and `vxsat` separate CSRs can also be accessed via fields in the *XLEN*-bit-wide vector control and status CSR, `vcsr`.

Table 48. `vcsr` layout

Bits	Name	Description
XLEN-1:3		Reserved
2:1	<code>vfrm[1:0]</code>	Fixed-point rounding mode
0	<code>vxsat</code>	Fixed-point accrued saturation flag

30.3.11. 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`, `vfrm`, `vxsat` CSRs can have arbitrary values at reset.



Most uses of the vector unit will require an initial `vset{i}vl{i}`, which will reset `vstart`. The `vfrm` and `vxsat` fields should be reset explicitly in software before use.

The vector registers can have arbitrary values at reset.

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

The mapping was chosen to provide the simplest and most portable model for software, but might appear to incur large wiring cost for wider vector datapaths on certain operations. The vector instruction set was expressly designed to support implementations that internally rearrange vector data for different SEW to reduce datapath wiring costs, while externally preserving the simple software model.



For example, microarchitectures can track the EEW with which a vector register was written, and then insert additional scrambling operations to rearrange data if the register is accessed with a different EEW.

30.4.1. Mapping for LMUL = 1

When 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=32b

Byte 3 2 1 0

SEW=8b 3 2 1 0

SEW=16b 1 0

SEW=32b 0

VLEN=64b

Byte 7 6 5 4 3 2 1 0

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

SEW=16b 3 2 1 0

SEW=32b 1 0

SEW=64b 0

VLEN=128b

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

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

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

SEW=32b 3 2 1 0

SEW=64b 1 0

VLEN=256b

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

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

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

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

SEW=64b 3 2 1 0

30.4.2. Mapping for LMUL < 1

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, and hence must obey the `vta` setting.

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

Byte	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
<code>SEW=8b</code>	-	-	-	-	-	-	-	-	-	-	-	-	3	2	1	0
<code>SEW=16b</code>	-	-	-	-	-	-	-	-	-	-	-	-	1	0		
<code>SEW=32b</code>	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0

30.4.3. Mapping for `LMUL > 1`

When vector registers are grouped, the elements of the vector register group are packed contiguously in element order beginning with the lowest-numbered vector register and moving to the next-highest-numbered vector register in the group once each vector register is filled.

`LMUL > 1` examples

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

Byte	3	2	1	0
<code>v2*n</code>	3	2	1	0
<code>v2*n+1</code>	7	6	5	4

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

Byte	3	2	1	0
<code>v2*n</code>	1	0		
<code>v2*n+1</code>	3	2		

`VLEN=32b`, `SEW=16b`, `LMUL=4`

Byte	3	2	1	0
<code>v4*n</code>	1	0		
<code>v4*n+1</code>	3	2		
<code>v4*n+2</code>	5	4		
<code>v4*n+3</code>	7	6		

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

Byte	3	2	1	0
<code>v4*n</code>				0
<code>v4*n+1</code>				1
<code>v4*n+2</code>				2
<code>v4*n+3</code>				3

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

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

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

[illegible]

$v4*n+1$	3	2
$v4*n+2$	5	4
$v4*n+3$	7	6

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.

	SEW/LMUL						
	1	2	4	8	16	32	64
SEW= 8	8	4	2	1	1/2	1/4	1/8
SEW= 16		8	4	2	1	1/2	1/4
SEW= 32			8	4	2	1	1/2
SEW= 64				8	4	2	1

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

30.4.5. Mask Register Layout

A vector mask occupies only one vector register regardless of SEW and LMUL.

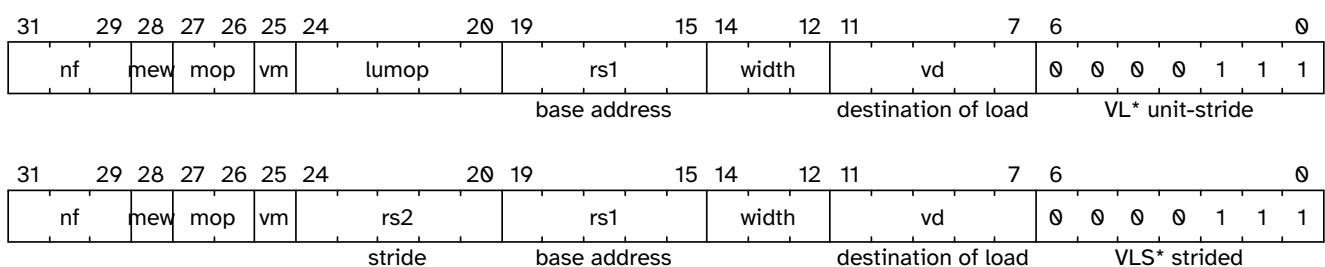
Each element is allocated a single mask bit in a mask vector register. The mask bit for element i is located in bit i of the mask register, independent of SEW or LMUL.

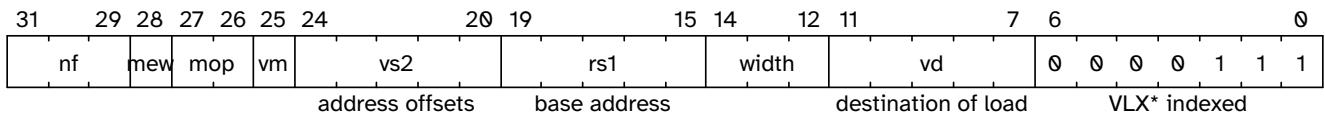
30.5. Vector Instruction Formats

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

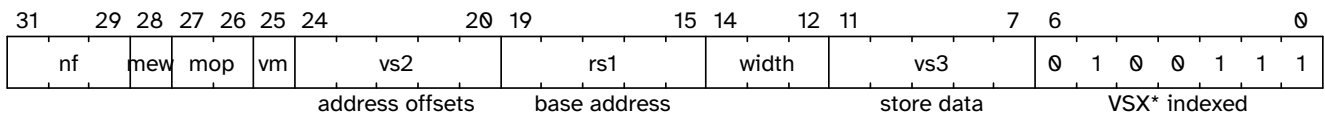
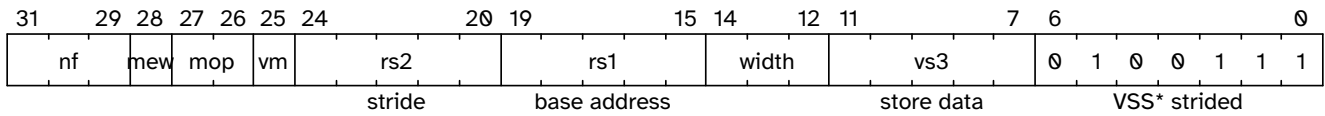
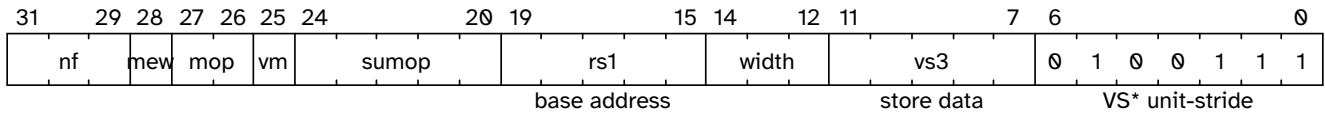
Vector loads and stores are encoded within the scalar floating-point load and store major opcodes (LOAD-FP/STORE-FP). The vector load and store encodings repurpose a portion of the standard scalar floating-point load/store 12-bit immediate field to provide further vector instruction encoding, with bit 25 holding the standard vector mask bit (see [Section 30.5.3.1](#)).

Format for Vector Load Instructions under LOAD-FP major opcode

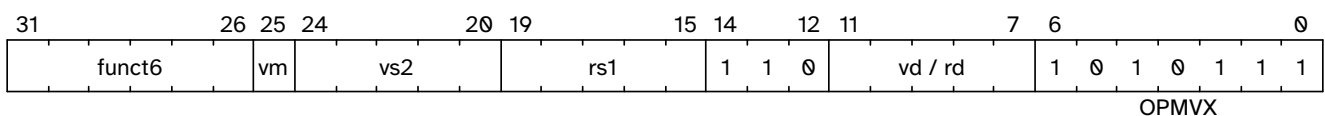
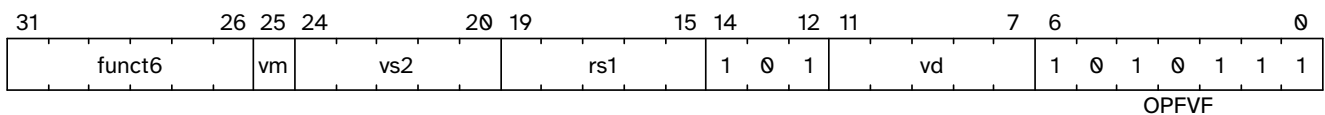
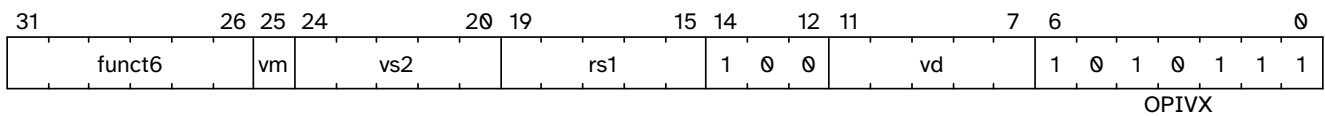
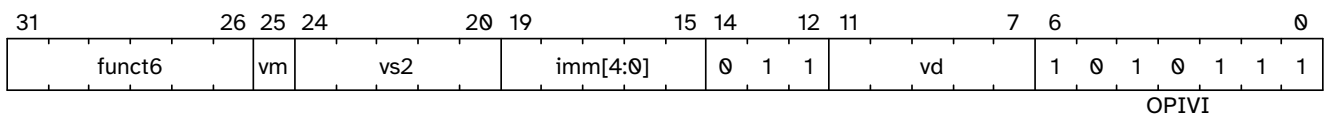
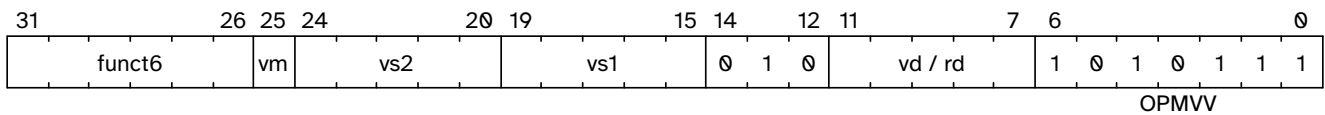
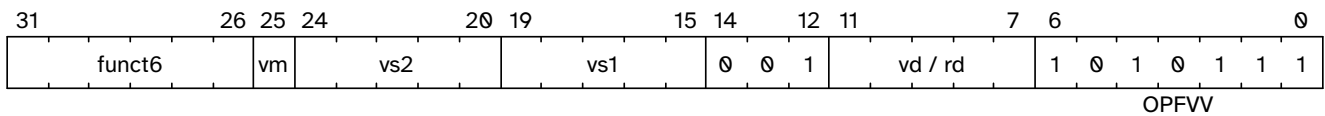
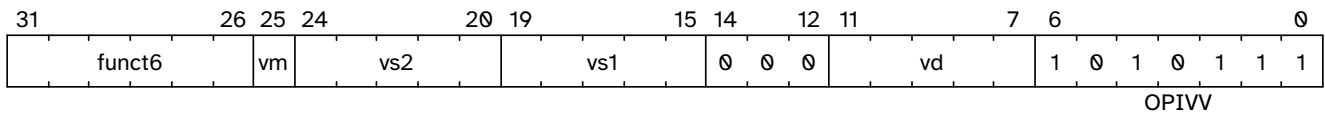




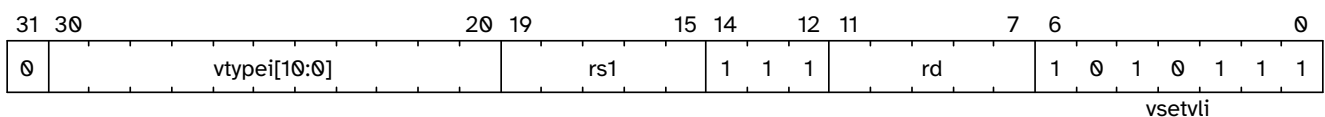
Format for Vector Store Instructions under STORE-FP major opcode

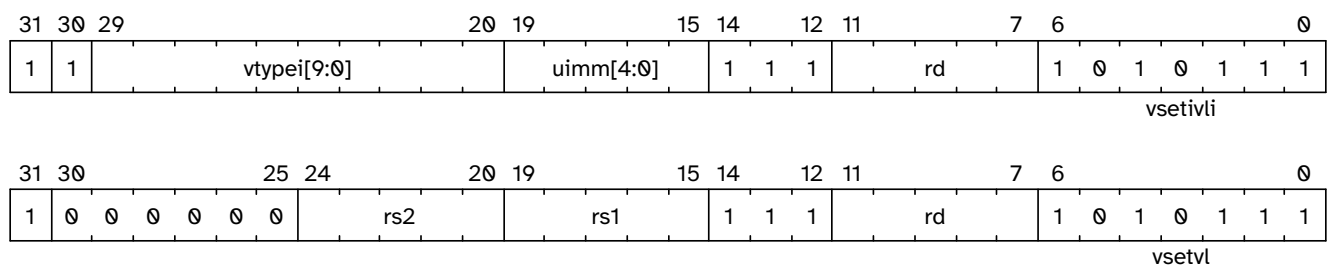


Formats for Vector Arithmetic Instructions under OP-V major opcode



Formats for Vector Configuration Instructions under OP-V major opcode





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.

30.5.1. 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.



*Zfinx ("F in X") is a new ISA extension where floating-point instructions take their arguments from the integer register file. The vector extension is also compatible with Zfinx, where the Zfinx vector extension has vector-scalar floating-point instructions taking their scalar argument from the *x* registers.*



*We considered but did not pursue overlaying the *f* registers on *v* registers. The adopted approach 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.*

30.5.2. 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 have a 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 with a 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 is a reserved encoding.

A vector register cannot be used to provide source operands with more than one EEW for a single instruction. A mask register source is considered to have EEW=1 for this constraint. An encoding that

would result in the same vector register being read with two or more different EEWs, including when the vector register appears at different positions within two or more vector register groups, is reserved.



In practice, there is no software benefit to reading the same register with different EEW in the same instruction, and this constraint reduces complexity for implementations that internally rearrange data dependent on EEW.

A destination vector register group can overlap a source vector register group only if one of the following holds:

- The destination EEW equals the source EEW.
- The destination EEW is smaller than the source EEW and the overlap is in the lowest-numbered part of the source register group (e.g., when LMUL=1, `vnsr1.wi v0, v0, 3` is legal, but a destination of `v1` is not).
- The destination EEW is greater than the source EEW, the source EMUL is at least 1, and the overlap is in the highest-numbered part of the destination register group (e.g., when LMUL=8, `vzext.vf4 v0, v6` is legal, but a source of `v0, v2, or v4` is not).

For the purpose of determining register group overlap constraints, mask elements have EEW=1.



The overlap constraints are designed to support resumable exceptions in machines without register renaming.

Any instruction encoding that violates the overlap constraints is reserved.

When source and destination registers overlap and have different EEW, the instruction is mask- and tail-agnostic, regardless of the setting of the `vta` and `vma` bits in `vtype`.

