RISC-V Vector (V) Extension Intrinsics

Document Number UM231-15

Date Issued 2024-06-12

Copyright Notice

Contact Information

Should you have any problems with the information contained herein, you may contact Andes Technology Corporation through

- email – support@andestech.com
- Website – https://es.andestech.com/eservice/

Please include the following information in your inquiries:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem

General suggestions for improvements are welcome.

Revision History

Rev.	Revision Date	Revised Content
1.5	2024/06/12	Fixed typos on zvfbfmin intrinsics (Section 7.16.19, 7.16.20, 7.17.19 and 7.17.20)
1.4	2023/12/05	Upgraded from RVV Intrinsic spec v0.11 to v0.12
1.3	2023/07/24	Upgraded from RVV Intrinsic spec v0.10 to v0.11.
1.2	2021/05/28	1. Updated to support RVV spec. v0.10. 2. Synced with upstream changes to support explicit VL argument.
1.1	2021/02/19	1. Added and modified some prototypes of vector register gather functions. 2. Removed two instructions of Andes load Int4 functions.
1.0	2020/12/29	Initial release

Table of Contents

Typographical Convention Index

Document Element	Font	Font Style	Size	Color
Normal text	Georgia	Normal	12	Black
`Command line, source code or file paths`	`Lucida console`	`Normal`	`11`	`Indigo`
`VARIABLES OR PARAMETERS IN COMMAND LINE, SOURCE CODE OR FILE PATHS`	**`LUCIDA CONSOLE`**	**`BOLD + ALL-CAPS`**	**`11`**	**`INDIGO`**
Hyperlink	Georgia	Underlined	12	Blue

1. Overview of vector extension ISA

This section is a brief overview of the RVV extension ISA and it focuses on the necessary fundamentals that are used in the C intrinsic APIs. These descriptions are not exhaustive. For more details, please see the RVV extension ISA. Every effort has been made to provide definitions ahead of an usage to simplify the reading of this document.

1.1. Vector registers and vector elements

A basic understanding of the vector registers and vector control registers is needed before using the RVV C API intrinsic functions since this information is part of the API.

1.1.1. Number of registers and register size

There are 32 vector registers, named v0..v31. Each register has an implementation defined `VLEN`, which is the maximum size in bits for a single vector register. Typically, the value could be 256, 512, 1024, etc., and it is a constant value that cannot be dynamically changed. Different SoCs or chips can have a different `VLEN` that needs to be taken into account when using the different vector programming models.

1.1.2. Element widths – SEW and EEW

`ELEN` is the maximum size in bits of a vector element that any operation can produce or consume, which is a constant value used by instruction encodings. The number of elements in a single vector register is `VLEN/ELEN`.

`EEW` is the effective element width of a vector operand with which a vector register is written. This value is used with mixed width encodings where the source register element widths differ from the destination vector element widths and vice versa.

`SEW` is the selected element width of a vector operand. In general, `VLEN/SEW` defines the number of elements in a vector register. Typically, `EEW` is equal to `SEW`, although certain instructions have differing `EEW` values from the selected SEW. For example, a widening operation will have an `EEW` equal to `SEW*2` and a narrowing instruction would have an `EEW` of `SEW/2`.

For example, with a VLEN equal to 128:

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

1.1.3. Register groups – ELMUL/LMUL

Multiple vector registers can be grouped together, so that a single vector instruction can operate on multiple vector registers. The term vector register group, or LMUL, is used 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.

Note that 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 only occupy 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 a few cases, having more shorter register-resident vectors improves efficiency relative to fewer longer register-resident vectors.

For ELEN equal to 64, there are three fractional LMUL values: 1/8, 1/4, and 1/2. For ELEN equal to 32 implementations, there would be only two fractional LMUL values: 1/2 and 1/4.

1.1.4. VLMAX

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

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

1.1.5. Mask register

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 in bit i of the mask register, independent of SEW or LMUL.

NOTE

The current implementation only supports one vector mask register, v0, which is used for predication.

1.1.6. Vector start register vstart and vector length register vl

The `vstart` register specifies the index of the first element to be executed by the current vector instruction. This register sets the index of the first element to zero when executing a vector instruction. It is updated via hardware, and it plays a part in cases like operating system context switches. When an interrupt or exception is taken, the value of `vstart` is set to the element at which the exception occurred. Upon restart, the element pointed to by `vstart` is the first element to be executed while the elements < `vstart` are unchanged.

The `vl` register contains the number of elements to be updated with results from a vector instruction and is set by the vector configuration instructions. The use of `vstart` and `vl` are described here since they play a part in how the regions of a vector register are acted upon.

1.1.6.1 Prestart, active, inactive, body, and tail elements definition

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 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, 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.

Programmatically, these definitions can be described as:

```
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
      actve(x)         = body(x) && mask(x)
      inactive(x)      = body(x) && !mask(x)
```


NOTE

When `vl=0`, no elements, including agnostic elements, are updated in the destination vector register group regardless of `vstart`.

1.1.7. Vector tail agnostic and vector agnostic subregisters vta and vma

These registers 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.

1.1.7.1 Tail policy register vta

Vector instructions support two types of tail policy:

- Tail-undisturbed (`vta = 0`) means the tail elements of the destination are preserved.
- Tail-agnostic (`vta = 1`) means that each tail element of the destination can be:
  - left undisturbed,
  - overwritten by ones, or,
  - for mask-producing instructions besides mask loads, which are overwritten by the result of the instruction as if `vl` were larger.

This behavior may vary element-by-element, and may vary across executions of the same instruction for the same inputs.

Some instructions have the same behavior regardless of the `vta` register:

- Instructions that produce a mask register have tail-agnostic policy regardless of `vta`.

- Instructions with no tail elements, like stores or those whose output is an X- or F-register, have the same behavior regardless of `vta`.
- Some instructions have tail elements whose values are guaranteed to be unchanged regardless of `vta`, for example, a vector load whose destination's tail is already all ones.

1.1.7.2 Masking and mask policy

Vector instructions can be considered masked or unmasked.

- Masked instructions are those that use of the mask register `v0` as an indicator of which body elements are active/inactive.
- Unmasked instructions are those that take all body elements as active, independent of the mask register `v0`.

Inactive elements' behavior depends on the mask policy, specified by the `vma` register:

- Mask-undisturbed (`vma = 0`) means the inactive elements are preserved.
- Mask-agnostic (`vma = 1`) means that each inactive element can be left undisturbed or overwritten by ones.

This behavior may vary element-by-element and may vary across executions of the same instruction for the same inputs.

Some instructions have the same behavior regardless of `vma`:

- Unmasked instructions,
- Instructions with all body elements active
- Reductions
- Stores
- Instructions whose output is an X- or F-register
- Instructions with no body elements (`vl = 0` or `vstart >= vl`).

Programmatically, there are four options for using `vta` and `vma` registers:

vta	vma	Tail elements	Inactive elements
0 (`tu`)	0 (`mu`)	undisturbed	undisturbed
0 (`tu`)	1 (`ma`)	undisturbed	agnostic
1 (`ta`)	0 (`mu`)	agnostic	undisturbed
1 (`ta`)	1 (`ma`)	agnostic	agnostic

1.1.8. Vector type register vtype

`vtype` provides the default type to interpret the contents of the vector register file and can only be updated by the configuration 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.

The register is a combination of the following registers and values:

- `vma` – vector masked agnostic
- `vta` – vector tail agnostic
- `vsew` – selected element width (`SEW`)
- `vlmul` – vector register group multiplier (`LMUL`)

For details on `vma` and `vta` registers, please refer to Section 1.1.7.

1.2. Mapping of vector elements to vector register state

The diagrams in the following sections 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.

NOTE

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

1.2.1. 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. In the following LUML=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=8  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
```

1.2.2. 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.

```
VLEN=128b, LMUL=1/4
Byte      F E D C B A 9 8 7 6 5 4 3 2 1 0
SEW=8b    - - - - - - - - - - - - 3 2 1 0
SEW=16b     -   -   -   -   -   -   1   0
SEW=32b       -       -       -       0
```

1.2.3. 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. For example,


```
VLEN=32b, SEW=8b, LMUL=2
Byte      3 2 1 0

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

VLEN=32b, SEW=16b, LMUL=2
Byte      3 2 1 0

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

VLEN=32b, SEW=16b, LMUL=4
Byte      3 2 1 0

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

VLEN=32b, SEW=32b, LMUL=4
Byte      3 2 1 0

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

VLEN=64b, SEW=32b, LMUL=2
Byte      7 6 5 4 3 2 1 0
v2*n            1       0
v2*n+1          3       2

VLEN=64b, SEW=32b, LMUL=4
Byte      7 6 5 4 3 2 1 0
v4*n            1       0
v4*n+1          3       2
v4*n+2          5       4
v4*n+3          7       6


VLEN=128b, SEW=32b, LMUL=2
Byte      F E D C B A 9 8 7 6 5 4 3 2 1 0

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

VLEN=128b, SEW=32b, LMUL=4
```

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

1.2.4. Mapping across mixed-width operations

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). The following example shows four different packed element widths (8b, 16b, 32b, 64b) in a **VLEN**=128b implementation. The vector register grouping factor (**LMUL**) is increased by the relative element size such that each group can hold the same number of vector elements (**VLMAX**=8 in this example) to simplify strip-mining code.


```
VLEN=128b, with SEW/LMUL=16
Byte   F E D C B A 9 8 7 6 5 4 3 2 1 0
vn     - - - - - - - - 7 6 5 4 3 2 1 0 SEW=8b, LMUL=1/2
vn       7   6   5   4   3   2   1   0 SEW=16b, LMUL=1

v2*n         3       2       1       0 SEW=32b, LMUL=2
v2*n+1       7       6       5       4

v4*n                 1               0 SEW=64b, LMUL=4
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 the table are **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

2. RVV C API intrinsic function formats

The C API is made up of data types, intrinsic functions that map to specific assembly-level instructions, and utility functions that aid in the programming of the vector unit.

2.1. Vector data types

The `SEW` and `LMUL` values are encoded in the data types and `LMUL >= SEW/ELEN` is enforced.

Vector data types are encoded as follows:

```
v DTYPE ELEN LMUL _t
```

where `DTYPE ELEN` is one of the following

`int64`	Signed long
`uint64`	Unsigned long
`int32`	Signed integer
`uint32`	Unsigned integer
`int16`	Signed short
`uint16`	Unsigned short
`int8`	Signed char
`uint8`	Unsigned char
`float64`	double
`float32`	float
`float16`	_Float16
`bfloat16`*	__bf16

* `vbfloat16` is an interchange format used exclusively for floating-point conversions to and from `vfloat32` types as well as loads and store instructions.

and `LMUL` is one of the following:

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

For example, `vint16m4_t` represents 16-bit signed integer elements with `LMUL = 4`.

The disambiguation of types with the same size allows for a strongly typed implementation that does not have implicit conversions between data types.

2.2. Vector mask types

The ratio `SEW/LMUL` is encoded into the mask type as follows:

```
vbool N _t
```

Where `N` is 1, 2, 4, 8, 16, 32, or 64.

2.3. Tuple types (structure of sizeless vector data types)

For return values of segmented load instructions, source arguments of store instructions, and arguments of get and set intrinsic functions, additional syntax is needed to describe the data type they use. The tuple types defined here match the limitations of the vector hardware, which is `NFIELD * ELMUL <= 8`.

Data type	ELEN value	NFIELDS value
`v DTYPE ELEN mf8 x NFIELDS _t`	8	2, 3, 4, 5, 6, 7, 8
`v DTYPE ELEN mf4 x NFIELDS _t`	8, 16	2, 3, 4, 5, 6, 7, 8
`v DTYPE ELEN mf2 x NFIELDS _t`	8, 16, 32	2, 3, 4, 5, 6, 7, 8
`v DTYPE ELEN m1 x NFIELDS _t`	8, 16, 32, 64	2, 3, 4, 5, 6, 7, 8
`v DTYPE ELEN m2 x NFIELDS _t`	8, 16, 32, 64	2, 3, 4
`v DTYPE ELEN m4 x NFIELDS _t`	8, 16, 32, 64	2
`v DTYPE ELEN m8 x NFIELDS _t`	X	X

For example, the tuple data type `vint32m1x3_t` is defined to be `3` vectors of data type `vint32m1_t` where one can get or set `3` values.

2.4. Intrinsic function naming rules

The intrinsic functions is the interface to the low level assembly from a high level programming language. The intrinsic API has the goal to make all the RVV instructions accessible from C/C++. The intrinsic names are as close as possible to the assembly mnemonics.

The intrinsic names may encode a return type so that it is easier to know the output type of an intrinsic function from its name. In addition, if an intrinsic function call is used as an operand to another intrinsic function call, it is simple to see what kind of data type the operand has since it is defined in the intrinsic function name. If there is no return value, the intrinsic functions will encode the input value types.

In general, the naming rules of an intrinsic function are as follows:

```
INTRINSIC ::= MNEMONIC '_' RET_TYPE ['_' TM]
MNEMONIC ::= Instruction name in RVV specification. Replace '.' with '_'.
RET_TYPE ::= SEW LMUL
SEW ::= ( i8 | i16 | i32 | i64 | u8 | u16 | u32 | u64 | f16 | f32 | f64 |
bf16)
LMUL ::= ( m1 | m2 | m4 | m8 | mf2 | mf4 | mf8 )
TM ::= ( m | mu | tu | tum | tumu )
```

where `SEW` is one of the following

`i64`	Signed long
`u64`	Unsigned long
`i32`	Signed integer
`u32`	Unsigned integer
`i16`	Signed short
`u16`	Unsigned short
`i8`	Signed char
`u8`	Unsigned char
`f64`	double
`f32`	float
`f16`	_Float16
`bf16`*	__bf16

* `bf16` is an interchange format used exclusively for floating-point conversions to and from `vfloat32` types as well as loads and store instructions.

For example,

```
vadd.vv vd, vs2, vs1: // add instruction defined in RVV ISA
specification
vint8m1_t vadd_vv_i8m1(vint8m1_t vs2, vint8m1_t vs1, size_t vl);

vwaddu.vv vd, vs2, vs1:
vint16m2_t vwaddu_vv_i16m2(vint8m1_t vs2, vint8m1_t vs1, size_t vl);
```


There are exceptions to the naming rules. They will be described in later sections. An example of this would be store instructions which do not have a return type.

To support the implied semantics, the compiler may generate multiple instructions for the intrinsic function. That is, this API does not constrain the compiler in which instructions it actually generates.

There are policy intrinsic functions where `TM` has either of the following options and associated attributes. Exceptions to the policy rules will be described in later sections.

TM in suffix	Masked?	Tail undisturbed?	Mask undisturbed?
(none)	no	no	N/A
`tu`	no	yes	N/A
`tum`	yes	yes	no
`m`	yes	no	no
`mu`	yes	no	yes
`tumu`	yes	yes	yes

For the policy rules, an N/A means a compiler defined value. In the case, the compiler has the flexibility to choose which policy to use based on surrounding instructions. This helps reduce the required configuration changes when dealing with instructions with differing policies.

For mask undisturbed policy intrinsic functions, there is an additional argument known as `maskedoff` to the functions. This argument provides elements to be inserted into the destination register when the mask element is zero.

```
if (mask[i] == 0) dest[i] = maskedoff[i];
else dest[i] = op_result[i];
```

For example,

```
vint16m2_t vwaddu_vv_i16m2_mu(vbool16_t mask, vint16m2_t maskedoff,
                              vint8m1_t vs2, vint8m1_t vs1, size_t vl);
```

Note that `maskedoff` always has the type of the return operand, not that of the sources.

2.5. psABI calling conventions

At this point, the psABI has not been extended to formalize the calling conventions for functions using vector types. This implies that vector function arguments, vector return, and vector callee save/restore may alter in the future. Care should be taken if calling functions are written in assembly language. Passing vector arguments may also have a performance penalty, depending on how many argument registers are defined using vector registers instead of memory.

2.6. Conversion instructions

Vector data types are strongly typed. There is no implicit conversions between these types.

2.6.1. Reinterpret cast conversion instructions

The instructions convert one data type to another. The `SEW` is different between the source and the destination while keeping the same `vl`.

The generalized form of the reinterpret cast intrinsic functions is:

```
__riscv_vreinterpret_v_ SEW LMUL _ SEW LMUL
```

Where the first `SEW`/`LMUL` pair is the type of the source operand while the second `SEW`/`LMUL` is the return type.

For example

```
vfloat32m1_t vreinterpret_i_f (vint32m1_t src)
{
  return __riscv_vreinterpret_v_i32m1_f32m1 (src);
}
```

2.6.2. Vector LMUL extension instructions

The instructions extend the current LMUL value while maintaining the same SEW.

The generalized form of the LMUL extension intrinsic functions is:

```
__riscv_vlmul_ext_v_ SEW LMUL _ SEW LMUL
```

Where the first SEW/LMUL pair is the type of the source operand and the second SEW/LMUL is the return type.

For example,

```
vfloat32m2_t lmul_ext (vfloat32m1_t opd1)
{
  return __riscv_vlmul_ext_v_f32m1_f32m2 (opd1);
}
```

2.6.3. Vector LMUL truncation instructions

The instructions truncate the current LMUL value while maintaining the same SEW.

The generalized form of the LMUL truncation intrinsic functions is:

```
__riscv_vlmul_trunc_v_ SEW LMUL _ SEW LMUL
```

Where the first SEW/LMUL pair is the type of the source operand while the second SEW/LMUL is the return type.

For example,

```
int32m1_t lmul_trunc (int32m2_t opd1)
{
return  __riscv_vlmul_trunc_v_i32m2_i32m1 (opd1);
}
```

2.6.4. Vector insertion instructions

The instructions insert a vector into another vector at the specified index.

The generalized form of the vector insertion intrinsic functions is:

```
__riscv_vset_v_ SEW LMUL _ SEW LMUL [x NFIELDS]
```

Where the first SEW/LMUL pair is the type of the vector to be inserted while the second SEW/LMUL is the type of the destination vector and the return type of the newly created vector.

For example,

```
vfloat32m2_t vset(vfloat32m1_t val,
                  vfloat32m2_t dest,
                  size_t index)
{
  return __riscv_vset_v_f32m1_f32m2 (dest, index, val);
}
```


2.6.5. Vector extract instructions

The instructions extract a vector at the specified index.

The generalized form of the vector extract intrinsic functions is:

```
__riscv_vget_v_ SEW LMUL [x NFIELDS] _ SEW LMUL
```

Where the first `SEW`/`LMUL` pair is the type of the vector to be extracted from while the second `SEW`/`LMUL` is the return type of the newly created vector.

For example,

```
vfloat32m1_t vget(vfloat32m2_t src,
                  size_t index)
{
return __riscv_vget_v_f32m2_f32m1 (src, index);
}
```

2.6.6. Vector tuple creation instructions

The instructions create a tuple from the specified vectors.

The generalized form of the vector tuple creation intrinsic functions is:

```
__riscv_vcreate_v_ SEW LMUL x NFIELDS
```

Where the **`SEW`**/**`LMUL`** is the type of the destination vector and the return type of the newly created vector and the **`NFIELDS`** value is the number of the source arguments.

For example,

```
vfloat32m1x2_t vcreate(vfloat32m1_t opd1,
                       vfloat32m1_t opd2)
{
  return __riscv_vcreate_v_f32m1x2 (opd1, opd2);
}
```

2.7. Fixed-point and floating-point rounding modes

You can use intrinsic functions with specific rounding modes to control the rounding of instructions. Note that there is no need to specify a rounding mode, as there are intrinsic functions available with and without rounding modes.

2.7.1. Fixed-point rounding mode

The following enumeration defines rounding modes that can be used to control the rounding of fixed-point intrinsic functions.

```
enum __RISCV_VXRM {
  __RISCV_VXRM_RNU = 0,
  __RISCV_VXRM_RNE = 1,
  __RISCV_VXRM_RDN = 2,
  __RISCV_VXRM_ROD = 3,
  };
```

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

enum	Rounding operation	Rounding increment r
`__RISCV_VXRM_RNU`	Round to nearest up (add +0.5 to LSB)	`v[d-1]`
`__RISCV_VXRM_RNE`	Round to nearest even	`v[d-1] & (v[d-2:0] = 0 | v[d])`
`__RISCV_VXRM_RDN`	Round down (truncate)	`0`
`__RISCV_VXRM_ROD`	Round to odd (OR bits into LSB)	`!v[d] & (v[di1:0] = 0)`

2.7.2. Floating-point rounding mode

There are two classes of floating-point intrinsic functions: implicit `frm` and explicit `frm`.

2.7.2.1 Implicit `frm` intrinsic functions

The implicit `frm` intrinsic functions are intended to be used regardless of `FENV_ACCESS`. Behaving like any C-language floating-point expressions, they use the `fenv` dynamic

rounding mode when FENV_ACCESS is on for programmers already using fenv and provide the default rounding mode when FENV_ACCESS is off.

2.7.2.2 Explicit frm intrinsic functions

The explicit frm intrinsic functions contain the frm operand which indicates the rounding mode control. The floating-point intrinsic functions with the frm operand are followed by an _rm suffix in their function names.

The explicit frm intrinsic functions are intended to be used when FENV_ACCESS is off, to enable more aggressive optimization while still providing programmers with control over the rounding mode. Using explicit frm intrinsic functions when FENV_ACCESS is on will still work correctly, but it is expected to lead to extra saving and restoring of frm, which could be avoided by using fenv functionality and implicit frm.

The following enumeration defines rounding modes that can be used to control the rounding of floating-point intrinsic functions.

```
enum __RISCV_FRM {
  __RISCV_FRM_RNE = 0,
  __RISCV_FRM_RTZ = 1,
  __RISCV_FRM_RDN = 2,
  __RISCV_FRM_RUP = 3,
  __RISCV_FRM_RMM = 4,
};
```

where the rounding modes are specified as follows.

enum	Rounding operation
__RISCV_FRM_RNE	Round to nearest, ties to even
__RISCV_FRM_RTZ	Round towards zero
__RISCV_FRM_RDN	Round down (towards -∞)
__RISCV_FRM_RUP	Round up (towards +∞)
__RISCV_FRM_RMM	Round to nearest, ties to max magnitude

2.8. Source code examples using intrinsic functions

While each instruction comes in many variations having 12 possible SEW options, 7 possible LMUL options, 6 possible TM options, and 2 possible RM options, the example code within this

document only covers a subset of these variations. In general, each instruction is provided with example code for the following variations, each of which uses one `DTYPE`, one `SEW`, and one `LMUL`:

- Normal instruction – no `TM` or `RM`
- Instruction using `RM` when applicable
- Masked instruction using `_m` from `TM` when applicable
- Mask undisturbed policy instruction using `_mu` from `TM` when applicable

Additionally, no intrinsic function is used before being defined in each example. The examples are simplistic in this regard – they are there for clarity.

3. Configuration instructions vsetvl and vsetvlmax

The RVV instruction set is a very flexible set of instructions that can be programmed to operate on differing number of elements and element types. The `vsetvl` function is used to get the active vector length (`vl`) according to the given application vector length (**AVL**), **SEW** and **LMUL**. The instruction sets the `vltype` register as well. You can treat `vsetvl` as a function which returns the minimum value between **AVL** and **VLMAX** and `vsetvlmax` as returning **VLMAX**.

One of the common approaches to handling a large number of elements is "strip-mining" 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.

The naming of the `vsetvl` function is

```
vsetvl_SEW LMUL
```

where **SEW** specifies the element width – e8, e16, e32, e64 and **LMUL** specifies the desired **LMUL** value. Both **SEW** and **LMUL** are static information for the intrinsic functions.

Many of the intrinsic functions have a `vl` argument to specify the active vector length. Exceptions are described in the appropriate sections below.

The intrinsic functions will only operate at most **VLMAX** elements if the `vl` arguments are larger than **VLMAX**.

The following are some example intrinsic function prototypes:

```
size_t __riscv_vsetvl_e8m1 (size_t avl);
size_t __riscv_vsetvl_e8m2 (size_t avl);
size_t __riscv_vsetvlmax_e8m2();
```

Code example:

```
size_t vl = __riscv_vsetvl_e8m1 (avl);
vint8m1_t va, vb, vc;
va = __riscv_vadd_vv_i8m1(vb, vc, vl);
```

To effectively utilize the resources available, the following code example shows how to write portable code that is agnostic to the underlying hardware. If you were to have a C loop such as this:


```
for (int I = 0; I < AVL; i++)
  {
    …
  }
```

You can modify the code like below:

```
If (AVL) do {
  size_t vl = __riscv_vsetvl_e8m1(AVL); // vl <= vlmax
  …
  AVL -= vl;
  } while (AVL > 0);
}
```

Please realize that there are many ways to write the loop in C and the above example is for clarity.

4. Load/store instructions

Vector loads and stores move values between vector registers and memory. Vector loads and stores are masked and do not raise exceptions on inactive elements. Masked vector loads do not update inactive elements in the destination vector register group, unless masked agnostic is specified. Masked vector stores only update active memory elements. All vector loads and stores may generate and accept a non-zero `vstart` value.

4.1. Load/store addressing modes

The vector extension supports unit-stride, strided, and indexed (scatter/gather) addressing modes.

- Vector unit-stride operations access elements stored contiguously in memory.
- Vector constant-strided operations access the first memory element at the base effective address, and then access subsequent elements at address increments provided in the instruction.

Vector indexed operations add the contents of each element of the vector offset operand 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`.

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 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.

4.2. Vector load/store width encoding

Vector loads and stores have an EEW encoded directly in the intrinsic function. The corresponding EMUL is calculated as EMUL = (EEW/SEW)*LMUL.

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.

4.3. Vector unit-stride load/store instructions

Unit stride instructions are simply an array of values such that the EEW of the instruction is equivalent to SEW – the stride of the memory accesses is one.

The generalized form of the unit-stride load intrinsic functions is:

```
__riscv_vl EEW _v_ SEW LMUL [_ TM]
```

Since there is no return type for a store instruction, the SEW LMUL of the source operand is encoded in the intrinsic name. The generalized form of the unit-stride store intrinsic functions is:

```
__riscv_vs EEW _v_ SEW LMUL
```

Example:

```
float16_t *base, *dest;
size_t vl;
vfloat16mf4_t v1 = __risc_vle16_v_f16mf4(base, vl);
__riscv_vse16_v_f16mf4(dest, v1, vl);
```

4.4. Vector unit-stride load/store masked instructions

Unit-stride masked 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=ceil(vl/8) (i.e. EMUL=1), and the destination register is always written with a tail-agnostic policy.

These instructions also provide a convenient mechanism to use packed bit vectors in memory as mask values and reduce the cost of mask spill/fill by reducing the need to change vl.

The generalized form of the unit-stride masked load intrinsic functions is:

```
__riscv_vl EEW _v_ SEW LMUL _ TM
```

Since there is no return type for a store instruction, the `SEW LMUL` of the source operand is encoded in the intrinsic name. The generalized form of the unit-stride store masked intrinsic functions is:


```
__riscv_vs EEW _v_ SEW LMUL _m
```

Example:

```
vfloat16mf4_t v1;
vbool64_t mask;
float16_t *base, *dest;
size_t vl;
v1 = __riscv_vle16_v_f16mf4_m (mask, base, vl);
__riscv_vse16_v_f16mf4_m (mask, dest, v1, vl);
```

4.5. Vector strided load/store instructions

Strided loads are essentially a gather operation while strided stores are a scatter operation. The stride of the instruction can be either negative or positive. Element access is unordered, and the accesses may be different across dynamic execution of the same instruction.

The generalized form of the strided load intrinsic functions is:

```
__riscv_vls EEW _v_ SEW LMUL [_ TM]
```

Since there is no return type for a store instruction, the `SEW LMUL` of the source operand is encoded in the intrinsic name. The generalized form of the strided store intrinsic functions is:

```
__riscv_vss EEW _v_ SEW LMUL
```

Example:

```
struct foo {
  float16_t a,b,c,d;
  } array[100];

vfloat16mf4_t v1;
size_t vl;
```

```
v1 = __riscv_vlse16_v_f16m4 (&array[0].a, sizeof (struct foo), vl);
__riscv_vsse16_v_f16mf4 (&array[0].b, sizeof (struct foo), v1, vl);
```

4.6. Vector strided masked load/store instructions

Strided masked 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=ceil(vl/8)` (i.e. `EMUL=1`), and the destination register is always written with a tail-agnostic policy.

These instructions also provide a convenient mechanism to use packed bit vectors in memory as mask values and reduce the cost of mask spill/fill by reducing the need to change `vl`.

The generalized form of the strided masked load intrinsic functions is:

```
__riscv_vls EEW _v_ SEW LMUL _ TM
```

Since there is no return type for a store instruction, the `SEW LMUL` of the source operand is encoded in the intrinsic name. The generalized form of the strided masked store intrinsic functions is:

```
__riscv_vss EEW _v_ SEW LMUL _m
```

Example:

```
struct foo {
  float16_t a,b,c,d;
  } array[100];

vfloat16mf4_t value;
size_t vl;
vbool64_t mask;
ptrdiff_t bstride = sizeof (struct foo);
float16_t *base_a = &array[0].a, *base_b = &array[0].b;

value = __riscv_vlse16_v_f16m4_m (mask, base_a, bstride, vl);
__riscv_vsse16_v_f16mf4_m mask, base_b, bstride, value, vl);
```

4.7. Vector indexed-unordered load/store instructions

The load/store indexed instructions take a vector of indices that are used to calculate the offsets into the base address when the gather/scatter operation is done. The indices are of type `EEW` of the load/store instruction.

As with the regular load/store instructions, the accesses into memory are unordered. That is to say that any element may be accessed before another and when the instruction is executed again, a different ordering may occur.

The generalized form of the indexed-unordered load intrinsic functions is:

```
__riscv_vlux EEW _v_ SEW LMUL [_ TM]
```

Since there is no return type for a store instruction, the `SEW LMUL` of the source operand is encoded in the intrinsic name. The generalized form of the index-unordered store intrinsic functions is:

```
__riscv_vsux EEW _v_ SEW LMUL
```

Example:

```
Float16_t *base, *dest;
vuint8mf8_t bindex; // vector of unsigned offsets
size_t vl;
vfloat16mf4_t value;

value = __riscv_vluxei8_v_f16mf4 (base, bindex, vl);
__riscv_vluxei8_v_f16mf4 (dest, bindex, value, vl);
```

4.8. Vector indexed-unordered masked load/store instructions

The generalized form of the indexed-unordered masked load intrinsic functions is:

```
__riscv_vlux EEW _v_ SEW LMUL _ TM
```

Since there is no return type for a store instruction, the `SEW LMUL` of the source operand is encoded in the intrinsic name. The generalized form of the indexed-unordered masked store intrinsic functions is:

```
__riscv_vsux EEW _v_ SEW LMUL _m
```

Example:

```
Float16_t *base, *dest;
vuint8mf8_t bindex; // vector of unsigned offsets
size_t vl;
vbool64_t mask;
vfloat16mf4_t value;

value = __riscv_vluxei8_v_f16mf4_m (mask, base, bindex, vl);
__riscv_vsuxei8_v_f16mf4_m (mask, dest, bindex, value, vl);
```


4.9. Vector indexed-ordered load/store instructions

The generalized form of the indexed-ordered load intrinsic functions is:

```
__riscv_vlox EEW _v_ SEW LMUL [_ TM]
```

Since there is no return type for a store instruction, the `SEW LMUL` of the source operand is encoded in the intrinsic name. The generalized form of the indexed-ordered store intrinsic functions is:

```
__riscv_vsox EEW _v_ SEW LMUL
```

Example:

```
Float16_t *base, *dest;
vuint8mf8_t bindex; // vector of unsigned offsets
size_t vl;
vfloat16mf4_t value;

value = __riscv_vloxei8_v_f16mf4 (base, bindex, vl);
__riscv_vloxei8_v_f16mf4 (dest, bindex, value, vl);
```

4.10. Vector load/store indexed-ordered masked instructions

The generalized form of the indexed-ordered masked load intrinsic functions is:

```
__riscv_vlox EEW _v_ SEW LMUL _ TM
```

Since there is no return type for a store instruction, the `SEW LMUL` of the source operand is encoded in the intrinsic name. The generalized form of the indexed-ordered masked store intrinsic functions is:

```
__riscv_vsox EEW _v_ SEW LMUL _m
```

Example:

```
Float16_t *base, *dest;
vuint8mf8_t bindex; // vector of unsigned offsets
```

```
size_t vl;
vbool64_t mask;
vfloat16mf4_t value;

value = __riscv_vloxei8_v_f16mf4_m (mask, base, bindex, vl);
__riscv_vsoxei8_v_f16mf4_m (mask, dest, bindex, value, vl);
```


4.11. Vector unit-stride fault-only-first load instructions

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, `vl` 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 `vl` 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.

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.

The generalized form of the unit-stride fault-only-first load intrinsic functions is:

```
__riscv_vl EEW ff_v_ SEW LMUL [_ TM]
```

Example:

```
size_t strlen_vec(char *src) {
  size_t vlmax = __riscv_vsetvlmax_e8m8();
  char *copy_src = src;
  long first_set_bit = -1;
  size_t vl;
  while (first_set_bit < 0) {
    // load may go beyond page or end of memory. Only access memory
    // up to that point.
    vint8m8_t vec_src = __riscv_vle8ff_v_i8m8(copy_src, &vl, vlmax);
    // is there a zero in this vector?
    vbool1_t string_terminate =
      __riscv_vmseq_vx_i8m8_b1(vec_src, 0, vl);
    copy_src += vl;
    // which element is set to zero?
    first_set_bit = __riscv_vfirst_m_b1(string_terminate, vl);
  }
  copy_src -= vl - first_set_bit;
  return (size_t)(copy_src - src);
}
```

4.12. Vector unit-stride fault-only-first masked load instructions

The generalized form of the unit-stride fault-only-first masked load intrinsic functions is:

```
__riscv_vl EEW ff_v_ SEW LMUL _ TM
```

Example:

```
char *base;
size_t vlmax = __riscv_vsetvlmax_e8m8();
vint8m8_t vec_src = __riscv_vle8ff_v_i8m8_m(mask, base, &vl, vlmax);
```


4.13. Vector segment load/store instructions

The vector segment load/store 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 number of fields to be accessed in the instructions can be between one and eight and the value is encoded in the instruction mnemonic. The `EMUL` setting must be such that `EMUL * NFIELDS ≤ 8`. The product `EMUL * NFIELDS` represents the number of underlying vector registers that will be touched by a segmented load or store instruction. This constraint makes this total no larger than 1/4 of the architectural register file, and the same as for regular operations with `EMUL=8`.