The largest vector register group used by an instruction can not be greater than 8 vector registers (i.e., $EMUL \leq 8$), and if a vector instruction would require greater than 8 vector registers in a group, the instruction encoding is reserved. For example, a widening operation that produces a widened vector register group result when LMUL=8 is reserved as this would imply a result EMUL=16.

Widened scalar values, e.g., input and output to a widening reduction operation, are held in the first element of a vector register and have EMUL=1.

30.5.3. 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 30.3.4.3](#)).

The mask value used to control execution of a masked vector instruction is always supplied by vector register `v0`.



Masks are held in vector registers, rather than in a separate mask register file, to reduce total architectural state and to simplify the ISA.



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., compares) or the scalar result of a reduction. These instruction encodings are reserved.



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.

As specified in [Section 30.3.4.3](#), mask destination values are always treated as tail-agnostic, regardless of the setting of `vta`.

30.5.3.1. Mask Encoding

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

<code>vm</code>	Description
0	vector result, only where <code>v0.mask[i] = 1</code>
1	unmasked

Vector masking is represented in assembler code as another vector operand, with `.t` indicating that the operation occurs when `v0.mask[i]` is 1 (`t` for "true"). 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, vm=0
vop.v*    v1, v2, v3        # unmasked vector operation, vm=1
```



Even though the current vector extensions only support 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 support both true and complement mask values. The `.t` suffix on the masking operand also helps to visually encode the use of a mask.



The `.mask` suffix is not part of the assembly syntax. We only append it in contexts where a mask vector is subscripted, e.g., `v0.mask[i]`.

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

The destination element indices operated on during a vector instruction's execution can be divided into three 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 *body* elements are those whose element index is greater than or equal to the initial value in the `vstart` register, and less than the current vector length setting in `vl`. The body can be split into two disjoint subsets:
 - The *active* elements during a vector instruction's execution are the elements within the body

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 body 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 specified in `vl`. 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, or with the result of the instruction in the case of mask-producing instructions except for mask loads. When $LMUL < 1$, the tail includes the elements past `VLMAX` that are held in the same vector register.

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

When $vstart \geq vl$, there are no body elements, and no elements are updated in any destination vector register group, including that no tail elements are updated with agnostic values.



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

Instructions that write an `x` register or `f` register do so even when $vstart \geq vl$, including when $vl=0$.



Some instructions such as `vsllidedown` and `vrgather` may read indices past `vl` or even `VLMAX` in source vector register groups. The general policy is to return the value 0 when the index is greater than `VLMAX` in the source vector register group.

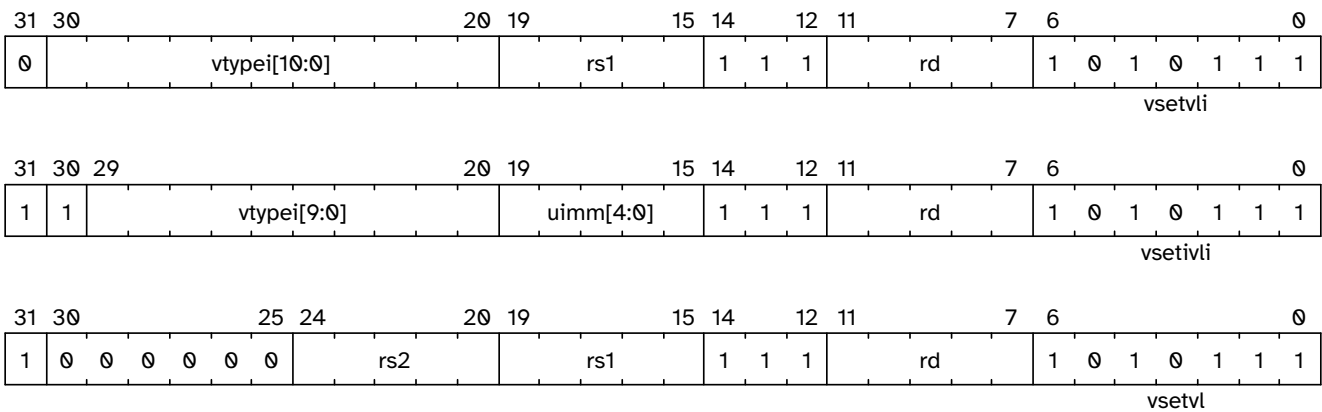
30.6. Configuration-Setting Instructions (vsetvli/vsetivli/vsetvl)

One of the common approaches to handling a large number of elements is "stripmining" where each iteration of a loop handles some number of elements, and the iterations continue until all elements have been processed. The RISC-V vector specification provides direct, portable support for this approach. The application specifies the total number of elements to be processed (the application vector length or AVL) as a candidate value for `vl`, and the hardware responds via a general-purpose register with the (frequently smaller) number of elements that the hardware will handle per iteration (stored in `vl`), based on the microarchitectural implementation and the `vtype` setting. A straightforward loop structure, shown in [Section 30.6.4](#), depicts the ease with which the code keeps track of the remaining number of elements and the amount per iteration handled by hardware.

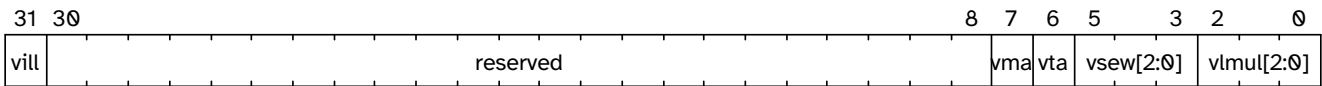
A set of instructions is provided to allow rapid configuration of the values in `vl` and `vtype` to match application needs. The `vset{i}vl{i}` instructions set the `vtype` and `vl` CSRs based on their arguments, and write the new value of `vl` into `rd`.

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

Formats for Vector Configuration Instructions under OP-V major opcode



30.6.1. vtype encoding



This diagram shows the layout for RV32 systems, whereas in general vll should be at bit XLEN-1.

Table 49. vtype register layout

Bits	Name	Description
XLEN-1	vll	Illegal value if set
XLEN-2:8	0	Reserved if non-zero
7	vma	Vector mask agnostic
6	vta	Vector tail agnostic
5:3	vsew[2:0]	Selected element width (SEW) setting
2:0	vlmul[2:0]	Vector register group multiplier (LMUL) setting

The new vtype value is encoded in the immediate fields of vsetvli and vsetivli, and in the rs2 register for vsetvl.

```
Suggested assembler names used for vset{i}vli vtypei immediate

e8    # SEW=8b
e16   # SEW=16b
e32   # SEW=32b
e64   # SEW=64b
```

```
mf8 # LMUL=1/8
mf4 # LMUL=1/4
mf2 # LMUL=1/2
m1  # LMUL=1
m2  # LMUL=2
m4  # LMUL=4
m8  # LMUL=8
```

Examples:

```
vsetvli t0, a0, e8, m1, ta, ma    # SEW= 8, LMUL=1
vsetvli t0, a0, e8, m2, ta, ma    # SEW= 8, LMUL=2
vsetvli t0, a0, e32, mf2, ta, ma  # SEW=32, LMUL=1/2
```

The `vsetvl` variant operates similarly to `vsetvli` except that it takes a `vtype` value from `rs2` and can be used for context restore.

30.6.1.1. Unsupported `vtype` Values

If the `vtype` value 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. Implementations could choose to trap when illegal values are written to `vtype` instead of setting `vill`, to allow emulation to support new configurations for forward-compatibility. The current scheme supports light-weight runtime interrogation of the supported vector unit configurations by checking if `vill` is clear for a given setting.

A `vtype` value with `vill` set is treated as an unsupported configuration.

Implementations must consider all bits of the `vtype` value to determine if the configuration is supported. An unsupported value in any location within the `vtype` value must result in `vill` being set.



In particular, all `XLEN` bits of the register `vtype` argument to the `vsetvl` instruction must be checked. Implementations cannot ignore fields they do not implement. All bits must be checked to ensure that new code assuming unsupported vector features in `vtype` traps instead of executing incorrectly on an older implementation.

30.6.2. AVL encoding

The new vector length setting is based on AVL, which for `vsetvli` and `vsetvl` is encoded in the `rs1` and `rd` fields as follows:

Table 50. AVL used in `vsetvli` and `vsetvl` instructions

rd	rs1	AVL value	Effect on <code>vl</code>
-	!x0	Value in <code>x[rs1]</code>	Normal stripmining
!x0	x0	~0	Set <code>vl</code> to <code>VLMAX</code>
x0	x0	Value in <code>vl</code> register	Keep existing <code>vl</code> (of course, <code>vtype</code> may change)

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 $rd\neq x0$, 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 instruction operates as if the current vector length in vl is used as the AVL, and the resulting value is written to vl , but not to a destination register. This form can only be used when VLMAX and hence vl is not actually changed by the new SEW/LMUL ratio. Use of the instruction with a new SEW/LMUL ratio that would result in a change of VLMAX is reserved. Use of the instruction is also reserved if vll was 1 beforehand. Implementations may set vll in either case.



This last form of the instructions allows the $vtype$ register to be changed while maintaining the current vl , provided VLMAX is not reduced. 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. The vl value could be reduced by this instruction if the new SEW/LMUL ratio causes VLMAX to shrink, and so this case has been reserved as it is not clear this is a generally useful operation, and implementations can otherwise assume vl is not changed by this instruction to optimize their microarchitecture.

For the `vsetivli` instruction, the AVL is encoded as a 5-bit zero-extended immediate (0—31) in the $rs1$ field.



The encoding of AVL for `vsetivli` is the same as for regular CSR immediate values.



The `vsetivli` instruction provides more compact code when the dimensions of vectors are small and known to fit inside the vector registers, in which case there is no stripmining overhead.

30.6.3. Constraints on Setting vl

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

1. $vl = AVL$ if $AVL \leq VLMAX$
2. $\text{ceil}(AVL / 2) \leq vl \leq VLMAX$ if $AVL < (2 * VLMAX)$
3. $vl = VLMAX$ if $AVL \geq (2 * VLMAX)$
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. $vl \leq VLMAX$
 - d. $vl \leq AVL$
 - e. a value read from vl when used as the AVL argument to `vset{i}vl{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 $AVL \leq VLMAX$, yet flexible enough to enable implementations to improve vector lane utilization for $AVL > VLMAX$.

For example, this permits an implementation to set $vl = \text{ceil}(AVL / 2)$ for $VLMAX < AVL < 2 \cdot VLMAX$ 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 $AVL < 2 \cdot VLMAX$. This allows software to avoid needing to explicitly calculate a running maximum of vector lengths observed during a stripmined loop. Requirement 2 also allows an implementation to set vl to $VLMAX$ for $VLMAX < AVL < 2 \cdot VLMAX$

30.6.4. Example of stripmining and changes to SEW

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.
# On entry:
# a0 holds the total number of elements to process
# a1 holds the address of the source array
# a2 holds the address of the destination array

loop:
    vsetvli a3, a0, e16, m4, ta, ma # vtype = 16-bit integer vectors;
                                    # also update a3 with vl (# of elements
this iteration)
    vle16.v v4, (a1)                # Get 16b vector
    slli t1, a3, 1                  # Multiply # elements this iteration by 2
bytes/source element
    add a1, a1, t1                  # Bump pointer
    vwmul.vx v8, v4, x10            # Widening multiply into 32b in <v8--v15>

    vsetvli x0, x0, e32, m8, ta, ma # Operate on 32b values
    vsrl.vi v8, v8, 3
    vse32.v v8, (a2)                # Store vector of 32b elements
    slli t1, a3, 2                  # Multiply # elements this iteration by 4
bytes/destination element
    add a2, a2, t1                  # Bump pointer
    sub a0, a0, a3                  # Decrement count by vl
    bnez a0, loop                   # Any more?
```

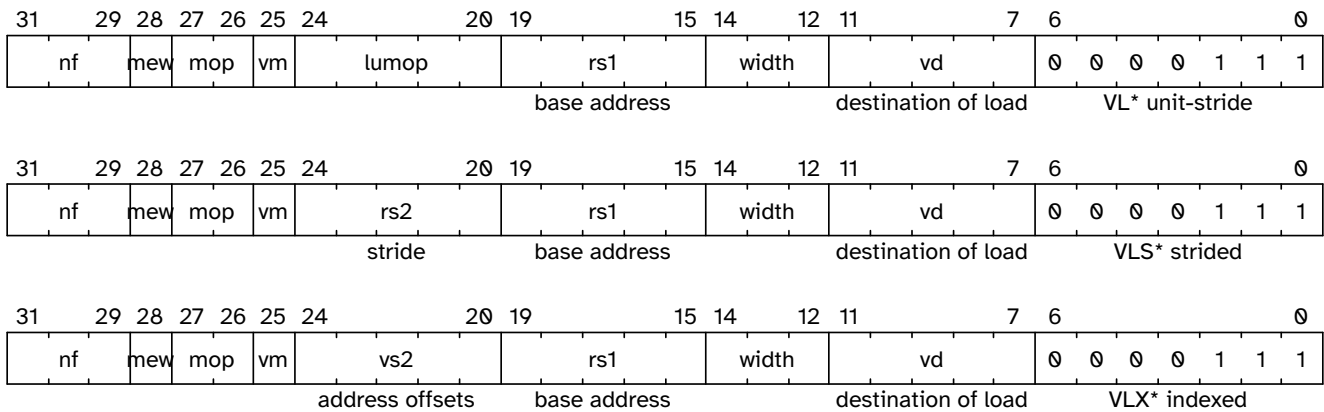
30.7. Vector Loads and Stores

Vector loads and stores move values between vector registers and memory. Vector loads and stores can be masked, and they only access memory or raise exceptions for active elements. Masked vector loads do not update inactive elements in the destination vector register group, unless masked agnostic is specified (`vtype.vma=1`). All vector loads and stores may generate and accept a non-zero `vstart` value.

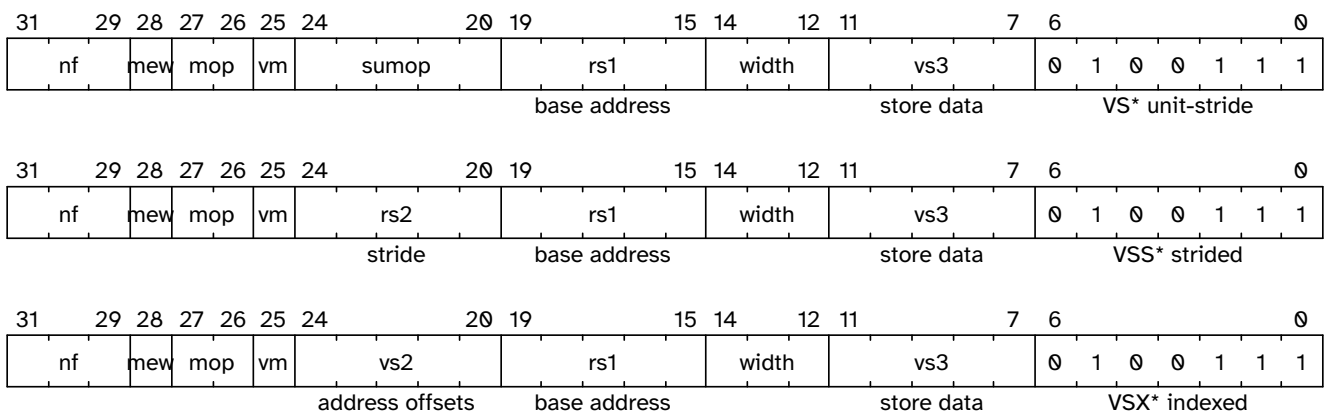
30.7.1. Vector Load/Store Instruction Encoding

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 [Section 30.5.3.1](#)).