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.

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.

4.14. Vector unit-stride segment load/store instructions

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 generalized form of the unit-stride segment load/store intrinsic functions is:

```
__riscv_vlseg NFIELDS EEW _v_ SEW LMUL x NFIELDS [_ TM]
```

Since there is no return type for a store instruction, the `SEW LMUL` of the source operand is encoded in the intrinsic name. The generalized form of the unit-stride segment store intrinsic functions is:

```
__riscv_vsseg NFIELDS EEW _v_ SEW LMUL x NFIELDS
```

Example:

```
struct RGB_values {
   unsigned short R, G, B;
  } RGB_array[100];

size_t vl = __riscv_vsetvl_e16m1 (100);
uint16_t *base = (uint16_t *) RGB_array;
struct RGB_values sbase[100];

vuint16m1x3 res = __riscv_vlseg3e16_v_u16m1x3 (base, vl);
__riscv_vsseg3e16_v_u16m1 (sbase, res, vl);
```

4.15. Vector unit-stride segment masked load/store instructions

The generalized form of the unit-stride segment masked load intrinsic functions is:

```
__riscv_vlseg NFIELDS EEW _v_ SEW LMUL _ x NFIELDS _ TM
```

Since there is no return type for a store instruction, the `SEW LMUL` of the source operand is encoded in the intrinsic name. The generalized form of the unit-stride segment masked store intrinsic functions is:

```
__riscv_vsseg NFIELDS EEW _v_ SEW LMUL _ x NFIELDS _m
```

Example:

```
struct RGB_values {
   unsigned short R, G, B;
  } RGB_array[100];

size_t vl = __riscv_vsetvl_e16m1 (100);
uint16_t *base = (uint16_t *) RGB_array;
struct RGB_values sbase[100];
bool64_t mask;

vuint16m1x3_t res = __riscv_vlseg3e16_v_u16m1x3_m (mask, base, vl);
__riscv_vsseg3e16_v_u16m1x3_m (mask, sbase, res, vl);
```


4.16. Vector strided segment load/store instructions

Strided loads are essentially a gather operation while strided stores are a scatter operation. The stride of the instruction can be either negative or positive. Element access is unordered, and the accesses may be different across dynamic execution of the same instruction.

The generalized form of the strided segment load intrinsic functions is:

```
__riscv_vlsseg NFIELDS EEW _v_ SEW LMUL x NFIELDS [_ TM]
```

Since there is no return type for a store instruction, the `SEW LMUL` of the source operand is encoded in the intrinsic name. The generalized form of the strided segment store intrinsic functions is:

```
__riscv_vssseg NFIELDS EEW _v_ SEW LMUL x NFIELDS
```

Example:

```
struct RGB_values {
   unsigned short R, G, B;
   unsigned short greyscale;
  } RGB_array[100];

size_t vl = __riscv_vsetvl_e16m1 (100);
uint16_t *base = (uint16_t *) RGB_array;
ptrdiff_t bstride = sizeof (struct RGB_values);
struct RGB_values sbase[100];

// notice that the fourth member of the RGB_values structure is
```

```
// skipped in this example. The first three members are accessed
// as normal.
vuint16m1x3_t res = __riscv_vlsseg3e16_v_u16m1x3 (base, bstride, vl);
__riscv_vssseg3e16_v_u16m1 (sbase, bstride, res, vl);
```

4.17. Vector strided segment masked load/store instructions

Strided masked 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=ceil(vl/8)` (i.e. `EMUL=1`), and the destination register is always written with a tail-agnostic policy.

These instructions also provide a convenient mechanism to use packed bit vectors in memory as mask values and reduce the cost of mask spill/fill by reducing the need to change `vl`.

The generalized form of the strided segment masked load intrinsic functions is:

```
__riscv_vlsseg NFIELDS EEW _v_ SEW LMUL x NFIELDS _ TM
```

Since there is no return type for a store instruction, the `SEW LMUL` of the source operand is encoded in the intrinsic name. The generalized form of the strided segment masked store intrinsic functions is:

```
__riscv_vssseg NFIELDS EEW _v_ SEW LMUL x NFIELDS _m
```

Example:

```
struct RGB_values {
   unsigned short R, G, B;
   unsigned short greyscale;
  } RGB_array[100];

size_t vl = __riscv_vsetvl_e16m1 (100);
uint16_t *base = (uint16_t *) RGB_array;
ptrdiff_t bstride = sizeof (struct RGB_values);
struct RGB_values sbase[100];
vbool64_t mask;

// notice that the fourth member of the RGB_values structure is
// skipped in this example. The first three members are accessed
// as normal.
vuint16m1x3_t res =
```

```
  __riscv_vlsseg3e16_v_u16m1x3_m (mask, base, bstride, vl);
__riscv_vssseg3e16_v_u16m1x3_m (mask, sbase, bstride, res, vl);
```

4.18. Vector indexed segment load/store instructions

The segmented load/store indexed instructions take a vector of indices that are used to calculate the offsets into the base address when the gather/scatter operation is done. The indices are of type `EEW` of the load/store instructions.

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

4.19. Vector indexed-unordered segment load/store instructions

The generalized form of the indexed-unordered segment load intrinsic functions is:

```
__riscv_vluxseg NFIELDS EEW _v_ SEW LMUL x NFIELDS [_TM]
```

Since there is no return type for a store instruction, the `SEW LMUL` of the source operand is encoded in the intrinsic name. The generalized form of the unit stride store intrinsic functions is:

```
__riscv_vsuxseg NFIELDS EEW _v_ SEW LMUL x NFIELDS
```

Example:

```
Float16_t *base, *dest;
vuint8mf8_t bstride; // vector of unsigned offsets
size_t vl;

vuint16m1x3_t res = __riscv_vluxseg3e16_v_u16m1x3 (base, bstride, vl);
__riscv_vsuxseg3e16_v_u16m1x3 (dest, bstride, vl);
```

4.20. Vector indexed-unordered segment masked load/store instructions

The generalized form of the indexed-unordered segment masked load intrinsic functions is:

```
__riscv_vluxseg NFIELDS EEW _v_ SEW LMUL x NFIELDS _ TM
```


Since there is no return type for a store instruction, the `SEW LMUL` of the source operand is encoded in the intrinsic name. The generalized form of the indexed-unordered segment masked store intrinsic functions is:

```
__riscv_vsuxseg NFIELDS EEW _v_ SEW LMUL x NFIELDS _m
```

Example:

```
Float16_t *base, *dest;
vuint8mf8_t bstride; // vector of unsigned offsets
vbool64_t mask;
size_t vl;

vuint16m1x3_t res =
  __riscv_vluxseg3e16_v_u16m1x3_m (mask, base, bstride, vl);
__riscv_vsuxseg3e16_v_u16m1x3_m (mask, dest, bstride, res, vl);
```

4.21. Vector indexed-ordered segment load/store instructions

The generalized form of the indexed-ordered segment load intrinsic functions is:

```
__riscv_vlox NFIELDS EEW _v_ SEW LMUL x NFIELDS [_ TM]
```

Since there is no return type for a store instruction, the `SEW LMUL` of the source operand is encoded in the intrinsic name. The generalized form of the indexed-ordered segment store intrinsic functions is:

```
__riscv_vsox NFIELDS EEW _v_ SEW LMUL x NFIELDS
```

Example:

```
Float16_t *base, *dest;
vuint8mf8_t bstride; // vector of unsigned offsets
size_t vl;

vuint16m1x3_t res = __riscv_vloxseg3e16_v_u16m1x3 (base, bstride, vl);
__riscv_vsoxseg3e16_v_u16m1x3 (dest, bstride, res, vl);
```

4.22. Vector indexed-ordered segment masked load/store instructions

The generalized form of the indexed-ordered segment masked load intrinsic functions is:

```
__riscv_vlox NFIELDS EEW _v_ SEW LMUL x NFIELDS _ TM
```

Since there is no return type for a store instruction, the `SEW LMUL` of the source operand is encoded in the intrinsic name. The generalized form of the indexed-ordered segment masked store intrinsic functions is:


```
__riscv_vsox NFIELDS EEW _v_ SEW LMUL x NFIELDS _m
```

Example:

```
Float16_t *base, *dest;
vuint8mf8_t bstride; // vector of unsigned offsets
vbool64_t mask;
size_t vl;

vuint16m1x3_t res =
  __riscv_vloxseg3e16_v_u16m1x3_m (mask, base, bstride, vl);
__riscv_vsoxseg3e16_v_u16m1x3_m (mask, dest, bstride, res, vl);
```

5. Vector integer arithmetic instructions

5.1. Vector single-width integer add/subtract instructions

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

The generalized form of the single-width add/subtract intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where **OPDS** is one of:

vv	Operation takes two vectors from vector register groups.
vx	Operation takes one vector from register group and a 5-bit immediate that is signed extended to **SEW** bits unless otherwise specified.
vx	Operation takes one vector from register group and a scalar integer variable that is sign extended to **SEW** bits unless otherwise specified.

OP is defined in the following sections.

5.1.1. Vector single-width signed integer add

Example:

```
void vector_add (int32_t *src1,
                 int32_t *src2,
                 int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vadd_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.1.2. Vector single-width signed integer masked add

Example:

```
void vector_add_m (vbool32_t mask,
                   int32_t *src1,
```

```
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res = __riscv_vadd_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_add_mu (vbool32_t mask,
                    vint32m1_t maskedoff,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vadd_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.1.3. Vector single-width unsigned integer add

Example:

```
void vector_addu (uint32_t *src1,
                  uint32_t *src2,
                  uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vadd_vv_u32m1 (opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.1.4. Vector single-width unsigned integer masked add

Example:

```
void vector_addu_m (vbool32_t mask,
                    uint32_t *src1,
                    uint32_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res = __riscv_vadd_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_addu_mu (vbool32_t mask,
                     vuint32m1_t maskedoff,
                     uint32_t *src1,
                     uint32_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vadd_vv_u32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.1.5. Vector single-width signed integer subtract

Example:

```
void vector_sub (int32_t *src1,
                 int32_t *src2,
                 int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vsub_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.1.6. Vector single-width signed integer masked subtract

Example:

```
void vector_sub_m (vbool32_t mask,
                   int32_t *src1,
                   int32_t *src2,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res = __riscv_vsub_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_sub_mu (vbool32_t mask,
                    vint32m1_t maskedoff,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vsub_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.1.7. Vector single-width signed integer reverse subtract

There is no vector-vector form of this instruction.

Example:

```
void vector_rsub (int32_t *src1,
                  int32_t src2,
                  int32_t *dest)
{
  // there is no vv form of this instruction. Only vx.
  size_t vl = __riscv_vsetvlmax_e32m1();
```

```
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t res = __riscv_vrsub_vx_i32m1 (opd1, src2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.1.8. Vector single-width signed integer reverse masked subtract

There is no vector-vector form of this instruction.

Example:

```
void vector_rsub_m (vbool32_t mask,
                    int32_t *src1,
                    int32_t src2,
                    int32_t *dest)
{
   // there is no vv form of this instruction. Only vx.
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t res = __riscv_vrsub_vx_i32m1_m (mask, opd1, src2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vector_rsub_mu (vbool32_t mask,
                     vint32m1_t maskedoff,
                     int32_t *src1,
                     int32_t src2,
                     int32_t *dest)
{
   // there is no vv form of this instruction. Only vx.
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t res =
    __riscv_vrsub_vx_i32m1_mu (mask, maskedoff, opd1, src2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.1.9. Vector single-width unsigned integer subtract

Example:

```
void vector_subu (uint32_t *src1,
                  uint32_t *src2,
```

```
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vsub_vv_u32m1 (opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```


5.1.10. Vector single-width unsigned integer masked subtract

Example:

```
void vector_subu_m (vbool32_t mask,
                    uint32_t *src1,
                    uint32_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res = __riscv_vsub_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vector_subu_mu (vbool32_t mask,
                     vuint32m1_t maskedoff,
                     uint32_t *src1,
                     uint32_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vsub_vv_u32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.1.11. Vector unsigned integer reverse subtract

Example:

```
void vector_rsubu (uint32_t *src1,
                   uint32_t src2,
                   uint32_t *dest)
{
  // there is no vv form of this instruction. Only vx.
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t res = __riscv_vrsub_vx_u32m1 (opd1, src2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```


5.1.12. Vector unsigned integer reverse masked subtract

Example:

```
void vector_rsubu_m (vbool32_t mask,
                     uint32_t *src1,
                     uint32_t src2,
                     uint32_t *dest)
{
  // there is no vv form of this instruction. Only vx.
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t res = __riscv_vrsub_vx_u32m1_m (mask, opd1, src2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vector_rsubu_mu (vbool32_t mask,
                      vuint32m1_t maskedoff,
                      uint32_t *src1,
                      uint32_t src2,
                      uint32_t *dest)
{
  // there is no vv form of this instruction. Only vx.
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t res =
    __riscv_vrsub_vx_u32m1_mu (mask, maskedoff, opd1, src2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.1.13. Vector single-width signed integer negate

Example:

```
void vector_neg (int32_t *src1,
                 int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t res = __riscv_vneg_v_i32m1 (opd1, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


5.1.14. Vector single-width signed integer masked negate

Example:

```
void vector_neg_m (vbool32_t mask,
                   int32_t *src1,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t res = __riscv_vneg_v_i32m1_m (mask, opd1, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vector_neg_m (vbool32_t mask,
                   vint32m1_t maskedoff,
                   int32_t *src1,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t res = __riscv_vneg_v_i32m1_mu (mask, maskedoff, opd1,
vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.2. Vector widening integer add/subtract instructions

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.

The generalized form of the widening add/subtract intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where OPDS is one of:

vv	Operation takes two vectors from vector register groups of SEW size. The return result is SEW*2.
wv	Operation takes two vectors from vector register groups where one argument is of size SEW*2 while the other is of size SEW. The return result is SEW*2.
vx	Operation takes one vector from register group of SEW size and a 5-bit immediate that is signed extended to SEW bits unless otherwise specified. The return result is SEW*2.
vx	Operation takes one vector from register group of SEW size and a scalar integer variable that is sign extended to SEW bits unless otherwise specified. The return result is SEW*2.

OP is defined in the following sections.

5.2.1. Vector widening signed integer add – 2*SEW = SEW + SEW

Example:

```
void vector_wadd (int16_t *src1,
                  int16_t *src2,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint32m2_t res = __riscv_vwadd_vv_i32m2 (opd1, opd2, vl);
 __riscv_vse32_v_i32m2 (dest, res, vl);
}
```

5.2.2. Vector widening signed integer masked add – 2*SEW = SEW + SEW

Example:

```
void vector_wadd_m (vbool16_t mask,
                    int16_t *src1,
                    int16_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1_m (mask, src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1_m (mask, src2, vl);
  vint32m2_t res = __riscv_vwadd_vv_i32m2_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m2_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_wadd_mu (vbool16_t mask,
                     vint32m2_t maskedoff,
                     int16_t *src1,
                     int16_t *src2,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint32m2_t res =
    __riscv_vwadd_vv_i32m2_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m2 (dest, res, vl);
}
```

5.2.3. Vector widening unsigned integer add – 2*SEW = SEW + SEW

Example:

```
void vector_waddu (uint16_t *src1,
                   uint16_t *src2,
                   uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vuint32m2_t res = __riscv_vwaddu_vv_u32m2 (opd1, opd2, vl);
  __riscv_vse32_v_u32m2 (dest, res, vl);
}
```

5.2.4. Vector widening unsigned integer masked add – 2*SEW = SEW + SEW

Example:

```
void vector_waddu_m (vbool16_t mask,
                     uint16_t *src1,
                     uint16_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1_m (mask, src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1_m (mask, src2, vl);
  vuint32m2_t res = __riscv_vwaddu_vv_u32m2_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m2_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_waddu_mu (vbool16_t mask,
                      vuint32m2_t maskedoff,
                      uint16_t *src1,
                      uint16_t *src2,
                      uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vuint32m2_t res =
    __riscv_vwaddu_vv_u32m2_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_u32m2 (dest, res, vl);
}
```

5.2.5. Vector widening signed integer add – 2*SEW = 2*SEW + SEW

Example:

```
void vector_wadd (int16_t *src1,
                  int16_t *src2,
                  int16_t *src3,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1(src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1(src2, vl);
  vint16m1_t opd3 = __riscv_vle16_v_i16m1(src3, vl);
  vint32m2_t res1 = __riscv_vwadd_vv_i32m2 (opd1, opd2, vl);
  vint32m2_t res2 = __riscv_vwadd_wv_i32m2 (res1, opd3, vl);
 __riscv_vse32_v_i32m2 (dest, res2, vl);
}
```


5.2.6. Vector widening signed integer masked add – 2*SEW = 2*SEW + SEW

Example:

```
void vector_wadd_m (vbool6_t mask,
                    int16_t *src1,
                    int16_t *src2,
                    int16_t *src3,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1_m (mask, src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1_m (mask, src2, vl);
  vint16m1_t opd3 = __riscv_vle16_v_i16m1_m (mask, src3, vl);
  vint32m2_t res1 = __riscv_vwadd_vv_i32m2_m (mask, opd1, opd2, vl);
  vint32m2_t res2 = __riscv_vwadd_wv_i32m2_m (mask, res1, opd3, vl);
 __riscv_vse32_v_i32m2_m (mask, dest, res2, vl);
}
```

Example with mask undisturbed policy:

```
void vector_wadd_mu (vbool6_t mask,
                     vint32m2_t maskedoff,
                     int16_t *src1,
                     int16_t *src2,
                     int16_t *src3,
                     int32_t *dest)
{
```

```
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint16m1_t opd3 = __riscv_vle16_v_i16m1 (src3, vl);
  vint32m2_t res1 =
   __riscv_vwadd_vv_i32m2_mu (mask, maskedoff, opd1, opd2, vl);
  vint32m2_t res2 =
   __riscv_vwadd_wv_i32m2_mu (mask, maskedoff, res1, opd3, vl);
 __riscv_vse32_v_i32m2 (dest, res2, vl);
}
```


5.2.7. Vector widening unsigned integer add – 2*SEW = 2*SEW + SEW

Example:

```
void vector_wadd (uint16_t *src1,
                  uint16_t *src2,
                  uint16_t *src3,
                  uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vuint16m1_t opd3 = __riscv_vle16_v_u16m1 (src3, vl);
  vuint32m2_t res1 = __riscv_vwaddu_vv_u32m2 (opd1, opd2, vl);
  vuint32m2_t res2 = __riscv_vwaddu_wv_u32m2 (res1, opd3, vl);
 __riscv_vse32_v_u32m2 (dest, res2, vl);
}
```

5.2.8. Vector widening unsigned integer masked add – `2*SEW = 2*SEW + SEW`

Example:

```
void vector_wadd_m (vbool6_t mask,
                    uint16_t *src1,
                    uint16_t *src2,
                    uint16_t *src3,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1_m (mask, src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1_m (mask, src2, vl);
  vuint16m1_t opd3 = __riscv_vle16_v_u16m1_m (mask, src3, vl);
  vuint32m2_t res1 = __riscv_vwaddu_vv_u32m2_m (mask, opd1, opd2, vl);
```

```
  vuint32m2_t res2 = __riscv_vwaddu_wv_u32m2_m (mask, res1, opd3, vl);
 __riscv_vse32_v_u32m2_m (mask, dest, res2, vl);
}
```

Example with mask undisturbed policy:

```
void vector_wadd_mu (vbool6_t mask,
                     vuint32m2_t maskedoff,
                     uint16_t *src1,
                     uint16_t *src2,
                     uint16_t *src3,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vuint16m1_t opd3 = __riscv_vle16_v_u16m1 (src3, vl);
  vuint32m2_t res1 =
    __riscv_vwaddu_vv_u32m2_mu (mask, maskedoff, opd1, opd2, vl);
  vuint32m2_t res2 =
    __riscv_vwaddu_wv_u32m2_mu (mask, maskedoff, res1, opd3, vl);
 __riscv_vse32_v_u32m2 (dest, res2, vl);
}
```


5.2.9. Vector widening signed integer subtract – 2*SEW = SEW – SEW

Example:

```
void vector_wsub (int16_t *src1,
                  int16_t *src2,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint32m2_t res = __riscv_vwsub_vv_i32m2 (opd1, opd2, vl);
 __riscv_vse32_v_i32m2 (dest, res, vl);
}
```

5.2.10. Vector widening signed integer masked subtract – 2*SEW = SEW – SEW

Example:

```
void vector_wsub_m (vbool16_t mask,
                    int16_t *src1,
```

```
                    int16_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1_m (mask, src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1_m (mask, src2, vl);
  vint32m2_t res = __riscv_vwsub_vv_i32m2_m (mask, opd1, opd2, vl);
 __riscv_vse32_v_i32m2_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_wsub_mu (vbool16_t mask,
                    vint32m2_t maskedoff,
                    int16_t *src1,
                    int16_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint32m2_t res =
    __riscv_vwsub_vv_i32m2_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m2 (dest, res, vl);
}
```

5.2.11. Vector widening unsigned integer subtract – 2*SEW = SEW – SEW

Example:

```
void vector_wsubu (uint16_t *src1,
                   uint16_t *src2,
                   uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vuint32m2_t res = __riscv_vwsubu_vv_u32m2 (opd1, opd2, vl);
 __riscv_vse32_v_u32m2 (dest, res, vl);
}
```

5.2.12. Vector widening unsigned integer masked subtract – 2*SEW = SEW – SEW

Example:

```
void vector_wsubu_m (vbool16_t mask,
                     uint16_t *src1,
                     uint16_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1_m (mask, src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1_m (mask, src2, vl);
  vuint32m2_t res = __riscv_vwsubu_vv_u32m2_m (mask, opd1, opd2, vl);
 __riscv_vse32_v_u32m2_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_wsubu_mu (vbool16_t mask,
                      vuint32m2_t maskedoff,
                      uint16_t *src1,
                      uint16_t *src2,
                      uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vuint32m2_t res =
    __riscv_vwsubu_vv_u32m2_mu (mask, maskedoff, opd1, opd2, vl);
 __riscv_vse32_v_u32m2 (dest, res, vl);
}
```

5.2.13. Vector widening signed integer subtract – 2*SEW = 2*SEW – SEW

Example:

```
void vector_wsub (int16_t *src1,
                  int16_t *src2,
                  int16_t *src3,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint16m1_t opd3 = __riscv_vle16_v_i16m1 (src3, vl);
  vint32m2_t res1 = __riscv_vsubu_vv_i32m2 (opd1, opd2, vl);
  vint32m2_t res2 = __riscv_vwaddu_wv_i32m2 (res1, opd3, vl);
 __riscv_vse32_v_i32m2 (dest, res2, vl);
```

```
}
```

5.2.14. Vector widening signed integer masked subtract – 2*SEW = 2*SEW – SEW

Example:

```
void vector_wsub_m (vbool6_t mask,
                    int16_t *src1,
                    int16_t *src2,
                    int16_t *src3,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1_m (mask, src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1_m (mask, src2, vl);
  vint16m1_t opd3 = __riscv_vle16_v_i16m1_m (mask, src3, vl);
  vint32m2_t res1 = __riscv_vwsub_vv_i32m2_m (mask, opd1, opd2, vl);
  vint32m2_t res2 = __riscv_vwsub_wv_i32m2_m (mask, res1, opd3, vl);
 __riscv_vse32_v_i32m2_m (mask, dest, res2, vl);
}
```


Example with mask undisturbed policy:

```
void vector_wsub_mu (vbool6_t mask,
                     vint32m2_t maskedoff,
                     int16_t *src1,
                     int16_t *src2,
                     int16_t *src3,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint16m1_t opd3 = __riscv_vle16_v_i16m1 (src3, vl);
  vint32m2_t res1 =
    __riscv_vwsub_vv_i32m2_mu (mask, maskedoff, opd1, opd2, vl);
  vint32m2_t res2 =
    __riscv_vwsub_wv_i32m2_mu (mask, maskedoff, res1, opd3, vl);
 __riscv_vse32_v_i32m2 (dest, res2, vl);
}
```

5.2.15. Vector widening unsigned integer subtract – 2*SEW = 2*SEW – SEW

Example:

```
void vector_wsubu (uint16_t *src1,
                   uint16_t *src2,
                   uint16_t *src3,
                   uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vuint16m1_t opd3 = __riscv_vle16_v_u16m1 (src3, vl);
  vuint32m2_t res1 = __riscv_vwsub_vv_i32m2 (opd1, opd2, vl);
  vuint32m2_t res2 = __riscv_vwsub_wv_i32m2 (res1, opd3, vl);
 __riscv_vse32_v_u32m2 (dest, res2, vl);
}
```


5.2.16. Vector widening unsigned integer masked subtract – 2*SEW = 2*SEW – SEW

Example:

```
void vector_wsubu_m (vbool6_t mask,
                     uint16_t *src1,
                     uint16_t *src2,
                     uint16_t *src3,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1_m (mask, src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1_m (mask, src2, vl);
  vuint16m1_t opd3 = __riscv_vle16_v_u16m1_m (mask, src3, vl);
  vuint32m2_t res1 = __riscv_vwsubu_vv_u32m2_m (mask, opd1, opd2, vl);
  vuint32m2_t res2 = __riscv_vwsubu_wv_u32m2_m (mask, res1, opd3, vl);
 __riscv_vse32_v_u32m2_m (mask, dest, res2, vl);
}
```

Example with mask undisturbed policy:

```
void vector_wsubu_mu (vbool6_t mask,
                      vuint32m2_t maskedoff,
                      uint16_t *src1,
                      uint16_t *src2,
                      uint16_t *src3,
                      uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
```

```
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vuint16m1_t opd3 = __riscv_vle16_v_u16m1 (src3, vl);
  vuint32m2_t res1 =
    __riscv_vwsubu_vv_u32m2_mu (mask, maskedoff, opd1, opd2, vl);
  vuint32m2_t res2 =
    __riscv_vwsubu_wv_u32m2_mu (mask, maskedoff, res1, opd3, vl);
 __riscv_vse32_v_u32m2 (dest, res2, vl);
}
```

5.3. Vector integer extension instructions

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`.

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), explicit extension instructions that can be used are added instead if an appropriate widening arithmetic instruction is not available.

The generalized form of the integer extension intrinsic functions is:

```
__riscv_v OP _v WIDTH _ SEW LMUL [_ TM]
```

Where `WIDTH` is one of:

f2	`EEW = SEW / 2` and `EMUL = (EEW / SEW) * LMUL`
f4	`EEW = SEW / 4` and `EMUL = (EEW / SEW) * LMUL`
F8	`EEW = SEW / 8` and `EMUL = (EEW / SEW) * LMUL`

`OP` is defined in the following sections.

5.3.1. Vector integer zero extension instruction

Example:

```
void zext (uint16_t *src,
           uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src, vl);
  vuint32m2_t res = __riscv_vzext_vf2_u32m2 (opd1, vl);
  __riscv_vse32_v_u32m2 (dest, res, vl);
}
```

5.3.2. Vector integer masked zero extension instruction

Example:

```
void zext_m (vbool16_t mask,
             uint16_t *src,
             uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1_m (mask, src, vl);
  vuint32m2_t res = __riscv_vsext_vf2_u32m2_m (mask, opd1, vl);
  __riscv_vse32_v_u32m2_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void zext_mu (vbool16_t mask,
              vuint32m2_t maskedoff,
              uint16_t *src,
              uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src, vl);
  vuint32m2_t res =
    __riscv_vsext_vf2_u32m2_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_u32m2 (dest, res, vl);
}
```

5.3.3. Vector integer sign extension instruction

Example:

```
void sext (int16_t *src,
           int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src, vl);
  vint32m2_t res = __riscv_vsext_vf2_i32m2 (opd1, vl);
  __riscv_vse32_v_i32m2 (dest, res, vl);
}
```

5.3.4. Vector integer masked sign extension instruction

Example:

```
void sext_m (vbool16_t mask,
```

```
                int16_t *src,
                int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1_m (mask, src, vl);
  vint32m2_t res = __riscv_vsext_vf2_i32m2_m (mask, opd1, vl);
  __riscv_vse32_v_i32m2_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void sext_mu (vbool16_t mask,
              vint32m2_t maskedoff,
              int16_t *src,
              int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src, vl);
  vint32m2_t res =
    __riscv_vsext_vf2_i32m2_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_i32m2 (dest, res, vl);
}
```

5.4. Vector integer add-with-carry and 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 is to provide the result (SEW width), and the second to generate the carry output (single bit encoded as a mask boolean).

These instructions are encoded as masked instructions (vm=0), but they operate on and write back all body elements. vmadc and vmsbc add or subtract the source operands, optionally add the carry-in or subtract the borrow-in if masked and write the result back to mask register result. 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.

Although there are signed and unsigned versions of the carry out functions, there is only one signed version of the carry in functions.

The generalized form of the add-carry/subtract-binary-borrow intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL _ [BTYPE] [_ TM]
```

Where OPDS is one of:

vvm	Operation takes two vectors from vector register groups and one mask register.
vim	Operation takes one vector from vector register groups, a 5-bit immediate that is signed extended to SEW bits unless otherwise specified, and a mask vector.
vxm	Operation takes one vector from vector register groups, a scalar integer variable that is sign extended to SEW bits unless otherwise specified, and a mask register.
vv	Operation takes two vectors from vector register groups and no carry-in mask register.
vi	Operation takes one vector from vector register groups, a 5-bit immediate that is signed extended to SEW bits unless otherwise specified, and no carry-in mask vector
vx	Operation takes one vector from vector register groups, a scalar integer variable that is sign extended to SEW bits unless otherwise specified, and no carry-in mask register.

OP is defined in the following sections.

5.4.1. Vector signed add with carry-out instruction

Example:


```
vbool16_t madc (int16_t *src1,
                int16_t *src2)
{
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vbool16_t carryout = __riscv_vmadc_vv_i16m1_b16 (opd1, opd2, vl);

  return carryout;
}
```

5.4.2. Vector unsigned add with carry-out instruction

Example:

```
vbool16_t adcu (uint16_t *src1,
                uint16_t *src2)
{
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vbool16_t carryout = __riscv_vmadc_vv_u16m1_b16 (opd1, opd2, vl);

  return carryout;
}
```

5.4.3. Vector signed add with carry-in instruction

Example:

```
void madc (int16_t *src1,
           int16_t *src2,
           int16_t *dest)
{
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vbool16_t carryin = __riscv_vmadc_vv_i16m1_b16 (opd1, opd2, vl);
  vbool16_t res = __riscv_vadc_vvm_i16m1 (opd1, opd2, carryin, vl);
  __riscv_vse32_v_i16m1 (dest, res, vl);
```

```
}
```

5.4.4. Vector signed subtract binary borrow with carry-out instruction

Example:

```
vbool16_t msbc (int16_t *src1,
                int16_t *src2)
{
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vbool16_t carryout = __riscv_vmsbc_vv_i16m1_b16 (opd1, opd2, vl);

  return carryout;
}
```


5.4.5. Vector unsigned subtract binary borrow with carry-out instruction

Example:

```
vbool16_t sbcu (uint16_t *src1,
                uint16_t *src2)
{
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vbool16_t carryout = __riscv_vmsbc_vv_u16m1_b16 (opd1, opd2, vl);

  return carryout;
}
```

5.4.6. Vector signed subtract binary borrow with carry-in instruction

Example:

```
void sbc (int16_t *src1,
          int16_t *src2,
          int16_t *dest)
{
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vbool16_t borrowin = __riscv_vmsbc_vv_i16m1_b16 (opd1, opd2, vl);
  vbool16_t res = __riscv_vsbc_vvm_i16m1_b16 (opd1, opd2, borrowin,
vl);
  __riscv_vse16_v_i16m1 (dest, res, vl);
```

```
}
```

5.4.7. Vector unsigned subtract binary borrow with carry-in instruction

Example:

```
void sbcu (int16_t *src1,
           uint16_t *src2,
           uint16_t *dest)
{
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vbool16_t borrowin = __riscv_vmsbc_vv_u16m1_b16 (opd1, opd2, vl);
  vbool16_t res = __riscv_vsbc_vvm_u16m1_b16 (opd1, opd2, borrowin,
vl);
  __riscv_vse16_v_u16m1 (dest, res, vl);
}
```

5.5. Vector bitwise logical instructions

The generalized form of the bitwise logical intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where **OPDS** is one of:

vv	Operation takes two vectors from vector register groups of **SEW** size returning **SEW***2 result.
vx	Operation takes one vector from vector register groups and a 5-bit immediate that is signed extended to **SEW** bits unless otherwise specified.
vx	Operation takes one vector from register vector group and a scalar integer variable that is sign extended to **SEW** bits unless otherwise specified.

OP is defined in the following sections.

5.5.1. Vector signed logical and instruction

Example:

```
void vector_and (int32_t *src1,
                 int32_t *src2,
                 int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vand_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.5.2. Vector signed logical masked and instruction

Example:

```
void vector_and_m (vbool32_t mask,
                   int32_t *src1,
                   int32_t *src2,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
```

```
  vint32m1_t res = __riscv_vand_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vector_and_mu (vbool32_t mask,
                    vint32m1_t maskedoff,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vand_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


5.5.3. Vector unsigned logical and instruction

Example:

```
void vector_andu (uint32_t *src1,
                  uint32_t *src2,
                  uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vand_vv_u32m1 (opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.5.4. Vector unsigned logical masked and instruction

Example:

```
void vector_andu_m (vbool32_t mask,
                    uint32_t *src1,
                    uint32_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
```

```
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res = __riscv_vand_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:


```
void vector_andu_mu (vbool32_t mask,
                     vuint32m1_t maskedoff,
                     uint32_t *src1,
                     uint32_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vand_vv_u32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.5.5. Vector signed logical or instruction

Example:

```
void vector_or (int32_t *src1,
                int32_t *src2,
                int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vor_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.5.6. Vector signed logical masked or instruction

Example:

```
void vector_or_m (vbool32_t mask,
                  int32_t *src1,
                  int32_t *src2,
                  int32_t *dest)
```

```
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res = __riscv_vor_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_or_mu (vbool32_t mask,
                   vint32m1_t maskedoff,
                   int32_t *src1,
                   int32_t *src2,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vor_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.5.7. Vector unsigned logical or instruction

Example:

```
void vector_oru (uint32_t *src1,
                 uint32_t *src2,
                 uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vor_vv_u32m1 (opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.5.8. Vector unsigned logical masked or instruction

Example:

```
void vector_oru_m (vbool32_t mask,
                   uint32_t *src1,
```

```
                    uint32_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (mask, src2, vl);
  vuint32m1_t res = __riscv_vor_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_oru_mu (vbool32_t mask,
                    vuint32m1_t maskedoff,
                    uint32_t *src1,
                    uint32_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vor_vv_u32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.5.9. Vector signed logical exclusive or instruction

Example:

```
void vector_xor (int32_t *src1,
                 int32_t *src2,
                 int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vxor_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.5.10. Vector signed logical masked exclusive or instruction

Example:

```
void vector_xor_m (vbool32_t mask,
                   int32_t *src1,
                   int32_t *src2,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res = __riscv_vxor_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_xor_mu (vbool32_t mask,
                    vint32m1_t maskedoff,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vxor_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.5.11. Vector unsigned logical exclusive or instruction

Example:

```
void vector_xoru (uint32_t *src1,
                  uint32_t *src2,
                  uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vxor_vv_u32m1 (opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.5.12. Vector unsigned logical masked exclusive or instruction

Example:

```
void vector_xoru_m (vbool32_t mask,
                    uint32_t *src1,
                    uint32_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_iu32m1_m (mask, src2, vl);
  vuint32m1_t res = __riscv_vxor_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_xoru_mu (vbool32_t mask,
                     vuint32m1_t maskedoff,
                     uint32_t *src1,
                     uint32_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_iu32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vxor_vv_u32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.5.13. Vector logical signed not instruction

Example:

```
void vector_not (int32_t *src1,
                 int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t res = __riscv_vnot_v_i32m1 (opd1, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.5.14. Vector logical signed masked not instruction

Example:

```
void vector_not_m (vbool32_t mask,
                   int32_t *src1,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t res = __riscv_vnot_v_i32m1_m (mask, opd1, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_not_mu (vbool32_t mask,
                    vint32m1_t maskedoff,
                    int32_t *src1,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t res = __riscv_vnot_v_i32m1_mu (mask, maskedoff, opd1,
vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.5.15. Vector logical unsigned not instruction

Example:

```
void vector_notu (uint32_t *src1,
                  uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t res = __riscv_vnot_v_u32m1 (opd1, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.5.16. Vector logical unsigned masked not instruction

Example:

```
void vector_notu_m (vbool32_t mask,
```

```
                    uint32_t *src1,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t res = __riscv_vnot_v_u32m1_m (mask, opd1, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_notu_mu (vbool32_t mask,
                    vuint32m1_t maskedoff,
                    uint32_t *src1,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t res = __riscv_vnot_v_u32m1_mu (mask, maskedoff, opd1,
vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.6. 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 and the shift amount value can come from a vector register group, a scalar integer register, or a zero-extended 5-bit immediate. Only the low `lg2(SEW)` bits of the shift-amount value are used to control the shift amount.