Format for Vector Load Instructions under LOAD-FP major opcode



Format for Vector Store Instructions under STORE-FP major opcode



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 whether vector masking is enabled (0 = mask enabled, 1 = mask disabled)
width[2:0]	specifies size of memory elements, and distinguishes from FP scalar
mew	extended memory element width. See Section 30.7.3
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 unit-stride and constant-stride 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.

30.7.2. Vector Load/Store Addressing Modes

The 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 constant-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 encoded in the instruction and EMUL=(EEW/SEW)*LMUL.

The vector offset operand is treated as a vector of byte-address offsets.



The indexed operations can also be used to access fields within a vector of objects, where the vs2 vector holds pointers to the base of the objects and the scalar x register holds the offset of the member field in each object. Supporting this case is why the indexed operations were not defined to scale the element indices by the data EEW.

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. An implementation must raise an illegal-instruction exception if the EEW is not supported for offset elements.



A profile may place an upper limit on the maximum supported index EEW (e.g., only up to XLEN) smaller than ELEN.

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

Table 51. encoding for loads

mop [1:0]		Description	Opcodes
0	0	unit-stride	VLE<EEW>
0	1	indexed-unordered	VLUXEI<EEW>
1	0	strided	VLSE<EEW>
1	1	indexed-ordered	VLOXEI<EEW>

Table 52. 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	VSOXEI<EEW>

Vector unit-stride and constant-stride memory accesses do not guarantee ordering between individual element accesses. The vector indexed load and store memory operations have two forms, ordered and unordered. The indexed-ordered variants preserve element ordering on memory accesses.

For unordered instructions (`mop[1:0] != 11`) there is no guarantee on element access order. If the accesses are to a strongly ordered IO region, the element accesses can be initiated in any order.



To provide ordered vector accesses to a strongly ordered IO region, the ordered indexed instructions should be used.

For implementations with precise vector traps, exceptions on indexed-unordered stores must also be 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 53. lumop

lumop[4:0]					Description
0	0	0	0	0	unit-stride load
0	1	0	0	0	unit-stride, whole register load
0	1	0	1	1	unit-stride, mask load, EEW=8
1	0	0	0	0	unit-stride fault-only-first
x	x	x	x	x	other encodings reserved

Table 54. sumop

sumop[4:0]					Description
0	0	0	0	0	unit-stride store
0	1	0	0	0	unit-stride, whole register store
0	1	0	1	1	unit-stride, mask store, EEW=8
x	x	x	x	x	other encodings 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 30.7.8](#).

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

30.7.3. Vector Load/Store Width Encoding

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

Vector unit-stride and constant-stride use the EEW/EMUL encoded in the instruction for the data values, while vector indexed loads and stores use the EEW/EMUL encoded in the instruction for the index values and the SEW/LMUL encoded in `vtype` for the data values.

Vector loads and stores are encoded using width values that are not claimed by the standard scalar floating-point loads and stores.

Implementations must provide vector loads and stores with EEWs corresponding to all supported SEW settings. Vector load/store encodings for unsupported EEW widths must raise an illegal instruction exception.

Table 55. Width encoding for vector loads and stores.

	me w	width [2:0]			Mem bits	Data Reg bits	Index 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 8b index	0	0	0	0	SEW	SEW	8	VLxEI8/VSxEI8
Vector 16b index	0	1	0	1	SEW	SEW	16	VLxEI16/VSxEI16
Vector 32b index	0	1	1	0	SEW	SEW	32	VLxEI32/VSxEI32
Vector 64b index	0	1	1	1	SEW	SEW	64	VLxEI64/VSxEI64
Reserved	1	X	X	X	-	-	-	

Mem bits is the size of each element accessed in memory.

Data reg bits is the size of each data element accessed in register.

Index bits is the size of each index accessed in register.

The `mew` bit (`inst[28]`) when set is expected to be used to encode expanded memory sizes of 128 bits and above, but these encodings are currently reserved.

30.7.4. Vector Unit-Stride Instructions

```
# Vector unit-stride loads and stores

# vd destination, rs1 base address, vm is mask encoding (v0.t or <missing>)
vle8.v    vd, (rs1), vm # 8-bit unit-stride load
vle16.v   vd, (rs1), vm # 16-bit unit-stride load
vle32.v   vd, (rs1), vm # 32-bit unit-stride load
vle64.v   vd, (rs1), vm # 64-bit unit-stride load

# vs3 store data, rs1 base address, vm is mask encoding (v0.t or <missing>)
vse8.v    vs3, (rs1), vm # 8-bit unit-stride store
vse16.v   vs3, (rs1), vm # 16-bit unit-stride store
vse32.v   vs3, (rs1), vm # 32-bit unit-stride store
vse64.v   vs3, (rs1), vm # 64-bit unit-stride store
```

Additional unit-stride mask load and store instructions are provided to transfer mask values to/from memory. These operate similarly to unmasked byte loads or stores (`EEW=8`), except that the effective vector length is $evl = \text{ceil}(vl/8)$ (i.e. `EMUL=1`), and the destination register is always written with a tail-agnostic policy.

```
# Vector unit-stride mask load
vlm.v vd, (rs1) # Load byte vector of length ceil(vl/8)

# Vector unit-stride mask store
vsm.v vs3, (rs1) # Store byte vector of length ceil(vl/8)
```

`vlm.v` and `vsm.v` are encoded with the same `width[2:0]=0` encoding as `vle8.v` and `vse8.v`, but are distinguished by different `lumop` and `sumop` encodings. Since `vlm.v` and `vsm.v` operate as byte loads and stores, `vstart` is in units of bytes for these instructions.



`vlm.v` and `vsm.v` respect the `vll` field in `vtype`, as they depend on `vtype` indirectly through its constraints on `vl`.



The previous assembler mnemonics `vle1.v` and `vse1.v` were confusing as length was handled differently for these instructions versus other element load/store instructions. To avoid software churn, these older assembly mnemonics are being retained as aliases.



The primary motivation to provide mask load and store is to support machines that internally rearrange data to reduce cross-datapath wiring. However, these instructions also provide a convenient mechanism to use packed bit vectors in memory as mask values, and also reduce the cost of mask spill/fill by reducing need to change `vl`.

30.7.5. Vector Strided Instructions

```
# Vector strided loads and stores

# vd destination, rs1 base address, rs2 byte stride
vlse8.v    vd, (rs1), rs2, vm # 8-bit strided load
vlse16.v   vd, (rs1), rs2, vm # 16-bit strided load
vlse32.v   vd, (rs1), rs2, vm # 32-bit strided load
vlse64.v   vd, (rs1), rs2, vm # 64-bit strided load

# vs3 store data, rs1 base address, rs2 byte stride
vsse8.v    vs3, (rs1), rs2, vm # 8-bit strided store
vsse16.v   vs3, (rs1), rs2, vm # 16-bit strided store
vsse32.v   vs3, (rs1), rs2, vm # 32-bit strided store
vsse64.v   vs3, (rs1), rs2, vm # 64-bit strided store
```

Negative and zero strides are supported.

Element accesses within a strided instruction are unordered with respect to each other.

When $rs2=x0$, then an implementation is allowed, but not required, to perform fewer memory operations than the number of active elements, and may perform different numbers of memory operations across different dynamic executions of the same static instruction.



Compilers must be aware to not use the $x0$ form for $rs2$ when the immediate stride is 0 if the intent is to require all memory accesses are performed.

When $rs2! = x0$ and the value of $x[rs2]=0$, the implementation must perform one memory access for each active element (but these accesses will not be ordered).



As with other architectural mandates, implementations must appear to perform each memory access. Microarchitectures are free to optimize away accesses that would not be observed by another agent, for example, in idempotent memory regions obeying RVWMO. For non-idempotent memory regions, where by definition each access can be observed by a device, the optimization would not be possible.



When repeating ordered vector accesses to the same memory address are required, then an ordered indexed operation can be used.

30.7.6. Vector Indexed Instructions

```
# Vector indexed loads and stores

# Vector indexed-unordered load instructions
# vd destination, rs1 base address, vs2 byte offsets
vluxei8.v  vd, (rs1), vs2, vm # unordered 8-bit indexed load of SEW data
vluxei16.v vd, (rs1), vs2, vm # unordered 16-bit indexed load of SEW data
vluxei32.v vd, (rs1), vs2, vm # unordered 32-bit indexed load of SEW data
vluxei64.v vd, (rs1), vs2, vm # unordered 64-bit indexed load of SEW data
```

```

# Vector indexed-ordered load instructions
# vd destination, rs1 base address, vs2 byte offsets
vloxei8.v    vd, (rs1), vs2, vm # ordered 8-bit indexed load of SEW data
vloxei16.v   vd, (rs1), vs2, vm # ordered 16-bit indexed load of SEW data
vloxei32.v   vd, (rs1), vs2, vm # ordered 32-bit indexed load of SEW data
vloxei64.v   vd, (rs1), vs2, vm # ordered 64-bit indexed load of SEW data

# Vector indexed-unordered store instructions
# vs3 store data, rs1 base address, vs2 byte offsets
vsuxei8.v    vs3, (rs1), vs2, vm # unordered 8-bit indexed store of SEW data
vsuxei16.v   vs3, (rs1), vs2, vm # unordered 16-bit indexed store of SEW data
vsuxei32.v   vs3, (rs1), vs2, vm # unordered 32-bit indexed store of SEW data
vsuxei64.v   vs3, (rs1), vs2, vm # unordered 64-bit indexed store of SEW data

# Vector indexed-ordered store instructions
# vs3 store data, rs1 base address, vs2 byte offsets
vsoxei8.v    vs3, (rs1), vs2, vm # ordered 8-bit indexed store of SEW data
vsoxei16.v   vs3, (rs1), vs2, vm # ordered 16-bit indexed store of SEW data
vsoxei32.v   vs3, (rs1), vs2, vm # ordered 32-bit indexed store of SEW data
vsoxei64.v   vs3, (rs1), vs2, vm # ordered 64-bit indexed store of SEW data

```



The assembler syntax for indexed loads and stores uses `eix` instead of `ex` to indicate the statically encoded EEW is of the index not the data.



The indexed operations mnemonics have a "U" or "O" to distinguish between unordered and ordered, while the other vector addressing modes have no character. While this is perhaps a little less consistent, this approach minimizes disruption to existing software, as `VSXEI` previously meant "ordered" - and the opcode can be retained as an alias during transition to help reduce software churn.

30.7.7. Unit-stride Fault-Only-First Loads

The unit-stride fault-only-first load instructions are used to vectorize loops with data-dependent exit conditions ("while" loops). These instructions execute as a regular load except that they will only take a trap caused by a synchronous exception on element 0. If element 0 raises an exception, `v1` is not modified, and the trap is taken. If an element > 0 raises an exception, the corresponding trap is not taken, and the vector length `v1` is reduced to the index of the element that would have raised an exception.

Load instructions may overwrite active destination vector register group elements past the element index at which the trap is reported. Similarly, fault-only-first load instructions may update active destination elements past the element that causes trimming of the vector length (but not past the original vector length). The values of these spurious updates do not have to correspond to the values in memory at the addressed memory locations. Non-idempotent memory locations can only be accessed when it is known the corresponding element load operation will not be restarted due to a trap or vector-length trimming.

```

# Vector unit-stride fault-only-first loads

# vd destination, rs1 base address, vm is mask encoding (v0.t or <missing>)

```

```

vle8ff.v    vd, (rs1), vm # 8-bit unit-stride fault-only-first load
vle16ff.v   vd, (rs1), vm # 16-bit unit-stride fault-only-first load
vle32ff.v   vd, (rs1), vm # 32-bit unit-stride fault-only-first load
vle64ff.v   vd, (rs1), vm # 64-bit unit-stride fault-only-first load

```

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

```

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

strlen:
    mv a3, a0                # Save start
loop:
    vsetvli a1, x0, e8, m8, ta, ma # Vector of bytes of maximum length
    vle8ff.v v8, (a3)          # Load bytes
    csrr a1, vl               # Get bytes read
    vmseq.vi v0, v8, 0        # Set v0[i] where v8[i] = 0
    vfirst.m a2, v0           # Find first set bit
    add a3, a3, a1            # Bump pointer
    bltz a2, loop            # Not found?

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

    ret

```



There is a security concern with fault-on-first loads, as they can be used to probe for valid effective addresses. The unit-stride versions only allow probing a region immediately contiguous to a known region, and so reduce the security impact when used in unprivileged code. However, code running in S-mode can establish arbitrary page translations that allow probing of random guest physical addresses provided by a hypervisor. Strided and scatter/gather fault-only-first instructions are not provided due to lack of encoding space, but they can also represent a larger security hole, allowing even unprivileged software to easily check multiple random pages for accessibility without experiencing a trap. This standard does not address possible security mitigations for fault-only-first instructions.

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.

When the fault-only-first instruction takes a trap due to an interrupt, implementations should not reduce `vl` and should instead set a `vstart` value.



When the fault-only-first instruction would trigger a debug data-watchpoint trap on an element after the first, implementations should not reduce `vl` but instead should trigger the debug trap as otherwise the event might be lost.

30.7.8. Vector Load/Store Segment Instructions

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



The name "segment" reflects that the items moved are subarrays with homogeneous elements. These operations can be used to transpose arrays between memory and registers, and can support operations on "array-of-structures" datatypes by unpacking each field in a structure into a separate vector register.

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*.

Table 56. *NFIELDS* Encoding

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 $EMUL * NFIELDS \leq 8$, otherwise the instruction encoding is reserved.



*The product $\text{ceil}(EMUL) * NFIELDS$ 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 $EMUL=8$.*

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).

If the vector register numbers accessed by the segment load or store would increment past 31, then the instruction encoding is reserved.



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

The `vl` register gives the number of segments 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 segments.

For segment loads and stores, the individual memory accesses used to access fields within each segment are unordered with respect to each other even for ordered indexed segment loads and stores.

The `vstart` value is in units of whole segments. If a trap occurs during access to a segment, it is implementation-defined whether a subset of the faulting segment's accesses are performed before the trap is taken.

30.7.8.1. Vector Unit-Stride Segment Loads and Stores

The vector unit-stride load and store segment instructions move packed contiguous segments into multiple destination vector register groups.



Where the segments hold structures with heterogeneous-sized fields, software can later unpack individual structure fields using additional instructions after the segment load brings data into the vector registers.

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

```
# Format
# In this syntax, <nf> equals NFIELDS and is an integer in the range [2, 8].
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,vs3+2 to memory
```

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

```
# Example 1
# Memory structure holds packed RGB pixels (24-bit data structure, 8bpp)
vsetvli a1, t0, e8, m1, ta, ma
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, m1, ta, ma
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.
```

```
vlseg<nf>e<eew>ff.v vd, (rs1), vm    # Unit-stride fault-only-first segment
loads
```

For fault-only-first segment loads, if an exception is detected partway through accessing the zeroth segment, the trap is taken. If an exception is detected partway through accessing a subsequent segment, `vl` is reduced to the index of that segment. In both cases, it is implementation-defined whether a subset of the segment is loaded.

These instructions may overwrite destination vector register group elements past the point at which a trap is reported or past the point at which vector length is trimmed.