The generalized form of the single-width shift intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where **OPDS** is one of:

vv	Operation takes two vectors from vector register groups of **SEW** size returning **SEW***2 result.
vx	Operation takes one vector from vector register groups and a 5-bit immediate that is signed extended to **SEW** bits unless otherwise specified.
vx	Operation takes one vector from vector register groups and a scalar integer variable that is sign extended to **SEW** bits unless otherwise specified.

OP is defined in the following sections.

5.6.1. Vector signed shift left logical instruction

Example:

```
void vector_sll (int32_t *src1,
                 int32_t *src2,
                 int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vsll_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.6.2. Vector signed masked shift left logical instruction

Example:

```
void vector_sll_m (vbool32_t mask,
                   int32_t *src1,
                   int32_t *src2,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res = __riscv_vsll_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_sll_mu (vbool32_t mask,
                    vint32m1_t maskedoff,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vsll_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.6.3. Vector unsigned shift left logical instruction

Example:

```
void vector_sllu (uint32_t *src1,
                  uint32_t *src2,
                  uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vsll_vv_u32m1 (opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.6.4. Vector unsigned masked shift left logical instruction

Example:

```
void vector_sllu_m (vbool32_t mask,
                    uint32_t *src1,
                    uint32_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res = __riscv_vsll_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_sllu_mu (vbool32_t mask,
                     vuint32m1_t maskedoff,
                     uint32_t *src1,
                     uint32_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vsll_vv_u32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.6.5. Vector unsigned shift right logical instruction

Example:

```
void vector_srlu (uint32_t *src1,
                  uint32_t *src2,
                  uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vsrl_vv_u32m1 (opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
```

```
}
```

5.6.6. Vector unsigned masked shift right logical instruction

Example:

```
void vector_srlu_m (vbool32_t mask,
                    uint32_t *src1,
                    uint32_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res = __riscv_vsrl_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_srlu_mu (vbool32_t mask,
                     vuint32m1_t maskedoff,
                     uint32_t *src1,
                     uint32_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vsrl_vv_u32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.6.7. Vector signed shift right arithmetic instruction

Example:

```
void vector_sra (int32_t *src1,
                 uint32_t *src2,
                 int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
```

```
  vint32m1_t res = __riscv_vsra_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.6.8. Vector signed masked shift right arithmetic instruction

Example:

```
void vector_sra_m (vbool32_t mask,
                   int32_t *src1,
                   uint32_t *src2,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vint32m1_t res = __riscv_vsra_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vector_sra_mu (vbool32_t mask,
                    vint32m1_t maskedoff,
                    int32_t *src1,
                    uint32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vsra_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.7. 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, a scalar register, or a zero-extended 5-bit immediate. The low `lg2(2*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).

The generalized form of the narrowing shift right intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where `OPDS` is one of:

wv	Operation takes one vector from vector register groups of size `SEW*2` and a shift amount. The result is a vector of size `SEW`.
wx	Operation takes one vector register of size `SEW*2` and an immediate 5-bit unsigned shift amount that is zero extended to `SEW`. The result is a vector of size `SEW`.
wx	Operation takes one vector register of size `SEW*2` and scalar register containing an unsigned shift amount that is zero extended to `SEW`. The result is a vector of size `SEW`.

`OP` is defined in the following sections.

5.7.1. Vector narrowing unsigned shift right instruction

Example:

```
void vector_nsrl (uint16_t *src1,
                  size_t shift,
                  uint8_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m2();
  vuint16m2_t opd1 = __riscv_vle16_v_u16m2 (src1, vl);
  vuint8m1_t res = __riscv_vnsrl_wx_u8m1 (opd1, shift, vl);
  __riscv_vse8_v_u8m1 (dest, res, vl);
}
```

5.7.2. Vector narrowing unsigned masked shift right instruction

Example:

```
void vector_nsrl_m (vbool8_t mask,
                    uint16_t *src1,
                    size_t shift,
                    uint8_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m2();
  vuint16m2_t opd1 = __riscv_vle16_v_u16m2_m (mask, src1, vl);
  vuint8m1_t res = __riscv_vnsrl_wx_u8m1_m (mask, opd1, shift, vl);
  __riscv_vse8_v_u8m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_nsrl_mu (vbool8_t mask,
                     vuint8m1_t maskedoff,
                     uint16_t *src1,
                     size_t shift,
                     uint8_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m2();
  vuint16m2_t opd1 = __riscv_vle16_v_u16m2 (src1, vl);
  vuint8m1_t res =
    __riscv_vnsrl_wx_u8m1_mu (mask, maskedoff, opd1, shift, vl);
  __riscv_vse8_v_u8m1 (dest, res, vl);
}
```

5.7.3. Vector narrowing signed shift right arithmetic instruction

Example:

```
void vector_nsra (int16_t *src1,
                  size_t shift,
                  int8_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m2();
  vint16m2_t opd1 = __riscv_vle16_v_i16m2 (src1, vl);
  vint8m1_t res = __riscv_vnsra_wx_i8m1 (opd1, shift, vl);
  __riscv_vse8_v_i8m1 (dest, res, vl);
}
```

5.7.4. Vector narrowing signed masked shift right arithmetic instruction

Example:

```
void vector_nsra_m (vbool8_t mask,
                    int16_t *src1,
                    size_t shift,
                    int8_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m2();
  vint16m2_t opd1 = __riscv_vle16_v_i16m2_m (mask, src1, vl);
  vint8m1_t res = __riscv_vnsra_wx_i8m1_m (mask, opd1, shift, vl);
  __riscv_vse8_v_i8m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_nsra_mu (vbool8_t mask,
                     vint8m1_t maskedoff,
                     int16_t *src1,
                     size_t shift,
                     int8_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m2();
  vint16m2_t opd1 = __riscv_vle16_v_i16m2 (src1, vl);
  vint8m1_t res =
    __riscv_vnsra_wx_i8m1_mu (mask, maskedoff, opd1, shift, vl);
  __riscv_vse8_v_i8m1 (dest, res, vl);
}
```

5.8. 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.

The generalized form of the integer compare intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL _ [BTYPE] [_ TM]
```

Where **OPDS** is one of:

vv	Operation takes two vectors from vector register groups.
vx	Operation takes one vector from vector register groups and a 5-bit immediate that is signed extended to **SEW** bits unless otherwise specified.
vx	Operation takes one vector from vector register groups and a scalar integer variable that is sign extended to **SEW** bits unless otherwise specified.

And **BTYPE** is the `boolean` return type and is one of:

b8	`vbool8_t` return type
b16	`vbool16_t` return type
b32	`vbool32_t` return type
b64	`vbool64_t` return type
b1	`vbool1_t` return type
b2	`vbool2_t` return type
b4	`vbool4_t` return type
b8	`vbool8_t` return type

OP is defined in the following sections.

5.8.1. Vector set if equal

Example:

```
vbool32_t vector_seq (int32_t *src1,
                      int32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
```

```
  vbool32_t res = __riscv_vmseq_vv_i32m1_b32 (opd1, opd2, vl);
  return res;
}
```

5.8.2. Vector set if masked equal

Example:


```
vbool32_t vector_seq_m (vbool32_t mask,
                        int32_t *src1,
                        int32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vbool32_t res = __riscv_vmseq_vv_i32m1_b32_m (mask, opd1, opd2, vl);
  return res;
}
```

Example with mask undisturbed policy:

```
vbool32_t vector_seq_mu (vbool32_t mask,
                         vbool32_t maskedoff,
                         int32_t *src1,
                         int32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vbool32_t res =
    __riscv_vmseq_vv_i32m1_b32_mu (mask, maskedoff, opd1, opd2, vl);
  return res;
}
```

5.8.3. Vector set if not equal

Example:

```
vbool32_t vector_sne (int32_t *src1,
                      int32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
```

```
  vbool32_t res = __riscv_vmsne_vv_i32m1_b32 (opd1, opd2, vl);
  return res;
}
```

5.8.4. Vector set if masked not equal

Example:

```
vbool32_t vector_sne_m (vbool32_t mask,
                        int32_t *src1,
                        int32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vbool32_t res = __riscv_vmsne_vv_i32m1_b32_m (mask, opd1, opd2, vl);
  return res;
}
```

Example with mask undisturbed policy:

```
vbool32_t vector_sne_mu (vbool32_t mask,
                         vbool32_t maskedoff,
                         int32_t *src1,
                         int32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vbool32_t res =
    __riscv_vmsne_vv_i32m1_b32_mu (mask, maskedoff, opd1, opd2, vl);
  return res;
}
```

5.8.5. Vector set if less than, unsigned

Example:

```
vbool32_t vector_sltu (uint32_t *src1,
                       uint32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vbool32_t res = __riscv_vmsltu_vv_u32m1_b32 (opd1, opd2, vl);
  return res;
```

```
}
```

5.8.6. Vector set if masked less than, unsigned

Example:

```
vbool32_t vector_sltu_m (vbool32_t mask,
                         uint32_t *src1,
                         uint32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vubool32_t res = __riscv_vmsltu_vv_u32m1_b32_m (mask, opd1, opd2,
vl);
  return res;
}
```


Example with mask undisturbed policy:

```
vbool32_t vector_sltu_mu (vbool32_t mask,
                          vubool32_t maskedoff,
                          uint32_t *src1,
                          uint32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vubool32_t res =
    __riscv_vmsltu_vv_u32m1_b32_mu (mask, maskedoff, opd1, opd2, vl);
  return res;
}
```

5.8.7. Vector set if less than, signed

Example:

```
vbool32_t vector_slt (int32_t *src1,
                      int32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vbool32_t res = __riscv_vmslt_vv_i32m1_b32 (opd1, opd2, vl);
  return res;
}
```

5.8.8. Vector set if masked less than, signed

Example:

```
vbool32_t vector_slt_m (vbool32_t mask,
                        int32_t *src1,
                        int32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vbool32_t res = __riscv_vmslt_vv_i32m1_b32_m (mask, opd1, opd2, vl);
  return res;
}
```


Example with mask undisturbed policy:

```
vbool32_t vector_slt_mu (vbool32_t mask,
                         vbool32_t maskedoff,
                         int32_t *src1,
                         int32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vbool32_t res =
    __riscv_vmslt_vv_i32m1_b32_mu (mask, maskedoff, opd1, opd2, vl);
  return res;
}
```

5.8.9. Vector set if less than or equal, unsigned

Example:

```
vbool32_t vector_sleu (uint32_t *src1,
                       uint32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vbool32_t res = __riscv_vmsleu_vv_u32m1_b32 (opd1, opd2, vl);
  return res;
}
```

5.8.10. Vector set if masked less than or equal, unsigned

Example:

```
vbool32_t vector_sleu_m (vbool32_t mask,
                         uint32_t *src1,
                         uint32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vbool32_t res = __riscv_vmsleu_vv_u32m1_b32_m (mask, opd1, opd2,
vl);
  return res;
}
```


Example with mask undisturbed policy:

```
vbool32_t vector_sleu_mu (vbool32_t mask,
                          vbool32_t maskedoff,
                          uint32_t *src1,
                          uint32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (msrc1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vbool32_t res =
    __riscv_vmsleu_vv_u32m1_b32_mu (mask, maskedoff, opd1, opd2, vl);
  return res;
}
```

5.8.11. Vector set if less than or equal, signed

Example:

```
vbool32_t vector_sle (int32_t *src1,
                      int32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vbool32_t res = __riscv_vmsle_vv_i32m1_b32 (opd1, opd2, vl);
  return res;
}
```

5.8.12. Vector set if masked less than or equal, signed

Example:

```
vbool32_t vector_sle_m (vbool32_t mask,
                        int32_t *src1,
                        int32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vbool32_t res = __riscv_vmsle_vv_i32m1_b32_m (mask, opd1, opd2, vl);
  return res;
}
```


Example with mask undisturbed policy:

```
vbool32_t vector_sle_mu (vbool32_t mask,
                         vbool32_t maskedoff,
                         int32_t *src1,
                         int32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vbool32_t res =
    __riscv_vmsle_vv_i32m1_b32_mu (mask, maskedoff, opd1, opd2, vl);
  return res;
}
```

5.8.13. Vector set if greater than, unsigned

Example:

```
vbool32_t vector_sgtu (uint32_t *src1,
                       uint32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vbool32_t res = __riscv_vmsgtu_vv_u32m1_b32 (opd1, opd2, vl);
  return res;
}
```

5.8.14. Vector set if masked greater than, unsigned

Example:

```
vbool32_t vector_sgtu_m (vbool32_t mask,
                         uint32_t *src1,
                         uint32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vbool32_t res = __riscv_vmsgtu_vv_u32m1_b32_m (mask, opd1, opd2,
vl);
  return res;
}
```


Example with mask undisturbed policy:

```
vbool32_t vector_sgtu_mu (vbool32_t mask,
                          vbool32_t maskedoff,
                          uint32_t *src1,
                          uint32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vbool32_t res =
    __riscv_vmsgtu_vv_u32m1_b32_mu (mask, maskedoff, opd1, opd2, vl);
  return res;
}
```

5.8.15. Vector set if greater than, signed

Example:

```
vbool32_t vector_sgt (int32_t *src1,
                      int32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vbool32_t res = __riscv_vmsgt_vv_i32m1_b32 (opd1, opd2, vl);
  return res;
}
```

5.8.16. Vector set if masked greater than, signed

Example:

```
vbool32_t vector_sgt_m (vbool32_t mask,
                        int32_t *src1,
                        int32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vbool32_t res = __riscv_vmsgt_vv_i32m1_b32_m (mask, opd1, opd2, vl);
  return res;
}
```


Example:

```
vbool32_t vector_sgt_mu (vbool32_t mask,
                         vbool32_t maskedoff,
                         int32_t *src1,
                         int32_t *src2)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vbool32_t res =
    __riscv_vmsgt_vv_i32m1_b32_mu (mask, maskedoff, opd1, opd2, vl);
  return res;
}
```

5.9. Vector integer minimum/maximum instructions

Signed and unsigned integer minimum and maximum instructions are supported.

The generalized form of the minimum/maximum intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```


Where OPDS is one of:

vv	Operation takes two vectors from vector register groups.
vx	Operation takes one vector from vector register groups and a 5-bit immediate that is signed extended to SEW bits unless otherwise specified.
vx	Operation takes one vector from vector register groups a scalar integer that is sign extended to SEW bits unless otherwise specified.

OP is defined in the following sections.

5.9.1. Vector signed integer minimum

Example:

```
void vector_min (int32_t *src1,
                 int32_t *src2,
                 int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vmin_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.9.2. Vector signed integer masked minimum

Example:

```
void vector_min_m (vbool32_t mask,
                   int32_t *src1,
                   int32_t *src2,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
```

```
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res = __riscv_vmin_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:


```
void vector_min_mu (vbool32_t mask,
                    vint32m1_t maskedoff,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vmin_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.9.3. Vector unsigned integer minimum

Example:

```
void vector_vminu (uint32_t *src1,
                   uint32_t *src2,
                   uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vminu_vv_u32m1 (opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.9.4. Vector unsigned integer masked minimum

Example:

```
void vector_minu_m (vbool32_t mask,
                    uint32_t *src1,
                    uint32_t *src2,
                    uint32_t *dest)
```

```
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res = __riscv_vminu_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_minu_mu (vbool32_t mask,
                     vuint32m1_t maskedoff,
                     uint32_t *src1,
                     uint32_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vminu_vv_u32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.9.5. Vector signed integer maximum

Example:

```
void vector_max (int32_t *src1,
                 int32_t *src2,
                 int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vmax_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.9.6. Vector signed integer masked maximum

Example:

```
void vector_max_m (vbool32_t mask,
                   int32_t *src1,
```

```
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res = __riscv_vmax_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_max_mu (vbool32_t mask,
                    vint32m1_t maskedoff,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vmax_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.9.7. Vector unsigned integer maximum

Example:

```
void vector_maxu (uint32_t *src1,
                  uint32_t *src2,
                  uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vmaxu_vv_u32m1 (opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.9.8. Vector unsigned integer masked maximum

Example:

```
void vector_maxu_m (vbool32_t mask,
                    uint32_t *src1,
                    uint32_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res = __riscv_vmaxu_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_maxu_mu (vbool32_t mask,
                     vuint32m1_t maskedoff,
                     uint32_t *src1,
                     uint32_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vmaxu_vv_u32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.10. 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.

The generalized form of the single-width integer multiply intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where `OPDS` is one of:

vv	Operation takes two vectors from vector register groups.
vx	Operation takes one vector from vector register groups and a 5-bit immediate that is signed extended to `SEW` bits unless otherwise specified.
vx	Operation takes one vector from vector register groups and a scalar integer that is sign extended to `SEW` bits unless otherwise specified.

`OP` is defined in the following sections.

5.10.1. Vector single-width signed multiply, returning low bits of product

Example:

```
void vector_mul (int32_t *src1,
                 int32_t *src2,
                 int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vmul_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.10.2. Vector single-width signed integer masked multiply, returning low bits of product

Example:

```
void vector_mul_m (vbool32_t mask,
```

```
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res = __riscv_vmul_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_mul_mu (vbool32_t mask,
                    vint32m1_t maskedoff,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vmul_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.10.3. Vector single-width signed integer multiply, returning high bits of product

Example:

```
void vector_mulh (int32_t *src1,
                  int32_t *src2,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vmulh_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.10.4. Vector single-width signed integer masked multiply, returning high bits of product

Example:

```
void vector_mulh_m (int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res = __riscv_vmulh_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_mulh_mu (int32_t *src1,
                     vint32m1_t maskedoff,
                     int32_t *src2,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vmulh_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.10.5. Vector single-width unsigned integer multiply, returning low bits of product

Example:

```
void vector_mulu (uint32_t *src1,
                  uint32_t *src2,
                  uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vmul_vv_u32m1 (opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.10.6. Vector single-width unsigned integer masked multiply, returning low bits of product

Example:

```
void vector_mulu_m (vbool32_t mask,
                    uint32_t *src1,
                    uint32_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res = __riscv_vmul_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_mulu_mu (vbool32_t mask,
                     vuint32m1_t maskedoff,
                     uint32_t *src1,
                     uint32_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vmul_vv_u32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.10.7. Vector single-width unsigned integer multiply, returning high bits of product

Example:

```
void vector_mulhu (uint32_t *src1,
                   uint32_t *src2,
                   uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
```

```
  vuint32m1_t opd2 = __riscv_vle32_v_y32m1 (src2, vl);
  vuint32m1_t res = __riscv_vmulhu_vv_u32m1 (opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.10.8. Vector single-width unsigned integer masked multiply, returning high bits of product

Example:

```
void vector_mulhu_m (vbool32_t mask,
                     uint32_t *src1,
                     uint32_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_y32m1_m (mask, src2, vl);
  vuint32m1_t res = __riscv_vmulhu_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vector_mulhu_mu (vbool32_t mask,
                      vuint32m1_t maskedoff,
                      uint32_t *src1,
                      uint32_t *src2,
                      uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_y32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vmulhu_vv_u32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.10.9. Vector single-width signed*unsigned integer multiply, returning high bits of product

Example:

```
void vector_mulhsu (int32_t *src1,
                    uint32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vint32m1_t res = __riscv_vmulhsu_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


5.10.10. Vector single-width signed*unsigned integer masked multiply, returning high bits of product

Example:

```
void vector_mulhsu_m (vbool32_t mask,
                      int32_t *src1,
                      uint32_t *src2,
                      int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vint32m1_t res = __riscv_vmulhsu_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vector_mulhsu_mu (vbool32_t mask,
                       vint32m1_t maskedoff,
                       int32_t *src1,
                       uint32_t *src2,
                       int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vmulhsu_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.11. Vector widening integer multiply instructions

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

The generalized form of the widening integer multiply intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```


Where `OPDS` is one of:

vv	Operation takes two vectors from vector register groups.
vx	Operation takes one vector from vector register groups and a 5-bit immediate that is signed extended to `SEW` bits unless otherwise specified.
vx	Operation takes one vector from vector register groups and a scalar integer that is sign extended to `SEW` bits unless otherwise specified.

`OP` is defined in the following sections.

5.11.1. Vector widening signed integer multiply – 2*SEW = SEW + SEW

Example:

```
void vector_wmul (int16_t *src1,
                  int16_t *src2,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint32m2_t res = __riscv_vwmul_vv_i32m2 (opd1, opd2, vl);
 __riscv_vse32_v_i32m2 (dest, res, vl);
}
```

5.11.2. Vector widening signed integer masked multiply – 2*SEW = SEW + SEW

Example:

```
void vector_wmul_m (vbool16_t mask,
                    int16_t *src1,
                    int16_t *src2,
                    int32_t *dest)
{
```

```
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1_m (mask, src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1_m (mask, src2, vl);
  vint32m2_t res = __riscv_vwmul_vv_i32m2_m (mask, opd1, opd2, vl);
 __riscv_vse32_v_i32m2_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_wmul_mu (vbool16_t mask,
                     vint32m2_t maskedoff,
                     int16_t *src1,
                     int16_t *src2,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint32m2_t res =
    __riscv_vwmul_vv_i32m2_mu (mask, maskedoff, opd1, opd2, vl);
 __riscv_vse32_v_i32m2 (dest, res, vl);
}
```

5.11.3. Vector widening unsigned integer multiply – 2*SEW = SEW + SEW

Example:

```
void vector_wmulu (uint16_t *src1,
                   uint16_t *src2,
                   uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vuint32m2_t res = __riscv_vwmulu_vv_u32m2 (opd1, opd2, vl);
 __riscv_vse32_v_u32m2 (dest, res, vl);
}
```

5.11.4. Vector widening unsigned integer masked multiply – 2*SEW = SEW + SEW

Example:

```
void vector_wmulu_m (vbool16_t mask,
                     uint16_t *src1,
                     uint16_t *src2,
```

```
                        uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1_m (mask, src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1_m (mask, src2, vl);
  vuint32m2_t res = __riscv_vwmulu_vv_u32m2_m (mask, opd1, opd2, vl);
 __riscv_vse32_v_u32m2_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_wmulu_mu (vbool16_t mask,
                      vuint32m2_t maskedoff,
                      uint16_t *src1,
                      uint16_t *src2,
                      uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vuint32m2_t res =
    __riscv_vwmulu_vv_u32m2_mu (mask, maskedoff, opd1, opd2, vl);
 __riscv_vse32_v_u32m2 (dest, res, vl);
}
```

5.11.5. Vector widening signed*unsigned integer multiply – 2*SEW = SEW + SEW

Example:

```
void vector_wmulsu (int32_t *src1,
                    uint32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vint32m1_t res = __riscv_vwmulsu_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.11.6. Vector widening signed*unsigned integer masked multiply – 2*SEW = SEW + SEW

Example:

```
void vector_wmulsu_m (vbool32_t mask,
                      int32_t *src1,
                      uint32_t *src2,
                      int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vint32m2_t res = __riscv_vwmulsu_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m2_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_wmulsu_mu (vbool32_t mask,
                       vint32m2_t maskedoff,
                       int32_t *src1,
                       uint32_t *src2,
                       int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vint32m2_t res =
    __riscv_vwmulsu_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m2 (dest, res, vl);
}
```

5.12. Vector single-width integer multiply and add instructions

The integer multiply-add instructions are destructive and are provided in two forms, one that overwrites the addend or minuend and one that overwrites the first multiplicand.

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

The generalized form of the single-width integer multiply and add intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where **OPDS** is one of:

vv	Operation takes two vectors from vector register groups.
vx	Operation takes one vector from vector register groups and a 5-bit immediate that is signed extended to **SEW** bits unless otherwise specified
vx	Operation takes one vector from vector register groups and a scalar integer that is sign extended to **SEW** bits unless otherwise specified

OP is defined in the following sections.

5.12.1. Vector single-width signed integer multiply and add, overwrite addend

Example:

```
void vector_macc (int32_t *src1,
                  int32_t *src2,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t vd = __riscv_vle32_v_i32m1 (dest, vl);
  vd = __riscv_vmacc_vv_i32m1 (vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, vd, vl);
}
```

5.12.2. Vector single-width signed integer masked multiply and add, overwrite addend

Example:

```
void vector_macc_m (vbool32_t mask,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t vd = __riscv_vle32_v_i32m1_m (mask, dest, vl);
  vd = __riscv_vmacc_vv_i32m1_m (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, vd, vl);
}
```


Example with mask undisturbed policy:

```
void vector_macc_mu (vbool32_t mask,
                     int32_t *src1,
                     int32_t *src2,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t vd = __riscv_vle32_v_i32m1 (dest, vl);
  // 'vd' is the equivalent of 'maskedoff' in general instructions.
  vd = __riscv_vmacc_vv_i32m1_mu (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, vd, vl);
}
```

5.12.3. Vector single-width unsigned integer multiply and add, overwrite addend

Example:

```
void vector_maccu (uint32_t *src1,
                   uint32_t *src2,
                   uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t vd = __riscv_vle32_v_u32m1 (dest, vl);
  vd = __riscv_vmacc_vv_u32m1 (vd, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, vd, vl);
}
```

5.12.4. Vector single-width unsigned integer masked multiply and add, overwrite addend

Example:

```
void vector_maccu_m (vbool32_t mask,
                     uint32_t *src1,
                     uint32_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t vd = __riscv_vle32_v_u32m1_m (mask, dest, vl);
  vd = __riscv_vmacc_vv_u32m1_m (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, vd, vl);
}
```


Example with mask undisturbed policy:

```
void vector_maccu_mu (vbool32_t mask,
                      uint32_t *src1,
                      uint32_t *src2,
                      uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t vd = __riscv_vle32_v_u32m1 (dest, vl);
  // 'vd' is the equivalent of 'maskedoff' in general instructions.
  vd = __riscv_vmacc_vv_u32m1_mu (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, vd, vl);
}
```

5.12.5. Vector single-width signed integer multiply and add, overwrite multiplicand

Example:

```
void vector_madd (int32_t *src1,
                  int32_t *src2,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t vd = __riscv_vle32_v_i32m1 (dest, vl);
```

```
  vd = __riscv_vmadd_vv_i32m1 (vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, vd, vl);
}
```

5.12.6. Vector single-width signed integer masked multiply and add, overwrite multiplicand

Example:

```
void vector_madd_m (vbool32_t mask,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t vd = __riscv_vle32_v_i32m1_m (mask, dest, vl);
  vd = __riscv_vmadd_vv_i32m1_m (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, vd, vl);
}
```

Example with mask undisturbed policy:

```
void vector_madd_mu (vbool32_t mask,
                     int32_t *src1,
                     int32_t *src2,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t vd = __riscv_vle32_v_i32m1 (dest, vl);
  // 'vd' is the equivalent of 'maskedoff' in general instructions.
  vd = __riscv_vmadd_vv_i32m1_m (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, vd, vl);
}
```

5.12.7. Vector single-width unsigned integer multiply and add, overwrite multiplicand

Example:

```
void vector_maddu (uint32_t *src1,
                   uint32_t *src2,
                   uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t vd = __riscv_vle32_v_u32m1 (dest, vl);
  vd = __riscv_vmadd_vv_u32m1 (vd, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, vd, vl);
}
```


5.12.8. Vector single-width unsigned integer masked multiply and add, overwrite multiplicand

Example:

```
void vector_maddu_m (vbool32_t mask,
                     uint32_t *src1,
                     uint32_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t vd = __riscv_vle32_v_u32m1_m (mask, dest, vl);
  vd = __riscv_vmadd_vv_u32m1_m (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, vd, vl);
}
```

Example with mask undisturbed policy:

```
void vector_maddu_mu (vbool32_t mask,
                      uint32_t *src1,
                      uint32_t *src2,
                      uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t vd = __riscv_vle32_v_u32m1 (dest, vl);
  // 'vd' is the equivalent of 'maskedoff' in general instructions.
  vd = __riscv_vmadd_vv_u32m1_mu (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, vd, vl);
}
```

5.12.9. Vector single-width signed integer multiply and sub, overwrite minuend

Example:

```
void vector_nmsac (int32_t *src1,
                   int32_t *src2,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t vd = __riscv_vle32_v_i32m1 (dest, vl);
  vd = __riscv_vnmsac_vv_i32m1 (vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, vd, vl);
}
```


5.12.10. Vector single-width signed integer masked multiply and sub, overwrite minuend

Example:

```
void vector_nmsac_m (vbool32_t mask,
                     int32_t *src1,
                     int32_t *src2,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t vd = __riscv_vle32_v_i32m1_m (mask, dest, vl);
  vd = __riscv_vnmsac_vv_i32m1_m (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, vd, vl);
}
```

Example with mask undisturbed policy:

```
void vector_nmsac_mu (vbool32_t mask,
                      int32_t *src1,
                      int32_t *src2,
                      int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t vd = __riscv_vle32_v_i32m1 (dest, vl);
```

```
  // 'vd' is the equivalent of 'maskedoff' in general instructions.
  vd = __riscv_vnmsac_vv_i32m1_mu (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, vd, vl);
}
```

5.12.11. Vector single-width unsigned integer multiply and sub, overwrite minuend

Example:

```
void vector_nmsacu (uint32_t *src1,
                    uint32_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t vd = __riscv_vle32_v_u32m1 (dest, vl);
  vd = __riscv_vnmsac_vv_u32m1 (vd, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, vd, vl);
}
```

5.12.12. Vector single-width unsigned integer masked multiply and sub, overwrite minuend

Example:

```
void vector_nmsacu_m (vbool32_t mask,
                      uint32_t *src1,
                      uint32_t *src2,
                      uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t vd = __riscv_vle32_v_u32m1_m (mask, dest, vl);
  vd = __riscv_vnmsac_vv_u32m1_m (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, vd, vl);
}
```

Example with mask undisturbed policy:

```
void vector_nmsacu_mu (vbool32_t mask,
                       uint32_t *src1,
```

```
                        uint32_t *src2,
                        uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t vd = __riscv_vle32_v_u32m1 (dest, vl);
  // 'vd' is the equivalent of 'maskedoff' in general instructions.
  vd = __riscv_vnmsac_vv_u32m1_mu (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, vd, vl);
}
```


5.12.13. Vector single-width signed integer multiply and sub, overwrite multiplicand

Example:

```
void vector_nmsub (int32_t *src1,
                   int32_t *src2,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t vd = __riscv_vle32_v_i32m1 (dest, vl);
  vd = __riscv_vnmsub_vv_i32m1 (vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, vd, vl);
}
```

5.12.14. Vector single-width signed integer masked multiply and sub, overwrite multiplicand

Example:

```
void vector_nmsub_m (vbool32_t mask,
                     int32_t *src1,
                     int32_t *src2,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t vd = __riscv_vle32_v_i32m1_m (mask, dest, vl);
```

```
  vd = __riscv_vnmsub_vv_i32m1_m (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, vd, vl);
}
```

Example with mask undisturbed policy:

```
void vector_nmsub_mu (vbool32_t mask,
                      int32_t *src1,
                      int32_t *src2,
                      int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t vd = __riscv_vle32_v_i32m1 (dest, vl);
  // 'vd' is the equivalent of 'maskedoff' in general instructions.
  vd = __riscv_vnmsub_vv_i32m1_mu (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, vd, vl);
}
```


5.12.15. Vector single-width unsigned integer multiply and sub, overwrite multiplicand

Example:

```
void vector_nmsubu (uint32_t *src1,
                    uint32_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t vd = __riscv_vle32_v_u32m1 (dest, vl);
  vd = __riscv_vnmsub_vv_u32m1 (vd, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, vd, vl);
}
```

5.12.16. Vector single-width unsigned integer masked multiply and sub, overwrite multiplicand

Example:

```
void vector_nmsubu_m (vbool32_t mask,
                      uint32_t *src1,
```

```
                        uint32_t *src2,
                        uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t vd = __riscv_vle32_v_u32m1_m (mask, dest, vl);
  vd = __riscv_vnmsub_vv_u32m1_m (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, vd, vl);
}
```


Example with mask undisturbed policy:

```
void vector_nmsubu_mu (vbool32_t mask,
                        uint32_t *src1,
                        uint32_t *src2,
                        uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t vd = __riscv_vle32_v_u32m1 (dest, vl);
  // 'vd' is the equivalent of 'maskedoff' in general instructions.
  vd = __riscv_vnmsub_vv_u32m1_mu (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, vd, vl);
}
```

5.13. Vector widening integer multiply and 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.

The generalized form of the widening integer multiply and add intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where `OPDS` is one of:

vv	Operation takes two vectors from vector register groups.
vx	Operation takes one vector from vector register groups and a 5-bit immediate that is signed extended to `SEW` bits unless otherwise specified.
vx	Operation takes one vector from vector register groups and a scalar integer that is sign extended to `SEW` bits unless otherwise specified.