30.7.8.2. 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, m1, ta, ma
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, m1, ta, ma
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
```

Accesses to the fields within each segment can occur in any order, including the case where the byte stride is such that segments overlap in memory.

30.7.8.3. 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`. Both ordered and unordered forms are provided, where the ordered forms access segments in element order. However, even for the ordered form, accesses to the fields within an individual segment are not ordered with respect to each other.

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`. The `EMUL * NFIELDS ≤ 8` constraint applies to the data vector register group.

```

# Format
vluxseg<nf>ei<eew>.v vd, (rs1), vs2, vm # Indexed-unordered segment loads
vloxseg<nf>ei<eew>.v vd, (rs1), vs2, vm # Indexed-ordered segment loads
vsuxseg<nf>ei<eew>.v vs3, (rs1), vs2, vm # Indexed-unordered segment stores
vsoxseg<nf>ei<eew>.v vs3, (rs1), vs2, vm # Indexed-ordered segment stores

# Examples
vsetvli a1, t0, e8, m1, ta, ma
vluxseg3ei8.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, m1, ta, ma
vsuxseg2ei32.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

```

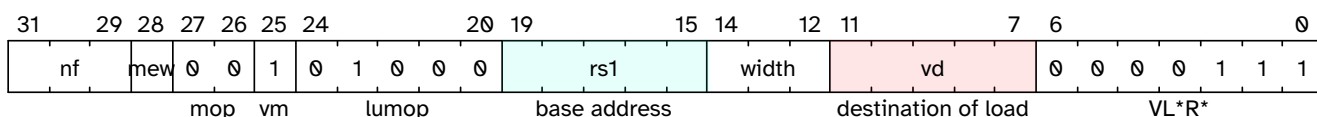
For vector indexed segment loads, the destination vector register groups cannot overlap the source vector register group (specified by `vs2`), else the instruction encoding is reserved.



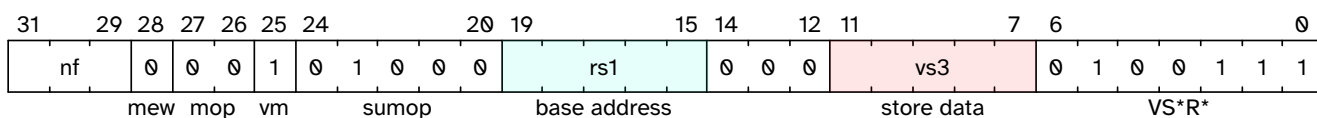
This constraint supports restart of indexed segment loads that raise exceptions partway through loading a structure.

30.7.9. Vector Load/Store Whole Register Instructions

Format for Vector Load Whole Register Instructions under LOAD-FP major opcode



Format for Vector Store Whole Register Instructions under STORE-FP major opcode



These instructions load and store whole vector register groups.



These instructions are intended to be used to save and restore vector registers when the type or 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.

The load instructions have an EEW encoded in the `mew` and `width` fields following the pattern of regular unit-stride loads.



Because in-register byte layouts are identical to in-memory byte layouts, the same data is written to the destination register group regardless of EEW. Hence, it would have sufficed to provide only EEW=8 variants. The full set of EEW variants is provided so that the encoded EEW can be used as a hint to indicate the destination register group will next be accessed with this EEW, which aids implementations that rearrange data internally.

The vector whole register store instructions are encoded similar to unmasked unit-stride store of elements with EEW=8.

The `nf` field encodes how many vector registers to load and store using the NFIELDS encoding (Figure Table 56). The encoded number of registers must be a power of 2 and the vector register numbers must be aligned as with a vector register group, otherwise the instruction encoding is reserved. NFIELDS indicates the number of vector registers to transfer, numbered successively after the base. Only NFIELDS values of 1, 2, 4, 8 are supported, with other values reserved. When multiple registers are transferred, the lowest-numbered vector register is held in the lowest-numbered memory addresses and successive vector register numbers are placed contiguously in memory.

The instructions operate with an effective vector length, $evl = NFIELDS * VLEN / EEW$, regardless of current settings in `vtype` and `vl`. The usual property that no elements are written if $vstart \geq vl$ does not apply to these instructions. Instead, no elements are written if $vstart \geq evl$.

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

Implementations are allowed to raise a misaligned address exception on whole register loads and stores if the base address is not naturally aligned to the larger of the size of the encoded EEW in bytes (EEW/8) or the implementation's smallest supported SEW size in bytes ($SEW_{MIN}/8$).



Allowing misaligned exceptions to be raised based on non-alignment to the encoded EEW simplifies the implementation of these instructions. Some subset implementations might not support smaller SEW widths, so are allowed to report misaligned exceptions for the smallest supported SEW even if larger than encoded EEW. An extreme non-standard implementation might have $SEW_{MIN} > XLEN$ for example. Software environments can mandate the minimum alignment requirements to support an ABI.

```
# Format of whole register load and store instructions.
vl1r.v v3, (a0)      # Pseudoinstruction equal to vl1re8.v

vl1re8.v  v3, (a0)    # Load v3 with VLEN/8 bytes held at address in a0
vl1re16.v v3, (a0)    # Load v3 with VLEN/16 halfwords held at address in a0
vl1re32.v v3, (a0)    # Load v3 with VLEN/32 words held at address in a0
vl1re64.v v3, (a0)    # Load v3 with VLEN/64 doublewords held at address in a0

vl2r.v v2, (a0)      # Pseudoinstruction equal to vl2re8.v

vl2re8.v  v2, (a0)    # Load v2-v3 with 2*VLEN/8 bytes from address in a0
vl2re16.v v2, (a0)    # Load v2-v3 with 2*VLEN/16 halfwords held at address in
a0
vl2re32.v v2, (a0)    # Load v2-v3 with 2*VLEN/32 words held at address in a0
vl2re64.v v2, (a0)    # Load v2-v3 with 2*VLEN/64 doublewords held at address
```

```

in a0

vl4r.v v4, (a0)          # Pseudoinstruction equal to vl4re8.v

vl4re8.v    v4, (a0)     # Load v4-v7 with 4*VLEN/8 bytes from address in a0
vl4re16.v   v4, (a0)
vl4re32.v   v4, (a0)
vl4re64.v   v4, (a0)

vl8r.v v8, (a0)          # Pseudoinstruction equal to vl8re8.v

vl8re8.v    v8, (a0)     # Load v8-v15 with 8*VLEN/8 bytes from address in a0
vl8re16.v   v8, (a0)
vl8re32.v   v8, (a0)
vl8re64.v   v8, (a0)

vs1r.v v3, (a1)          # Store v3 to address in a1
vs2r.v v2, (a1)          # Store v2-v3 to address in a1
vs4r.v v4, (a1)          # Store v4-v7 to address in a1
vs8r.v v8, (a1)          # Store v8-v15 to address in a1

```



Implementations should raise illegal-instruction exceptions on `vl<nf>r` instructions for EEW values that are not supported.



We have considered adding a whole register mask load instruction (`vl1rm.v`) but have decided to omit from initial extension. The primary purpose would be to inform the microarchitecture that the data will be used as a mask. The same effect can be achieved with the following code sequence, whose cost is at most four instructions. Of these, the first could likely be removed as `vl` is often already in a scalar register, and the last might already be present if the following vector instruction needs a new SEW/LMUL. So, in best case only two instructions (of which only one performs vector operations) are needed to synthesize the effect of the dedicated instruction:

```

csrr t0, vl              # Save current vl (potentially not needed)
vsetvli t1, x0, e8, m8, ta, ma  # Maximum VLMAX
vlm.v v0, (a0)           # Load mask register
vsetvli x0, t0, <new type>    # Restore vl (potentially already present)

```

30.8. Vector Memory Alignment Constraints

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

Support for misaligned vector memory accesses is independent of an implementation's support for misaligned scalar memory accesses.



An implementation may have neither, one, or both scalar and vector memory

accesses support some or all misaligned accesses in hardware. A separate PMA should be defined to determine if vector misaligned accesses are supported in the associated address range.

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

30.9. 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. If the Ztso extension is implemented, vector memory instructions additionally follow RVTSO at the instruction level.

Except for vector indexed-ordered loads and stores, element operations are unordered within the instruction.

Vector indexed-ordered loads and stores read and write elements from/to memory in element order respectively, obeying RVWMO at the element level.



Ztso only imposes RVTSO at the instruction level; intra-instruction ordering follows RVWMO regardless of whether Ztso is implemented.



More formal definitions required.

Instructions affected by the vector length register v_l have a control dependency on v_l , rather than a data dependency. Similarly, masked vector instructions have a control dependency on the source mask register, rather than a data dependency.

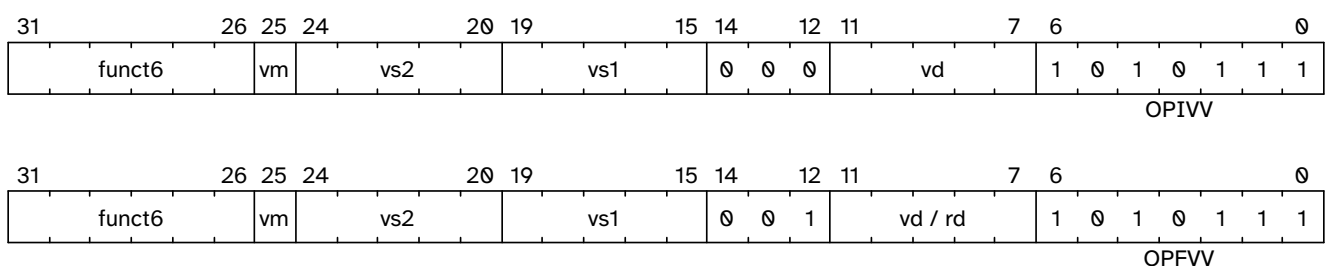


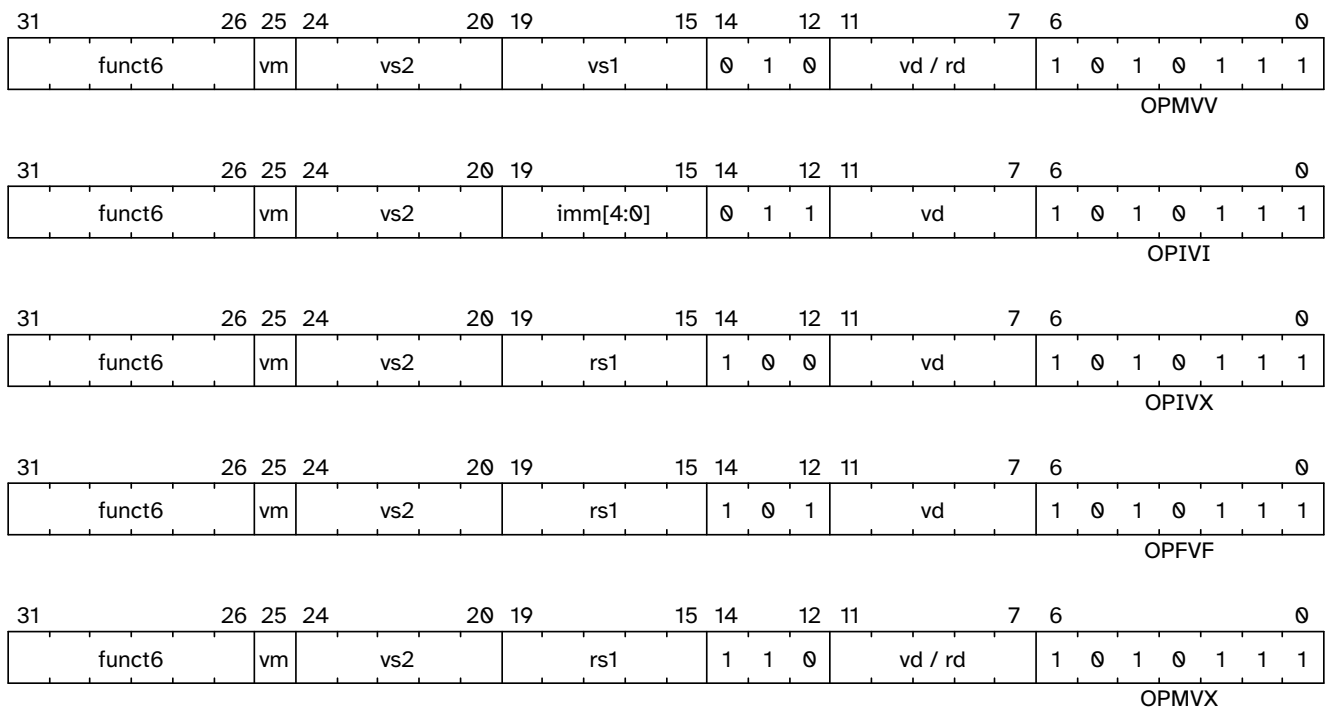
Treating the vector length and mask as control rather than data typically matches the semantics of the corresponding scalar code, where branch instructions ordinarily would have been used. Treating the mask as control allows masked vector load instructions to access memory before the mask value is known, without the need for a misspeculation-recovery mechanism.

30.10. 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





30.10.1. Vector Arithmetic Instruction encoding

The funct3 field encodes the operand type and source locations.

Table 57. funct3

funct3[2:0]			Category	Operands	Type of scalar operand
0	0	0	OPIVV	vector-vector	N/A
0	0	1	OPFVV	vector-vector	N/A
0	1	0	OPMVV	vector-vector	N/A
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.



In this discussion, fixed-point operations are considered to be integer operations.

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. Use of the `frm` field when it contains an invalid rounding mode by any vector floating-point instruction—even those that do not depend on the rounding mode, or when `vl=0`, or when `vstart ≥ vl--` is reserved.



All vector floating-point code will rely on a valid value in `frm`. Implementations can make all vector FP instructions report exceptions when the rounding mode is invalid to simplify control logic.

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. In all three forms, the vector register group operand is specified by `vs2`. The second scalar source operand comes from one of three alternative sources:

1. For integer operations, the scalar can be a 5-bit immediate, `imm[4:0]`, encoded in the `rs1` field. The value is sign-extended to SEW bits, unless otherwise specified.
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, unless otherwise specified. 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. Vector instructions where any floating-point vector operand's EEW is not a supported floating-point type width (which includes when `FLEN < SEW`) are reserved.



Some instructions zero-extend the 5-bit immediate, and denote this by naming the immediate `vimm` in the assembly syntax.



When adding a vector extension to the `Zfinx/Zdinx/Zhinx` extensions, floating-point scalar arguments are taken from the `x` registers. NaN-boxing is not supported in these extensions, and so the vector floating-point scalar value is produced using the same rules as for an integer scalar operand (i.e., when `XLEN > SEW` use the lowest SEW bits, when `XLEN < SEW` use the sign-extended value).

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/imm` 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.

30.10.2. Widening Vector Arithmetic Instructions

A few vector arithmetic instructions are defined to be *widening* operations where the destination vector register group has $EEW=2*SEW$ and $EMUL=2*LMUL$. These are generally given a *vw** prefix on the opcode, or *vwf** for vector floating-point instructions.

The first vector register group operand can be either single or double-width.

```

# Assembly syntax pattern for vector widening arithmetic instructions

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

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

```



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.*

Widening instruction encodings must follow the constraints in [Section 30.5.2](#).

30.10.3. 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 specified by `vs2` with $EEW/EMUL=2*SEW/2*LMUL$ to a vector register group with the current $SEW/LMUL$ setting. Where there is a second source vector register group (specified by `vs1`), this has the same (narrower) width as the result (i.e., $EEW=SEW$).



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



Compare operations that set a mask register are also implicitly a narrowing operation.

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`)

Assembly syntax pattern for vector narrowing arithmetic instructions

```
# Single-width result vd, double-width source vs2, single-width source vs1/rs1
# SEW = 2*SEW op SEW
vop.wv  vd, vs2, vs1, vm # integer vector-vector      vd[i] = vs2[i] op
vs1[i]
vop.wx  vd, vs2, rs1, vm # integer vector-scalar      vd[i] = vs2[i] op
x[rs1]
```

Narrowing instruction encodings must follow the constraints in [Section 30.5.2](#).

30.11. Vector Integer Arithmetic Instructions

A set of vector integer arithmetic instructions is provided. Unless otherwise stated, integer operations wrap around on overflow.

30.11.1. 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
vrsb.vx vd, vs2, rs1, vm # vd[i] = x[rs1] - vs2[i]
vrsb.vi vd, vs2, imm, vm # vd[i] = imm - vs2[i]
```



A vector of integer values can be negated using a reverse-subtract instruction with a scalar operand of x0. An assembly pseudoinstruction `vneg.v vd,vs = vrsb.vx vd,vs,x0` is provided.

30.11.2. 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
vwsbu.vv vd, vs2, vs1, vm # vector-vector
vwsbu.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
vwsb.vv vd, vs2, vs1, vm # vector-vector
vwsb.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
vwsbu.wv vd, vs2, vs1, vm # vector-vector
vwsbu.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
vwsb.wv vd, vs2, vs1, vm # vector-vector
vwsb.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. Assembly pseudoinstructions `vwcv.t.x.x.v vd,vs,vm = vwadd.vx vd,vs,x0,vm` and `vwcv.tu.x.x.v vd,vs,vm = vwaddu.vx vd,vs,x0,vm` are provided.

30.11.3. 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 SEW, while EMUL of the source is (EEW/SEW)*LMUL. The destination has EEW equal to SEW and EMUL equal to LMUL.

```

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

```

If the source EEW is not a supported width, or source EMUL would be below the minimum legal LMUL, the instruction encoding is reserved.



Standard vector load instructions access memory values that are the same size as the destination register elements. Some application code needs to operate on a range of operand widths in a wider element, for example, loading a byte from memory and adding to an eight-byte element. To avoid having to provide the cross-product of the number of vector load instructions by the number of data types (byte, word, halfword, and also signed/unsigned variants), we instead add explicit extension instructions that can be used if an appropriate widening arithmetic instruction is not available.

30.11.4. 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 30.4.5](#). 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.

`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 `vmvsbc` add or subtract the source operands, optionally add the carry-in or subtract the borrow-in if masked (`vm=0`), and write the resulting carry-out or borrow-out 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. Because these instructions produce a mask value, they always operate with a tail-agnostic policy.

```

# Produce sum with carry.

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

```

```

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

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

# Produce carry out in mask register format

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

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

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

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

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

# vd.mask[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
vmmv.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.mask[i]
vsbc.vvm    vd, vs2, vs1, v0 # Vector-vector

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

# Produce borrow out in mask register format

```

```

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

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

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

# vd.mask[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 `vsubc`, the instruction encoding is reserved if the destination vector register is `v0`.



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

30.11.5. 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 is provided as an assembler pseudoinstruction `vnot.v vd,vs,vm = vxor.vi vd,vs,-1,vm`.

30.11.6. Vector Single-Width Shift Instructions

A full set of vector shift instructions are provided, including logical shift left (`sll`), and logical (zero-extending `srl`) and arithmetic (sign-extending `sra`) shift right. The data to be shifted is in the vector register group specified by `vs2` and the shift amount value can come from a vector register group `vs1`, a scalar integer register `rs1`, or a zero-extended 5-bit immediate. Only the low $\lg_2(\text{SEW})$ bits of the shift-amount value are used to control the shift amount.

```

# 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

```

30.11.7. 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 register group, or a scalar `x` register, or a zero-extended 5-bit immediate. The low $\lg(2 \cdot \text{SEW})$ bits of the shift-amount value are used (e.g., the low 6 bits for a SEW=64-bit to SEW=32-bit narrowing operation).

```

# 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

```



Future extensions might add support for versions that narrow to a destination that is 1/4 the width of the source.



An integer value can be halved in width using the narrowing integer shift instructions with a scalar operand of `x0`. An assembly pseudoinstruction is provided `vncvt.x.x.w vd,vs,vm = vnsrl.wx vd,vs,x0,vm`.

30.11.8. Vector Integer Compare 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 30.4.5](#). 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 giving an effective `vmsltu.vi` range of 1 to 16 (Note, `vmsltu.vi` with immediate 0 is not useful as it is always false).



Because the 5-bit vector immediates are always sign-extended, when the high bit of the `simm5` immediate is set, `vmsleu.vi` also supports unsigned immediate values in the range $2^{\text{SEW}}-16$ to $2^{\text{SEW}}-1$, allowing corresponding `vmsltu.vi` compares against unsigned immediates in the range $2^{\text{SEW}}-15$ to 2^{SEW} . Note that `vmsltu.vi` with immediate 2^{SEW} is not useful as it is always true.

Similarly, `vmsge{u}.vi` is not provided and the compare 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 compare 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 $va \geq x$

```
pseudoinstruction: vmsge{u}.vx vd, va, x
expansion: vmslt{u}.vx vd, va, x; vmnand.mm vd, vd, vd
```

masked $va \geq x$, $vd \neq v0$

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

masked $va \geq x$, $vd == v0$

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

masked $va \geq x$, any vd

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

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

Compares effectively AND in the mask under a mask-undisturbed policy if the destination register is $v0$, e.g.,

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

Compares write mask registers, and so always operate under a tail-agnostic policy.

30.11.9. 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
```

30.11.10. Vector Single-Width Integer Multiply Instructions

The single-width multiply instructions perform a SEW-bit*SEW-bit multiply to generate a 2*SEW-bit product, then return one half of the product in the SEW-bit-wide destination. The `mul` versions write the low word of the product to the destination register, while 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.vx` opcode to return high half of unsigned-vector * signed-scalar product. The scalar can be splatted to a vector, then a `vmulhsu.vv` used.*



The current `vmulh` opcodes perform simple fractional multiplies, but with no option to scale, round, and/or saturate the result. A possible future extension can consider variants of `vmulh`, `vmulhu`, `vmulhsu` that use the `vxrm` rounding mode when discarding low half of product. There is no possibility of overflow in these cases.*

30.11.11. 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.

30.11.12. 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(vs2)-unsigned integer multiply
vwmulsu.vv vd, vs2, vs1, vm # vector-vector
vwmulsu.vx vd, vs2, rs1, vm # vector-scalar

```

30.11.13. 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]
```

30.11.14. Vector Widening Integer Multiply-Add Instructions

The widening integer multiply-add instructions add the full 2*SEW-bit product from a SEW-bit*SEW-bit multiply to 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]
```

30.11.15. 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 encoded as masked instructions (`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]
```

30.11.16. 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 encoded as unmasked instructions (`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] = x[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`.

The form `vmv.v.v vd, vd`, which leaves body elements unchanged, can be used to indicate that the register will next be used with an EEW equal to SEW.



Implementations that internally reorganize data according to EEW can shuffle the internal representation according to SEW. Implementations that do not internally reorganize data can dynamically elide this instruction, and treat as a NOP.



The `vmv.v.v vd, vd` instruction is not a RISC-V HINT as a tail-agnostic setting may cause an architectural state change on some implementations.

30.12. Vector Fixed-Point Arithmetic Instructions

The preceding set of integer arithmetic instructions is extended to support fixed-point arithmetic.

A fixed-point number is a two's-complement signed or unsigned integer interpreted as the numerator in a fraction with an implicit denominator. The fixed-point instructions are intended to be applied to the numerators; it is the responsibility of software to manage the denominators. An N -bit element can hold two's-complement signed integers in the range $-2^{N-1} \dots +2^{N-1}-1$, and unsigned integers in the range $0 \dots +2^N-1$. 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 avoid overflow.

30.12.1. 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
```

30.12.2. 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`. Computation is performed in infinite precision before rounding and truncating. Both unsigned and signed versions are provided. For `vaaddu` and `vaadd` there can be no overflow in the result. For `vasub` and `vasubu`, overflow is ignored and the result wraps around.



For `vasub`, overflow occurs only when subtracting the smallest number from the largest number under `rne` or `rnu` rounding.

```
# 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)
```

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

The signed fractional multiply instruction produces a $2 \times \text{SEW}$ product of the two SEW inputs, then shifts the result right by SEW-1 bits, rounding these bits according to `vfrm`, 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 vfrm 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.



We do not provide an equivalent fractional multiply where one input is unsigned, as these would retain all upper SEW bits and would not need to saturate. This operation is partly covered by the `vmulhu` and `vmulhsu` instructions, for the case where rounding is simply truncation (`rdn`).

30.12.4. Vector Single-Width Scaling Shift Instructions

These instructions shift the input value right, and round off the shifted out bits according to `vfrm`. The scaling right shifts have both zero-extending (`vssrl`) and sign-extending (`vssra`) forms. The data to be shifted is in the vector register group specified by `vs2` and the shift amount value can come from a vector register group `vs1`, a scalar integer register `rs1`, or a zero-extended 5-bit immediate. Only the low $\lg 2(\text{SEW})$ bits of the shift-amount value are used to control the shift amount.

```
# 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)
```

30.12.5. 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 source data is in the vector register group specified by `vs2`. The scaling shift amount value can come from a vector register group `vs1`, a scalar integer register `rs1`, or a zero-extended 5-bit immediate. The low $\lg(2 \cdot \text{SEW})$ bits of the vector or scalar shift-amount value (e.g., the low 6 bits for a $\text{SEW}=64$ -bit to $\text{SEW}=32$ -bit narrowing operation) are used to control the right shift amount, which provides the scaling.

```
# 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], uimm))

# 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], uimm))
```

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.



There is no single instruction that can saturate a signed value into an unsigned destination. A sequence of two vector instructions that first removes negative numbers by performing a max against 0 using `vmax` then clips the resulting unsigned value into the destination using `vnclipu` can be used if setting `vxsat` value for negative numbers is not required. A `vsetvli` is required inbetween these two instructions to change SEW.

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.

30.13. Vector Floating-Point Instructions

The standard vector floating-point instructions treat 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, the instruction encoding is reserved.



Whether floating-point is supported, and for which element widths, is determined by the specific vector extension. The current set of extensions include support for 32-bit and 64-bit floating-point values. When 16-bit and 128-bit element widths are added, they will be also be treated as IEEE-754/2008-compatible values. Other floating-point formats may be supported in future extensions.

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



In particular, future vector extensions supporting 16-bit half-precision floating-point values will also require some scalar half-precision floating-point support.

If the floating-point unit status field `mstatus.FS` is 0ff 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.

If the hypervisor extension is implemented and `V=1`, the `vsstatus.FS` field is additionally in effect for vector floating-point instructions. If `vsstatus.FS` or `mstatus.FS` is 0ff 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 both `mstatus.FS` and `vsstatus.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 floating-point vector-scalar operations are sourced as described in [Section 30.10.1](#).

30.13.1. 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.

30.13.2. 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]
```

30.13.3. 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
```

```

vfwsb.vv vd, vs2, vs1, vm # vector-vector
vfwsb.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
vfwsb.wv  vd, vs2, vs1, vm # vector-vector
vfwsb.wf  vd, vs2, rs1, vm # vector-scalar

```

30.13.4. 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]

```

30.13.5. 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

```

30.13.6. 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
vfmac.vv vd, vs1, vs2, vm # vd[i] = +(vs1[i] * vs2[i]) + vd[i]
vfmac.vf vd, rs1, vs2, vm # vd[i] = +(f[rs1] * vs2[i]) + vd[i]

# FP negate-(multiply-accumulate), overwrites subtrahend
vfnmac.vv vd, vs1, vs2, vm # vd[i] = -(vs1[i] * vs2[i]) - vd[i]
vfnmac.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
vfmsadd.vv vd, vs1, vs2, vm  # vd[i] = +(vs1[i] * vd[i]) + vs2[i]
vfmsadd.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]

```



While we considered using the two unused rounding modes in the scalar FP FMA encoding to provide a few non-destructive FMAs, these would complicate microarchitectures by being the only maskable operation with three inputs and separate output.

30.13.7. 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
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-accumulate), overwrites addend
vfnwmacc.vv vd, vs1, vs2, vm  # vd[i] = -(vs1[i] * vs2[i]) - vd[i]
vfnwmacc.vf vd, rs1, vs2, vm  # vd[i] = -(f[rs1] * vs2[i]) - vd[i]

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

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

```

30.13.8. 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
```

30.13.9. Vector Floating-Point Reciprocal Square-Root Estimate Instruction

```
# Floating-point reciprocal square-root estimate to 7 bits.
vfrsqrt7.v vd, vs2, vm
```

This is a unary vector-vector instruction that returns an estimate of $1/\sqrt{x}$ accurate to 7 bits.



An earlier draft version had used the assembler name `vfrsqste7` but this was deemed to cause confusion with the `ex` notation for element width. The earlier name can be retained as alias in tool chains for backward compatibility.

The following table describes the instruction's behavior for all classes of floating-point inputs:

Input	Output	Exceptions raised
$-\infty \leq x < -0.0$	canonical NaN	NV
-0.0	$-\infty$	DZ
$+0.0$	$+\infty$	DZ
$+0.0 < x < +\infty$	estimate of $1/\sqrt{x}$	
$+\infty$	$+0.0$	
qNaN	canonical NaN	
sNaN	canonical NaN	NV



All positive normal and subnormal inputs produce normal outputs.



The output value is independent of the dynamic rounding mode.

For the non-exceptional cases, the low bit of the exponent and the six high bits of significand (after the leading one) are concatenated and used to address the following table. The output of the table becomes the seven high bits of the result significand (after the leading one); the remainder of the result significand is zero. Subnormal inputs are normalized and the exponent adjusted appropriately before the lookup. The output exponent is chosen to make the result approximate the reciprocal of the square root of the argument.

More precisely, the result is computed as follows. Let the normalized input exponent be equal to the input exponent if the input is normal, or 0 minus the number of leading zeros in the significand otherwise. If the input is subnormal, the normalized input significand is given by shifting the input significand left by 1 minus the normalized input exponent, discarding the leading 1 bit. The output exponent equals $\text{floor}((3 \cdot B - 1 - \text{the normalized input exponent}) / 2)$, where B is the exponent bias. The output sign equals the input sign.

The following table gives the seven MSBs of the output significand as a function of the LSB of the normalized input exponent and the six MSBs of the normalized input significand; the other bits of the output significand are zero.