`OP` is defined in the following sections.

5.13.1. Widening signed-integer multiply-add, overwrite addend

Example:

```
void vector_wmacc (int16_t *src1,
                   int16_t *src2,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint32m2_t vd = __riscv_vle32_v_i32m2 (dest, vl);
  vd = __riscv_vwmacc_vv_i32m2 (vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m2 (dest, vd, vl);
}
```

5.13.2. Widening signed-integer masked multiply-add, overwrite addend

Example:

```
void vector_wmacc_m (vbool16_t mask,
                     int16_t *src1,
                     int16_t *src2,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1_m (mask, src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1_m (mask, src2, vl);
  vint32m2_t vd = __riscv_vle32_v_i32m2_m (mask, dest, vl);
  vd = __riscv_vwmacc_vv_i32m2_m (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m2_m (mask, dest, vd, vl);
}
```


Example with mask undisturbed policy:

```
void vector_wmacc_mu (vbool16_t mask,
                      int16_t *src1,
                      int16_t *src2,
                      int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint32m2_t vd = __riscv_vle32_v_i32m2 (dest, vl);
  // 'vd' is the equivalent of 'maskedoff' in general instructions.
  vd = __riscv_vwmacc_vv_i32m2_m (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m2 (dest, vd, vl);
}
```

5.13.3. Widening unsigned-integer multiply-add, overwrite addend

Example:

```
void vector_wmaccu (uint16_t *src1,
                    uint16_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vuint32m2_t vd = __riscv_vle32_v_u32m2 (dest, vl);
```

```
  vd = __riscv_vwmaccu_vv_u32m2 (vd, opd1, opd2, vl);
  __riscv_vse32_v_u32m2 (dest, vd, vl);
}
```

5.13.4. Widening unsigned-integer masked multiply-add, overwrite addend

Example:


```
void vector_wmaccu_m (vbool16_t mask,
                      int16_t *src1,
                      int16_t *src2,
                      int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1_m (mask, src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1_m (mask, src2, vl);
  vint32m2_t vd = __riscv_vle32_v_i32m2_m (mask, dest, vl);
  vd = __riscv_vwmaccu_vv_i32m2_m (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m2_m (mask, dest, vd, vl);
}
```

Example with mask undisturbed policy:

```
void vector_wmaccu_mu (vbool16_t mask,
                       int16_t *src1,
                       int16_t *src2,
                       int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint32m2_t vd = __riscv_vle32_v_i32m2 (dest, vl);
  // 'vd' is the equivalent of 'maskedoff' in general instructions.
  vd = __riscv_vwmaccu_vv_i32m2_mu (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m2 (dest, vd, vl);
}
```

5.13.5. Widening signed*unsigned multiply-add, overwrite addend

Example:

```
void vector_wmaccsu (int16_t *src1,
                     uint16_t *src2,
                     int32_t *dest)
{
```

```
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vint32m2_t vd = __riscv_vle32_v_i32m2 (dest, vl);
  vd = __riscv_vwmaccsu_vv_i32m2 (vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m2 (dest, vd, vl);
}
```


5.13.6. Widening signed*unsigned masked multiply-add, overwrite addend

Example:

```
void vector_wmaccsu_m (vbool16_t mask,
                       int16_t *src1,
                       uint16_t *src2,
                       int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1_m (mask, src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1_m (mask, src2, vl);
  vint32m2_t vd = __riscv_vle32_v_i32m2_m (mask, dest, vl);
  vd = __riscv_vwmaccsu_vv_i32m2_m (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m2_m (mask, dest, vd, vl);
}
```

Example with mask undisturbed policy:

```
void vector_wmaccsu_mu (vbool16_t mask,
                        int16_t *src1,
                        uint16_t *src2,
                        int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vint32m2_t vd = __riscv_vle32_v_i32m2 (dest, vl);
  // 'vd' is the equivalent of 'maskedoff' in general instructions.
  vd = __riscv_vwmaccsu_vv_i32m2_mu (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m2 (dest, vd, vl);
}
```

5.13.7. Widening unsigned*signed multiply-add, overwrite addend

Example:

```
void vector_wmaccus (uint16_t src1,
                     int16_t *src2,
                     int32_t *dest)
{
  // no vv instruction available. Only rx.
  size_t vl = __riscv_vsetvlmax_e16m1();
  vunt16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint32m2_t vd = __riscv_vle32_v_i32m2 (dest, vl);
  vd = __riscv_vwmaccus_vx_i32m2 (vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m2 (dest, vd, vl);
}
```


5.13.8. Widening unsigned*signed masked multiply-add, overwrite addend

Example:

```
void vector_wmaccus_m (vbool16_t mask,
                       uint16_t src1,
                       int16_t *src2,
                       int32_t *dest)
{
  // no vv instruction available. Only rx.
  size_t vl = __riscv_vsetvlmax_e16m1();
  vunt16m1_t opd1 = __riscv_vle16_v_u16m1_m (mask, src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1_m (mask, src2, vl);
  vint32m2_t vd = __riscv_vle32_v_i32m2_m (mask, dest, vl);
  vd = __riscv_vwmaccus_vx_i32m2_m (mask, vd, opd1, opd2, vl);
  __riscv_vse32_v_i32m2_m (mask, dest, vd, vl);
}
```

Example with mask undisturbed policy:

```
void vector_wmaccus_mu (vbool16_t mask,
                        uint16_t src1,
                        int16_t *src2,
                        int32_t *dest)
{
  // no vv instruction available. Only rx.
  size_t vl = __riscv_vsetvlmax_e16m1();
  vunt16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint32m2_t vd = __riscv_vle32_v_i32m2 (dest, vl);
  // 'vd' is the equivalent of 'maskedoff' in general instructions.
  vd = __riscv_vwmaccus_vx_i32m2_mu (mask, vd, opd1, opd2, vl);
```

```
    __riscv_vse32_v_i32m2 (dest, vd, vl);
}
```

5.14. Vector integer divide/remainder instructions

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

The generalized form of the divide and remainder intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```


Where OPDS is one of:

vv	Operation takes two vectors from vector register groups.
vx	Operation takes one vector from vector register groups and a 5-bit immediate that is signed extended to SEW bits unless otherwise specified.
vx	Operation takes one vector from vector register groups and a scalar integer that is sign extended to SEW bits unless otherwise specified.

OP is defined in the following sections.

5.14.1. Vector signed integer divide

Example:

```
void vector_div (int32_t *src1,
                 int32_t *src2,
                 int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vdiv_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.14.2. Vector signed integer masked divide

Example:

```
void vector_div_m (vbool32_t mask,
                   int32_t *src1,
                   int32_t *src2,
                   int32_t *dest)
{
```

```
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res = __riscv_vdiv_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_div_mu (vbool32_t mask,
                    vint32m1_t maskedoff,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vdiv_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.14.3. Vector unsigned integer divide

Example:

```
void vector_divu (uint32_t *src1,
                  uint32_t *src2,
                  uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vdivu_vv_u32m1 (opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.14.4. Vector unsigned integer masked divide

Example:

```
void vector_divu_m (vbool32_t mask,
                    uint32_t *src1,
                    uint32_t *src2,
```

```
                        uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res = __riscv_vdivu_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_divu_mu (vbool32_t mask,
                     vuint32m1_t maskedoff,
                     uint32_t *src1,
                     uint32_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vdivu_vv_u32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.14.5. Vector signed integer remainder

Example:

```
void vector_rem (int32_t *src1,
                 int32_t *src2,
                 int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vrem_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.14.6. Vector signed integer masked remainder

Example:

```
void vector_rem_m (vbool32_t mask,
```

```
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res = __riscv_vrem_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_rem_mu (vbool32_t mask,
                    vint32m1_t maskedoff,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vrem_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.14.7. Vector unsigned integer remainder

Example:

```
void vector_remu (uint32_t *src1,
                  uint32_t *src2,
                  uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vremu_vv_u32m1 (opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.14.8. Vector unsigned integer masked remainder

Example:

```
void vector_remu_m (vbool32_t mask,
                    uint32_t *src1,
                    uint32_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res = __riscv_vremu_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_remu_mu (vbool32_t mask,
                     vuint32m1_t maskedoff,
                     uint32_t *src1,
                     uint32_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vremu_vv_u32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.15. 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.

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 generalized form of the merge intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL
```

Where **OPDS** is one of:

vvm	Operation takes two vectors from vector register groups and one mask vector.
vxm	Operation takes one vector from vector register groups, a 5-bit immediate that is signed extended to **SEW** bits unless otherwise specified, and one mask vector.
vxm	Operation takes one vector from vector register groups, a scalar integer that is sign extended to **SEW** bits unless otherwise specified, and one mask vector.

OP is defined in the following sections.

There are no masked forms of these instructions.

5.15.1. Vector signed merge instruction

Example:

```
void vector_merge (vbool32_t mask,
                   int32_t *src1,
                   int32_t *src2,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vmerge_vvm_i32m1 (opd1, opd2, mask, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.15.2. Vector unsigned merge instruction

Example:

```
void vector_mergeu (vbool32_t mask,
                    uint32_t *src1,
                    uint32_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vmerge_vvm_u32m1 (opd1, opd2, mask, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

5.16. Vector integer move instructions

The vector integer move instructions copy a source operand to a vector register group.

The generalized form of the integer move intrinsic functions is:

```
__riscv_v OP _v_ OPDS _ SEW LMUL
```


Where OPDS is one of:

v	Operation takes two vectors from vector register groups.
x	Operation takes one vector from vector register groups and a 5-bit immediate that is signed extended to SEW bits unless otherwise specified.
x	Operation takes one vector from vector register groups and a scalar integer that is sign extended to SEW bits unless otherwise specified.

OP is defined in the following sections.

There are no masked forms of these instructions.

5.16.1. Vector signed move instruction

Example:

```
void vector_mv (int32_t *src1,
                int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t res = __riscv_vmv_v_v_i32m1 (opd1, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

5.16.2. Vector unsigned move instruction

Example:

```
void vector_mvu (uint32_t *src1,
                 uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vuint32m1_t res = __riscv_vmv_v_v_u32m1 (opd1, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

6. 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 $-2^{N-1}...+2^{N-1}-1$, and unsigned integers in the range $0...+^{2N-1}$. 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.

6.1. 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.

The generalized form of the saturating add and subtract intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where **OPDS** is one of:

vv	Operation takes two vectors from vector register groups.
vi	Operation takes one vector from vector register groups and a 5-bit immediate that is zero extended to **SEW** bits unless otherwise specified.
vx	Operation takes one vector from vector register groups and a scalar integer that is zero extended to **SEW** bits unless otherwise specified.

OP is defined in the following sections.

No explicit rounding mode needs to be set.

6.1.1. Saturating addition of signed integers

Example:

```
void vector_sadd (int32_t *src1,
                  int32_t *src2,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vsadd_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

6.1.2. Saturating masked addition of signed integers

Example:

```
void vector_sadd_m (vbool32_t mask,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res = __riscv_vsadd_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_sadd_mu (vbool32_t mask,
                     vint32m1_t maskedoff,
                     int32_t *src1,
                     int32_t *src2,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vsadd_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

6.1.3. Saturating addition of unsigned integers

Example:

```
void vector_saddu (uint32_t *src1,
                   uint32_t *src2,
                   uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vsaddu_vv_u32m1 (opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

6.1.4. Saturating masked addition of unsigned integers

Example:

```
void vector_vsaddu_m (vbool32_t mask,
                      uint32_t *src1,
                      uint32_t *src2,
                      uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res = __riscv_vsaddu_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_vsaddu_mu (vbool32_t mask,
                       vuint32m1_t maskedoff,
                       uint32_t *src1,
                       uint32_t *src2,
                       uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vsaddu_vv_u32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

6.1.5. Saturating subtract of signed integers

Example:

```
void vector_ssub (int32_t *src1,
                  int32_t *src2,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vssub_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

6.1.6. Saturating masked subtract of signed integers

Example:

```
void vector_ssub_m (vbool32_t mask,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res = __riscv_vssub_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_ssub_mu (vbool32_t mask,
                     vint32m1_t maskedoff,
                     int32_t *src1,
                     int32_t *src2,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vssub_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

6.1.7. Saturating subtract of unsigned integers

Example:

```
void vector_ssubu (uint32_t *src1,
                   uint32_t *src2,
                   uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vssubu_vv_u32m1 (opd1, opd2, vl);
```

```
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

6.1.8. Saturating masked subtract of unsigned integers

Example:

```
void vector_vssubu_m (vbool32_t mask,
                      uint32_t *src1,
                      uint32_t *src2,
                      uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res = __riscv_vssubu_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_vssubu_mu (vbool32_t mask,
                       vuint32m1_t maskedoff,
                       uint32_t *src1,
                       uint32_t *src2,
                       uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vssubu_vv_u32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

6.2. Vector single-width averaging addition and subtraction

The averaging add and subtract instructions right shift the result by one bit and round off the result according to the setting in `vxrm`. Both unsigned and signed versions are provided. For `vaaddu` 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 `rnu` or `rne` rounding.

The generalized form of the averaging addition and subtraction intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where **OPDS** is one of:

vv	Operation takes two vectors from vector register groups.
vx	Operation takes one vector from vector register groups and a scalar integer that is zero or signed extended to **SEW** bits unless otherwise specified

OP is defined in the following sections.

6.2.1. Averaging add of signed integers

Example:

```
void vector_aadd (int32_t *src1,
                  int32_t *src2,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vaadd_vv_i32m1 (opd1, opd2, __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

6.2.2. Averaging masked add of signed integers

Example:

```
void vector_aadd_m (vbool32_t mask,
```

```
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res =
    __riscv_vaadd_vv_i32m1_m (mask, opd1, opd2, __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_aadd_mu (vbool32_t mask,
                    vint32m1_t maskedoff,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vaadd_vv_i32m1_mu (mask, maskedoff,
                                              opd1, opd2,
                                              __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

6.2.3. Averaging add of unsigned integers

Example:

```
void vector_aaddu (uint32_t *src1,
                   uint32_t *src2,
                   uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vaaddu_vv_u32m1 (opd1, opd2, __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

6.2.4. Averaging masked add of unsigned integers

Example:

```
void vector_vaaddu_m (vbool32_t mask,
                      uint32_t *src1,
                      uint32_t *src2,
                      uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res =
    __riscv_vaaddu_vv_u32m1_m (mask, opd1, opd2, __RISCV_VXRM_RNE,
vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_vaaddu_mu (vbool32_t mask,
                       vuint32m1_t maskedoff,
                       uint32_t *src1,
                       uint32_t *src2,
                       uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vaaddu_vv_u32m1_mu (mask, maskedoff,
                                                opd1, opd2,
                                                __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

6.2.5. Averaging sub of signed integers

Example:

```
void vector_asub (int32_t *src1,
                  int32_t *src2,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
```

```
  vint32m1_t res =
    __riscv_vasub_vv_i32m1 (opd1, opd2, __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

6.2.6. Averaging masked sub of signed integers

Example:

```
void vector_asub_m (vbool32_t mask,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res =
    __riscv_vasub_vv_i32m1_m (mask, opd1, opd2, __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vector_asub_mu (vbool32_t mask,
                     vint32m1_t maskedoff,
                     int32_t *src1,
                     int32_t *src2,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res = __riscv_vasub_vv_i32m1_mu (mask, maskedoff,
                                              opd1, opd2,
                                              __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

6.2.7. Averaging sub of unsigned integers

Example:

```
void vector_asubu (uint32_t *src1,
                   uint32_t *src2,
```

```
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vasubu_vv_u32m1 (opd1, opd2, __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```


6.2.8. Averaging masked sub of unsigned integers

Example:

```
void vector_asubu_m (vbool32_t mask,
                     uint32_t *src1,
                     uint32_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res =
    __riscv_vasubu_vv_u32m1_m (mask, opd1, opd2, __RISCV_VXRM_RNE,
vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vector_asubu_mu (vbool32_t mask,
                      vuint32m1_t maskedoff,
                      uint32_t *src1,
                      uint32_t *src2,
                      uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vasubu_vv_u32m1_mu (mask, maskedoff,
                                                opd1, opd2,
                                                __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

6.3. Vector single-width fractional multiply with rounding and saturation

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

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.

There is no 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`).

The generalized form of the single-width fractional multiply with mounding and maturation intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where `OPDS` is one of:

`vv`	Operation takes two vectors from vector register groups.
`vx`	Operation takes one vector from vector register groups and a scalar integer that is zero or signed extended to `SEW` bits unless otherwise specified

`OP` is defined in the following sections.

6.3.1. Signed saturating and rounding fractional multiply

Example:

```
void vector_smul (int32_t *src1,
                  int32_t *src2,
                  int32_t *dest)
{
  // there is no masked form of this instruction
  size_t vl = __riscv_vsetvlmax_e32m1();
```

```
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vsmul_vv_i32m1 (opd1, opd2, __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


6.3.2. Signed masked saturating and rounding fractional multiply

Example:

```
void vector_smul_m (vbool32_t mask,
                    int32_t *src1,
                    int32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res =
    __riscv_vsmul_vv_i32m1_m (mask, opd1, opd2, __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vector_smul_mu (vbool32_t mask,
                     vint32m1_t maskedoff,
                     int32_t *src1,
                     int32_t *src2,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
   vint32m1_t res = __riscv_vsmul_vv_i32m1_mu (mask, maskedoff,
                                               opd1, opd2,
                                               __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

6.4. Vector single-width scaling shift instructions

These instructions shift the input value right and round off the shifted-out bits according to `vxrm`. The scaling right shifts have both zero-extending and sign-extending forms. The data to be shifted is in the vector register group and the shift amount value can come from a vector register group, a scalar integer register, or a zero-extended 5-bit immediate. Only the low `lg2(SEW)` bits of the shift-amount value are used to control the shift amount.

The generalized form of the single-width scaling shift intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where **OPDS** is one of:

vv	Operation takes two vectors from vector register groups.
vx	Operation takes one vector from register group and a 5-bit immediate that is signed extended to **SEW** bits unless otherwise specified.
vx	Operation takes one vector from register group and a scalar integer variable that is sign extended to **SEW** bits unless otherwise specified.

OP is defined in the following sections.

6.4.1. Vector single-width scaling shift right logical

Example:

```
void vector_ssrl (uint32_t *src1,
                  uint32_t *src2,
                  uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res =
    __riscv_vssrl_vv_u32m1 (opd1, opd2, __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

6.4.2. Vector single width scaling masked shift right logical

Example:

```
void vector_ssrl_m (vbool32_t mask,
```

```
                    uint32_t *src1,
                    uint32_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res =
    __riscv_vssrl_vv_u32m1_m (mask, opd1, opd2, __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_ssrl_mu (vbool32_t mask,
                    vuint32m1_t maskedoff,
                    uint32_t *src1,
                    uint32_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vssrl_vv_u32m1_mu (mask, maskedoff,
                                               opd1, opd2,
                                               __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

6.4.3. Vector single-width scaling shift right arithmetic

Example:

```
void vector_ssra (int32_t *src1,
                  uint32_t *src2,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vint32m1_t res =
    __riscv_vssra_vv_i32m1 (opd1, opd2, __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

6.4.4. Vector single width scaling masked shift right arithmetic

Example:

```
void vector_ssra_m (vbool32_t mask,
                    int32_t *src1,
                    uint32_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vint32m1_t res =
    __riscv_vssra_vv_i32m1_m (mask, opd1, opd2, __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vector_ssra_mu (vbool32_t mask,
                     vint32m1_t maskedoff,
                     int32_t *src1,
                     uint32_t *src2,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vint32m1_t res = __riscv_vssra_vv_i32m1_mu (mask, maskedoff,
                                              opd1, opd2,
                                              __RISCV_VXRM_RNE, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

6.5. 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 a vector in the vector register group. The scaling shift amount value can come from a vector register group, a scalar integer register, or a zero-extended 5-bit immediate. The low `lg2(2*SEW)` bits of the vector or scalar shift-amount value (e.g., the low 6 bits for a `SEW=64-bit` to `SEW=32-bit` narrowing operation) are used to control the right shift amount, which provides the scaling.

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 in between 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.

The generalized form of the narrowing fixed-point clip intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where OPDS is one of:

vv	Operation takes two vectors from vector register groups of SEW size. The return result is SEW*2.
wv	Operation takes two vectors from vector register groups where one argument is of size SEW*2 while the other is of size SEW. The return result is SEW*2.
vx	Operation takes one vector from register group of SEW size and a 5-bit immediate that is signed extended to SEW bits unless otherwise specified. The return result is SEW*2.
vx	Operation takes one vector from register group of SEW size and a scalar integer variable that is sign extended to SEW bits unless otherwise specified. The return result is SEW*2.

OP is defined in the following sections.

6.5.1. Vector narrowing signed clip

Example:

```
void vector_nclip (int32_t *src1,
                   uint16_t *src2,
                   int16_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint32m2_t opd1 = __riscv_vle32_v_i32m2 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle32_v_u16m1 (src2, vl);
  vint16m1_t res =
    __riscv_vnclip_wv_i16m1 (opd1, opd2, __RISCV_VXRM_RNE, vl);
  __riscv_vse16_v_i16m1 (dest, res, vl);
}
```


6.5.2. Vector narrowing signed masked clip

Example:

```
void vector_nclip_m (vbool16_t mask,
                     int32_t *src1,
                     uint16_t *src2,
                     int16_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint32m2_t opd1 = __riscv_vle32_v_i32m2_m (mask, src1, vl);
  vuint16m1_t opd2 = __riscv_vle32_v_u16m1_m (mask, src2, vl);
  vint16m1_t res =
    __riscv_vnclip_wv_i16m1_m (mask, opd1, opd2, __RISCV_VXRM_RNE,
vl);
  __riscv_vse16_v_i16m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vector_nclip_mu (vbool16_t mask,
                      vint16m1_t maskedoff,
                      int32_t *src1,
                      uint16_t *src2,
                      int16_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint32m2_t opd1 = __riscv_vle32_v_i32m2 (msrc1, vl);
  vuint16m1_t opd2 = __riscv_vle32_v_u16m1 (src2, vl);
  vint16m1_t res = __riscv_vnclip_wv_i16m1_mu (mask, maskedoff,
                                               opd1, opd2,
```

```
                                                    __RISCV_VXRM_RNE, vl);
  __riscv_vse16_v_i16m1 (dest, res, vl);
}
```

6.5.3. Vector narrowing unsigned clip

Example:


```
void vector_nclipu (uint32_t *src1,
                    uint16_t *src2,
                    uint16_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint32m2_t opd1 = __riscv_vle32_v_u32m2 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle32_v_u16m1 (src2, vl);
  vuint16m1_t res =
    __riscv_vnclipu_wv_u16m1 (opd1, opd2, __RISCV_VXRM_RNE, vl);
  __riscv_vse16_v_u16m1 (dest, res, vl);
}
```

6.5.4. vector narrowing unsigned masked clip

Example:

```
void vector_nclip_m (vbool16_t mask,
                     uint32_t *src1,
                     uint16_t *src2,
                     uint16_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint32m2_t opd1 = __riscv_vle32_v_u32m2_m (mask, src1, vl);
  vuint16m1_t opd2 = __riscv_vle32_v_u16m1_m (mask, src2, vl);
  vuint16m1_t res =
    __riscv_vnclip_wv_u16m1_m (mask, opd1, opd2, __RISCV_VXRM_RNE,
vl);
  __riscv_vse16_v_u16m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vector_nclip_mu (vbool16_t mask,
                      vuint16m1_t maskedoff,
                      uint32_t *src1,
                      uint16_t *src2,
                      uint16_t *dest)
```

```
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint32m2_t opd1 = __riscv_vle32_v_u32m2 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle32_v_u16m1 (src2, vl);
  vuint16m1_t res = __riscv_vnclip_wv_u16m1_mu (mask, maskedoff,
                                                opd1, opd2,
                                                __RISCV_VXRM_RNE, vl);
  __riscv_vse16_v_u16m1 (dest, res, vl);
}
```

7. Vector floating point instructions

The standard vector floating-point instructions treat elements as IEEE-754/2008-compatible values.

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.

The vector floating-point instructions have the same behavior as the scalar floating-point instructions regarding NaNs.

7.1. Vector single-width floating-point add and subtract instructions

The generalized form of the single-width floating-point add and subtract intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_rm] [_ TM]
```

Where OPDS is one of:

vv	Operation takes two vectors from vector register groups.
vf	Operation takes one vector from vector register groups and a scalar floating-point register.

OP is defined in the following sections.

7.1.1. Vector single-width floating-point add instruction

Example:

```
void fadd (float32_t *src1,
           float32_t *src2,
           float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vfadd_vv_f32m1 (opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void fadd (float32_t *src1,
           float32_t *src2,
           float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res =
    __riscv_vfadd_vv_f32m1_rm (opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.1.2. Vector single-width floating-point masked add instruction

Example:

```
void fadd_m (vbool32_t mask,
             float32_t *src1,
             float32_t *src2,
             float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res = __riscv_vfadd_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void fadd_mu (vbool32_t mask,
              vfloat32m1_t maskedoff,
              float32_t *src1,
              float32_t *src2,
              float32_t *dest)
(
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res =
    __riscv_vfadd_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.1.3. Vector single-width floating-point subtract instruction

Example:

```
void fsub (float32_t *src1,
           float32_t *src2,
           float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vfsub_vv_f32m1 (opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
```

```
}
```

Example with rounding mode:

```
void fsub (float32_t *src1,
           float32_t *src2,
           float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res =
    __riscv_vfsub_vv_f32m1_rm (opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


7.1.4. Vector single-width floating-point masked subtract instruction

Example:

```
void fsub_m (vbool32_t mask,
             float32_t *src1,
             float32_t *src2,
             float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res = __riscv_vfsub_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fsub_mu (vbool32_t mask,
              vfloat32m1_t maskedoff,
              float32_t *src1,
              float32_t *src2,
              float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res =
    __riscv_vfsub_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
```

```
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.1.5. Vector single-width floating-point reverse subtract instruction

Example:

```
void frsub (float32_t *src1,
            float32_t opd2,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res = __riscv_vfrsub_vv_f32m1 (opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


Example with rounding mode:

```
void frsub (float32_t *src1,
            float32_t opd2,
            float32_t *dest)
(
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfrsub_vv_f32m1 (opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.1.6. Vector single-width floating-point masked reverse subtract instruction

Example:

```
void frsub_m (vbool32_t mask,
              float32_t *src1,
              float opd2,
              float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t res = __riscv_vfrsub_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void frsub_mu (vbool32_t mask,
               vfloat32m1_t maskedoff,
               float32_t *src1,
               float opd2,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfrsub_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.2. Vector widening floating-point add and subtract instructions

The generalized form of the widening floating-point add and subtract intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_rm] [_ TM]
```

Where OPDS is one of:

vv	Operation takes two vectors from vector register groups.
vf	Operation takes one vector from vector register groups and a scalar floating-point register.
wv	Operation takes two vectors from vector register groups where the first argument is of size SEW*2 while the other is of size SEW. The return result is SEW*2.
wf	Operation takes one vector from vector register groups of size SEW*2 and a scalar floating-point register of size SEW. The return result is SEW*2.

OP is defined in the following sections.

7.2.1. Vector widening floating-point add instructions – 2*SEW = SEW + SEW

Example:

```
void fwadd (float32_t *src1,
            float32_t *src2,
            float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res = __riscv_vfwadd_vv_f64m2 (opd1, opd2, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

Example with rounding mode:

```
void fwadd (float32_t *src1,
            float32_t *src2,
            float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
```

```
  vfloat64m2_t res =
    __riscv_vfwadd_vv_f64m2_rm (opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.2.2. Vector widening floating-point masked add instructions – 2*SEW = SEW + SEW

Example:

```
void fwadd_m (vbool32_t mask,
              float32_t *src1,
              float32_t *src2,
              float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat64m2_t res = __riscv_vfwadd_vv_f64m2_m (mask, opd1, opd2, vl);
  __riscv_vse64_v_f64m2_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fwadd_mu (vbool32_t mask,
               vfloat64m2_t maskedoff,
               float32_t *src1,
               float32_t *src2,
               float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res =
    __riscv_vfwadd_vv_f64m2_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.2.3. Vector widening floating-point sub instructions – 2*SEW = SEW – SEW

Example:

```
void fwsub (float32_t *src1,
            float32_t *src2,
```

```
                float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res = __riscv_vfwsub_vv_f64m2 (opd1, odp2, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```


Example with rounding mode:

```
void fwsub (float32_t *src1,
            float32_t *src2,
            float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res =
    __riscv_vfwsub_vv_f64m2_rm (opd1, odp2, __RISCV_FRM_RNE, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.2.4. Vector widening floating-point masked sub instructions – 2*SEW = SEW – SEW

Example:

```
void fwsub_m (vbool32_t mask,
              float32_t *src1,
              float32_t *src2,
              float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat64m2_t res = __riscv_vfwsub_vv_f64m2_m (mask, opd1, opd2, vl);
  __riscv_vse64_v_f64m2_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fwsub_mu (vbool32_t mask,
               vfloat64m2_t maskedoff,
```

```
                float32_t *src1,
                float32_t *src2,
                float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res =
    __riscv_vfwsub_vv_f64m2_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```


7.2.5. Vector widening floating-point add instruction – 2*SEW = 2*SEW – SEW

Example:

```
void fwadd (float64_t *src1,
            float32_t *src2,
            float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res = __riscv_vfwadd_wv_f64m2 (opd1, opd2, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

Example with rounding mode:

```
void fwadd (float64_t *src1,
            float32_t *src2,
            float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res =
    __riscv_vfwadd_wv_f64m2_rm (opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.2.6. Vector widening floating-point masked add instruction – 2*SEW = 2*SEW – SEW

Example:

```
void fwadd_m (vbool32_t mask,
              float64_t *src1,
              float32_t *src2,
              float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat64m2_t res = __riscv_vfwadd_wv_f64m2_m (mask, opd1, opd2, vl);
  __riscv_vse64_v_f64m2_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void fwadd_mu (vbool32_t mask,
               vfloat64m2_t maskedoff,
               float64_t *src1,
               float32_t *src2,
               float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res =
    __riscv_vfwadd_wv_f64m2_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.2.7. Vector widening floating-point sub instruction – 2*SEW = 2*SEW – SEW

Example:

```
void fwsub (float64_t *src1,
            float32_t *src2,
            float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res = __riscv_vfwsub_wv_f64m2 (opd1, opd2, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```


Example with rounding mode:

```
void fwsub (float64_t *src1,
            float32_t *src2,
            float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res =
    __riscv_vfwsub_wv_f64m2_rm (opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.2.8. Vector widening floating-point masked sub instruction – 2*SEW = 2*SEW – SEW

Example:

```
void fwsub_m (vbool32_t mask,
              float64_t *src1,
              float32_t *src2,
              float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat64m2_t res = __riscv_vfwsub_wv_f64m2_m (mask, opd1, opd2, vl);
  __riscv_vse64_v_f64m2_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fwsub_mu (vbool32_t mask,
               vfloat64m2_t maskedoff,
               float64_t *src1,
               float32_t *src2,
               float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res =
    __riscv_vfwsub_wv_f64m2_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.3. Vector single-width floating-point multiply and divide instructions

The generalized form of the single-width floating-point multiply and divide intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_rm] [_ TM]
```


Where OPDS is one of:

vv	Operation takes two vectors from vector register groups
vf	Operation takes one vector from vector register groups and a scalar floating-point register

OP is defined in the following sections.

7.3.1. Vector single-width floating-point multiply instruction

Example:

```
void fmul (float32_t *src1,
           float32_t *src2,
           float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vfmul_vv_f32m1 (opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void fmul (float32_t *src1,
           float32_t *src2,
           float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res =
    __riscv_vfmul_vv_f32m1_rm (opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.3.2. Vector single-width floating-point masked multiply instruction

Example:

```
void fmul_m (vbool32_t mask,
             float32_t *src1,
             float32_t *src2,
             float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res = __riscv_vfmul_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void fmul_mu (vbool32_t mask,
              vfloat32m1_t maskedoff,
              float32_t *src1,
              float32_t *src2,
              float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res =
    __riscv_vfmul_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.3.3. Vector widening -point integer multiply - 2*SEW = SEW + SEW

Example:

```
void vector_fwmul (float32_t *src1,
                   float32_t *src2,
                   float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vfloat64m2_t res = __riscv_vfwmul_vv_f64m2 (opd1, opd2, vl);
  __riscv_vse64_v_i64m2 (dest, res, vl);
```

```
}
```

Example with rounding mode:

```
void vector_fwmul (float32_t *src1,
                   float32_t *src2,
                   float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  vfloat64m2_t res =
    __riscv_vfwmul_vv_f64m2_rm (opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse64_v_i64m2 (dest, res, vl);
}
```


7.3.4. Vector widening floating-point masked multiply – 2*SEW = SEW + SEW

Example:

```
void vector_fwmul_m (vbool16_t mask,
                     float32_t *src1,
                     float32_t *src2,
                     float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vfloat32m1_t opd1 = __riscv_vle16_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle16_v_f32m1_m (mask, src2, vl);
  vfloat64m2_t res = __riscv_vfwmul_vv_f64m2_m (mask, opd1, opd2, vl);
  __riscv_vse64_v_i64m2_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vector_fwmul_mu (vbool16_t mask,
                      vfloat64m2_t maskedoff,
                      float32_t *src1,
                      float32_t *src2,
                      float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vfloat32m1_t opd1 = __riscv_vle16_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle16_v_f32m1 (src2, vl);
  vfloat64m2_t res =
    __riscv_vfwmul_vv_f64m2_mu (mask, maskedoff, opd1, opd2, vl);
```

```
  __riscv_vse64_v_i64m2 (dest, res, vl);
}
```

7.3.5. Vector single-width floating-point divide instruction

Example:

```
void fdiv (float32_t *src1,
           float32_t *src2,
           float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vfdiv_vv_f32m1 (opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


Example with rounding mode:

```
void fdiv (float32_t *src1,
           float32_t *src2,
           float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res =
    __riscv_vfdiv_vv_f32m1_rm (opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.3.6. Vector single-width floating-point masked divide instruction

Example:

```
void fdiv_m (vbool32_t mask,
             float32_t *src1,
             float32_t *src2,
             float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res = __riscv_vfdiv_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void fdiv_mu (vbool32_t mask,
              vfloat32m1_t maskedoff,
              float32_t *src1,
              float32_t *src2,
              float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res =
    __riscv_vfdiv_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.3.7. Vector single-width floating-point reversed divide instruction

Example:

```
void frdiv (float32_t *src1,
            float32_t opd2,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res = __riscv_vfrdiv_vv_f32m1 (opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void frdiv (float32_t *src1,
            float32_t opd2,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfrdiv_vv_f32m1_rm (opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


7.3.8. Vector single-width floating-point masked reversed divide instruction

Example:

```
void frdiv_m (vbool32_t mask,
              float32_t *src1,
              float opd2,
              float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t res = __riscv_vfrdiv_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void frdiv_mu (vbool32_t mask,
               vfloat32m1_t maskedoff,
               float32_t *src1,
               float opd2,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfrdiv_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.4. Vector single-width floating-point fused multiply and 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.

The generalized form of the single-width floating-point fused multiply and add intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_rm] [_ TM]
```

Where **OPDS** is one of:

vv	Operation takes two vectors from vector register groups.
vf	Operation takes one vector from vector register groups and a scalar floating-point register.

OP is defined in the following sections.

7.4.1. Vector single-width floating-point multiply-accumulate, overwrites addend

Example:

```
void fmacc (float32_t *src1,
            float32_t *src2,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res = __riscv_vfmacc_vv_f32m1 (res, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void fmacc (float32_t *src1,
            float32_t *src2,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
```

```
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res =
    __riscv_vfmacc_vv_f32m1_rm (res, opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


7.4.2. Vector single-width floating-point masked multiply-accumulate, overwrites addend

Example:

```
void fmacc_m (vbool32_t mask,
              float32_t *src1,
              float32_t *src2,
              float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1_m (mask, dest, vl);
  res = __riscv_vfmacc_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fmacc_mu (vbool32_t mask,
               vfloat32m1_t maskedoff,
               float32_t *src1,
               float32_t *src2,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res = __riscv_vfmacc_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.4.3. Vector single-width floating-point negate-(multiply-accumulate), overwrites subtrahend

Example:

```
void fnmacc (float32_t *src1,
             float32_t *src2,
             float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res = __riscv_vfnmacc_vv_f32m1 (res, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


7.4.4. Vector single-width floating-point masked negate-(multiply-accumulate), overwrites subtrahend

Example:

```
void fnmacc_m (vbool32_t mask,
               float32_t *src1,
               float32_t *src2,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1_m (mask, dest, vl);
  res = __riscv_vfnmacc_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fnmacc_mu (vbool32_t mask,
               vfloat32m1_t maskedoff,
               float32_t *src1,
               float32_t *src2,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
```

```
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res = __riscv_vfnmacc_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


7.4.5. Vector single-width floating-point multiply-subtract-accumulator, overwrites subtrahend

Example:

```
void fmsac (float32_t *src1,
            float32_t *src2,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res = __riscv_vfmsac_vv_f32m1 (res, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void fmsac (float32_t *src1,
            float32_t *src2,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res =
    __riscv_vfmsac_vv_f32m1_rm (res, opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.4.6. Vector single-width floating-point masked multiply-subtract-accumulator, overwrites subtrahend

Example:

```
void fmsac_m (vbool32_t mask,
              float32_t *src1,
              float32_t *src2,
              float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1_m (mask, dest, vl);
  res = __riscv_vfmsac_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void fmsac_mu (vbool32_t mask,
               vfloat32m1_t maskedoff,
               float32_t *src1,
               float32_t *src2,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res = __riscv_vfmsac_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.4.7. Vector single-width floating-point negate-(multiply-subtract-accumulator), overwrites minuend

Example:

```
void fnmsac (float32_t *src1,
             float32_t *src2,
             float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
```

```
  res = __riscv_vfnmsac_vv_f32m1 (res, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void fnmsac (float32_t *src1,
             float32_t *src2,
             float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res =
    __riscv_vfnmsac_vv_f32m1_rm (res, opd1, opd2, __RISCV_FRM_RNE,
vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


7.4.8. Vector single-width floating-point masked negate-(multiply-subtract-accumulator), overwrites minuend

Example:

```
void fnmsac_m (vbool32_t mask,
               float32_t *src1,
               float32_t *src2,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1_m (mask, dest, vl);
  res = __riscv_vfnmsac_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fnmsac_mu (vbool32_t mask,
                vfloat32m1_t maskedoff,
                float32_t *src1,
                float32_t *src2,
```

```
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res = __riscv_vfnmsac_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


7.4.9. Vector single-width floating-point multiply-add, overwrites multiplicand

Example:

```
void fmadd (float32_t *src1,
            float32_t *src2,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res = __riscv_vfmadd_vv_f32m1 (res, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void fmadd (float32_t *src1,
            float32_t *src2,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res =
    __riscv_vfmadd_vv_f32m1_rm (res, opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.4.10. Vector single-width floating-point masked multiply-add, overwrites multiplicand

Example:

```
void fmadd_m (vbool32_t mask,
              float32_t *src1,
              float32_t *src2,
              float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1_m (mask, dest, vl);
  res = __riscv_vfmadd_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void fmadd_mu (vbool32_t mask,
               vfloat32m1_t maskedoff,
               float32_t *src1,
               float32_t *src2,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res = __riscv_vfmadd_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.4.11. Vector single-width floating-point negate-(multiply-add), overwrites multiplicand

Example:

```
void fmadd (float32_t *src1,
            float32_t *src2,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
```

```
  res = __riscv_vfmadd_vv_f32m1 (opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void fmadd (float32_t *src1,
            float32_t *src2,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res = __riscv_vfmadd_vv_f32m1_rm (opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


7.4.12. Vector single-width floating-point masked negate-(multiply-add), overwrites multiplicand

Example:

```
void fnmadd_m (vbool32_t mask,
               float32_t *src1,
               float32_t *src2,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1_m (mask, dest, vl);
  res = __riscv_vfnmadd_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fnmadd_mu (vbool32_t mask,
                vfloat32m1_t maskedoff,
                float32_t *src1,
                float32_t *src2,
                float32_t *dest)
{
```

```
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res = __riscv_vfnmadd_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


7.4.13. Vector single-width floating-point multiply-sub, overwrites multiplicand

Example:

```
void fmsub (float32_t *src1,
            float32_t *src2,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res = __riscv_vfmsub_vv_f32m1 (opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void fmsub (float32_t *src1,
            float32_t *src2,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res = __riscv_vfmsub_vv_f32m1_rm (opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.4.14. Vector single-width floating-point masked multiply-sub, overwrites multiplicand

Example:

```
void fmsub_m (vbool32_t mask,
```

```
                float32_t *src1,
                float32_t *src2,
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1_m (mask, dest, vl);
  res = __riscv_vfmsub_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void fmsub_mu (vbool32_t mask,
               vfloat32m1_t maskedoff,
               float32_t *src1,
               float32_t *src2,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res = __riscv_vfmsub_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.4.15. Vector single-width floating-point negate-(multiply-sub), overwrites multiplicand

Example:

```
void fnmsub (float32_t *src1,
             float32_t *src2,
             float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res = __riscv_vfnmsub_vv_f32m1 (opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void fnmsub (float32_t *src1,
             float32_t *src2,
             float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
  res = __riscv_vfnmsub_vv_f32m1_rm (opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


7.4.16. Vector single-width floating-point masked negate-(multiply-sub), overwrites multiplicand

Example:

```
void fnmsub_m (vbool32_t mask,
               float32_t *src1,
               float32_t *src2,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1_m (mask, dest, vl);
  res = __riscv_vfnmsub_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fnmsub_mu (vbool32_t mask,
               float32_t *src1,
               float32_t *src2,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vle32_v_f32m1 (dest, vl);
```

```
  res = __riscv_vfnmsub_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.5. 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.

The generalized form of the widening floating-point fused multiply and add intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_rm] [_ TM]
```

Where OPDS is one of:

vv	Operation takes two vectors from vector register groups.
vf	Operation takes one vector from vector register groups and a scalar floating-point register

OP is defined in the following sections.

7.5.1. Vector widening multiply-accumulate, overwrites addend

Example:

```
void fwmacc (float32_t *src1,
             float32_t *src2,
             float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res = __riscv_vle64_v_f64m2 (dest, vl);
  res = __riscv_vfwmac_vv_f64m2 (res, opd1, opd2, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

Example with rounding mode:

```
void fwmacc (float32_t *src1,
             float32_t *src2,
             float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
```

```
  vfloat64m2_t res = __riscv_vle64_v_f64m2 (dest, vl);
  res =
    __riscv_vfwmac_vv_f64m2_rm (res, opd1, opd2, __RISCV_FRM_RNE, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```


7.5.2. Vector widening masked multiply-accumulate, overwrites addend

Example:

```
void fwmacc_m (vbool32_t mask,
               float32_t *src1,
               float32_t *src2,
               float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat64m2_t res = __riscv_vle64_v_f64m2_m (mask, dest, vl);
  res = __riscv_vfwmac_vv_f64m2_m (mask, res, opd1, opd2, vl);
  __riscv_vse64_v_f64m2_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fwmacc_mu (vbool32_t mask,
               vfloat64m2_t maskedoff,
               float32_t *src1,
               float32_t *src2,
               float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res = __riscv_vle64_v_f64m2 (dest, vl);
  res =
    __riscv_vfwmac_vv_f64m2_mu (mask, maskedoff, res, opd1, opd2, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.5.3. Vector widening negate-(multiply-accumulate), overwrites addend

Example:

```
void fwnmacc (float32_t *src1,
              float32_t *src2,
```

```
                float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res = __riscv_vle64_v_f64m2 (dest, vl);
  res = __riscv_vfwnmacc_vv_f64m2 (res, opd1, opd2, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```


Example with rounding mode:

```
void fwnmacc (float32_t *src1,
              float32_t *src2,
              float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res = __riscv_vle64_v_f64m2 (dest, vl);
  res =
    __riscv_vfwnmacc_vv_f64m2_rm (res, opd1, opd2, __RISCV_FRM_RNE,
vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.5.4. Vector widening masked negate-(multiply-accumulate), overwrites addend

Example:

```
void fwnmacc_m (vbool32_t mask,
                float32_t *src1,
                float32_t *src2,
                float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat64m2_t res = __riscv_vle64_v_f64m2_m (mask, dest, vl);
  res = __riscv_vfwnmacc_vv_f64m2_m (mask, res, opd1, opd2, vl);
  __riscv_vse64_v_f64m2_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fwnmacc_mu (vbool32_t mask,
                 vfloat64m2_t maskedoff,
                 float32_t *src1,
                 float32_t *src2,
                 float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res = __riscv_vle64_v_f64m2 (dest, vl);
  res =
    __riscv_vfwnmacc_vv_f64m2_mu (mask, maskedoff, res, opd1, opd2,
vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```


7.5.5. Vector widening multiply-subtract-accumulator, overwrites addend

Example:

```
void fwmsac (float32_t *src1,
             float32_t *src2,
             float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res = __riscv_vle64_v_f64m2 (dest, vl);
  res = __riscv_vfwmsac_vv_f64m2 (res, opd1, opd2, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

Example with rounding mode:

```
void fwmsac (float32_t *src1,
             float32_t *src2,
             float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res = __riscv_vle64_v_f64m2 (dest, vl);
  res =
    __riscv_vfwmsac_vv_f64m2_rm (res, opd1, opd2, __RISCV_FRM_RNE,
```

```
vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.5.6. Vector widening masked multiply-subtract-accumulator, overwrites addend

Example:

```
void fwmsac_m (vbool32_t mask,
               float32_t *src1,
               float32_t *src2,
               float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat64m2_t res = __riscv_vle64_v_f64m2_m (mask, dest, vl);
  res = __riscv_vfwmsac_vv_f64m2_m (mask, res, opd1, opd2, vl);
  __riscv_vse64_v_f64m2_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fwmsac_mu (vbool32_t mask,
                vfloat64m2_t maskedoff,
                float32_t *src1,
                float32_t *src2,
                float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res = __riscv_vle64_v_f64m2 (dest, vl);
  res =
    __riscv_vfwmsac_vv_f64m2_mu (mask, maskedoff, res, opd1, opd2,
vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.5.7. Vector widening negate-(multiply-subtract-accumulator), overwrites addend

Example:

```
void fwnmsac (float32_t *src1,
              float32_t *src2,
              float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res = __riscv_vle64_v_f64m2 (dest, vl);
  res = __riscv_vfwnmsac_vv_f64m2 (res, opd1, opd2, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```


Example with rounding mode:

```
void fwnmsac (float32_t *src1,
              float32_t *src2,
              float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res = __riscv_vle64_v_f64m2 (dest, vl);
  res =
    __riscv_vfwnmsac_vv_f64m2_rm (res, opd1, opd2, __RISCV_FRM_RNE,
vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.5.8. Vector widening masked negate-(multiply-subtract-accumulator), overwrites addend

Example:

```
void fwnmsac_m (vbool32_t mask,
                float32_t *src1,
                float32_t *src2,
                float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat64m2_t res = __riscv_vle64_v_f64m2_m (mask, dest, vl);
  res = __riscv_vfwnmsac_vv_f64m2_m (mask, res, opd1, opd2, vl);
  __riscv_vse64_v_f64m2_m (mask, dest, res, vl);
```

```
}
```

Example with mask undisturbed policy:

```
void fwnmsac_mu (vbool32_t mask,
                 vfloat64m2_t maskedoff,
                 float32_t *src1,
                 float32_t *src2,
                 float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat64m2_t res = __riscv_vle64_v_f64m2 (dest, vl);
  res =
    __riscv_vfwnmsac_vv_f64m2_mu (mask, maskedoff, res, opd1, opd2,
vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.6. Vector floating-point square-root instructions

The generalized form of the floating-point square-root intrinsic functions is:

```
__riscv_vfsqrt_v_ SEW LMUL [_rm] [_ TM]
```

They are unary vector-vector instructions.

7.6.1. Vector floating-point square-root

Example:

```
void fsqrt (float32_t *src1,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res = __riscv_vfsqrt_v_f32m1 (opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void fsqrt (float32_t *src1,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfsqrt_v_f32m1_rm (opd1, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.6.2. Vector floating-point masked square root

Example:

```
void fsqrt_m (vbool32_t mask,
              float32_t *src1,
              float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t res = __riscv_vfsqrt_v_f32m1_m (mask, opd1, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
```

```
}
```

Example with mask undisturbed policy:

```
void fsqrt_mu (vbool32_t mask,
               vfloat32m1_t maskedoff,
               float32_t *src1,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfsqrt_v_f32m1_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.7. Vector floating-point reciprocal square-root estimate to 7 bits instructions

These are unary vector-vector instructions that return an estimate of 1/x accurate to 7 bits. The following table describes their behavior for all classes of floating-point inputs:

Input	Output	Exceptions raised
-∞ ≤ x < -0.0	canonical NaN	NV
-0.0	-∞	DZ
+0.0	+∞	DZ
+0.0 < x < +∞	estimate of 1/sqrt(x)	
+∞	+0.0	
qNaN	canonical NaN	
sNaN	canonical NaN	NV

All positive normal and subnormal inputs produce normal outputs, and the output value is independent of the dynamic rounding mode.

The generalized form of the floating-point square-root intrinsic functions is:

```
__riscv_vfrsqrt7_v_ SEW LMUL [_ TM]
```

7.7.1. Vector floating-point reciprocal square-root estimate

Example:

```
void frsqrt7 (float32_t *src1,
                 float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res = __riscv_vfrsqrt7_v_f32m1 (opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.7.2. Vector floating-point masked reciprocal square-root estimate

Example:

```
void frsqrt7_m (vbool32_t mask,
```

```
                    float32_t *src1,
                    float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t res = __riscv_vfrsqrt7_v_f32m1_m (mask, opd1, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void frsqrt7_mu (vbool32_t mask,
                 vfloat32m1_t maskedoff,
                 float32_t *src1,
                 float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfrsqrt7_v_f32m1_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.8. Vector floating-point reciprocal estimate to 7 bits instructions

The generalized form of the floating-point reciprocal estimate intrinsic functions is:

```
__riscv_vfrsqrt7_v_ SEW LMUL [_rm] [_ TM]
```

They are unary vector-vector instructions that return an estimate of 1/x accurate to 7 bits.

7.8.1. Vector floating-point reciprocal estimate to 7 bits

Example:

```
void frec7 (float32_t *src1,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res = __riscv_vfrec7_v_f32m1 (opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void frec7 (float32_t *src1,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfrec7_v_f32m1_rm (opd1, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.8.2. Vector floating-point masked reciprocal estimate to 7 bits

Example:

```
void frec7_m (vbool32_t mask,
              float32_t *src1,
              float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t res = __riscv_vfrec7_v_f32m1_m (mask, opd1, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
```

```
}
```

Example with mask undisturbed policy:

```
void frec7_mu (vbool32_t mask,
               vfloat32m1_t maskedoff,
               float32_t *src1,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfrec7_v_f32m1_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.9. Vector floating-point minimum and maximum instructions

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

The generalized form of the floating-point minimum and maximum intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where `OPDS` is one of:

`vv`	Operation takes two vectors from vector register groups.
`vf`	Operation takes one vector from vector register groups and a scalar floating-point register.

`OP` is defined in the following sections.

7.9.1. Vector floating-point minimum

Example:

```
void fmin (float32_t *src1,
           float32_t *src2,
           float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vfmin_vv_f32m1 (opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.9.2. Vector floating-point masked minimum

Example:

```
void fmin_m (vbool32_t mask,
             float32_t *src1,
             float32_t *src2,
             float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
```

```
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res = __riscv_vfmin_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fmin_mu (vbool32_t mask,
              vfloat32m1_t maskedoff,
              float32_t *src1,
              float32_t *src2,
              float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res =
    __riscv_vfmin_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


7.9.3. Vector floating-point maximum

Example:

```
void fmax (float32_t *src1,
           float32_t *src2,
           float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vfmax_vv_f32m1 (opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.9.4. Vector floating-point masked maximum

Example:

```
void fmax_m (vbool32_t mask,
             float32_t *src1,
             float32_t *src2,
             float32_t *dest)
{
```

```
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res = __riscv_vfmax_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void fmax_mu (vbool32_t mask,
              vfloat32m1_t maskedoff,
              float32_t *src1,
              float32_t *src2,
              float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res =
    __riscv_vfmax_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.10. Vector floating-point sign-injection instructions

Vector versions of the scalar sign-injection instructions. The result takes all bits except the sign bit from the first vector operand.

The generalized form of the floating-point sign-injection intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```


Where OPDS is one of:

vv	Operation takes two vectors from vector register groups
vf	Operation takes one vector from vector register groups and a scalar floating-point register

OP is defined in the following sections.

7.10.1. Vector vfsgnj instruction

Example:

```
void fsgnj (float32_t *src1,
            float32_t *src2,
            float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vfsgnj_vv_f32m1 (opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

7.10.2. Vector masked vfsgnj instruction

Example:

```
void fsgnj_m (vbool32_t mask,
              float32_t *src1,
              float32_t *src2,
              float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
```

```
  vfloat32m1_t res = __riscv_vfsgnj_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fsgnj_mu (vbool32_t mask,
               vfloat32m1_t maskedoff,
               float32_t *src1,
               float32_t *src2,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res =
    __riscv_vfsgnj_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


7.10.3. Vector vfsgnjn instruction

Example:

```
void fsgnjn (float32_t *src1,
             float32_t *src2,
             float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vfsgnjn_vv_f32m1 (opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

7.10.4. Vector masked vfsgnjn instruction

Example:

```
void fsgnjn_m (vbool32_t mask,
               float32_t *src1,
               float32_t *src2,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res = __riscv_vfsgnjn_vv_f32m1_m (mask, opd1, opd2,vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void fsgnjn_mu (vbool32_t mask,
               float32_t *src1,
               float32_t *src2,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res =
    __riscv_vfsgnjn_vv_f32m1_mu (mask, maskedoff, opd1, opd2,vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.10.5. Vector vfsgnjx instruction

Example:

```
void fsgnjx (float32_t *src1,
             float32_t *src2,
             float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vfsgnjx_vv_f32m1 (opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

7.10.6. Vector masked vfsgnjx instruction

Example:

```
void fsgnjx_m (vbool32_t mask,
               float32_t *src1,
               float32_t *src2,
               float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res = __riscv_vfsgnjx_vv_f32m1_m (mask, opd1, opd2,vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void fsgnjx_mu (vbool32_t mask,
                vfloat32m1_t maskedoff,
                float32_t *src1,
                float32_t *src2,
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res =
    __riscv_vfsgnjx_vv_f32m1_mu (mask, maskedoff, opd1, opd2,vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.11. Vector floating-point absolute value and negate instructions

The generalized form of the floating-point absolute value and negate intrinsic functions is:

```
__riscv_v OP _v_ SEW LMUL [_ TM]
```

They are vector versions of the scalar instructions.

`OP` is defined in the following sections.

7.11.1. Vector floating-point absolute value

Example:

```
void fabs (float32_t *src1,
           float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res = __riscv_vfabs_v_f32m1 (opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.11.2. Vector floating-point masked absolute value

Example:

```
void fabs_m (vbool32_t mask,
             float32_t *src1,
             float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t res = __riscv_vfabs_v_f32m1_m (mask, opd1, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fabs_mu (vbool32_t mask,
              vfloat32m1_t maskedoff,
              float32_t *src1,
              float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
```

```
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfabs_v_f32m1_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


7.11.3. Vector floating-point negate

Example:

```
void fneg (float32_t *src1,
           float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res = __riscv_vfneg_v_f32m1 (opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.11.4. vector floating-point masked negate

Example:

```
void fneg_m (vbool32_t mask,
             float32_t *src1,
             float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t res = __riscv_vfneg_v_f32m1_m (mask, opd1, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fneg_mu (vbool32_t mask,
              vfloat32m1_t maskedoff,
              float32_t *src1,
              float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfneg_v_f32m1_mu (mask, maskedoff, opd1, vl);
```

```
    __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.12. Vector floating-point compare instructions

These vector floating-point 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. The instructions compare 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.

The generalized form of the floating-point compare intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL _ BTYPE [_m | _mu]
```

Where **OPDS** is one of:

vv	Operation takes two vectors from vector register groups.
vf	Operation takes one vector from vector register groups and a floating-point register.

And **BTYPE** is the `boolean` return type and is one of:

b8	`vbool8_t` return type
b16	`vbool16_t` return type
b32	`vbool32_t` return type
b64	`vbool64_t` return type
b1	`vbool1_t` return type
b2	`vbool2_t` return type
b4	`vbool4_t` return type
b8	`vbool8_t` return type

OP is defined in the following sections.

7.12.1. Vector floating-point compare equal

Example:

```
vbool32_t mfeq (float32_t *src1,
                float32_t *src2,
                float32_t *dest)
```

```
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vbool32_t res = __riscv_vmfeq_vv_f32m1_b32 (opd1, opd2, vl);
  return res;
}
```


7.12.2. Vector floating-point masked compare equal

Example:

```
vbool32_t mfeq_m (vbool32_t mask,
                  float32_t *src1,
                  float32_t *src2,
                  float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vbool32_t res = __riscv_vmfeq_vv_f32m1_b32_m (mask, opd1, opd2, vl);
  return res;
}
```

Example with mask undisturbed policy:

```
vbool32_t mfeq_mu (vbool32_t mask,
                   float32_t *src1,
                   float32_t *src2,
                   float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vbool32_t res =
    __riscv_vmfeq_vv_f32m1_b32_mu (mask, maskedoff, opd1, opd2, vl);
  return res;
}
```

7.12.3. Vector floating-point compare not equal

Example:

```
vbool32_t mfne (float32_t *src1,
                float32_t *src2,
```

```
                    float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vbool32_t res = __riscv_vmfne_vv_f32m1_b32 (opd1, opd2, vl);
  return res;
}
```


7.12.4. Vector floating-point masked compare not equal

Example:

```
vbool32_t mfne_m (vbool32_t mask,
                  float32_t *src1,
                  float32_t *src2,
                  float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vbool32_t res = __riscv_vmfne_vv_f32m1_b32_m (mask, opd1, opd2, vl);
  return res;
}
```

Example with mask undisturbed policy:

```
vbool32_t mfne_mu (vbool32_t mask,
                   float32_t *src1,
                   float32_t *src2,
                   float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vbool32_t res =
    __riscv_vmfne_vv_f32m1_b32_m u(mask, maskedoff, opd1, opd2, vl);
  return res;
}
```

7.12.5. Vector floating-point compare less than

Example:

```
vbool32_t mflt (float32_t *src1,
```

```
                    float32_t *src2,
                    float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vbool32_t res = __riscv_vmflt_vv_f32m1_b32 (opd1, opd2, vl);
  return res;
}
```


7.12.6. Vector floating-point masked compare less than

Example:

```
vbool32_t mflt_m (vbool32_t mask,
                  float32_t *src1,
                  float32_t *src2,
                  float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vbool32_t res = __riscv_vmflt_vv_f32m1_b32_m (mask, opd1, opd2, vl);
  return res;
}
```

Example with mask undisturbed policy:

```
vbool32_t mflt_mu (vbool32_t mask,
                   vbool32_t maskedoff,
                   float32_t *src1,
                   float32_t *src2,
                   float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vbool32_t res =
    __riscv_vmflt_vv_f32m1_b32_mu (mask, maskedoff, opd1, opd2, vl);
  return res;
}
```

7.12.7. Vector floating-point compare less than or equal

Example:

```
vbool32_t mfle (float32_t *src1,
                float32_t *src2,
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vbool32_t res = __riscv_vmfle_vv_f32m1_b32 (opd1, opd2, vl);
  return res;
}
```


7.12.8. Vector floating-point masked compare less than or equal

Example:

```
vbool32_t mfle_m (vbool32_t mask,
                  float32_t *src1,
                  float32_t *src2,
                  float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vbool32_t res = __riscv_vmfle_vv_f32m1_b32_m (mask, opd1, opd2, vl);
  return res;
}
```

Example with mask undisturbed policy:

```
vbool32_t mfle_mu (vbool32_t mask,
                   vbool32_t maskedoff,
                   float32_t *src1,
                   float32_t *src2,
                   float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vbool32_t res =
    __riscv_vmfle_vv_f32m1_b32_mu (mask, maskedoff, opd1, opd2, vl);
  return res;
```

```
}
```

7.12.9. Vector floating-point compare greater than

Example:

```
vbool32_t mfgt (float32_t *src1,
                float32_t *src2,
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vbool32_t res = __riscv_vmfgt_vv_f32m1_b32 (opd1, opd2, vl);
  return res;
}
```


7.12.10. Vector floating-point masked compare greater than

Example:

```
vbool32_t mfgt_m (vbool32_t mask,
                  float32_t *src1,
                  float32_t *src2,
                  float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vbool32_t res = __riscv_vmfgt_vv_f32m1_b32_m (mask, opd1, opd2, vl);
  return res;
}
```

Example with mask undisturbed policy:

```
vbool32_t mfgt_mu (vbool32_t mask,
                   vbool32_t maskedoff,
                   float32_t *src1,
                   float32_t *src2,
                   float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vbool32_t res =
```

```
    __riscv_vmfgt_vv_f32m1_b32_mu (mask, maskedoff, opd1, opd2, vl);
  return res;
}
```

7.12.11. vector floating-point compare greater than or equal

Example:


```
vbool32_t mfge (float32_t *src1,
                float32_t *src2,
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vbool32_t res = __riscv_vmfge_vv_f32m1_b32 (opd1, opd2, vl);
  return res;
}
```

7.12.12. vector floating-point masked compare greater than or equal

Example:

```
vbool32_t mfge_m (vbool32_t mask,
                  float32_t *src1,
                  float32_t *src2,
                  float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vbool32_t res = __riscv_vmfge_vv_f32m1_b32_m (mask, opd1, opd2, vl);
  return res;
}
```

Example with mask undisturbed policy:

```
vbool32_t mfge_mu (vbool32_t mask,
                   vbool32_t maskedoff,
                   float32_t *src1,
                   float32_t *src2,
                   float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
```

```
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vbool32_t res =
    __riscv_vmfge_vv_f32m1_b32_mu (mask, maskedoff, opd1, opd2, vl);
  return res;
}
```

7.13. Vector floating-point classify instructions

These are unary vector-vector instructions that operate in the same way as scalar classify instructions.

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.

The generalized form of the floating-point classify intrinsic functions is:

```
__riscv_vfclass_v_ SEW LMUL [_ TM]
```

7.13.1. Vector floating-point classify

Example:

```
void fclass (float32_t *src1,
             uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vuint32m1_t res = __riscv_vfclass_v_u32m1 (opd1, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

7.13.2. Vector floating-point masked classify

Example:

```
void fclass_m (vbool32_t mask,
               float32_t *src1,
               uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vuint32m1_t res =
  __riscv_vfclass_v_u32m1_m (opd1, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fclass_mu (vbool32_t mask,
```

```
                vuint32m1 maskedoff,
                float32_t *src1,
                uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vuint32m1_t res =
   __riscv_vfclass_v_u32m1_mu (mask, maskedoff, opd1, vl);
   __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

7.14. Vector floating-point merge instructions

Vector-scalar floating-point merge instructions are provided, which operate on all body elements from `vstart` up to the current vector length regardless of the mask value.

The `vfmerge.vfm` instruction is encoded as a masked instruction. 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.

The generalized form of the floating-point merge intrinsic functions is:

```
__riscv_vfmerge _ OPDS _ SEW LMUL [_tu]
```

Where **OPDS** is one of:

vvm	Operation takes two vectors from vector register groups and one mask vector.
vfm	Operation takes one vector from vector register groups, a scalar floating-point register, and a mask vector.

There are no masked forms of these instructions.

7.14.1. Vector floating-point merge

Example:

```
void fmerge (float32_t *src1,
             float32_t *src2,
             vbool32_t mask,
             float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vfmerge_vvm_f32m1 (opd1, opd2, mask, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

7.14.2. Vector floating-point move instructions

There are two forms to these instructions. The first is a copy from a vector register group to vector register group. The second splats a floating-point scalar operand to a vector register group. The instructions copy a scalar f register value to all active elements of a vector register

group.

The generalized form of the floating-point compare intrinsic functions is:

```
__riscv_v OP _v_ OPDS _ SEW LMUL [_tu]
```

Where **OPDS** is one of:

v	Operation takes one vector from vector register groups.
f	floating-point scalar value

OP is defined in the following sections.

7.14.3. Vector floating-point vector-vector copy

Example:

```
void mv (float32_t *src1,
         float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res = __riscv_vmv_v_v_f32m1 (opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.14.4. Vector floating-point splat

Example:

```
void fmv (float opd1,
          float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t res = __riscv_vfmv_v_f_f32m1 (opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.15. Vector single-width floating-point/integer type conversion 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.

The generalized form of the floating-point/integer conversion intrinsic functions is:

```
__riscv_v OP _v_ OPDS _ SEW LMUL [_rm] [_ TM]
```

Where **OPDS** is one of:

xu_f_v	Operation takes one vector from vector register groups and converts float to unsigned integer.
x_f_v	Operation takes one vector from vector register groups and converts float to signed integer.
f_xu_v	Operation takes one vector from vector register groups and converts unsigned integer to float.
f_x_v	Operation takes one vector from vector register groups and converts signed integer to float.

OP is defined in the following sections.