Table 58. *vfrsqr7.v* common-case lookup table contents

exp[0]	sig[MSB -: 6]	sig_out[MSB -: 7]
0	0	52
0	1	51
0	2	50
0	3	48
0	4	47
0	5	46
0	6	44
0	7	43
0	8	42
0	9	41
0	10	40
0	11	39
0	12	38
0	13	36
0	14	35
0	15	34
0	16	33
0	17	32
0	18	31
0	19	30
0	20	30
0	21	29
0	22	28
0	23	27
0	24	26
0	25	25
0	26	24
0	27	23
0	28	23
0	29	22
0	30	21

exp[0]	sig[MSB -: 6]	sig_out[MSB -: 7]
0	31	20
0	32	19
0	33	19
0	34	18
0	35	17
0	36	16
0	37	16
0	38	15
0	39	14
0	40	14
0	41	13
0	42	12
0	43	12
0	44	11
0	45	10
0	46	10
0	47	9
0	48	9
0	49	8
0	50	7
0	51	7
0	52	6
0	53	6
0	54	5
0	55	4
0	56	4
0	57	3
0	58	3
0	59	2
0	60	2
0	61	1
0	62	1
0	63	0
1	0	127
1	1	125

exp[0]	sig[MSB -: 6]	sig_out[MSB -: 7]
1	2	123
1	3	121
1	4	119
1	5	118
1	6	116
1	7	114
1	8	113
1	9	111
1	10	109
1	11	108
1	12	106
1	13	105
1	14	103
1	15	102
1	16	100
1	17	99
1	18	97
1	19	96
1	20	95
1	21	93
1	22	92
1	23	91
1	24	90
1	25	88
1	26	87
1	27	86
1	28	85
1	29	84
1	30	83
1	31	82
1	32	80
1	33	79
1	34	78
1	35	77
1	36	76

exp[0]	sig[MSB -: 6]	sig_out[MSB -: 7]
1	37	75
1	38	74
1	39	73
1	40	72
1	41	71
1	42	70
1	43	70
1	44	69
1	45	68
1	46	67
1	47	66
1	48	65
1	49	64
1	50	63
1	51	63
1	52	62
1	53	61
1	54	60
1	55	59
1	56	59
1	57	58
1	58	57
1	59	56
1	60	56
1	61	55
1	62	54
1	63	53



For example, when $SEW=32$, $vfrsqrt7(0x00718abc \approx 1.043e-38) = 0x5f080000 (\approx 9.800e18)$, and $vfrsqrt7(0x7f765432 \approx 3.274e38) = 0x1f820000 (\approx 5.506e-20)$.



The 7 bit accuracy was chosen as it requires 0,1,2,3 Newton-Raphson iterations to converge to close to bfloat16, FP16, FP32, FP64 accuracy respectively. Future instructions can be defined with greater estimate accuracy.

30.13.10. Vector Floating-Point Reciprocal Estimate Instruction

Floating-point reciprocal estimate to 7 bits.

vfred7.v vd, vs2, vm



An earlier draft version had used the assembler name `vfrec7` but this was deemed to cause confusion with `ex` notation for element width. The earlier name can be retained as alias in tool chains for backward compatibility.

This is a unary vector-vector instruction that returns an estimate of $1/x$ accurate to 7 bits.

The following table describes the instruction's behavior for all classes of floating-point inputs, where B is the exponent bias:

Input (x)	Rounding Mode	Output ($y \approx 1/x$)	Exceptions raised
$-\infty$	<i>any</i>	-0.0	
$-2^{B+1} < x \leq -2^B$ (normal)	<i>any</i>	$-2^{-(B+1)} \geq y > -2^{-B}$ (subnormal, sig=01...)	
$-2^B < x \leq -2^{B-1}$ (normal)	<i>any</i>	$-2^{-B} \geq y > -2^{B+1}$ (subnormal, sig=1...)	
$-2^{B-1} < x \leq -2^{B-1}$ (normal)	<i>any</i>	$-2^{B+1} \geq y > -2^{B-1}$ (normal)	
$-2^{B+1} < x \leq -2^{-B}$ (subnormal, sig=1...)	<i>any</i>	$-2^{B-1} \geq y > -2^B$ (normal)	
$-2^{-B} < x \leq -2^{-(B+1)}$ (subnormal, sig=01...)	<i>any</i>	$-2^B \geq y > -2^{B+1}$ (normal)	
$-2^{-(B+1)} < x < -0.0$ (subnormal, sig=00...)	RUP, RTZ	greatest-mag. negative finite value	NX, OF
$-2^{-(B+1)} < x < -0.0$ (subnormal, sig=00...)	RDN, RNE, RMM	$-\infty$	NX, OF
-0.0	<i>any</i>	$-\infty$	DZ
$+0.0$	<i>any</i>	$+\infty$	DZ
$+0.0 < x < 2^{-(B+1)}$ (subnormal, sig=00...)	RUP, RNE, RMM	$+\infty$	NX, OF
$+0.0 < x < 2^{-(B+1)}$ (subnormal, sig=00...)	RDN, RTZ	greatest finite value	NX, OF
$2^{-(B+1)} \leq x < 2^{-B}$ (subnormal, sig=01...)	<i>any</i>	$2^{B+1} > y \geq 2^B$ (normal)	
$2^{-B} \leq x < 2^{B-1}$ (subnormal, sig=1...)	<i>any</i>	$2^B > y \geq 2^{B-1}$ (normal)	
$2^{B+1} \leq x < 2^{B-1}$ (normal)	<i>any</i>	$2^{B-1} > y \geq 2^{B+1}$ (normal)	
$2^{B-1} \leq x < 2^B$ (normal)	<i>any</i>	$2^{B+1} > y \geq 2^{-B}$ (subnormal, sig=1...)	
$2^B \leq x < 2^{B+1}$ (normal)	<i>any</i>	$2^{-B} > y \geq 2^{-(B+1)}$ (subnormal, sig=01...)	
$+\infty$	<i>any</i>	$+0.0$	
qNaN	<i>any</i>	canonical NaN	
sNaN	<i>any</i>	canonical NaN	NV



Subnormal inputs with magnitude at least $2^{-(B+1)}$ produce normal outputs; other subnormal inputs produce infinite outputs. Normal inputs with magnitude at least 2^{B-1} produce subnormal outputs; other normal inputs produce normal outputs.



The output value depends on the dynamic rounding mode when the overflow exception is raised.

For the non-exceptional cases, the seven high bits of significand (after the leading one) are used to address the following table. The output of the table becomes the seven high bits of the result significand (after the leading one); the remainder of the result significand is zero. Subnormal inputs are normalized and the exponent adjusted appropriately before the lookup. The output exponent is chosen to make the result approximate the reciprocal of the argument, and subnormal outputs are denormalized accordingly.

More precisely, the result is computed as follows. Let the normalized input exponent be equal to the input exponent if the input is normal, or 0 minus the number of leading zeros in the significand otherwise. The normalized output exponent equals $(2*B - 1 - \text{the normalized input exponent})$. If the normalized output exponent is outside the range $[-1, 2*B]$, the result corresponds to one of the exceptional cases in the table above.

If the input is subnormal, the normalized input significand is given by shifting the input significand left by 1 minus the normalized input exponent, discarding the leading 1 bit. Otherwise, the normalized input significand equals the input significand. The following table gives the seven MSBs of the normalized output significand as a function of the seven MSBs of the normalized input significand; the other bits of the normalized output significand are zero.

Table 59. vfrec7.v common-case lookup table contents

sig[MSB -: 7]	sig_out[MSB -: 7]
0	127
1	125
2	123
3	121
4	119
5	117
6	116
7	114
8	112
9	110
10	109
11	107
12	105
13	104
14	102
15	100
16	99
17	97
18	96

sig[MSB -: 7]	sig_out[MSB -: 7]
19	94
20	93
21	91
22	90
23	88
24	87
25	85
26	84
27	83
28	81
29	80
30	79
31	77
32	76
33	75
34	74
35	72
36	71
37	70
38	69
39	68
40	66
41	65
42	64
43	63
44	62
45	61
46	60
47	59
48	58
49	57
50	56
51	55
52	54
53	53

sig[MSB -: 7]	sig_out[MSB -: 7]
54	52
55	51
56	50
57	49
58	48
59	47
60	46
61	45
62	44
63	43
64	42
65	41
66	40
67	40
68	39
69	38
70	37
71	36
72	35
73	35
74	34
75	33
76	32
77	31
78	31
79	30
80	29
81	28
82	28
83	27
84	26
85	25
86	25
87	24
88	23

sig[MSB -: 7]	sig_out[MSB -: 7]
89	23
90	22
91	21
92	21
93	20
94	19
95	19
96	18
97	17
98	17
99	16
100	15
101	15
102	14
103	14
104	13
105	12
106	12
107	11
108	11
109	10
110	9
111	9
112	8
113	8
114	7
115	7
116	6
117	5
118	5
119	4
120	4
121	3
122	3
123	2

sig[MSB -: 7]	sig_out[MSB -: 7]
124	2
125	1
126	1
127	0

If the normalized output exponent is 0 or -1, the result is subnormal: the output exponent is 0, and the output significand is given by concatenating a 1 bit to the left of the normalized output significand, then shifting that quantity right by 1 minus the normalized output exponent. Otherwise, the output exponent equals the normalized output exponent, and the output significand equals the normalized output significand. The output sign equals the input sign.



For example, when $SEW=32$, $vfrec7(0x00718abc \approx 1.043e-38) = 0x7e900000 \approx 9.570e37$, and $vfrec7(0x7f765432 \approx 3.274e38) = 0x00214000 \approx 3.053e-39$.



The 7 bit accuracy was chosen as it requires 0,1,2,3 Newton-Raphson iterations to converge to close to bfloat16, FP16, FP32, FP64 accuracy respectively. Future instructions can be defined with greater estimate accuracy.

30.13.11. Vector Floating-Point MIN/MAX Instructions

The vector floating-point `vfmn` and `vfmn` instructions have the same behavior as the corresponding scalar floating-point instructions in version 2.2 of the RISC-V F/D/Q extension: they perform the `minimumNumber` or `maximumNumber` operation on active elements.

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

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

30.13.12. 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
```



A vector of floating-point values can be negated using a sign-injection instruction with both source operands set to the same vector operand. An assembly pseudoinstruction is provided: `vfneg.v vd,vs = vfsgnjn.vv vd,vs,vs`.



The absolute value of a vector of floating-point elements can be calculated using a sign-injection instruction with both source operands set to the same vector operand. An assembly pseudoinstruction is provided: `vfabs.v vd,vs = vfsgnjx.vv vd,vs,vs`.

30.13.13. 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 30.4.5](#). The destination mask vector register may be the same as the source vector mask register (`v0`). Compares write mask registers, and so always operate under a tail-agnostic policy.

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 compares 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
<code>va < vb</code>	<code>vmflt.vv vd, va, vb, vm</code>	
<code>va <= vb</code>	<code>vmfle.vv vd, va, vb, vm</code>	
<code>va > vb</code>	<code>vmflt.vv vd, vb, va, vm</code>	<code>vmfgt.vv vd, va, vb, vm</code>

```

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 compares for NaNs.



C99 floating-point quiet compares can be implemented by masking the signaling compares 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.

30.13.14. 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.

30.13.15. 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 encoded as a masked instruction (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]
```

30.13.16. 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 encoded as an unmasked instruction (*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*.

30.13.17. 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.

30.13.18. 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 [Section 30.10.2](#)).



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.

30.13.19. 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.

```

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

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

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

vfnvcvt.f.f.w vd, vs2, vm        # Convert double-width float to single-width
float.
vfnvcvt.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 [Section 30.10.3](#)).



A full set of floating-point narrowing 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 (`vfnvrt.rnd.f.f.w`). Only the final step should use the desired rounding mode.



For `vfnvrt.rnd.f.f.w`, a finite value that exceeds the range of the destination format is converted to the destination format's largest finite value with the same sign.

30.14. 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 source operands, including the mask register.



Vector reductions read and write the scalar operand and result into element 0 of a vector register instead of a scalar 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 ($0 < \text{index} < \text{VLEN}/\text{SEW}$) are considered the tail and are managed with the current tail agnostic/undisturbed policy.

If $v_l=0$, no operation is performed and the destination register is not updated.



This choice of behavior for $v_l=0$ reduces implementation complexity as it is consistent with other operations on vector register state. For the common case that the source and destination scalar operand are the same vector register, this behavior also produces the expected result. For the uncommon case that the source and destination scalar operand are in different vector registers, this instruction will not copy the source into the destination when $v_l=0$. However, it is expected that in most of these cases it will be statically known that v_l is not zero. In other cases, a check for $v_l=0$ will have to be added to ensure that the source scalar is copied to the destination (e.g., by explicitly setting $v_l=1$ and performing a register-register copy).

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.

30.14.1. 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[*] )
```

30.14.2. 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.

For both `vwredsumu.vs` and `vwredsum.vs`, overflows wrap around.

```
# 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))
```

30.14.3. Vector Single-Width Floating-Point Reduction Instructions

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



Older assembler mnemonic `vfredsum` is retained as alias for `vfredusum`.

30.14.3.1. 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:

```
vd[0] = `(((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.

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

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

The implementation must 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 and raising exception flags as necessary. 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 $-\infty$) 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 `vfredusum` instruction.

30.14.3.3. Vector Single-Width Floating-Point Max and Min Reductions

The `vfredmin` and `vfredmax` instructions reduce the scalar argument in `vs1[0]` and active elements in `vs2` using the `minimumNumber` and `maximumNumber` operations, respectively.



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



If no elements are active, the scalar in `vs1[0]` is simply copied to the destination register, without canonicalizing NaN values and without setting any exception flags.

30.14.4. 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
vfwredusum.vs vd, vs2, vs1, vm # Unordered sum
```



Older assembler mnemonic `vfwredsum` is retained as alias for `vfwredusum`.

The reduction of the SEW-width elements is performed as in the single-width reduction case, with the elements in `vs2` promoted to $2 \times \text{SEW}$ bits before adding to the $2 \times \text{SEW}$ -bit accumulator.



`vfwredosum.vs` handles inactive elements and NaN payloads analogously to `vfredosum.vs`; `vfwredusum.vs` does so analogously to `vfredusum.vs`.

30.15. Vector Mask Instructions

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

30.15.1. 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, and the encodings with `vm=0` are reserved. Mask elements past `vl`, the tail elements, are always updated with a tail-agnostic policy.

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

```

vmnor.mm vd, vs2, vs1 # vd.mask[i] = !(vs2.mask[i] || vs1.mask[i])
vmorn.mm vd, vs2, vs1 # vd.mask[i] = vs2.mask[i] || !vs1.mask[i]
vmxnor.mm vd, vs2, vs1 # vd.mask[i] = !(vs2.mask[i] ^^ vs1.mask[i])

```



The previous assembler mnemonics `vmandnot` and `vmornot` have been changed to `vmandn` and `vmorn` to be consistent with the equivalent scalar instructions. The old `vmandnot` and `vmornot` mnemonics can be retained as assembler aliases for compatibility.

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 pseudoinstruction can be retained for compatibility. For implementations that internally rearrange bits according to EEW, a `vmmv.m` instruction with same source and destination can be used as idiom to force an internal reformat into a mask vector.

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	<code>vmxor.mm vd, vd, vd</code>	<code>vmclr.m vd</code>
1	0	0	0	<code>vmnor.mm vd, src1, src2</code>	
0	1	0	0	<code>vmandn.mm vd, src2, src1</code>	
1	1	0	0	<code>vmnand.mm vd, src1, src1</code>	<code>vmnot.m vd, src1</code>
0	0	1	0	<code>vmandn.mm vd, src1, src2</code>	
1	0	1	0	<code>vmnand.mm vd, src2, src2</code>	<code>vmnot.m vd, src2</code>
0	1	1	0	<code>vmxor.mm vd, src1, src2</code>	
1	1	1	0	<code>vmnand.mm vd, src1, src2</code>	
0	0	0	1	<code>vmand.mm vd, src1, src2</code>	
1	0	0	1	<code>vmxnor.mm vd, src1, src2</code>	
0	1	0	1	<code>vmand.mm vd, src2, src2</code>	<code>vmmv.m vd, src2</code>

output				instruction	pseudoinstruction
1	1	0	1	vmorn.mm vd, src2, src1	
0	0	1	1	vmand.mm vd, src1, src1	vmmv.m vd, src1
1	0	1	1	vmorn.mm vd, src1, src2	
0	1	1	1	vmor.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.

30.15.2. Vector count population in mask `vcpop.m`

```
vcpop.m rd, vs2, vm
```



This instruction previously had the assembler mnemonic `vpopc.m` but was renamed to be consistent with the scalar instruction. The assembler instruction alias `vpopc.m` is being retained for software compatibility.

The source operand is a single vector register holding mask register values as described in [Section 30.4.5](#).

The `vcpop.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.

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

The `vcpop.m` instruction writes `x[rd]` even if `vL=0` (with the value 0, since no mask elements are active).

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

30.15.3. `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 reach $2^{(XLEN-1)}$ on any implementation.

The `vfirst.m` instruction writes `x[rd]` even if `vl=0` (with the value -1, since no mask elements are active).

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.

30.15.4. `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
	<code>vmsbf.m v2, v3</code>
0 0 0 0 0 0 1 1	v2 contents
1 0 0 1 0 1 0 1	v3 contents
	<code>vmsbf.m v2, v3</code>
0 0 0 0 0 0 0 0	v2
0 0 0 0 0 0 0 0	v3 contents
	<code>vmsbf.m v2, v3</code>
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
	<code>vmsbf.m v2, v3, v0.t</code>
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 active 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 active elements of the source vector, then all active elements in the destination are written with a 1.

The tail elements in the destination mask register are updated under a tail-agnostic policy.

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 the source register and, if masked, cannot overlap the mask register ('v0').

30.15.5. `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 updated under a tail-agnostic policy.

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 the source register and, if masked, cannot overlap the mask register ('v0').

30.15.6. 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 updated under a tail-agnostic policy.

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 the source register and, if masked, cannot overlap the mask register ('v0').

30.15.7. Example using vector mask instructions

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

```
# char* strcpy(char *dst, const char* src)
strcpy:
    mv a2, a0          # Copy dst
    li t0, -1          # Infinite AVL
loop:
    vsetvli x0, t0, e8, m8, ta, ma # Max length vectors of bytes
    vle8ff.v v8, (a1)      # Get src bytes
    csrr t1, vl           # Get number of bytes fetched
    vmseq.vi v1, v8, 0     # Flag zero bytes
    vfirst.m a3, v1        # Zero found?
    add a1, a1, t1         # Bump pointer
    vmsif.m v0, v1         # Set mask up to and including zero byte.
    vse8.v v8, (a2), v0.t  # Write out bytes
    add a2, a2, t1         # Bump pointer
    bltz a3, loop         # Zero byte not found, so loop

    ret
```

```
# char* strncpy(char *dst, const char* src, size_t n)
strncpy:
    mv a3, a0          # Copy dst
loop:
    vsetvli x0, a2, e8, m8, ta, ma # Vectors of bytes.
    vle8ff.v v8, (a1)      # Get src bytes
    vmseq.vi v1, v8, 0     # Flag zero bytes
    csrr t1, vl           # Get number of bytes fetched
    vfirst.m a4, v1        # Zero found?
    vmsbf.m v0, v1         # Set mask up to before zero byte.
    vse8.v v8, (a3), v0.t  # Write out non-zero bytes
    bgez a4, zero_tail     # Zero remaining bytes.
    sub a2, a2, t1         # Decrement count.
    add a3, a3, t1         # Bump dest pointer
    add a1, a1, t1         # Bump src pointer
    bnez a2, loop         # Anymore?

    ret
```

```

zero_tail:
    sub a2, a2, a4          # Subtract count on non-zero bytes.
    add a3, a3, a4          # Advance past non-zero bytes.
    vsetvli t1, a2, e8, m8, ta, ma # Vectors of bytes.
    vmv.v.i v0, 0           # Splat zero.

zero_loop:
    vse8.v v0, (a3)         # Store zero.
    sub a2, a2, t1          # Decrement count.
    add a3, a3, t1          # Bump pointer
    vsetvli t1, a2, e8, m8, ta, ma # Vectors of bytes.
    bnez a2, zero_loop      # Anymore?

    ret

```

30.15.8. 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.

```

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, vtype.vma=0
    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.

The destination register group cannot overlap the source register and, if masked, cannot overlap the mask register (`v0`).

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;
#     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, ta, ma        # 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
    vcpop.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
```

30.15.9. Vector Element Index Instruction

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

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

The instruction can be masked. Masking does not change the index value written to active elements.

The `vs2` field of the instruction must be set to v_0 , 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.



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

30.16. Vector Permutation Instructions

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

30.16.1. 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] (vs1=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.



`vmv.x.s` performs its operation even if $v_{start} \geq v_l$ or $v_l = 0$.

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 ($0 < \text{index} < VLEN/SEW$) are treated as tail elements using the current tail agnostic/undisturbed policy. If $v_{start} \geq v_l$, no operation is performed and the destination register is not updated.



As a consequence, when $v_l = 0$, no elements are updated in the destination vector register group, regardless of v_{start} .

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

30.16.2. 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.



`vfmv.f.s` performs its operation even if $vstart \geq vl$ or $vl=0$.

The `vfmv.s.f` instruction copies the scalar floating-point register to element 0 of the destination vector register. The other elements in the destination vector register ($0 < \text{index} < VLEN/SEW$) are treated as tail elements using the current tail agnostic/undisturbed policy. If $vstart \geq vl$, 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.

30.16.3. 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 $vstart \geq vl$, 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 tail agnostic/undisturbed policy is followed for tail elements.

The slide instructions may be masked, with mask element i controlling whether *destination* element i is written. The mask undisturbed/agnostic policy is followed for inactive elements.

30.16.3.1. Vector Slideup Instructions

```
vslideup.vx vd, vs2, rs1, vm # vd[i+x[rs1]] = vs2[i]
vslideup.vi vd, vs2, uimm, 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, zero-extended to XLEN bits. 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 (*vstart* < *vl*)

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

$0 \leq i < \min(vl, \max(vstart, OFFSET))$	Unchanged
$\max(vstart, OFFSET) \leq i < vl$	$vd[i] = vs2[i-OFFSET]$
if <i>v0.mask</i> [<i>i</i>] enabled	
$vl \leq i < VLMAX$	Follow tail policy

The destination vector register group for *vslideup* cannot overlap the source vector register group, otherwise the instruction encoding is reserved.



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

30.16.3.2. Vector Slidedown Instructions

```
vslidedown.vx vd, vs2, rs1, vm    # vd[i] = vs2[i+x[rs1]]
vslidedown.vi vd, vs2, uimm, vm   # vd[i] = vs2[i+uimm]
```

For *vslidedown*, the value in *vl* specifies the maximum number of destination elements that are written. The remaining elements past *vl* are handled according to the current tail policy ([Section 30.3.4.3](#)).

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

vslidedown behavior for source elements for element *i* in slide (*vstart* < *vl*)

$0 \leq i+OFFSET < VLMAX$	$src[i] = vs2[i+OFFSET]$
$VLMAX \leq i+OFFSET$	$src[i] = 0$

vslidedown behavior for destination element *i* in slide (*vstart* < *vl*)

$0 \leq i < vstart$	Unchanged
$vstart \leq i < vl$	$vd[i] = src[i]$ if <i>v0.mask</i> [<i>i</i>] enabled
$vl \leq i < VLMAX$	Follow tail policy

30.16.3.3. 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]
```

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 update follows the current mask agnostic/undisturbed policy. If $XLEN < SEW$, the value is sign-extended to SEW bits. If $XLEN > SEW$, the least-significant bits are copied over and the high $XLEN-SEW$ 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 the maximum number of destination vector register elements updated with source values, and remaining elements past `vl` are handled according to the current tail policy ([Section 30.3.4.3](#)).

`vslide1up` behavior when $vl > 0$

$i < vstart$	unchanged
$0 = i = vstart$	$vd[i] = x[rs1]$ if <code>v0.mask[i]</code> enabled
$\max(vstart, 1) \leq i < vl$	$vd[i] = vs2[i-1]$ if <code>v0.mask[i]</code> enabled
$vl \leq i < VLMAX$	Follow tail policy

The `vslide1up` instruction requires that the destination vector register group does not overlap the source vector register group. Otherwise, the instruction encoding is reserved.

30.16.3.4. Vector Floating-Point Slide1up Instruction

```
vfslide1up.vf vd, vs2, rs1, vm      # vd[0]=f[rs1], vd[i+1] = vs2[i]
```

The `vfslide1up` instruction is defined analogously to `vslide1up`, but sources its scalar argument from an `f` register.

30.16.3.5. 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 the maximum number of destination vector register elements written with source values, and remaining elements past `vl` are handled according to the current tail policy ([Section 30.3.4.3](#)).

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

The `vslide1down` instruction places the `x` register argument at location $vl-1$ in the destination vector register, provided that element $vl-1$ is active, otherwise the destination element update follows the current mask agnostic/undisturbed policy. 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 v0.mask[i] enabled
vstart <= i = vl-1    vd[vl-1] = x[rs1] if v0.mask[i] enabled
        vl <= i < VLMAX    Follow tail policy

```



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.

30.16.3.6. Vector Floating-Point Slide1down Instruction

```
vfslide1down.vf vd, vs2, rs1, vm    # vd[i] = vs2[i+1], vd[vl-1]=f[rs1]
```

The `vfslide1down` instruction is defined analogously to `vslide1down`, but sources its scalar argument from an `f` register.

30.16.4. Vector Register Gather Instructions

The vector register gather instructions read 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 maximum 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 30.3.4.3). The operation can be masked, and the mask undisturbed/agnostic policy is followed for inactive elements.

```

vrgather.vv vd, vs2, vs1, vm    # vd[i] = (vs1[i] >= VLMAX) ? 0 : vs2[vs1[i]];
vrgatherei16.vv vd, vs2, vs1, vm # vd[i] = (vs1[i] >= VLMAX) ? 0 : vs2[vs1[i]];

```

The `vrgather.vv` form uses SEW/LMUL for both the data and indices. The `vrgatherei16.vv` form uses SEW/LMUL for the data in `vs2` but EEW=16 and EMUL = (16/SEW)*LMUL for the indices in `vs1`.



When $SEW=8$, `vrgather.vv` can only reference vector elements 0-255. The `vrgatherei16` form can index 64K elements, and can also be used to reduce the register capacity needed to hold indices when $SEW > 16$.

If an element index is out of range ($vs1[i] \geq VLMAX$) 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 active elements of the destination vector register. The index value in the scalar register and the immediate, zero-extended to XLEN bits, 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 the instruction encoding is reserved.

30.16.5. 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 30.3.4.3](#)).

Example use of `vcompress` instruction

8	7	6	5	4	3	2	1	0	Element number
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


```
vsetivli    t0, 9, e8, m1, tu, ma
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 the instruction encoding is reserved.

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 to do that but will instead restart from element 0. This does mean elements in destination register

after vstart will already have been updated.

30.16.5.1. Synthesizing vdecompress

There is no inverse vdecompress provided, as this operation can be readily synthesized using iota and a masked vrgather:

Desired functionality of 'vdecompress'

```
7 6 5 4 3 2 1 0      # vid

      e d c b a      # packed vector of 5 elements
1 0 0 1 1 1 0 1      # mask vector of 8 elements
p q r s t u v w      # destination register before vdecompress

e q r d c b v a      # result of vdecompress
```

```
# v0 holds mask
# v1 holds packed data
# v11 holds input expanded vector and result
viota.m v10, v0          # Calc iota from mask in v0
vrgather.vv v11, v1, v10, v0.t # Expand into destination
```

```
p q r s t u v w      # v11 destination register
      e d c b a      # v1 source vector
1 0 0 1 1 1 0 1      # v0 mask vector