7.15.1. Vector convert float to unsigned integer

Example:

```
void fcvt_xu_f (float32_t *src1,
                uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vuint32m1_t res = __riscv_vfcvt_xu_f_v_u32m1 (opd1, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void fcvt_xu_f (float32_t *src1,
                uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
```

```
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vuint32m1_t res =
    __riscv_vfcvt_xu_f_v_u32m1_rm (opd1, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```


7.15.2. Vector masked convert float to unsigned integer

Example:

```
void fcvt_xu_f_m (vbool32_t mask,
                  float32_t *src1,
                  uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vuint32m1_t res = __riscv_vfcvt_xu_f_v_u32m1_m (mask, opd1, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fcvt_xu_f_mu (vbool32_t mask,
                   vuint32m1_t maskedoff,
                   float32_t *src1,
                   uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vuint32m1_t res =
    __riscv_vfcvt_xu_f_v_u32m1_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

7.15.3. Vector convert float to signed integer

Example:

```
void fcvt_x_f (float32_t *src1,
               int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vint32m1_t res = __riscv_vfcvt_x_f_v_i32m1 (opd1, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
```

```
}
```

Example with rounding mode:

```
void fcvt_x_f (float32_t *src1,
               int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vint32m1_t res =
    __riscv_vfcvt_x_f_v_i32m1_rm (opd1, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


7.15.4. Vector masked convert float to signed integer

Example:

```
void fcvt_x_f_m (vbool32_t mask,
                 float32_t *src1,
                 int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vint32m1_t res = __riscv_vfcvt_x_f_v_i32m1_m (mask, opd1, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fcvt_x_f_mu (vbool32_t mask,
                  vint32m1_t maskedoff,
                  float32_t *src1,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vint32m1_t res =
    __riscv_vfcvt_x_f_v_i32m1_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

7.15.5. Vector convert float to unsigned integer. Truncating

Example:

```
void fcvt_rtz_xu_f (float32_t *src1,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vuint32m1_t res = __riscv_vfcvt_rtz_xu_f_v_u32m1 (opd1, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```


7.15.6. Vector masked convert float to unsigned integer. Truncating

Example:

```
void fcvt_rtz_xu_f_m (vbool32_t mask,
                      float32_t *src1,
                      uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vuint32m1_t res = __riscv_vfcvt_rtz_xu_f_v_u32m1_m (mask, opd1, vl);
__riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fcvt_rtz_xu_f_mu (vbool32_t mask,
                       vuint32m1_t maskedoff,
                       float32_t *src1,
                       uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vuint32m1_t res =
    __riscv_vfcvt_rtz_xu_f_v_u32m1_mu (mask, maskedoff, opd1, vl);
__riscv_vse32_v_u32m1 (dest, res, vl);
```

7.15.7. Vector convert float to signed integer, truncating

Example:

```
void fcvt_rtz_x_f (float32_t *src1,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
```

```
  vint32m1_t res = __riscv_vfcvt_rtz_x_f_v_i32m1 (opd1, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

7.15.8. Vector masked convert float to signed integer, truncating

Example:

```
void fcvt_rtz_x_f_m (vbool32_t mask,
                     float32_t *src1,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vint32m1_t res = __riscv_vfcvt_rtz_x_f_v_i32m1_m (mask, opd1, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fcvt_rtz_x_f_mu (vbool32_t mask,
                      vint32m1_t maskedoff,
                      float32_t *src1,
                      int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vint32m1_t res =
    __riscv_vfcvt_rtz_x_f_v_i32m1_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

7.15.9. Vector convert unsigned integer to float

Example:

```
void fcvt_f_xu (uint32_t *src1,
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vfloat32m1_t res = __riscv_vfcvt_f_xu_v_f32m1 (opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void fcvt_f_xu (uint32_t *src1,
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfcvt_f_xu_v_f32m1_rm (opd1, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


7.15.10. Vector masked convert unsigned integer to float

Example:

```
void fcvt_f_xu_m (vbool32_t mask,
                  uint32_t *src1,
                  float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vfloat32m1_t res = __riscv_vfcvt_f_xu_v_f32m1_m (mask, opd1, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fcvt_f_xu_mu (vbool32_t mask,
                   vfloat32m1_t maskedoff,
                   uint32_t *src1,
                   float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfcvt_f_xu_v_f32m1_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.15.11. Vector convert signed integer to float

Example:

```
void fcvt_f_x (int32_t *src1,
```

```
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vfloat32m1_t res = __riscv_vfcvt_f_x_v_f32m1 (opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


Example with rounding mode:

```
void fcvt_f_x (int32_t *src1,
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfcvt_f_x_v_f32m1_rm (opd1, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.15.12. Vector masked convert signed integer to float

Example:

```
void fcvt_f_x_m (bool32_t mask,
                int32_t *src1,
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vfloat32m1_t res = __riscv_vfcvt_f_x_v_f32m1_m (mask, opd1, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fcvt_f_x_mu (bool32_t mask,
                vfloat32m1_t maskedoff,
                int32_t *src1,
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfcvt_f_x_v_f32m1_mu (mask, maskedoff, opd1, vl);
```

```
    __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.16. Vector widening floating-point/integer type conversion instructions

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

The generalized form of the widening floating-point/integer conversion intrinsic functions is:

```
__riscv_v OP _v_ OPDS _ SEW LMUL [_rm] [_ TM]
```

Where `OPDS` is one of:

`xu_f_v`	Operation takes one vector from vector register groups and converts float to double-width unsigned integer.
`x_f_v`	Operation takes one vector from vector register groups and converts float to double-width signed integer.
`f_xu_v`	Operation takes one vector from vector register groups and converts unsigned integer to double-width float.
`f_x_v`	Operation takes one vector from vector register groups and converts signed integer to double-width float.
`f_f_v`	Operation takes one vector from vector register groups and converts single-width float to double-width float.
`x_x_v`	Operation takes one vector from vector register groups and converts single-width integer to double-width integer.

`OP` is defined in the following sections.

Rounding mode variants are only available for floating-point to integer conversions. In addition, the truncating forms of these conversions already incorporate the rounding mode in their names.

7.16.1. Vector convert single-width unsigned integer to double-width unsigned integer

Example:

```
void wcvtu_x_x (uint32_t *src1,
                uint64_t *dest)
```

```
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint64m2_t res = __riscv_vwcvtu_x_x_v_u64m2 (opd1, vl);
  __riscv_vse64_v_u64m2 (dest, res, vl);
}
```


7.16.2. Vector masked convert single-width unsigned integer to double-width unsigned integer

Example:

```
void wcvtu_x_x_m (vbool32_t mask,
                  uint32_t *src1,
                  uint64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint64m2_t res = __riscv_vwcvtu_x_x_v_u64m2_m (mask, src, vl);
  __riscv_vse64_v_u64m2_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void wcvtu_x_x_mu (vbool32_t mask,
                   vuint64m2_t maskedoff,
                   uint32_t *src1,
                   uint64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint64m2_t res =
    __riscv_vwcvtu_x_x_v_u64m2_mu (mask, maskedoff, src, vl);
  __riscv_vse64_v_u64m2 (dest, res, vl);
}
```

7.16.3. Vector convert single-width signed integer to double-width signed integer

Example:

```
void fwcvt_x_x (int32_t *src1,
                int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint64m2_t res = __riscv_vwcvt_x_x_v_i64m2 (opd1, vl);
  __riscv_vse64_v_i64m2 (dest, res, vl);
}
```


7.16.4. Vector masked convert single-width signed integer to double-width signed integer

Example:

```
void fwcvt_x_x_m (vbool32_t mask,
                  int32_t *src1,
                  int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint64m2_t res = __riscv_vwcvt_x_x_v_i64m2_m (mask, opd1, vl);
  __riscv_vse64_v_i64m2_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fwcvt_x_x_mu (vbool32_t mask,
                   vint64m2_t maskedoff,
                   int32_t *src1,
                   int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint64m2_t res =
    __riscv_vwcvt_x_x_v_i64m2_mu (mask, maskedoff, opd1, vl);
  __riscv_vse64_v_i64m2 (dest, res, vl);
}
```

7.16.5. Vector convert single-width float to double-width unsigned integer

Example:

```
void fwcvt_xu_f (float32_t *src1,
                 uint64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vuint64m2_t res = __riscv_vfwcvt_xu_f_v_u64m2 (opd1, vl);
  __riscv_vse64_v_u64m2 (dest, res, vl);
}
```


Example with rounding mode:

```
void fwcvt_xu_f (float32_t *src1,
                 uint64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vuint64m2_t res =
    __riscv_vfwcvt_xu_f_v_u64m2_rm (opd1, __RISCV_FRM_RNE, vl);
  __riscv_vse64_v_u64m2 (dest, res, vl);
}
```

7.16.6. Vector masked convert single-width float to double-width unsigned integer

Example:

```
void  fwcvt_xu_f_m (vbool32_t mask,
                    float32_t *src1,
                    uint64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vuint64m2_t res = __riscv_vfwcvt_xu_f_v_u64m2_m (mask, opd1, vl);
  __riscv_vse64_v_u64m2_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void  fwcvt_xu_f_mu (vbool32_t mask,
                     vuint64m2_t maskedoff,
                     float32_t *src1,
                     uint64_t *dest)
{
```

```
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vuint64m2_t res =
    __riscv_vfwcvt_xu_f_v_u64m2_mu (mask, maskedoff, opd1, vl);
  __riscv_vse64_v_u64m2 (dest, res, vl);
}
```


7.16.7. Vector widening convert single-width float to double-width signed integer

Example:

```
void fwcvt_x_f (float32_t *src1,
                int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vint64m2_t res = __riscv_vfwcvt_x_f_v_i64m2 (opd1, vl);
  __riscv_vse64_v_i64m2 (dest, res, vl);
}
```

Example with rounding mode:

```
void fwcvt_x_f (float32_t *src1,
                int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vint64m2_t res =
    __riscv_vfwcvt_x_f_v_i64m2_rm (opd1, __RISCV_FRM_RNE, vl);
  __riscv_vse64_v_i64m2 (dest, res, vl);
}
```

7.16.8. Vector masked widening convert single-width float to double-width signed integer

Example:

```
void fwcvt_x_f_m (vbool32_t mask,
                  float32_t *src1,
                  int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vint64m2_t res = __riscv_vfwcvt_x_f_v_i64m2_m (mask, opd1, vl);
```

```
  __riscv_vse64_v_i64m2_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fwcvt_x_f_mu (vbool32_t mask,
                   vint64m2_t maskedoff,
                   float32_t *src1,
                   int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vint64m2_t res =
    __riscv_vfwcvt_x_f_v_i64m2_mu (mask, maskedoff, opd1, vl);
  __riscv_vse64_v_i64m2 (dest, res, vl);
}
```


7.16.9. Vector convert single-width float to double-width unsigned integer, truncating

Example:

```
void fwcvt_rtz_xu_f (float32_t *src1,
                     uint64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vuint64m2_t res = __riscv_vfwcvt_rtz_xu_f_v_u64m2 (opd1, vl);
  __riscv_vse64_v_u64m2 (dest, res, vl);
}
```

7.16.10. Vector masked convert single-width float to double-width unsigned integer, truncating

Example:

```
void fwcvt_xu_f_m (vbool32_t mask,
                   float32_t *src1,
                   uint64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vuint64m2_t res = __riscv_vfwcvt_rtz_xu_f_v_u64m2_m (mask, opd1,vl);
  __riscv_vse64_v_u64m2_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void fwcvt_xu_f_mu (vbool32_t mask,
                    vuint64m2_t maskedoff,
                    float32_t *src1,
                    uint64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vuint64m2_t res =
    __riscv_vfwcvt_rtz_xu_f_v_u64m2_mu (mask, maskedoff, opd1,vl);
  __riscv_vse64_v_u64m2 (dest, res, vl);
}
```

7.16.11. Vector convert single-width float to double-width signed integer, truncating

Example:

```
void fwcvt_rtz_x_f (float32_t *src1,
                    int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vint64m2_t res = __riscv_vfwcvt_rtz_x_f_v_i64m2 (opd1, vl);
  __riscv_vse64_v_i64m2 (dest, res, vl);
}
```

7.16.12. Vector masked convert single-width float to double-width signed integer, truncating

Example:

```
void fwcvt_rtz_x_f_m (vbool32_t mask,
                      float32_t *src1,
                      int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vint64m2_t res = __riscv_vfwcvt_rtz_x_f_v_i64m2_m (mask, opd1, vl);
  __riscv_vse64_v_i64m2_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void fwcvt_rtz_x_f_mu (vbool32_t mask,
                       vint64m2_t maskedoff,
                       float32_t *src1,
                       int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vint64m2_t res =
    __riscv_vfwcvt_rtz_x_f_v_i64m2_mu (mask, maskedoff, opd1, vl);
  __riscv_vse64_v_i64m2 (dest, res, vl);
}
```

7.16.13. Vector convert single-width unsigned integer to double-width float

Example:

```
void fcvt_f_xu (uint32_t *src1,
                float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vfloat64m2_t res = __riscv_vfwcvt_f_xu_v_f64m2 (opd1, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.16.14. Vector masked convert single-width unsigned integer to double-width float

Example:

```
void fcvt_f_xu_m (vbool32_t mask,
                  uint32_t *src1,
                  float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vfloat64m2_t res = __riscv_vfwcvt_f_xu_v_f64m2_m (mask, opd1, vl);
  __riscv_vse64_v_f64m2_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void fcvt_f_xu_mu (vbool32_t mask,
                   vfloat64m2_t maskedoff,
                   uint32_t *src1,
                   float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vfloat64m2_t res =
    __riscv_vfwcvt_f_xu_v_f64m2_mu (mask, maskedoff, opd1, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.16.15. Vector convert single-width signed integer to double-width float

Example:

```
void fcvt_f_x (int32_t *src1,
               float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vfloat64m2_t res = __riscv_vfwcvt_f_x_v_f64m2 (opd1, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.16.16. Vector masked convert single-width signed integer to double-width float

Example:

```
void fcvt_f_x_m (vbool32_t mask,
                 int32_t *src1,
                 float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vfloat64m2_t res = __riscv_vfwcvt_f_x_v_f64m2_m (mask, opd1, vl);
  __riscv_vse64_v_f64m2_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void fcvt_f_x_mu (vbool32_t mask,
                  vfloat64m2_t maskedoff,
                  int32_t *src1,
                  float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vfloat64m2_t res =
    __riscv_vfwcvt_f_x_v_f64m2_mu (mask, maskedoff, opd1, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.16.17. Convert single-width float to double-width float

Example:

```
void wcvt_f_f (float32_t *src1.
               float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat64m2_t res = __riscv_vfwcvt_f_f_v_f64m2 (opd1, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.16.18. Convert masked single-width float to double-width float

Example:

```
void wcvt_f_f_m (vbool32_t mask,
                 float32_t *src1.
                 float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat64m2_t res = __riscv_vfwcvt_f_f_v_f64m2_m (mask, opd1, vl);
  __riscv_vse64_v_f64m2_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void wcvt_f_f_mu (vbool32_t mask,
                  vfloat64m2_t maskedoff,
                  float32_t *src1.
                  float64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat64m2_t res =
    __riscv_vfwcvt_f_f_v_f64m2_mu (mask, maskedoff, opd1, vl);
  __riscv_vse64_v_f64m2 (dest, res, vl);
}
```

7.16.19. Convert bfloat16 to single-width float

Example:

```
void wcvt_f_f (__bf16 *src1,
               Float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1 ();
  vint16m1_t opd0 = __riscv_vle16_v_i16m1 ((short *) src1, vl);
  vbfloat16_t opd1 = __riscv_vreinterpret_v_i16m1_bf16m1 (opd0);
  vfloat32m2_t res = __riscv_vfwcvtbf16_f_f_v_f32m2 (opd1, vl);
  __riscv_vse32_v_f32m2 (dest, res, vl);
}
```

7.16.20. Convert masked bfloat16 to single-width float

Example:

```
void wcvt_f_f_m (vbool16_t mask,
                 __bf16 *src1,
                 Float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1 ();
  vint16m1_t opd0 = __riscv_vle16_v_i16m1_m (mask, (short *) src1,
vl);
  vbfloat16m1_t opd1 = __riscv_vreinterpret_v_i16m1_bf16m1 (opd0);
  vfloat32m2_t res = __riscv_vfwcvtbf16_f_f_v_f32m2_m (mask, opd1,
vl);
  __riscv_vse32_v_f32m2_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void wcvt_f_f_mu (vbool16_t mask,
                  vfloat32m2_t maskedoff,
                  __bf16 *src1,
                  Float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1 ();
  vint16m1_t opd0 = __riscv_vle16_v_i16m1 ((short *) src1, vl);
  vbfloat16m1_t opd1 = __riscv_vreinterpret_v_i16m1_bf16m1 (opd0);
  vfloat32m2_t res =
    __riscv_vfwcvtbf16_f_f_v_f32m2_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_f32m2 (dest, res, vl);
}
```

7.17. Vector 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.

The generalized form of the narrowing floating-point/integer convert intrinsic functions is:

```
__riscv_v OP _v_ OPDS _ SEW LMUL [_rm] [_ TM]
```


Where OPDS is one of:

xu_f_w	Operation takes one vector from vector register groups and converts double-width float to single-width unsigned integer.
x_f_w	Operation takes one vector from vector register groups and converts double-width float to single-width signed integer.
f_xu_w	Operation takes one vector from vector register groups and converts double-width unsigned integer to single-width float.
f_x_w	Operation takes one vector from vector register groups and converts double-width signed integer to single-width float.
f_f_w	Operation takes one vector from vector register groups and converts double-width float to single-width float.
x_x_w	Operation takes one vector from vector register groups and converts double-width integer to single-width integer.

OP is defined in the following sections.

7.17.1. Vector convert double-width unsigned integer to single-width unsigned integer

Example:

```
void vncvtu_x_x (uint64_t *src1,
                 uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vuint64m2_t opd1 = __riscv_vle64_v_u64m2 (src1, vl);
  vuint32m1_t res = __riscv_vncvt_x_x_w_u32m1 (opd1, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

7.17.2. Vector masked convert double-width unsigned integer to single-width unsigned integer

Example:

```
void vncvtu_x_x_m (vbool32_t mask,
                   uint64_t *src1,
                   uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vuint64m2_t opd1 = __riscv_vle64_v_u64m2_m (mask, src1, vl);
  vuint32m1_t res = __riscv_vncvt_x_x_w_u32m1_m (mask, opd1, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vncvtu_x_x_mu (vbool32_t mask,
                    vuint32m1_t maskedoff,
                    uint64_t *src1,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vuint64m2_t opd1 = __riscv_vle64_v_u64m2 (src1, vl);
  vuint32m1_t res =
    __riscv_vncvt_x_x_w_u32m1_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

7.17.3. Vector convert double-width signed integer to single-width signed integer

Example:

```
void vncvt_x_x (int64_t *src1,
                int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vint64m2_t opd1 = __riscv_vle64_v_i64m2 (src1, vl);
  vint32m1_t res = __riscv_vncvt_x_x_w_i32m1 (opd1, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

7.17.4. Vector masked convert double-width signed integer to single-width signed integer

Example:

```
void vncvt_x_x_m (vbool32_t mask,
                  int64_t *src1,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vint64m2_t opd1 = __riscv_vle64_v_i64m2_m (mask, src1, vl);
  vint32m1_t res = __riscv_vncvt_x_x_w_i32m1_m (mask, opd1 vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vncvt_x_x_mu (vbool32_t mask,
                   vint32m1_t maskedoff,
                   int64_t *src1,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vint64m2_t opd1 = __riscv_vle64_v_i64m2 (src1, vl);
  vint32m1_t res =
    __riscv_vncvt_x_x_w_i32m1_mu (mask, maskedoff, opd1 vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

7.17.5. Vector convert double-width float to single-width unsigned integer

Example:

```
void fncvt_xu_f (float64_t *src1,
                 uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vuint32m1_t res = __riscv_vfncvt_xu_f_w_u32m1 (opd, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void fncvt_xu_f (float64_t *src1,
                 uint32_t *dest)
```

```
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vuint32m1_t res =
    __riscv_vfncvt_xu_f_w_u32m1_rm (opd, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```


7.17.6. Vector masked convert double-width float to single-width unsigned integer

Example:

```
void fncvt_xu_f_m (vbool32_t mask,
                   float64_t *src1,
                   uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2_m (mask, src1, vl);
  vuint32m1_t res = __riscv_vfncvt_xu_f_w_u32m1_m (mask, opd, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fncvt_xu_f_mu (vbool32_t mask,
                    vuint32m1_t maskedoff,
                    float64_t *src1,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vuint32m1_t res =
    __riscv_vfncvt_xu_f_w_u32m1_mu (mask, maskedoff, opd, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

7.17.7. Vector convert double-width float to single-width signed integer

Example:

```
void fncvt_x_f (float64_t *src1,
                int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
```

```
  vint32m1_t res = __riscv_vfncvt_x_f_w_i32m1 (opd1, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void fncvt_x_f (float64_t *src1,
                int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vint32m1_t res =
    __riscv_vfncvt_x_f_w_i32m1_rm (opd1, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


7.17.8. Vector masked convert double-width float to single-width signed integer

Example:

```
void fncvt_x_f_m (vbool32_t mask,
                  float64_t *src1,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2_m (mask, src1, vl);
  vint32m1_t res = __riscv_vfncvt_x_f_w_i32m1_m (mask, iod1, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fncvt_x_f_mu (vbool32_t mask,
                   vint32m1_t maskedoff,
                   float64_t *src1,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vint32m1_t res =
    __riscv_vfncvt_x_f_w_i32m1_mu (mask, maskedoff, iod1, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

7.17.9. Vector convert double-width float to single-width unsigned integer, truncating

Example:

```
void fncvt_rtz_xu_f (float64_t *src1,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vuint32m1_t res = __riscv_vfncvt_rtz_xu_f_w_u32m1 (opd1, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```


7.17.10. Vector masked convert double-width float to single-width unsigned integer, truncating

Example:

```
void fncvt_rtz_xu_f_m (vbool32_t mask,
                       float64_t *src1,
                       uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2_m (mask, src1, vl);
  vuint32m1_t res = __riscv_vfncvt_rtz_xu_f_w_u32m1_m (mask, opd1,vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fncvt_rtz_xu_f_mu (vbool32_t mask,
                        vuint32m1_t maskedoff,
                        float64_t *src1,
                        uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vuint32m1_t res =
    __riscv_vfncvt_rtz_xu_f_w_u32m1_mu (mask, maskedoff, opd1,vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

7.17.11. Vector convert double-width float to single-width signed integer, truncating

Example:

```
void fncvt_rtz_x_f (float64_t *src1,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vint32m1_t res = __riscv_vfncvt_rtz_x_f_w_i32m1 (opd1, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


7.17.12. Vector masked convert double-width float to single-width signed integer, truncating

Example:

```
void fncvt_rtz_x_f_m (vbool32_t mask,
                      float64_t *src1,
                      int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2_m (mask, src1, vl);
  vint32m1_t res = __riscv_vfncvt_rtz_x_f_w_i32m1_m (mask, opd1, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fncvt_rtz_x_f_mu (vbool32_t mask,
                       vint32m1_t maskedoff,
                       float64_t *src1,
                       int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vint32m1_t res =
    __riscv_vfncvt_rtz_x_f_w_i32m1_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

7.17.13. Vector convert double-width unsigned integer to single-width float

Example:

```
void fncvt_f_xu (uint64_t *src1,
                 float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vuint64m2_t opd1 = __riscv_vle64_v_u64m2 (src1, vl);
  vfloat32m1_t res = __riscv_vfncvt_f_xu_w_f32m1 (opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


Example with rounding mode:

```
void fncvt_f_xu (uint64_t *src1,
                 float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vuint64m2_t opd1 = __riscv_vle64_v_u64m2 (src1, vl);
  vfloat32m1_t res =
      __riscv_vfncvt_f_xu_w_f32m1_rm (opd1, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.17.14. Vector masked convert double-width unsigned integer to single-width float

Example:

```
void fncvt_f_xu_m (vbool32_t mask,
                   uint64_t *src1,
                   float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vuint64m2_t opd1 = __riscv_vle64_v_u64m2_m (mask, src1, vl);
  vfloat32m1_t res = __riscv_vfncvt_f_xu_w_f32m1_m (mask, opd1, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fncvt_f_xu_mu (vbool32_t mask,
                    vfloat32m1_t maskedoff,
                    uint64_t *src1,
                    float32_t *dest)
```

```
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vuint64m2_t opd1 = __riscv_vle64_v_u64m2 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfncvt_f_xu_w_f32m1_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


7.17.15. Vector convert double-width signed integer to single-width float

Example:

```
void fncvt_f_x (int64_t *src1,
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vint64m2_t opd1 = __riscv_vle64_v_i64m2 (src1, vl);
  vfloat32m1_t res = __riscv_vfncvt_f_x_w_f32m1 (opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void fncvt_f_x (int64_t *src1,
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vint64m2_t opd1 = __riscv_vle64_v_i64m2 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfncvt_f_x_w_f32m1_rm (opd1, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.17.16. Vector masked convert double-width signed integer to single-width float

Example:

```
void fncvt_f_x_m (vbool32_t mask,
                  int64_t *src1,
                  float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vint64m2_t opd1 = __riscv_vle64_v_i64m2_m (mask, src1, vl);
```

```
  vfloat32m1_t res = __riscv_vfncvt_f_x_w_f32m1_m (mask, opd1, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fncvt_f_x_mu (vbool32_t mask,
                   vfloat32m1_t maskedoff,
                   int64_t *src1,
                   float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vint64m2_t opd1 = __riscv_vle64_v_i64m2 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfncvt_f_x_w_f32m1_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


7.17.17. Convert double-width float to single-width float

Example:

```
void fncvt_f_f (float64_t *src1,
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vfloat32m1_t res = __riscv_vfncvt_f_f_w_f32m1 (opd, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

Example with rounding mode:

```
void fncvt_f_f (float64_t *src1,
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfncvt_f_f_w_f32m1_rm (opd, __RISCV_FRM_RNE, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.17.18. Convert masked double-width float to single-width float

Example:

```
void fncvt_f_f_m (vbool32_t mask,
                  float64_t *src1,
                  float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2_m (mask, src1, vl);
  vfloat32m1_t res = __riscv_vfncvt_f_f_w_f32m1_m (mask, opd1, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void fncvt_f_f_mu (vbool32_t mask,
                   vfloat32m1_t maskedoff,
                   float64_t *src1,
                   float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfncvt_f_f_w_f32m1_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.17.19. Convert single-width float to bfloat16

Example:

```
void fncvt_f_f (float32_t *src1,
                __bf16 *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m2 ();
  vfloat32m2_t opd1 = __riscv_vle64_v_f32m2 (src1, vl);
  vbfloat16m1_t res1 = __riscv_vfncvtbf16_f_f_w_bf16m1 (opd1, vl);
  vint16m1_t res = __riscv_vreinterpret_v_bf16m1_i16m1 (res1);
  __riscv_vse16_v_i16m1 ((short*) dest, res, vl);
}
```

7.17.20. Convert masked single-width float to bfloat16

Example:

```
void fncvt_f_f_m (vbool16_t mask,
                  float32_t *src1,
                  __bf16 *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m2 ();
  vfloat32m2_t opd1 = __riscv_vle32_v_f32m2_m (mask, src1, vl);
  vbfloat16m1_t res1 = __riscv_vfncvtbf16_f_f_w_bf16m1_m (mask, opd1,
vl);
  __riscv_vse16_v_i16m1_m (mask, (short*)dest, res, vl);
}
```


7.17.21. Convert double-width float to single-width float – round towards odd

Example:

```
void fncvt_rod_f_f (float64_t *src1,
                    float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vfloat32m1_t res = __riscv_vfncvt_rod_f_f_w_f32m1 (opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

7.17.22. Convert masked double-width float to single-width float – round towards odd

Example:

```
void fncvt_rod_f_f_m (vbool32_t mask,
                      float64_t *src1,
                      float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2_m (mask, src1, vl);
  vfloat32m1_t res =__riscv_vfncvt_rod_f_f_w_f32m1_m (mask, opd1, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void fncvt_rod_f_f_mu (vbool32_t mask,
                       vfloat32m1_t maskedoff,
                       float64_t *src1,
                       float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e64m2 ();
  vfloat64m2_t opd1 = __riscv_vle64_v_f64m2 (src1, vl);
  vfloat32m1_t res =
    __riscv_vfncvt_rod_f_f_w_f32m1_mu (mask, maskedoff, opd1, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

8. 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.

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

The other elements in the destination vector register (`0 < index < VLEN/SEW`) are considered the tail and are managed with the current tail agnostic/undisturbed policy.

8.1. Vector single-width integer reduction instructions

The generalized form of the single-width integer reduction intrinsic functions is:

```
__riscv_v OP _vs_ SEW LMUL SEW LMUL [_ TM]
```

The first `SEW/LMUL` pair is the type of the vector operand while the second `SEW/LMUL` pair is the type of the scalar vector as well as the type of the result.

There are no mask undisturbed (`_mu`) policy intrinsics for these instructions.

`OP` is defined in the following sections.

8.1.1. Vector single-width signed reduction sum

Example:

```
void vredsum (int32_t *src1,
              int32_t *src2,
              int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vector = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t scalar = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res;
  res = __riscv_vredsum_vs_i32m1_i32m1 (vector, scalar, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

8.1.2. Vector single-width masked signed reduction sum

Example:

```
void vredsum_m (vbool32_t mask,
                int32_t *src1,
                int32_t *src2,
                int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vector = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t scalar = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res;
  res = __riscv_vredsum_vs_i32m1_i32m1_m (mask, vector, scalar, vl);
```

```
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

8.1.3. Vector single-width unsigned reduction sum

Example:


```
void vredsumu (uint32_t *src1,
               uint32_t *src2,
               uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vector = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t scalar = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res;
  res = __riscv_vredsum_vs_u32m1_u32m1 (vector, scalar, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

8.1.4. Vector single-width masked unsigned reduction sum

Example:

```
void vredsumu_m (vbool32_t mask,
                 uint32_t *src1,
                 uint32_t *src2,
                 uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vector = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t scalar = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res;
  res = __riscv_vredsum_vs_u32m1_u32m1_m (mask, vector, scalar, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

8.1.5. Vector single-width reduction signed max

Example:

```
void vredmax (int32_t *src1,
              int32_t *src2,
              int32_t *dest)
{
```

```
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vector = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t scalar = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res;
  res = __riscv_vredmax_vs_i32m1_i32m1 (vector, scalar, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


8.1.6. Vector single-width masked reduction signed max

Example:

```
void vredmax_m (vbool32_t mask,
                int32_t *src1,
                int32_t *src2,
                int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vector = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t scalar = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res;
  res = __riscv_vredmax_vs_i32m1_i32m1_m (mask, vector, scalar, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

8.1.7. Vector single-width reduction unsigned max

Example:

```
void vredmaxu (uint32_t *src1,
               uint32_t *src2,
               uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vector = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t scalar = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res;
  res = __riscv_vredmaxu_vs_u32m1_u32m1 (vector, scalar, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

8.1.8. Vector single-width masked reduction unsigned max

Example:

```
void vredmaxu_m (vbool32_t mask,
                 uint32_t *src1,
                 uint32_t *src2,
                 uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vector = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t scalar = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res;
  res = __riscv_vredmaxu_vs_u32m1_u32m1_m (mask, vector, scalar, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


8.1.9. Vector single-width reduction signed min

Example:

```
void vredmin (int32_t *src1,
              int32_t *src2,
              int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vector = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t scalar = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res;
  res = __riscv_vredmin_vs_i32m1_i32m1 (vector, scalar, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

8.1.10. Vector single-width masked reduction signed min

Example:

```
void vredmin_m (vbool32_t mask,
                int32_t *src1,
                int32_t *src2,
                int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vector = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t scalar = __riscv_vle32_v_i32m1_m (mask, src2, vl);
```

```
  vint32m1_t res;
  res = __riscv_vredmin_vs_i32m1_i32m1_m (mask, vector, scalar, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

8.1.11. Vector single-width reduction unsigned min

Example:

```
void vredminu (uint32_t *src1,
               uint32_t *src2,
               uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vector = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t scalar = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res;
  res = __riscv_vredminu_vs_u32m1_u32m1 (vector, scalar, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

8.1.12. Vector single-width masked reduction unsigned min

Example:

```
void vredminu_m (vbool32_t mask,
                 uint32_t *src1,
                 uint32_t *src2,
                 uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vector = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t scalar = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res;
  res = __riscv_vredminu_vs_u32m1_u32m1_m (mask, vector, scalar, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

8.1.13. Vector single-width signed reduction and

Example:

```
void vredand (int32_t *src1,
              int32_t *src2,
```

```
                int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vector = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t scalar = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res;
  res = __riscv_vredand_vs_i32m1_i32m1 (vector, scalar, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


8.1.14. Vector single-width masked signed reduction and

Example:

```
void vredand_m (vbool32_t mask,
                int32_t *src1,
                int32_t *src2,
                int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vector = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t scalar = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res;
  res = __riscv_vredand_vs_i32m1_i32m1_m (mask, vector, scalar, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

8.1.15. Vector single-width unsigned reduction and

Example:

```
void vredandu (uint32_t *src1,
               uint32_t *src2,
               uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vector = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t scalar = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res;
  res = __riscv_vredand_vs_u32m1_u32m1 (vector, scalar, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

8.1.16. Vector single-width masked unsigned reduction and

Example:

```
void vredandu_m (vbool32_t mask,
                 uint32_t *src1,
                 uint32_t *src2,
                 uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vector = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t scalar = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res;
  res = __riscv_vredand_vs_u32m1_u32m1_m (mask, vector, scalar, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```


8.1.17. Vector single-width signed reduction or

Example:

```
void vredor (int32_t *src1,
             int32_t *src2,
             int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vector = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t scalar = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res;
  res = __riscv_vredor_vs_i32m1_i32m1 (vector, scalar, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

8.1.18. Vector single-width masked signed reduction or

Example:

```
void vredor_m (vbool32_t mask,
               int32_t *src1,
               int32_t *src2,
               int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vector = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t scalar = __riscv_vle32_v_i32m1_m (mask, src2, vl);
```

```
  vint32m1_t res;
  res = __riscv_vredor_vs_i32m1_i32m1_m (mask, vector, scalar, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

8.1.19. Vector single-width unsigned reduction or

Example:

```
void vredoru (uint32_t *src1,
              uint32_t *src2,
              uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vector = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t scalar = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res;
  res = __riscv_vredor_vs_u32m1_u32m1 (vector, scalar, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

8.1.20. Vector single-width masked unsigned reduction or

Example:

```
void vredoru_m (vbool32_t mask,
                uint32_t *src1,
                uint32_t *src2,
                uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vector = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t scalar = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res;
  res = __riscv_vredor_vs_u32m1_u32m1_m (mask, vector, scalar, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

8.1.21. Vector single-width signed reduction xor

Example:

```
void vredxor (int32_t *src1,
              int32_t *src2,
```

```
                int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vector = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t scalar = __riscv_vle32_v_i32m1 (src2, vl);
  vint32m1_t res;
  res = __riscv_vredxor_vs_i32m1_i32m1 (vector, scalar, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


8.1.22. Vector single-width masked signed reduction xor

Example:

```
void vredxor_m (vbool32_t mask,
                int32_t *src1,
                int32_t *src2,
                int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vector = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t scalar = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t res;
  res = __riscv_vredxor_vs_i32m1_i32m1_m (mask, vector, scalar, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

8.1.23. Vector single-width unsigned reduction xor

Example:

```
void vredxoru (uint32_t *src1,
               uint32_t *src2,
               uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vector = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t scalar = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res;
  res = __riscv_vredxor_vs_u32m1_u32m1 (vector, scalar, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

8.1.24. Vector single-width masked unsigned reduction xor

Example:

```
void vredxoru_m (vbool32_t mask,
                 uint32_t *src1,
                 uint32_t *src2,
                 uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vector = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t scalar = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res;
  res = __riscv_vredxor_vs_u32m1_u32m1_m (mask, vector, scalar, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

8.2. Vector widening integer reduction instructions

The unsigned `vwredsumu` instructions zero-extend the **`SEW`**-wide vector elements before summing them, then add the `2*`**`SEW`**-width scalar element and store the result in a `2*`**`SEW`**-width scalar element.

The `vwredsum` instructions sign-extend the `SEW`-wide vector elements before summing them.

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

The generalized form of the widening integer reduction intrinsic functions is:

```
__riscv_v OP _vs_ SEW LMUL SEW LMUL [_ TM]
```

The first **`SEW/LMUL`** pair is the type of the first operand while the second **`SEW/LMUL`** pair is the type of the scalar vector as well as the result of the instruction.

There are no mask undisturbed (`_mu`) policy intrinsics for these instructions.

`OP` is defined in the following sections.

8.2.1. Vector signed reduction sum into double-width accumulator

Example:

```
void vwredsum (int32_t *src1,
               int64_t *src2,
               int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vector = __riscv_vle32_v_i32m1 (src1, vl);
  vint64m1_t scalar = __riscv_vle64_v_i64m1 (src2, vl);
  vint64m1_t res;
  res = __riscv_vwredsum_vs_i32m1_i64m1 (vector, scalar, vl);
  __riscv_vse64_v_i64m1 (dest, res, vl);
}
```

8.2.2. Vector signed masked reduction sum into double-width accumulator

Example:

```
void vwredsum_m (int32_t *src1,
                 int64_t *src2,
                 int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vector = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint64m1_t scalar = __riscv_vle64_v_i64m1_m (mask, src2, vl);
  vint64m1_t res;
  res = __riscv_vwredsum_vs_i32m1_i64m1_m (mask, vector, scalar, vl);
  __riscv_vse64_v_i64m1_m (mask, dest, res, vl);
}
```


8.2.3. Vector unsigned reduction sum into double-width accumulator

Example:

```
void vwredsumu (uint32_t *src1,
                uint64_t *src2,
                uint64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vector = __riscv_vle32_v_u32m1 (src1, vl);
  vuint64m1_t scalar = __riscv_vle64_v_u64m1 (src2, vl);
  vuint64m1_t res;
  res = __riscv_vwredsumu_vs_u32m1_u64m1 (vector, scalar, vl);
  __riscv_vse64_v_u64m1 (dest, res, vl);
}
```

8.2.4. Vector unsigned masked reduction sum into double-width accumulator

Example:

```
void vwredsumu_m (uint32_t *src1,
                  uint64_t *src2,
                  uint64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vector = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint64m1_t scalar = __riscv_vle64_v_u64m1_m (mask, src2, vl);
  vuint64m1_t res;
  res = __riscv_vwredsumu_vs_u32m1_u64m1_m (mask, vector, scalar, vl);
  __riscv_vse64_v_i64m1_m (mask, dest, res, vl);
}
```

8.3. Vector single-width floating-point reduction instructions

The `vfredosum` instructions must sum the floating-point values in element order, starting with the scalar in `scalar[0]`. That is, it performs the following computation:

```
res[0] = (((scalar[0] + vector[0]) + vector[1]) + ...) + vector[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 unordered sum reduction instructions, `vfredusum`, produce a result equivalent to a reduction tree composed of binary operator nodes, with the inputs being elements from the source vector register group (`vector`) and the source scalar value (`scalar[0]`). Each operator in the tree accepts two inputs and produces one result. Each operator first computes an exact sum as a RISC-V scalar floating-point addition with infinite exponent range and precision, then converts this exact sum to a floating-point format with range and precision each at least as great as the element floating-point format indicated by `SEW`, rounding using the currently active floating-point dynamic rounding mode. A different floating-point range and precision may be chosen for the result of each operator. A node where one input is derived only from elements masked-off or beyond the active vector length may either treat that input as the additive identity of the appropriate **`EEW`** or simply copy the other input to its output. The rounded result from the root node in the tree is converted (rounded again, using the dynamic rounding mode) to the standard floating point format indicated by **`SEW`**.

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

The generalized form of the single-width floating-point reduction intrinsic functions is:

```
__riscv_v OP _vs_ SEW LMUL SEW LMUL [_ TM]
```

The first **`SEW/LMUL`** pair is the type of the first operand while the second **`SEW/LMUL`** pair is the type of the scalar vector as well as the result of the instruction.

There are no mask undisturbed (`_mu`) policy intrinsics for these instructions.
`OP` is defined in the following sections.

8.3.1. Vector single-width floating-point ordered sum

Example:

```
void fredosum (float32_t *src1,
               float32_t *src2,
               float32_t *dest)
{
   size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t vector = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t scalar = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res;
  res = __riscv_vfredosum_vs_f32m1_f32m1 (vector, scalar, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


8.3.2. Vector single-width floating-point masked ordered sum

Example:

```
void fredosum_m (vbool32_t mask,
                 float32_t *src1,
                 float32_t *src2,
                 float32_t *dest)
{
   size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t vector = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t scalar = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res;
  res = __riscv_vfredosum_vs_f32m1_f32m1_m (mask, vector, scalar, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

8.3.3. Vector single-width floating-point unordered sum

Example:

```
void fredusum (float32_t *src1,
               float32_t *src2,
               float32_t *dest)
{
   size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t vector = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t scalar = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res;
```

```
  res = __riscv_vfredusum_vs_f32m1_f32m1 (vector, scalar, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

8.3.4. Vector single-width floating-point masked unordered sum

Example:


```
void fredusum_m (vbool32_t mask,
                 float32_t *src1,
                 float32_t *src2,
                 float32_t *dest)
{
   size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t vector = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t scalar = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res;
  res = __riscv_vfredusum_vs_f32m1_f32m1_m (mask, vector, scalar, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

8.3.5. Vector single-width floating-point max

Example:

```
void fredmax (float32_t *src1,
              float32_t *src2,
              float32_t *dest)
{
   size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t vector = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t scalar = __riscv_vle64_v_f32m1 (src2, vl);
  vfloat32m1_t res;
  res = __riscv_vfredmax_vs_f32m1_f32m1 (vector, scalar, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

8.3.6. Vector single-width floating-point masked max

Example:

```
void fredmax_m (vbool32_t mask,
                float32_t *src1,
                float32_t *src2,
```

```
                 float32_t *dest)
{
   size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t vector = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t scalar = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res;
  res = __riscv_vfredmax_vs_f32m1_f32m1_m (mask, vector, scalar, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```


8.3.7. Vector single-width floating-point reduction min

Example:

```
void fredmin (float32_t *src1,
              float32_t *src2,
              float32_t *dest)
{
   size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t vector = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t scalar = __riscv_vle32_v_f32m1 (src2, vl);
  vfloat32m1_t res;
  res = __riscv_vfredmin_vs_f32m1_f32m1 (vector, scalar, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

8.3.8. Vector single-width floating-point masked reduction min

Example:

```
void fredmin_m (vbool32_t mask,
                float32_t *src1,
                float32_t *src2,
                float32_t *dest)
{
   size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t vector = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t scalar = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  vfloat32m1_t res;
  res = __riscv_vfredmin_vs_f32m1_f32m1_m (mask, vector, scalar, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

8.4. Vector widening float-point reduction instructions

These instructions provide widening forms of the sum reductions that read and write a double-width reduction result. The reduction of the SEW-width elements is performed as in the single-width reduction case, with the elements in vector promoted to 2*SEW bits before adding to the 2*SEW-bit accumulator.

The generalized form of the widening floating-point reduction intrinsic functions is:

```
__riscv_v OP _vs_ SEW LMUL SEW LMUL [_ TM]
```

The first SEW/LMUL pair is the type of the first argument while the second SEW/LMUL pair is the type of the scalar vector and the result of the instruction.

There are no mask undisturbed (_mu) policy intrinsics for these instructions.

OP is defined in the following sections.

8.4.1. Vector widening float-point reduction ordered sum

Example:

```
void fwredosum (float32_t *src1,
                float64_t *src2,
                float64_t *dest)
{
   size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t vector = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat64m1_t scalar = __riscv_vle64_v_f64m1 (src2, vl);
  vfloat64m1_t res;
  res = __riscv_vfwredosum_vs_f32m1_f64m1 (vector, scalar, vl);
  __riscv_vse64_v_f64m1 (dest, res, vl);
}
```

8.4.2. Vector widening float-point reduction masked ordered sum

Example:

```
void fwredosum_m (vbool32_t mask,
                  float32_t *src1,
                  float64_t *src2,
                  float64_t *dest)
{
```

```
    size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t vector = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat64m1_t scalar = __riscv_vle64_v_f64m1_m (mask, src2, vl);
  vfloat64m1_t res;
  res = __riscv_vfwredosum_vs_f32m1_f64m1_m (mask, vector, scalar,vl);
  __riscv_vse64_v_f64m1 (dest, res, vl);
}
```


8.4.3. Vector widening float-point reduction unordered sum

Example:

```
void fwredusum (float32_t *src1,
                float64_t *src2,
                float64_t *dest)
{
    size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t vector = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat64m1_t scalar = __riscv_vle64_v_f64m1 (src2, vl);
  vfloat64m1_t res;
  res = __riscv_vfwredusum_vs_f32m1_f64m1 (vector, scalar, vl);
  __riscv_vse64_v_f64m1 (dest, res, vl);
}
```

8.4.4. Vector widening float-point reduction masked unordered sum

Example:

```
void fwredusum_m (vbool32_t mask,
                  float32_t *src1,
                  float64_t *src2,
                  float64_t *dest)
{
    size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t vector = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat64m1_t scalar = __riscv_vle64_v_f64m1_m (mask, src2, vl);
  vfloat64m1_t res;
  res = __riscv_vfwredusum_vs_f32m1_f64m1_m (mask, vector, scalar,vl);
  __riscv_vse64_v_f64m1 (dest, res, vl);
}
```

9. Vector mask instructions

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

9.1. Vector mask load and store instructions

The generalized form of the mask load and store intrinsic functions is:

```
__riscv_v OP _v_ BTYPE
```

BTYPE is the `boolean` return type for the mask load and store instructions and is one of:

b8	`vbool8_t` return type
b16	`vbool16_t` return type
b32	`vbool32_t` return type
b64	`vbool64_t` return type
b1	`vbool1_t` return type
b2	`vbool2_t` return type
b4	`vbool4_t` return type
b8	`vbool8_t` return type

OP is defined in the following sections.

9.1.1. Vector mask load instruction

Example:

```
vbool32_t vlm (uint8_t *src1)
{
  size_t vl = __riscv_vsetvlmax_e8m1 ();
  const vuint8m1_t base = __riscv_vle8_v_u8m1 (src1, vl);
  return __riscv_vlm_v_b32 (base, vl);
}
```

9.1.2. Vector mask store instruction

Example:

```
void vsm (vbool32_t value,
          uint8_t *src1)
{
  size_t vl = __riscv_vsetvlmax_e8m1 ();
  vuint8m1_t base = __riscv_vle8_v_u8m1 (src1, vl);
  __riscv_vsm_v_b32 (base, value, vl);
}
```

9.2. 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. Vector mask logical instructions are always unmasked, so there are no inactive elements. Mask elements past `vl`, and the tail elements, are always updated with a tail-agnostic policy.

The generalized form of the mask logical intrinsic functions is:

```
__riscv_v OP _mm_ BTYPE
```

BTYPE is the `boolean` type for the arguments and return type of the mask logical instructions and is one of:

b8	`vbool8_t` return type
b16	`vbool16_t` return type
b32	`vbool32_t` return type
b64	`vbool64_t` return type
b1	`vbool1_t` return type
b2	`vbool2_t` return type
b4	`vbool4_t` return type
b8	`vbool8_t` return type

OP is defined in the following sections.

9.2.1. Vector mask-register and instruction

Example:

```
vbool32_t vmand (vbool32_t opd1,
                 vbool32_t opd2)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmand_mm_b32 (opd1, opd2, vl);
}
```

9.2.2. Vector mask-register nand instruction

Example:

```
vbool32_t vmnand (vbool32_t opd1,
```

```
                    vbool32_t opd2)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmnand_mm_b32 (opd1, opd2, vl);
}
```


9.2.3. Vector mask-register andn instruction

Example:

```
vbool32_t vmandn (vbool32_t opd1,
                  vbool32_t opd2)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmandn_mm_b32 (opd1, opd2, vl);
}
```

9.2.4. Vector mask-register xor instruction

Example:

```
vbool32_t vmxor (vbool32_t opd1,
                 vbool32_t opd2)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmxor_mm_b32 (opd1, opd2, vl);
}
```

9.2.5. Vector mask-register or instruction

Example:

```
vbool32_t vmor (vbool32_t opd1,
                vbool32_t opd2)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmor_mm_b32 (opd1, opd2, vl);
}
```

9.2.6. Vector mask-register nor instruction

Example:

```
vbool32_t vmnor (vbool32_t opd1,
                 vbool32_t opd2)
```

```
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmnor_mm_b32 (opd1, opd2, vl);
}
```

9.2.7. Vector mask-register orn instruction

Example:

```
vbool32_t vmorn (vbool32_t opd1,
                 vbool32_t opd2)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmorn_mm_b32 (opd1, opd2, vl);
}
```

9.2.8. Vector mask-register xnor instruction

Example:

```
vbool32_t vmxnor (vbool32_t opd1,
                  vbool32_t opd2)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmxnor_mm_b32 (opd1, opd2, vl);
}
```

9.2.9. Vector mask-register mv instruction

Example:

```
vbool32_t vmmv (vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmmv_m_b32 (opd1, vl);
}
```

9.2.10. Vector mask-register clr instruction

Example:

```
vbool32_t vmclr ()
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmclr_m_b32 (vl);
```

```
}
```

9.2.11. Vector mask-register set instruction

Example:

```
vbool32_t vmset ()
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmset_m_b32 (vl);
}
```


9.2.12. Vector mask-register not instruction

Example:

```
vbool32_t vmnot (vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmnot_m_b32 (opd1, vl);
}
```

9.3. Vector count population in mask register instructions

The instructions count the number of mask elements of the active elements of the vector source mask register that has the value 1 and writes the result to a scalar register.

The generalized form of the mask count population intrinsic functions is:

```
__riscv_v OP _m_ BTYPE [_m]
```


BTYPE is the boolean type for the source operand of the count population instructions and is one of:

b8	vbool8_t return type
b16	vbool16_t return type
b32	vbool32_t return type
b64	vbool64_t return type
b1	vbool1_t return type
b2	vbool2_t return type
b4	vbool4_t return type
b8	vbool8_t return type

OP is defined in the following sections.

9.3.1. Vector mask-register count population instruction

Example:

```
unsigned long vcpop (vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vcpop_m_b32 (opd1, vl);
}
```

9.3.2. Vector mask-register masked count population instruction

Example:

```
unsigned long vcpop_m (vbool32_t mask,
                       vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vcpop_m_b32_m (mask, opd1, vl);
}
```

9.4. Vector mask-register find-first set instructions

The `vfirst` instructions find the lowest-numbered active element of the source mask vector that has the value 1 and writes that element's index to a scalar x register. If no active element has the value 1, -1 is written to the scalar x register.

The generalized form of the mask find-first intrinsic functions is:

```
__riscv_v OP _m_ BTYPE [_m]
```

BTYPE is the `boolean` type for the source operand of the find-first instruction and is one of:

b8	`vbool8_t` return type
b16	`vbool16_t` return type
b32	`vbool32_t` return type
b64	`vbool64_t` return type
b1	`vbool1_t` return type
b2	`vbool2_t` return type
b4	`vbool4_t` return type
b8	`vbool8_t` return type

OP is defined in the following sections.

9.4.1. Vector mask-register find-first instruction

Example:

```
unsigned long vfirst (vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vfirst_m_b32 (opd1, vl);
}
```

9.4.2. Vector mask-register masked find-first instruction

Example:

```
unsigned long vfirst_m (vbool32_t mask,
                        vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vfirst_m_b32_m (mask, opd1, vl);
}
```

9.5. Vector mask-register set-before-first mask bit instructions

The instructions take a mask register as input and write results to a mask register. They write a 1 to all active mask elements before the first active source element that is a 1, then write 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.

The generalized form of the set-before-first intrinsic functions is:

```
__riscv_v OP _m_ BTYPE [_m | _mu]
```

BTYPE is the `boolean` type for the source operand and the return type of the set-before-first instructions and is one of:

b8	vbool8_t return type
b16	vbool16_t return type
b32	vbool32_t return type
b64	vbool64_t return type
b1	vbool1_t return type
b2	vbool2_t return type
b4	vbool4_t return type
b8	vbool8_t return type

OP is defined in the following sections.

9.5.1. Vector mask-register set-before-first instruction

Example:

```
vbool32_t vmsbf (vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmsbf_m_b32 (opd1, vl);
}
```

9.5.2. Vector mask-register masked set-before-first instruction

Example:

```
vbool32_t vmsbf_m (vbool32_t mask,
                   vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmsbf_m_b32_m (mask, opd1, vl);
}
```


Example with mask undisturbed policy:

```
vbool32_t vmsbf_mu (vbool32_t mask,
                    vbool32_t maskedoff,
                    vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmsbf_m_b32_mu (mask, maskedoff, opd1, vl);
}
```

9.6. Vector mask-register set-including-first instructions

The vector mask set-including-first instructions are similar to set-before-first instructions, except that they also include the element with a set bit.

The generalized form of the set-including-first intrinsic functions is:

```
__riscv_v OP _m_ BTYPE [_m | _mu]
```


BTYPE is the boolean type for the source operand and the return type of the set-including-first instructions and is one of:

b8	vbool8_t return type
b16	vbool16_t return type
b32	vbool32_t return type
b64	vbool64_t return type
b1	vbool1_t return type
b2	vbool2_t return type
b4	vbool4_t return type
b8	vbool8_t return type

OP is defined in the following sections.

9.6.1. Vector mask-register set-including-first instruction

Example:

```
vbool32_t vmsif (vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmsif_m_b32 (opd1, vl);
}
```

9.6.2. Vector mask-register masked set-including-first instruction

Example:

```
vbool32_t vmsif_m (vbool32_t mask,
                   vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmsif_m_b32_m (mask, opd1, vl);
```

```
}
```

Example with mask undisturbed policy:

```
vbool32_t vmsif_mu (vbool32_t mask,
                    vbool32_t maskedoff,
                    vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmsif_m_b32_mu (mask, maskedoff, opd1, vl);
}
```

9.7. Vector mask-register set-only-first instructions

The vector mask set-only-first instructions are similar to set-before-first instructions, except that they only set the first element with a bit set, if any.

The generalized form of the set-only-first intrinsic functions is:

```
__riscv_v OP _m_ BTYPE [_m | _mu]
```


BTYPE is the boolean type for the source operand and the return type of the set-only-first instructions and is one of:

b8	vbool8_t return type
b16	vbool16_t return type
b32	vbool32_t return type
b64	vbool64_t return type
b1	vbool1_t return type
b2	vbool2_t return type
b4	vbool4_t return type
b8	vbool8_t return type

OP is defined in the following sections.

9.7.1. Vector mask-register set-only-first instruction

Example:

```
vbool32_t vmsof (vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmsof_m_b32 (opd1, vl);
}
```

9.7.2. Vector mask-register masked set-only-first instruction

Example:

```
vbool32_t vmsof_m (vbool32_t mask,
                   vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmsof_m_b32_m (mask, opd1, vl);
```

```
}
```

Example with mask undisturbed policy:

```
vbool32_t vmsof_mu (vbool32_t mask,
                    vbool32_t maskedoff,
                    vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vmsof_m_b32_mu (mask, maskedoff, opd1, vl);
}
```

9.8. Vector mask-register iota instructions

The instructions read a source vector mask register and write 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.

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

Example

```
7 6 5 4 3 2 1 0    Element number

1 0 0 1 0 0 0 1    source contents
                   viota instruction

2 2 2 1 1 1 1 0    result
```

The generalized form of the `iota` intrinsic functions is:

```
__riscv_v viota_m_ SEW LMUL [_ TM]
```

The **SEW** of the instruction is always unsigned.

9.8.1. Vector mask-register iota instruction

Example:

```
vuint32_t viota (vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_viota_m_u32m1 (opd1, vl);
}
```

9.8.2. Vector mask-register masked iota instruction

Example:

```
vuint32_t viota_m (vbool32_t mask,
                   vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
```

```
  return __riscv_viota_m_u32m1_m (mask, opd1, vl);
}
```

Example with mask undisturbed policy:

```
vuint32_t viota_mu (vbool32_t mask,
                    vuint32m1_t maskedoff,
                    vbool32_t opd1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_viota_m_u32m1_mu (mask, maskedoff, opd1, vl);
}
```

9.9. Vector mask-register element index instructions

The instructions write each element's index to the destination vector register group, from `0` to `vl-1`. They can be masked. Masking does not change the index value written to active elements.

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

The generalized form of the element index intrinsic functions is:

```
__riscv_v vid_v_ SEW LMUL [_ TM]
```

The `SEW` of the instruction is always unsigned.

9.9.1. Vector mask-register element index instruction

Example:

```
vuint32_t vid ()
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vid_v_u32m1 (vl);
}
```

9.9.2. Vector mask-register masked element index instruction

Example:

```
vuint32_t vid_m (vbool32_t mask)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vid_v_u32m1_m (mask, vl);
}
```

Example with mask undisturbed policy:

```
vuint32_t vid_mu (vbool32_t mask,
                  vuint32m1_t maskedoff)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  return __riscv_vid_v_u32m1_mu (mask, maskedoff, vl);
}
```

10. Vector permutation instructions

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

10.1. Vector 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.

The generalized form of the vector floating-point scalar move intrinsic functions is:

```
__riscv_vfmv _ OPDS _ SEW LMUL _ [RTYPE]
```

Where **OPDS** is one of:

`f_s`	Operation takes element 0 from one vector and returns a floating-point scalar.
`s_f`	Operation takes a floating-point scalar register and inserts it into element zero of a vector.

And **RTYPE** is defined for the vector to register move intrinsic functions as:

`f16`	`float16_t` type
`f32`	`float32_t` type
`f64`	`float64_t` type

10.1.1. Vector floating-point vector to scalar float instruction

This instruction copies a single **SEW**-wide element from index 0 of the source vector register to a destination scalar floating-point register.

Example:

```
float32_t fmv_f_s (float32_t *src1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t vector = __riscv_vle32_v_f32m1 (src1, vl);
  return __riscv_vfmv_f_s_f32m1_f32 (vector);
```

```
}
```

10.1.2. Vector scalar float to floating-point vector instruction

This instruction copies the scalar floating-point register to element 0 of the destination vector register. Other elements in the destination vector register (`0 < index < VLEN/SEW`) are treated as tail elements using the current tail agnostic/undisturbed policy.

Example:

```
void fmv_s_f (float32_i src,
              float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t res = __riscv_vfmv_s_f_f32m1 (src, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

10.2. Vector 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.

There are no masked versions of these instructions.

The generalized form of the vector floating-point scalar move intrinsic functions is:

```
__riscv_vfmv_ OPDS _ SEW LMUL _ [RTYPE]
```

Where OPDS is one of:

x_s	Operation takes element 0 from a vector and returns an integer scalar.
s_x	Operation takes an integer scalar and inserts it into element zero of a vector.

And RTYPE is defined for the vector to register move intrinsic functions as:

i8	int8_t type
i16	int16_t type
i32	int32_t type
i64	int64_t type
u8	uint8_t type
u16	uint16_t type
u32	uint32_t type
u64	uint64_t type

10.2.1. Vector signed integer vector to signed scalar integer instruction

This 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.

Example:

```
int32_t fmv_x_s (int32_t *src1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vector = __riscv_vle32_v_i32m1 (src1, vl);
  return __riscv_vfmv_x_s_i32m1_i32 (vector);
}
```

10.2.2. Vector signed scalar integer to signed integer vector instruction

This instruction copies a 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. Other elements in the destination vector register (`0 < index < VLEN/SEW`) are treated as tail elements using the current tail agnostic/undisturbed policy.

Example:

```
void fmv_s_x (int32_t src,
              int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t res = __riscv_vmv_s_x_i32m1 (src, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

10.2.3. Vector unsigned integer vector to unsigned scalar integer instruction

Example:

```
uint32_t fmv_x_s (uint32_t *src1)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vector = __riscv_vle32_v_u32m1 (src1, vl);
  return __riscv_vfmv_x_s_u32m1_u32 (vector);
}
```

10.2.4. Vector unsigned scalar integer to unsigned integer vector instruction

Example:

```
void fmv_s_x (uint32_t src,
              uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t res = __riscv_vmv_s_x_u32m1 (src, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

10.3. Vector slide instructions

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

10.3.1. Vector integer or floating-point vslideup instructions

The value in `vl` specifies the maximum number of destination elements that are written.
If **`XLEN > SEW`**, `offset` is not truncated to **`SEW`** bits. Destination elements `offset` through `vl-1` are written if unmasked and if `offset < vl`.

```
vslideup  behavior for destination elements:
offset amounts to slideup.
                   0 < i < max(vstart,offset)   Unchanged
max(vstart, offset) <= i < vl                   vd[i] = vs2[i-offset]
                                                if v0.mask[i] enabled
                 vl <= i < VLMAX                Follow tail policy
```

The generalized form of the vector slideup intrinsic functions is:

```
__riscv_v OP _vx_ SEW LMUL [_ TM]
```

Where **`OP`** is defined in the following sections.

10.3.1.1 Vector floating-point slideup instruction

Example:

```
void vslideupf (float32_t *src1,
                float32_t *src2,
                float32_t *dest,
                size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  // elements 0..offset-1 of opd2 are moved into res.
  vfloat32m1_t res;
  res = __riscv_vslideup_vx_f32m1 (opd2, opd1, offset, vl);
```

```
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

10.3.1.2 Vector masked floating-point slideup instruction

Example:

```
void vslideupf_m (vbool32_t mask,
                  float32_t *src1,
                  float32_t *src2,
                  float32_t *dest,
                  size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1_m (mask, src2, vl);
  // elements 0..offset-1 of opd2 are moved into res.
  vfloat32m1_t res;
  res = __riscv_vslideup_vx_f32m1_m (mask, opd2, opd1,
                                     offset, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vslideupf_mu (vbool32_t mask,
                   vfloat32m1_t maskedoff,
                   float32_t *src1,
                   float32_t *src2,
                   float32_t *dest,
                   size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t opd2 = __riscv_vle32_v_f32m1 (src2, vl);
  // elements 0..offset-1 of opd2 are moved into res.
  vfloat32m1_t res =
    __riscv_vslideup_vx_f32m1_mu (mask, maskedoff, opd2, opd1,
                                  offset, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

10.3.1.3 Vector signed integer slideup instruction

Example:

```
void vslideupi (int32_t *src1,
                int32_t *src2,
                int32_t *dest,
                size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  // elements 0..offset-1 of opd2 are moved into res.
  vint32m1_t res;
  res = __riscv_vslideup_vx_i32m1 (opd2, opd1, offset, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


10.3.1.4 Vector masked signed integer vslideup instruction

Example:

```
void vslideupi_m (vbool32_t mask,
                  int32_t *src1,
                  int32_t *src2,
                  int32_t *dest,
                  size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t opd2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  // elements 0..offset-1 of opd2 are moved into res.
  vint32m1_t res;
  res = __riscv_vslideup_vx_i32m1_m (mask, opd2, opd1,
                                     offset, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vslideupi_mu (vbool32_t mask,
                   vint32m1_t maskedoff,
                   int32_t *src1,
                   int32_t *src2,
                   int32_t *dest,
                   size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
```

```
  vint32m1_t opd2 = __riscv_vle32_v_i32m1 (src2, vl);
  // elements 0..offset-1 of opd2 are moved into res.
  vint32m1_t res;
  res = __riscv_vslideup_vx_i32m1_mu (mask, maskedoff, opd2, opd1,
                                      offset, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


10.3.1.5 Vector unsigned integer vslideup instruction

Example:

```
void vslideupu (uint32_t *src1,
                uint32_t *src2,
                uint32_t *dest,
                size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  // elements 0..offset-1 of opd2 are moved into res.
  vuint32m1_t res;
  res = __riscv_vslideup_vx_u32m1 (opd2, opd1, offset, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

10.3.1.6 Vector masked unsigned integer vslideup instruction

Example:

```
void vslideupu_m (vbool32_t mask,
                  uint32_t *src1,
                  uint32_t *src2,
                  uint32_t *dest,
                  size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  // elements 0..offset-1 of opd2 are moved into res.
  vuint32m1_t res;
  res = __riscv_vslideup_vx_u32m1_m (mask, opd2, opd1,
                                     offset, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vslideupu_mu (vbool32_t mask,
                   vuint32m1_t maskedoff,
                   uint32_t *src1,
                   uint32_t *src2,
                   uint32_t *dest,
                   size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  // elements 0..offset-1 of opd2 are moved into res.
  vuint32m1_t res;
  res = __riscv_vslideup_vx_u32m1_mu (mask, maskedoff, opd2, opd1,
                                      offset, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

10.3.2. Vector integer or float-point vslide1up instructions

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.

The `vslide1up` instructions place the `x` register argument at location 0 of the destination vector register group if 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 `SEW-XLEN` bits are ignored.

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

```
vslide1up  behavior when vl > 0
                   i < vstart      Unchanged
               0 = i = vstart     vd[i] = x[rs1] if v0.mask[i] enabled
 Max(vstart, 1) <= i < vl         vd[i] = vs2[i-1] if v0.mask[i] enabled
             vl <= i < VLMAX      Follow tail policy
```

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.

The generalized form of the vector slide1up intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where `OPDS` is one of:

`v`	Operation takes one vector from vector register group and a scalar integer register.
`vf`	Operation takes one vector from vector register group and a scalar floating-point register

`OP` is defined in the following sections.

10.3.2.1 Vector floating-point vslide1up instruction

Example:

```
void vslide1upf (float32_t *src1,
                 float32_t value,
                 float32_t *dest,
                 size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vfloat32m1_t vector = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res;
  res = __riscv_vslide1up_vf_f32m1 (vector, value, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


10.3.2.2 Vector masked floating-point vslide1up instruction

Example:

```
void vslide1upf_m (vbool32_t mask,
                   float32_t *src1,
                   float32_t value,
                   float32_t *dest,
                   size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vfloat32m1_t vector = __riscv_vle32_v_f32m1_m (mask,
                                                 src1, vl);
  vfloat32m1_t res;
  res = __riscv_vslide1up_vf_f32m1_m (mask, vector, value, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vslide1upf_mu (vbool32_t mask,
                    vfloat32m1_t maskedoff,
                    float32_t *src1,
                    float32_t val,
                    float32_t *dest,
                    size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vfloat32m1_t vector = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vslide1up_vf_f32m1_mu (mask, maskedoff, vector, val,
vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
```

```
}
```

10.3.2.3 Vector signed integer vslide1up instruction

Example:

```
void vslide1upi (int32_t *src1,
                 int32_t value,
                 int32_t *dest,
                 size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vint32m1_t vector = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t res;
  res = __riscv_vslide1up_vx_i32m1 (vector, value, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


10.3.2.4 Vector masked signed integer vslide1up instruction

Example:

```
void vslide1upi_m (vbool32_t mask,
                   int32_t *src1,
                   int32_t value,
                   int32_t *dest,
                   size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vint32m1_t vector = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t res;
  res = __riscv_vslide1up_vx_i32m1_m (mask, vector, value, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vslide1upi_mu (vbool32_t mask,
                    vint32m1_t maskedoff,
                    int32_t *src1,
                    int32_t val,
                    int32_t *dest,
                    size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vint32m1_t vector = __riscv_vle32_v_i32m1 (src1, vl);
```

```
  vint32m1_t res =
    __riscv_vslide1up_vx_i32m1_mu (mask, maskedoff, vector, val,
vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

10.3.2.5 Vector unsigned integer vslide1up instruction

Example:

```
void vslide1upu (uint32_t *src1,
                 uint32_t value,
                 uint32_t *dest,
                 size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vuint32m1_t vector = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t res;
  res = __riscv_vslide1up_vx_u32m1 (vector, value, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

10.3.2.6 Vector masked unsigned integer vslide1up instruction

Example:

```
void vslide1upu_m (vbool32_t mask,
                   uint32_t *src1,
                   uint32_t value,
                   uint32_t *dest,
                   size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vuint32m1_t vector = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t res;
  res = __riscv_vslide1up_vx_u32m1_m (mask, vector, value, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vslide1upu_mu (vbool32_t mask,
                    vuint32m1_t maskedoff,
                    uint32_t *src1,
                    uint32_t val,
```

```
                    uint32_t *dest,
                    size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vuint32m1_t vector = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t res =
    __riscv_vslide1up_vx_u32m1_mu (mask, maskedoff, vector, val,
vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

10.3.3. Vector integer or floating-point slidedown instructions

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.