4 4 4 3 2 1 1 0      # v10 result of viota.m
e q r d c b v a      # v11 destination after vrgather using viota.m under mask
```

30.16.6. 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 `nr` value in the opcode is the number of individual vector registers, NREG, to copy. The instructions operate as if `EEW=SEW`, `EMUL = NREG`, effective length `evl= EMUL * VLEN/SEW`.



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



The usual property that no elements are written if $vstart \geq vL$ does not apply to these instructions. Instead, no elements are written if $vstart \geq evL$.



If vd is equal to vs2 the instruction is an architectural NOP, but is treated as a hint to implementations that rearrange data internally that the register group will next be accessed with an EEW equal to SEW.

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 (`simm[2:0]`) using the same encoding as the `nf[2:0]` field for memory instructions (Figure Table 56), i.e., `simm[2:0] = NREG-1`.

The value of `NREG` must be 1, 2, 4, or 8, and values of `simm[4:0]` other than 0, 1, 3, and 7 are reserved.



A future extension may support other numbers of registers to be moved.



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 `vmmerge` 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, and encodings with other vector register numbers are reserved.



A future extension may relax the vector register alignment restrictions.

30.17. 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 trapping vector instruction, while the `vstart` CSR contains the element index on which the trap was 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.

30.17.1. Precise vector traps



We assume most supervisor-mode environments with demand-paging will require 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 nevertheless produce the correct final 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.

In idempotent memory regions, vector store instructions may have updated elements in memory past the element causing a synchronous trap. Non-idempotent memory regions must not have been updated for indices equal to or greater than the element that caused a synchronous trap during a vector store instruction.

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` element index. However, there are a number of cases where this overwrite is prohibited to enable execution of the vector instructions to be idempotent and hence restartable from an earlier index location.

Implementations must ensure forward progress can be eventually guaranteed for the element or segment reported by `vstart`.

30.17.2. 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 profile might specify that interrupts are precise while other traps are imprecise. We assume many embedded implementations will generate only imprecise traps for vector instructions on fatal errors, as they will not require resumable traps.

Imprecise traps shall report the faulting element in `vstart` for traps caused by synchronous vector exceptions.

There is no support for imprecise traps in the current standard extensions.

30.17.3. Selectable precise/imprecise traps

Some profiles 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.

This mechanism is not defined in the current standard extensions.

30.17.4. 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 current standard extensions.



A future extension might define a standard way of saving and restoring opaque microarchitectural state from a vector unit implementation to support context switching with imprecise traps.

30.18. Standard Vector Extensions

This section describes the standard vector extensions. A set of smaller extensions intended for embedded use are named with a "Zve" prefix, while a larger vector extension designed for application processors is named as a single-letter V extension. A set of vector length extension names with prefix "Zvl" are also provided.

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

30.18.1. Zvl*: Minimum Vector Length Standard Extensions

All standard vector extensions have a minimum required VLEN as described below. A set of vector length extensions are provided to increase the minimum vector length of a vector extension.



The vector length extensions can be used to either specify additional software or architecture profile requirements, or to advertise hardware capabilities.

Table 60. Vector length extensions

Extension	Minimum VLEN
Zvl32b	32
Zvl64b	64
Zvl128b	128
Zvl256b	256
Zvl512b	512
Zvl1024b	1024



Longer vector length extensions should follow the same pattern.



Every vector length extension effectively includes all shorter vector length extensions.



Explicit use of the Zvl32b extension string is not required for any standard vector extension as they all effectively mandate at least this minimum, but the string can be useful when stating hardware capabilities.

30.18.2. Zve*: Vector Extensions for Embedded Processors

The following five standard extensions are defined to provide varying degrees of vector support and are intended for use with embedded processors. Any of these extensions can be added to base ISAs with XLEN=32 or XLEN=64. The table lists the minimum VLEN and supported EEWs for each extension as well as what floating-point types are supported.

Table 61. Embedded vector extensions

Extension	Minimum VLEN	Supported EEW	FP32	FP64
Zve32x	32	8, 16, 32	N	N
Zve32f	32	8, 16, 32	Y	N
Zve64x	64	8, 16, 32, 64	N	N
Zve64f	64	8, 16, 32, 64	Y	N
Zve64d	64	8, 16, 32, 64	Y	Y

The Zve32f and Zve64x extensions depend on the Zve32x extension. The Zve64f extension depends on the Zve32f and Zve64x extensions. The Zve64d extension depends on the Zve64f extension.

All Zve* extensions have precise traps.



There is currently no standard support for handling imprecise traps, so standard extensions have to provide precise traps.

All Zve* extensions provide support for EEW of 8, 16, and 32, and Zve64* extensions also support EEW of 64.

All Zve* extensions support the vector configuration instructions ([Section 30.6](#)).

All Zve* extensions support all vector load and store instructions ([Section 30.7](#)), except Zve64* extensions do not support EEW=64 for index values when XLEN=32.

All Zve* extensions support all vector integer instructions ([Section 30.11](#)), except that the `vmulh` integer multiply variants that return the high word of the product (`vmulh.vv`, `vmulh.vx`, `vmulhu.vv`, `vmulhu.vx`, `vmulhsu.vv`, `vmulhsu.vx`) are not included for EEW=64 in Zve64*.



Producing the high-word of a product can take substantial additional gates for large EEW.

All Zve* extensions support all vector fixed-point arithmetic instructions ([Section 30.12](#)), except that `vsmul.vv` and `vsmul.vx` are not included in EEW=64 in Zve64*.



As with `vmulh`, `vsmul` requires a large amount of additional logic, and 64-bit fixed-point multiplies are relatively rare.

All Zve* extensions support all vector integer single-width and widening reduction operations ([Section 30.14.1](#), [Section 30.14.2](#)).

All Zve* extensions support all vector mask instructions ([Section 30.15](#)).

All Zve* extensions support all vector permutation instructions ([Section 30.16](#)), except that Zve32x and Zve64x do not include those with floating-point operands, and Zve64f does not include those with EEW=64 floating-point operands.

The Zve32x extension depends on the Zicsr extension. The Zve32f and Zve64f extensions depend upon the F extension, and implement all vector floating-point instructions ([Section 30.13](#)) for floating-point operands with EEW=32. Vector single-width floating-point reduction operations ([Section 30.14.3](#)) for EEW=32 are supported.

The Zve64d extension depends upon the D extension, and implements all vector floating-point instructions ([Section 30.13](#)) for floating-point operands with EEW=32 or EEW=64 (including widening instructions and conversions between FP32 and FP64). Vector single-width floating-point reductions ([Section 30.14.3](#)) for EEW=32 and EEW=64 are supported as well as widening reductions from FP32 to FP64.

30.18.3. V: Vector Extension for Application Processors

The single-letter V extension is intended for use in application processor profiles.

The `misal.v` bit is set for implementations providing `misal` and supporting V.

The V vector extension has precise traps.

The V vector extension depends upon the Zvl128b and Zve64d extensions.



The value of 128 was chosen as a compromise for application processors. Providing a larger VLEN allows stripmining code to be elided in some cases for short vectors, but also increases the size of the minimum implementation. Note that larger LMUL can be used to avoid stripmining for longer known-size application vectors at the cost of having fewer available vector register groups. For example, an LMUL of 8 allows vectors of up to sixteen 64-bit elements to be processed without stripmining using four vector register groups.

The V extension supports EEW of 8, 16, and 32, and 64.

The V extension supports the vector configuration instructions ([Section 30.6](#)).

The V extension supports all vector load and store instructions ([Section 30.7](#)), except the V extension does not support EEW=64 for index values when XLEN=32.

The V extension supports all vector integer instructions ([Section 30.11](#)).

The V extension supports all vector fixed-point arithmetic instructions ([Section 30.12](#)).

The V extension supports all vector integer single-width and widening reduction operations ([Section 30.14.1](#), [Section 30.14.2](#)).

The V extension supports all vector mask instructions ([Section 30.15](#)).

The V extension supports all vector permutation instructions ([Section 30.16](#)).

The V extension depends upon the F and D extensions, and implements all vector floating-point instructions ([Section 30.13](#)) for floating-point operands with EEW=32 or EEW=64 (including widening instructions and conversions between FP32 and FP64). Vector single-width floating-point reductions ([Section 30.14.3](#)) for EEW=32 and EEW=64 are supported as well as widening reductions from FP32 to FP64.



As is the case with other RISC-V extensions, it is valid to include overlapping extensions in the same ISA string. For example, RV64GCV and RV64GCV_Zve64f are both valid and equivalent ISA strings, as is RV64GCV_Zve64f_Zve32x_Zvl128b.

30.18.4. Zvfhmin: Vector Extension for Minimal Half-Precision Floating-Point

The Zvfhmin extension provides minimal support for vectors of IEEE 754-2008 binary16 values, adding conversions to and from binary32. When the Zvfhmin extension is implemented, the `vfwcvt.f.f.v` and `vfnvcvt.f.f.w` instructions become defined when SEW=16. The EEW=16 floating-point operands of these instructions use the binary16 format.

The Zvfhmin extension depends on the Zve32f extension.

30.18.5. Zvfh: Vector Extension for Half-Precision Floating-Point

The Zvfh extension provides support for vectors of IEEE 754-2008 binary16 values. When the Zvfh extension is implemented, all instructions in [Section 30.13](#), [Section 30.14.3](#), [Section 30.14.4](#), [Section 30.16.2](#), [Section 30.16.3.4](#), and [Section 30.16.3.6](#) become defined when SEW=16. The EEW=16 floating-point operands of these instructions use the binary16 format.

Additionally, conversions between 8-bit integers and binary16 values are provided. The floating-point-to-integer narrowing conversions (`vfnvcvt[.rtz].x[u].f.w`) and integer-to-floating-point widening conversions (`vfwcvt.f.x[u].v`) become defined when SEW=8.

The Zvfh extension depends on the Zve32f and Zfhmin extensions.



Requiring basic scalar half-precision support makes Zvfh's vector-scalar instructions substantially more useful. We considered requiring more complete scalar half-precision support, but we reasoned that, for many half-precision vector workloads, performing the scalar computation in single-precision will suffice.

30.19. Vector Element Groups

Some vector instructions treat operands as a vector of one or more *element groups*, where each element group is a fixed number of elements. For example, complex numbers can be viewed as a two-element group (one real element and one imaginary element). As another example, the SHA-256 cryptographic instructions in the Zvknha extension operate on 128-bit values represented as a 4-element group of 32-bit elements.

This section describes recommendations and terminology for generic instruction set design for vector instructions that operate on element groups.

30.19.1. Element Group Size

The *element group size* (EGS) is the number of elements in one group, and must be a power-of-two (POT).



Support for non-POT EGS was considered but causes many practical complications and so has been dropped. Error checking for `vL` is a little more difficult. For `LMUL>1`, non-POT EGSs will result in groups straddling the individual vector registers in a vector register group. Non-POT EGS can also cause large increases in the lowest-common-multiple of element group sizes, which adds constraints to `vL` setting in order to avoid splitting an element group across stripmine iterations in vector-length-agnostic code.

The element group size is statically encoded in the instruction, often implicitly as part of the opcode.

Executing a vector instruction with $EGS > VLMAX$ causes an illegal instruction exception to be raised.



The vector instructions in the base V vector ISA can be viewed as all having an element group size of 1 for all operands statically encoded in the instruction.



Many operations only make sense with a certain number of elements per group (e.g., complex operations require a element group size of 2 and SHA-256 requires an element group size of 4).

30.19.2. Setting vL

Each source and destination operand to a vector instruction might be defined as either a single element group or a vector of element groups. When an operand is a vector of element groups, the vL setting must correspond to an integer multiple of the element group size, with other values of vL reserved.



For example, a SHA-256 instruction would require that vL is a multiple of 4.

When element group instructions are present, an additional constraint is placed on the setting of vL based on an AVL value (augmenting [Section 30.6.3](#)). $EGSMAX$ is the largest EGS supported by the implementation. When $AVL > VLMAX$, the value of vL must be set to either $VLMAX$ or a positive integer multiple of $EGSMAX$.



As the base vector extension only has element group size of 1, this constraint is backwards-compatible.



This constraint prevents element groups being broken across stripmining iterations in vector-length-agnostic code when a $VLMAX$ -size vector would otherwise be able to accommodate a whole number of element groups.



If EEW is encoded statically in the instruction, or if an instruction has multiple operands containing vectors of element groups with different EEW , an appropriate SEW must be chosen for $vsetvL$ instructions.



Additional constraints may be required for some element group instructions to ensure legal length values for all operands.

30.19.3. Determining EEW

The $vtype$ SEW can be used to indicate or calculate the effective element size (EEW) of one or more operands of an element group instruction. Where the operand is an element group, SEW and EEW refer to the number of bits in each individual element within a group not the number of bits in the group as a whole.

Alternatively, the opcode might encode EEW of all operands statically and ignore the value of SEW when the operation only makes sense for a single size on each operand.



Many operations are only defined for one EEW , e.g., SHA-256 requires $EEW=32$. Encoding $EEWs$ statically in the instruction removes a dynamic dependency on the SEW value and the need to check for errors in SEW values. However, ignoring SEW also prevents reuse of the static opcode with a different dynamic SEW , and in many cases, the SEW setting will be needed for regular vector instructions used to process

the individual elements in the vector.

30.19.4. Determining EMUL

The `vtype` LMUL setting can be used to indicate or calculate the effective length multiplier (EMUL) for one or more operands. Element group instructions tend to exhibit a much wider range of relationships between various operand EEW/EMUL values. For example, an instruction might take a vector of length N of 4-element groups with EEW=8b and reduce each group to produce a vector length N of 1-element groups with EEW=32b. In this case, the input and output EMUL values are equal even though the EEW settings differ by a factor of 4.

Each source and destination operand to a vector instruction may have a different element group size, different EMUL, and/or different EEW.

30.19.5. Element Group Width

The *element group width* (EGW) is the number of bits in the element group as a whole. For example, the SHA-256 instructions in the Zvknha extension operate on an EGW of 128, with EGS=4 and EEW=32. It is possible to use LMUL to concatenate multiple vector registers together to support larger EGW>VLEN.



If software using large-EGW instructions need be portable across a range of implementations, some of which may have VLEN<EGW and hence require LMUL>1, then software can only use a subset of the architectural registers. Profiles can set minimum VLEN requirements to inform authors of such software.



Element group operations by their nature will gather data from across a wider portion of a vector datapath than regular vector instructions. Some element group instructions might allow temporal execution of individual element operations in a larger group, while others will require all EGW bits of a group to be presented to a functional unit at the same time.

30.19.6. Masking

No ratified extensions include masked element-group instructions. Future extensions might extend the element-group scheme to support element-level masking, or might define the concept of a *mask element group* (which might, e.g., update the destination element group if any mask bit in the mask element group is set).

30.20. Vector Instruction Listing

Integer					Integer				FP			
funct3					funct3				funct3			
OPIV	V				OPMV	V			OPFV	V		
OPIVX		X			OPMVX		X		OPFVX		F	
OPIVI			I									

funct6					funct6					funct6			
000000	V	X	I	vadd	000000	V		vredsum	000000	V	F	vfadd	
000001					000001	V		vredand	000001	V		vfredusum	
000010	V	X		vsub	000010	V		vredor	000010	V	F	vfsub	
000011		X	I	vrsb	000011	V		vredxor	000011	V		vfredosum	
000100	V	X		vminu	000100	V		vredminu	000100	V	F	vfmin	
000101	V	X		vmin	000101	V		vredmin	000101	V		vfredmin	
000110	V	X		vmaxu	000110	V		vredmaxu	000110	V	F	vfmax	
000111	V	X		vmax	000111	V		vredmax	000111	V		vfredmax	
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	vrgather	001100				001100				
001101					001101				001101				
001110		X	I	vslideup	001110		X	vslide1up	001110		F	vfslide1up	
001110	V			vrgatheri16									
001111		X	I	vslidedown	001111		X	vslide1down	001111		F	vfslide1down	

funct6					funct6					funct6				
010000	V	X	I	vadc	010000	V		VWXU	NARY0	010000	V		VWFUN	ARY0
					010000		X	VRXUN	ARY0	010000		F	VRFUN	ARY0
010001	V	X	I	vadc	010001					010001				
010010	V	X		vsbc	010010	V		VXUNA	RY0	010010	V		VFUNA	RY0

funct6					funct6				funct6			
010011	V	X		vmsbc	010011				010011	V		VFUNARY1
010100					010100	V		VMUNARY0	010100			
010101					010101				010101			
010110					010110				010110			
010111	V	X	I	vmmerge/vmv	010111	V		vcompress	010111		F	vmmerge/vfmv
011000	V	X	I	vmseq	011000	V		vmmandn	011000	V	F	vmfeq
011001	V	X	I	vmsne	011001	V		vmmand	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		vmorn	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	vfdv
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
100111			I	vmv<nr>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

funct6					funct6				funct6			
101011	V	X	I	vssra	101011	V	X	vnmsu b	101011	V	F	vfnmsu b
101100	V	X	I	vnsrl	101100				101100	V	F	vfmacc
101101	V	X	I	vnsra	101101	V	X	vmacc	101101	V	F	vfnmac c
101110	V	X	I	vnclipu	101110				101110	V	F	vfmsac
101111	V	X	I	vnclip	101111	V	X	vnmsac	101111	V	F	vfnmsa c

funct6					funct6				funct6			
110000	V			vwredsu mu	110000	V	X	vwaddu 0	110000	V	F	vwadd
110001	V			vwredsu m	110001	V	X	vwadd	110001	V		vwred sum
110010					110010	V	X	vwsubu	110010	V	F	vfwsu b
110011					110011	V	X	vwsu b	110011	V		vwred osum
110100					110100	V	X	vwaddu .w	110100	V	F	vwadd. w
110101					110101	V	X	vwadd. w	110101			
110110					110110	V	X	vwsubu .w	110110	V	F	vfwsu b. w
110111					110111	V	X	vwsu b. w	110111			
111000					111000	V	X	vwmulu	111000	V	F	vfwmul
111001					111001				111001			
111010					111010	V	X	vwmuls u	111010			
111011					111011	V	X	vwmul	111011			
111100					111100	V	X	vwmac cu	111100	V	F	vfwmac c
111101					111101	V	X	vwmac c	111101	V	F	vfwnma cc
111110					111110		X	vwmac cus	111110	V	F	vfwm sa c
111111					111111	V	X	vwmac csu	111111	V	F	vfwnms ac

Table 62. VRXUNARY0 encoding space

vs2	
00000	vmv.s.x

Table 63. VWXUNARY0 encoding space

vs1	
00000	vmv.x.s
10000	vcpop
10001	vfirst

Table 64. VXUNARY0 encoding space

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

Table 65. VRFUNARY0 encoding space

vs2	
00000	vfmv.s.f

Table 66. VWFUNARY0 encoding space

vs1	
00000	vfmv.f.s

Table 67. 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

vs1	name
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 68. VFUNARY1 encoding space

vs1	name
00000	vfsqrt.v
00100	vfrsqrt7.v
00101	vfrec7.v
10000	vfclass.v

Table 69. VMUNARY0 encoding space

vs1	
00001	vmsbf
00010	vmsof
00011	vmsif
10000	viota
10001	vid