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

vslidedown behavior for destination element i in slide
             0 < i < vstart       Unchanged
       vstart <= i < vl           vd[i] = src[i] if v0.mask[i] enabled
           vl <= i < VLMAX        Follow tail policy
```


The generalized form of the vector slidedown intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where **OPDS** is one of:

vx	Operation takes one vector from vector register group and a scalar integer register
vf	Operation takes one vector from vector register group and a scalar floating-point register

OP is defined in the following sections.

10.3.3.1 Vector floating-point slidedown instruction

Example:

```
void vslidedownf (float32_t *src1,
                  float32_t value,
                  float32_t *dest,
                  size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vfloat32m1_t vector = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res;
  res = __riscv_vslidedown_vf_f32m1 (vector, value, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

10.3.3.2 Vector masked floating-point slidedown instruction

Example:

```
void vslidedownf_m (vbool32_t mask,
                    float32_t *src1,
                    float32_t value,
                    float32_t *dest,
                    size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vfloat32m1_t vector = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vfloat32m1_t res;
  res = __riscv_vslidedown_vf_f32m1_m (mask, vector, value, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```


Example with mask undisturbed policy:

```
void vslidedownf_mu (vbool32_t mask,
                     vfloat32m1_t maskedoff,
                     float32_t *src1,
                     float32_t val,
                     float32_t *dest,
                     size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vfloat32m1_t vec = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res =
    __riscv_vslidedown_vf_f32m1_mu (mask, maskedoff, vec, val,
vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

10.3.3.3 Vector signed integer slidedown instruction

Example:

```
void vslidedowni (int32_t *src1,
                  int32_t value,
                  int32_t *dest,
                  size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vint32m1_t vector = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t res;
```

```
  res = __riscv_vslidedown_vx_i32m1 (vector, value, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

10.3.3.4 Vector masked signed integer slidedown instruction

Example:


```
void vslidedowni_m (vbool32_t mask,
                    int32_t *src1,
                    int32_t value,
                    int32_t *dest,
                    size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vint32m1_t vector = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t res;
  res = __riscv_vslidedown_vx_i32m1_m (mask, vector, value, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vslidedowni_mu (vbool32_t mask,
                     vint32m1_t maskedoff,
                     int32_t *src1,
                     int32_t val,
                     int32_t *dest,
                     size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vint32m1_t vector = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t res =
    __riscv_vslidedown_vx_i32m1_mu (mask, maskedoff, vec, val,
vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

10.3.3.5 Vector unsigned integer slidedown instruction

Example:

```
void vslidedownu (uint32_t *src1,
                  uint32_t value,
                  uint32_t *dest,
                  size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
```

```
  vuint32m1_t vector = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t res;
  res = __riscv_vslidedown_vx_u32m1 (vector, value, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

10.3.3.6 Vector masked unsigned integer slidedown instruction

Example:

```
void vslidedownu_m (vbool32_t mask,
                    uint32_t *src1,
                    uint32_t value,
                    uint32_t *dest,
                    size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vuint32m1_t vector = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t res;
  res = __riscv_vslidedown_vx_u32m1_m (mask, vector, value, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vslidedownu_mu (vbool32_t mask,
                     vuint32m1_t maskedoff,
                     uint32_t *src1,
                     uint32_t val,
                     uint32_t *dest,
                     size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vuint32m1_t vec = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t res =
    __riscv_vslidedown_vx_u32m1_mu (mask, maskedoff, vec, val,
vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

10.3.4. Vector integer and floating-point slide1down instructions

The `vslide1down` instructions copy 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.

The `vslide1down` instructions place the x register argument at location `vl-1` in the destination vector register if element `vl-1` is active; otherwise, the destination element is unchanged. If **`XLEN < SEW`**, the value is sign-extended to **`SEW`** bits. If **`XLEN > SEW`**, the least-significant bits are copied over and the high **`SEW-XLEN`** bits are ignored.

```
vslide1down behavior

                  i < vstart     Unchanged
        vstart <= i < vl-1       vd[i] = vs2[i+1] if 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 generalized form of the vector slide1down intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where **`OPDS`** is one of:

`vx`	Operation takes one vector from vector register group and a scalar integer register.
`vf`	Operation takes one vector from vector register group and a scalar floating-point register .

`OP` is defined in the following sections.

10.3.4.1 Vector floating-point slide1down instruction

Example:

```
void vslide1downf (float32_t *src1,
                   float32_t value,
                   float32_t *dest,
                   size_t offset)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vfloat32m1_t vector = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res;
  res = __riscv_vslide1down_vf_f32m1 (vector, value, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


10.3.4.2 Vector masked floating-point slide1down instruction

Example:

```
void vslide1downf_m (vbool32_t mask,
                     float32_t *src1,
                     float32_t value,
                     float32_t *dest)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vfloat32m1_t vector = __riscv_vle32_v_f32m1_m (mask,
                                                 src1, vl);
  vfloat32m1_t res;
  res = __riscv_vslide1down_vf_f32m1_m (mask, vector, value, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vslide1downf_mu (vbool32_t mask,
                      vfloat32m1_t maskedoff,
                      float32_t *src1,
                      float32_t val,
                      float32_t *dest)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vfloat32m1_t vec = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res =
  __riscv_vslide1down_vf_f32m1_mu (mask, maskedoff, vec, val, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

10.3.4.3 Vector signed integer slide1down instruction

Example:

```
void vslide1downi (int32_t *src1,
                   int32_t value,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vint32m1_t vector = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t res;
  res = __riscv_vslide1down_vx_i32m1 (vector, value, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


10.3.4.4 Vector masked signed integer slide1down instruction

Example:

```
void vslide1downi_m (vbool32_t mask,
                     int32_t *src1,
                     int32_t value,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vint32m1_t vector = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t res;
  res = __riscv_vslide1down_vx_i32m1_m (mask, vector, value, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vslide1downi_mu (vbool32_t mask,
                      vint32m1_t maskedoff,
                      int32_t *src1,
                      int32_t val,
                      int32_t *dest)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vint32m1_t vec = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t res =
    __riscv_vslide1down_vx_i32m1_mu (mask, maskedoff, vec, val,
vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

10.3.4.5 Vector unsigned integer slide1down instruction

Example:

```
void vslide1downu (uint32_t *src1,
                   uint32_t value,
                   uint32_t *dest)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vuint32m1_t vector = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t res;
  res = __riscv_vslide1down_vx_u32m1 (vector, value, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```


10.3.4.6 Vector masked unsigned integer slide1down instruction

Example:

```
void vslide1downu_m (vbool32_t mask,
                     uint32_t *src1,
                     uint32_t value,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vuint32m1_t vector = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t res;
  res = __riscv_vslide1down_vx_u32m1_m (mask, vector, value, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vslide1downu_m (vbool32_t mask,
                     vuint32m1_t maskedoff,
                     uint32_t *src1,
                     uint32_t val,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvl_e32m1 (100);
  vuint32m1_t vec = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t res =
    __riscv_vslide1down_vx_u32m1_mu (mask, maskedoff, vec, val,
vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

10.4. 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. The operation can be masked, and the mask undisturbed/agnostic policy is followed for inactive elements.

The vrgather instruction uses SEW/LMUL for both the data and indices. The vrgatherei16 instruction uses SEW/LMUL for the first source vector but EEW=16 and EMUL = (16/SEW)*LMUL for the indices in the second source vector.

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.

The generalized form of the vector gather intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_ TM]
```

Where OPDS is one of:

vv	Operation takes two vectors from vector register groups.
vx	Operation takes one vector from register group and a 5-bit immediate that is signed extended to SEW bits unless otherwise specified.
vx	Operation takes one vector from register group and a scalar integer variable that is sign extended to SEW bits unless otherwise specified.

OP is defined in the following sections.

10.4.1. Vector floating-point gather instruction

Example:

```
void vrgatherf (float32_t *src1,
                uint32_t *src2,
                float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vfloat32m1_t res = __riscv_vrgather_vv_f32m1 (opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


10.4.2. Vector masked floating-point gather instruction

Example:

```
void vrgatherf_m (vbool32_t mask,
                  float32_t *src1,
                  uint32_t *src2,
                  float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vfloat32m1_t res;
  res = __riscv_vrgather_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vrgatherf_mu (vbool32_t mask,
                   vfloat32m1_t maskedoff,
                   float32_t *src1,
                   uint32_t *src2,
                   float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vfloat32m1_t res =
    __riscv_vrgather_vv_f32m1_mu (mask, maskedoff, opd1, opd2, vl);
```

```
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

10.4.3. Vector signed integer gather instruction

Example:

```
void vrgatheri (int32_t *src1,
                uint32_t *src2,
                int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vint32m1_t res = __riscv_vrgather_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


10.4.4. Vector masked signed integer gather instruction

Example:

```
void vrgatheri_m (vbool32_t mask,
                  int32_t *src1,
                  uint32_t *src2,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vint32m1_t res = __riscv_vrgather_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vrgatheri_mu (vbool32_t mask,
                   vint32m1_t maskedoff,
                   int32_t *src1,
                   uint32_t *src2,
                   int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
```

```
  vint32m1_t res =
    __riscv_vrgather_vv_i32m1_mu (mask, maskedoff, opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

10.4.5. Vector unsigned integer gather instruction

Example:

```
void vrgatheru (uint32_t *src1,
                uint32_t *src2,
                uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res = __riscv_vrgather_vv_u32m1 (opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

10.4.6. Vector masked unsigned integer gather instruction

Example:

```
void vrgatheru_m (vbool32_t mask,
                  uint32_t *src1,
                  uint32_t *src2,
                  uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t res;
  res = __riscv_vrgather_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i=u32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vrgatheru_mu (vbool32_t mask,
                   vuint32m1_t maskedoff,
                   uint32_t *src1,
                   uint32_t *src2,
                   uint32_t *dest)
{
```

```
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t opd2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t res;
  res = __riscv_vrgather_vv_u32m1_mu (mask, maskedoff, opd1, opd2,
vl);
  __riscv_vse32_v_i=u32m1 (dest, res, vl);
}
```


10.4.7. Vector floating-point gather16 instruction

Example:

```
void vrgather16f (float32_t *src1,
                  uint16_t *src2,
                  float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vuint16mf2_t opd2 = __riscv_vle32_v_u16mf2 (src2, vl);
  vfloat32m1_t res = __riscv_vrgatherei16_vv_f32m1 (opd1, opd2, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

10.4.8. Vector masked floating-point gather16 instruction

Example:

```
void vrgather16f_m (vbool32_t mask,
                    float32_t *src1,
                    uint16_t *src2,
                    float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1_m (mask, src1, vl);
  vuint16mf2_t opd2 = __riscv_vle32_v_u16mf2_m (mask, src2, vl);
  vfloat32m1_t res;
  res = __riscv_vrgatherei16_vv_f32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_f32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vrgather16f_mu (vbool32_t mask,
                     vfloat32m1_t maskedoff,
```

```
                    float32_t *src1,
                    uint16_t *src2,
                    float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vuint16mf2_t opd2 = __riscv_vle32_v_u16mf2 (src2, vl);
  vfloat32m1_t res =
    __riscv_vrgatherei16_vv_f32m1_mu (mask, maskedoff, opd1, opd2,
vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```


10.4.9. Vector signed integer gather16 instruction

Example:

```
void vrgather16i (int32_t *src1,
                  uint16_t *src2,
                  int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vuint16mf2_t opd2 = __riscv_vle32_v_u16mf2 (src2, vl);
  vint32m1_t res = __riscv_vrgatherei16_vv_i32m1 (opd1, opd2, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```

10.4.10. Vector masked signed integer gather16 instruction

Example:

```
void vrgather16i_m (vbool32_t mask,
                    int32_t *src1,
                    uint16_t *src2,
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vuint16mf2_t opd2 = __riscv_vle32_v_u16mf2_m (mask, src2, vl);
  vint32m1_t res;
  res = __riscv_vrgatherei16_vv_i32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vrgather16i_mu (vbool32_t mask,
                     vint32m1_t maskedoff,
                     int32_t *src1,
                     uint16_t *src2,
                     int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vuint16mf2_t opd2 = __riscv_vle32_v_u16mf2 (src2, vl);
  vint32m1_t res =
    __riscv_vrgatherei16_vv_i32m1_mu (mask, maskedoff, opd1, opd2,
vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


10.4.11. Vector unsigned integer gather16 instruction

Example:

```
void vrgather16u (uint32_t *src1,
                  uint16_t *src2,
                  uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint16mf2_t opd2 = __riscv_vle32_v_u16mf2 (src2, vl);
  vuint32m1_t res = __riscv_vrgatherei16_vv_u32m1 (opd1, opd2, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

10.4.12. Vector masked unsigned integer gather16 instruction

Example:

```
void vrgather16u_m (vbool32_t mask,
                    uint32_t *src1,
                    uint16_t *src2,
                    uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint16mf2_t opd2 = __riscv_vle32_v_u16mf2_m (mask, src2, vl);
```

```
  vuint32m1_t res;
  res = __riscv_vrgatherei16_vv_u32m1_m (mask, opd1, opd2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, res, vl);
}
```

Example with mask undisturbed policy:

```
void vrgather16u_mu (vbool32_t mask,
                     vuint32m1_t maskedoff,
                     uint32_t *src1,
                     uint16_t *src2,
                     uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint16mf2_t opd2 = __riscv_vle32_v_u16mf2 (src2, vl);
  vuint32m1_t res =
    __riscv_vrgatherei16_vv_u32m1_mu (mask, maskedoff, opd1, opd2,
vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

10.5. Vector compress instructions

The vector compress instructions allow 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.

The vector mask register specified by `opd2` indicates which of the first `vl` elements of vector register group `opd1` should be extracted and packed into contiguous elements at the beginning of vector register `res`. The remaining elements of `res` are treated as tail elements according to the current tail policy.

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     mask
8 7 6 5 4 3 2 1 0     opd1
1 2 3 4 5 6 7 8 9     res (as an example of what the data looks like)
                      vcompress res, opd1, mask
1 2 3 4 8 7 5 2 0     result
```

The generalized form of the vector compress intrinsic functions is:

```
__riscv_vcompress_vm_ SEW LMUL [_tu]
```

10.5.1. Vector floating-point compress instruction

Example:

```
void vcompressf (float32_t *src1,
                 vbool32_t maak,
                 float32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vfloat32m1_t opd1 = __riscv_vle32_v_f32m1 (src1, vl);
  vfloat32m1_t res = __riscv_vcompress_vm_f32m1 (opd1, mask, vl);
  __riscv_vse32_v_f32m1 (dest, res, vl);
}
```

10.5.2. Vector signed integer compress instruction

Example:

```
void vcompressi (int32_t *src1,
                 vbool32_t maak,
```

```
                    int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t opd1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t res = __riscv_vcompress_vm_i32m1 (opd1, mask, vl);
  __riscv_vse32_v_i32m1 (dest, res, vl);
}
```


10.5.3. Vector usnsigned integer compress instruction

Example:

```
void vcompressu (uint32_t *src1,
                 vbool32_t maak,
                 uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t opd1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t res = __riscv_vcompress_vm_u32m1 (opd1, mask, vl);
  __riscv_vse32_v_u32m1 (dest, res, vl);
}
```

11. Andes vector extensions

These extensions define vector instructions to enhance 8-bit and 16-bit integer/fixed-point data dot product operations to improve machine learning application performance.

11.1. Vector dot product instructions

These instructions calculate the dot product of four sets of `SEW/4` bits data between the elements of `vs1` and `vs2` and the result is `SEW` bits. And the `SEW` bits result is accumulated into the corresponding `SEW` bits element in `vd`.

The instructions are only valid for `SEW`=32 or 64-bit.

Operation:

```
D4 = SEW/4;
vs1d0[i] = vs1[i].D4[0]; vs2d0[i] = vs2[i].D4[0];
vs1d1[i] = vs1[i].D4[1]; vs2d1[i] = vs2[i].D4[1];
vs1d2[i] = vs1[i].D4[2]; vs2d2[i] = vs2[i].D4[2];
vs1d3[i] = vs1[i].D4[3]; vs2d3[i] = vs2[i].D4[3];
vd[i] = vd[i] + vs1d0[i] * vs2d0[i] +
                vs1d1[i] * vs2d1[i] +
                vs1d2[i] * vs2d2[i] +
                vs1d3[i] * vs2d3[i];
```

The generalized form of dot product intrinsic functions is:

```
__riscv_v OP _vv_ SEW LMUL [_m]
```

Where `OP` is defined in the following sections.

11.1.1. Vector signed dot product on ¼ of SEW-sized data

Example:

```
void vdot4s (int32_t *src1,
             int32_t *src2,
             int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vs1 = __riscv_vle32_v_i32m1 (src1, vl);
  vint32m1_t vs2 = __riscv_vle32_v_i32m1 (src2, vl);
```

```
  vint32m1_t vd = __riscv_vle32_v_i32m1 (dest, vl);
  vd = __riscv_vd4dots_vv_i32m1 (vd, vs1, vs2, vl);
  __riscv_vse32_v_i32m1 (dest, vd, vl);
}
```

11.1.2. Vector masked signed dot product on ¼ of SEW-sized data

Example:

```
void vdot4s_m (vbool32_t mask,
               int32_t *src1,
               int32_t *src2,
               int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vs1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vint32m1_t vs2 = __riscv_vle32_v_i32m1_m (mask, src2, vl);
  vint32m1_t vd = __riscv_vle32_v_i32m1_m (mask, dest, vl);
  vd = __riscv_vd4dots_vv_i32m1_m (mask, vd, vs1, vs2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, vd, vl);
}
```

11.1.3. Vector unsigned dot product on ¼ of SEW-sized data

Example:

```
void vdot4u (uint32_t *src1,
             uint32_t *src2,
             uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vs1 = __riscv_vle32_v_u32m1 (src1, vl);
  vuint32m1_t vs2 = __riscv_vle32_v_u32m1 (src2, vl);
  vuint32m1_t vd = __riscv_vle32_v_u32m1 (dest, vl);
  vd = __riscv_vd4dotu_vv_u32m1 (vd, vs1, vs2, vl);
  __riscv_vse32_v_u32m1 (dest, vd, vl);
}
```

11.1.4. Vector masked unsigned dot product on ¼ of SEW-sized data

Example:

```
void vdot4u_m (vbool32_t mask,
               uint32_t *src1,
```

```
                uint32_t *src2,
                uint32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vuint32m1_t vs1 = __riscv_vle32_v_u32m1_m (mask, src1, vl);
  vuint32m1_t vs2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vuint32m1_t vd = __riscv_vle32_v_u32m1_m (mask, dest, vl);
  vd = __riscv_vd4dotu_vv_u32m1_m (mask, vd, vs1, vs2, vl);
  __riscv_vse32_v_u32m1_m (mask, dest, vd, vl);
}
```


11.1.5. Vector signed and unsigned dot product on ¼ of SEW-sized data

Example:

```
void vdot4su (int32_t *src1,
              uint32_t *src2,
              int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vs1 = __riscv_vle32_v_i32m1 (src1, vl);
  vuint32m1_t vs2 = __riscv_vle32_v_u32m1 (src2, vl);
  vint32m1_t vd = __riscv_vle32_v_i32m1 (dest, vl);
  vd = __riscv_vd4dotsu_vv_i32m1 (vd, vs1, vs2, vl);
  __riscv_vse32_v_i32m1 (dest, vd, vl);
}
```

11.1.6. Vector masked signed and unsigned dot product on ¼ of SEW-sized data

Example:

```
void vdot4su_m (vbool32_t mask,
                int32_t *src1,
                uint32_t *src2,
                int32_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e32m1 ();
  vint32m1_t vs1 = __riscv_vle32_v_i32m1_m (mask, src1, vl);
  vuint32m1_t vs2 = __riscv_vle32_v_u32m1_m (mask, src2, vl);
  vint32m1_t vd = __riscv_vle32_v_i32m1_m (mask, dest, vl);
  vd = __riscv_vd4dotsu_vv_i32m1_m (mask, vd, vs1, vs2, vl);
  __riscv_vse32_v_i32m1_m (mask, dest, vd, vl);
}
```

11.2. Vector packed FP16 extensions

11.2.1. Vector single-width floating-point packed fused multiply-add with top FP16 as multiplicand instructions

The instructions extract a pair of `FP16` data from the source scalar floating-point register (`rs1`), and multiply the top `FP16` data in the pair (`rs1.FP16[1]`) with the `FP16` elements in the vector register (`vs2`). The multiplication results are added with the bottom `FP16` data in the pair (`rs1.FP16[0]`) and the element addition results are written back to `vd`.

The instructions are only valid for **`SEW`**=16.

Operation:

```
vd[i] = (f[rs1].FP16[1] * vs2[i]) + f[rs1].FP16[0];
```

The generalized form of the single-width floating-point packed fused multiply-add with top FP16 as multiplicand intrinsic functions is:

```
__riscv_vfpmadt_vf_f16 LMUL [_m]
```

11.2.1.1 single-width floating-point packed fused multiply-add with top FP16 as multiplicand instruction

Example:

```
void fpmadt (float32_t rs1,
             float16_t *src2,
             float16_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1 ();
  vfloat16m1_t vs2 = __riscv_vle16_v_f16m1 (src2, vl);
  vfloat16m1_t vd = __riscv_vle16_v_f16m1 (dest, vl);
  vd = __riscv_vfpmadt_vf_f16m1 (vd, rs1, vs2, vl);
  __riscv_vse16_v_f16m1 (dest, vd, vl);
}
```

11.2.1.2 single-width floating-point packed fused masked multiply-add with top FP16 as multiplicand instruction

Example:

```
void fpmadt_m (vbool16_t mask,
```

```
                float32_t rs1,
                float16_t *src2,
                float16_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1 ();
  vfloat16m1_t vs2 = __riscv_vle16_v_f16m1_m (mask, src2, vl);
  vfloat16m1_t vd = __riscv_vle16_v_f16m1_m (mask, dest, vl);
  vd = __riscv_vfpmadt_vf_f16m1_m (mask, vd, rs1, vs2, vl);
  __riscv_vse16_v_f16m1_m (mask, dest, vd, vl);
}
```

11.2.2. Vector single-width floating-point packed fused multiply-add with bottom FP16 as multiplicand instructions

The instructions extract a pair of `FP16` data from the source scalar floating-point register (`rs1`) and multiply the bottom `FP16` data in the pair (`rs1.FP16[0]`) with the `FP16` elements in the vector register (`vs2`). The multiplication results are added with the top `FP16` data in the pair (`rs1.FP16[1]`) and the element addition results are written back to `vd`.

The instructions are only valid for `SEW=16`.

Operations:

```
vd[i] = (f[rs1].FP16[0] * vs2[i]) + f[rs1].FP16[1];
```

The generalized form of the single-width floating-point packed fused multiply-add with bottom FP16 as multiplicand intrinsic functions is:

```
__riscv_vfpmadb_vf_f16 LMUL [_m]
```

11.2.2.1 single-width floating-point packed fused multiply-add with bottom FP16 as multiplicand instruction

Example:

```
void fpmadb (float32_t rs1,
             float16_t *src2,
             float16_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1 ();
  vfloat16m1_t vs2 = __riscv_vle16_v_f16m1 (src2, vl);
  vfloat16m1_t vd = __riscv_vle16_v_f16m1 (dest, vl);
  vd = __riscv_vfpmadb_vf_f16m1 (vd, rs1, vs2, vl);
  __riscv_vse16_v_f16m1 (dest, vd, vl);
}
```

11.2.2.2 single-width floating-point packed fused masked multiply-add with bottom FP16 as multiplicand instruction

Example:

```
void fpmadb_m (vbool16_t mask,
               float32_t rs1,
```

```
                float16_t *src2,
                float16_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1 ();
  vfloat16m1_t vs2 = __riscv_vle16_v_f16m1_m (mask, src2, vl);
  vfloat16m1_t vd = __riscv_vle16_v_f16m1_m (mask, dest, vl);
  vd = __riscv_vfpmadb_vf_f16m1_m (mask, vd, rs1, vs2, vl);
  __riscv_vse16_v_f16m1_m (mask, dest, vd, vl);
}
```

11.3. Vector quad-widening multiply and add Instructions

The quad-widening multiply and add instructions add a `SEW-bit*SEW-bit` multiply result to or from a `4*SEW-bit` value and produces a `4*SEW-bit` result. All combinations of signed and unsigned are supported.

The generalized form of the quad-widening multiply-add intrinsic functions is:

```
__riscv_v OP _ OPDS _ SEW LMUL [_m]
```

Where `OPDS` is one of:

vv	Operation takes two vectors from vector register groups.
vx	Operation takes one vector from register group and a 5-bit immediate that is signed extended to `SEW` bits unless otherwise specified.
vx	Operation takes one vector from register group and a scalar integer variable that is sign extended to `SEW` bits unless otherwise specified.

`OP` is defined in the following sections.

11.3.1. Quad-widening unsigned-integer multiply-add, overwrite addend

Example:

```
void vector_qmaccu (uint16_t *src1,
                    uint16_t *src2,
                    uint64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vuint64m4_t vd = __riscv_vle32_v_u64m4 (dest, vl);
  vd = __riscv_vqmaccu_vv_u64m4 (vd, opd1, opd2, vl);
  __riscv_vse64_v_u64m4 (dest, vd, vl);
}
```

11.3.2. Quad-widening masked unsigned-integer multiply-add, overwrite addend

Example:

```
void vector_qmaccu_m (vbool16_t mask,
                      uint16_t *src1,
                      uint16_t *src2,
                      uint64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vuint16m1_t opd1 = __riscv_vle16_v_u16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vuint64m4_t vd = __riscv_vle32_v_u64m4 (dest, vl);
  vd = __riscv_vqmaccu_vv_u64m4_m (mask, vd, opd1, opd2, vl);
  __riscv_vse64_v_u64m4 (dest, vd, vl);
}
```


11.3.3. Quad-widening signed-integer multiply-add, overwrite addend

Example:

```
void vector_qmacc (int16_t *src1,
                   int16_t *src2,
                   int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint64m4_t vd = __riscv_vle32_v_i64m4 (dest, vl);
  vd = __riscv_vqmacc_vv_i64m4 (vd, opd1, opd2, vl);
  __riscv_vse64_v_i64m4 (dest, vd, vl);
}
```

11.3.4. Quad-widening masked signed-integer multiply-add, overwrite addend

Example:

```
void vector_qmacc_m (vbool16_t mask,
                     int16_t *src1,
                     int16_t *src2,
                     int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
```

```
  vint64m4_t vd = __riscv_vle32_v_i64m4 (dest, vl);
  vd = __riscv_vqmacc_vv_i64m4_m (mask, vd, opd1, opd2, vl);
  __riscv_vse64_v_i64m4 (dest, vd, vl);
}
```

11.3.5. Quad-widening signed-unsigned-integer multiply-add, overwrite addend

Example:

```
void vector_qmaccsu (int16_t *src1,
                     uint16_t *src2,
                     int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vint64m4_t vd = __riscv_vle32_v_i64m4 (dest, vl);
  vd = __riscv_vqmaccsu_vv_i64m4 (vd, opd1, opd2, vl);
  __riscv_vse64_v_i64m4 (dest, vd, vl);
}
```

11.3.6. Quad-widening masked signed-unsigned-integer multiply-add, overwrite addend

Example:

```
void vector_qmaccsu_m (vbool16_t mask,
                       int16_t *src1,
                       uint16_t *src2,
                       int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd1 = __riscv_vle16_v_i16m1 (src1, vl);
  vuint16m1_t opd2 = __riscv_vle16_v_u16m1 (src2, vl);
  vint64m4_t vd = __riscv_vle32_v_i64m4 (dest, vl);
  vd = __riscv_vqmaccsu_vv_i64m4_m (mask, vd, opd1, opd2, vl);
  __riscv_vse64_v_i64m4 (dest, vd, vl);
}
```

11.3.7. Quad-widening unsigned-signed-integer multiply-add, overwrite addend

Example:

```
void vector_qmaccus (uint16_t opd1,
```

```
                    int16_t *src2,
                    int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint64m4_t vd = __riscv_vle32_v_i64m4 (dest, vl);
  vd = __riscv_vqmaccus_vx_i64m4 (vd, opd1, opd2, vl);
  __riscv_vse64_v_i64m4 (dest, vd, vl);
}
```


11.3.8. Quad-widening masked unsigned-signed-integer multiply-add, overwrite addend

Example:

```
void vector_qmaccus_m (vbool16_t mask,
                    uint16_t opd1,
                    int16_t *src2,
                    int64_t *dest)
{
  size_t vl = __riscv_vsetvlmax_e16m1();
  vint16m1_t opd2 = __riscv_vle16_v_i16m1 (src2, vl);
  vint64m4_t vd = __riscv_vle32_v_i64m4 (dest, vl);
  vd = __riscv_vqmaccus_vx_i64m4_m (mask, vd, opd1, opd2, vl);
  __riscv_vse64_v_i64m4 (dest, vd, vl);
}
```

12. RVV C intrinsic functions examples

12.1. Sgemm

The GEMM (General Matrix Multiplication) class of functions are the heart of a fully connected layer in most neural networks. Roughly 85% of a CPU's processing capabilities are spent on computing the matrices, which are often very large. One could have an 1152x192 matrix producing a 256x192 result. With a CPU-only implementation, this would amount to about fifty million floating point instructions.


```
#include <riscv_vector.h>
#include <stddef.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#define N 32

#define MAX_BLOCKSIZE 32
#define MLEN 4
#define KLEN 8
#define NLEN 4
#define OUTPUT_LEN 16

float a_array[MAX_BLOCKSIZE] = {
   0.4325648115282207, -1.6655843782380970,
   0.1253323064748307, 0.2876764203585489,
   -1.1464713506814637, 1.1909154656429988,
    1.1891642016521031, -0.0376332765933176,
    0.3272923614086541, 0.1746391428209245,
   -0.1867085776814394, 0.7257905482933027,
   -0.5883165430141887, 2.1831858181971011,
   -0.1363958830865957, 0.1139313135208096,
    1.0667682113591888, 0.0592814605236053,
   -0.0956484054836690, -0.8323494636500225,
   0.2944108163926404, -1.3361818579378040,
   0.7143245518189522, 1.6235620644462707,
   -0.6917757017022868, 0.8579966728282626,
   1.2540014216025324, -1.5937295764474768,
   -1.4409644319010200, 0.5711476236581780,
   -0.3998855777153632, 0.1
```

```
  };

float b_array[MAX_BLOCKSIZE] = {
  1.7491401329284098, 0.1325982188803279,
  0.3252281811989881, -0.7938091410349637,
  0.3149236145048914, -0.5272704888029532,
  0.9322666565031119, 1.1646643544607362,
  -2.0456694357357357, -0.6443728590041911,
  1.7410657940825480, 0.4867684246821860,
  1.0488288293660140, 1.4885752747099299,
  1.2705014969484090, -1.8561241921210170,
  2.1343209047321410, 1.4358467535865909,
  -0.9173023332875400, -1.1060770780029008,
  0.8105708062681296, 0.6985430696369063,
  -0.4015827425012831, 1.2687512030669628,
  -0.7836083053674872, 0.2132664971465569,
  0.7878984786088954, 0.8966819356782295,
  0.1869172943544062, 1.0131816724341454,
  0.2484350696132857, 0.1}
  ;

float golden_array[OUTPUT_LEN];
float c_array[OUTPUT_LEN];

void sgemm_golden() {
  for (size_t i = 0; i < MLEN; ++i)
    for (size_t j = 0; j < NLEN; ++j)
      for (size_t k = 0; k < KLEN; ++k)
        golden_array[i * NLEN + j] += a_array[i * KLEN + k] *
b_array[j + k * NLEN];
}

// reference https://github.com/riscv/riscv-v-
spec/blob/master/example/sgemm.S
// c += a*b (alpha=1, no transpose on input matrices)
// matrices stored in C row-major order
void sgemm_vec(size_t size_m, size_t size_n, size_t size_k,
               const float *a, // m * k matrix
               size_t lda,
               const float *b, // k * n matrix
               size_t ldb,
               float *c, // m * n matrix
               size_t ldc) {
```

```
  size_t vl;
  for (size_t m = 0; m < size_m; ++m) {
    const float *b_n_ptr = b;
    float *c_n_ptr = c;
    for (size_t c_n_count = size_n; c_n_count; c_n_count -= vl) {
      vl = __riscv_vsetvl_e32m1(c_n_count );
      const float *a_k_ptr = a;
      const float *b_k_ptr = b_n_ptr;
      vfloat32m1_t acc = __riscv_vle32_v_f32m1(c_n_ptr, vl);
      for (size_t k = 0; k < size_k; ++k) {
        vfloat32m1_t b_n_data = __riscv_vle32_v_f32m1(b_k_ptr, vl);
        acc = vfmacc_vf_f32m1(acc, *a_k_ptr, b_n_data, vl);
        b_k_ptr += ldb;
        a_k_ptr++;
      }
      vse32_v_f32m1(c_n_ptr, acc, vl);
      c_n_ptr += vl;
      b_n_ptr += vl;
    }
    a += lda;
    c += ldc;
  }
}

int fp_eq(float reference, float actual, float relErr)
{
  // if near zero, do absolute error instead.
  float absErr =
   relErr * ((fabsf(reference) > relErr) ? fabsf(reference) : relErr);
  return fabsf(actual - reference) < absErr;
}

int main() {
  // golden
  memcpy(golden_array, b_array, OUTPUT_LEN * sizeof(float));
  sgemm_golden();
  // vector
  memcpy(c_array, b_array, OUTPUT_LEN * sizeof(float));
  sgemm_vec(MLEN, NLEN, KLEN, a_array, KLEN, b_array, NLEN,
            c_array, NLEN);

  int pass = 1;
  for (int i = 0; i < OUTPUT_LEN; i++) {
```

```
    if (!fp_eq(golden_array[i], c_array[i], 1e-5)) {
      printf("index %d fail, %f=!%f\n", i, golden_array[i], c_array[i]);
      pass = 0;
    }
  }
  if (pass)
    printf("passed\n");
  return (pass == 0);
}
```

12.2. Vector branch example

This example shows how to perform a check for dividing by zero and replacing the result with a given constant.


```
#include "common.h"
#include <riscv_vector.h>

// branch and assign
void branch_golden(double *a, double *b, double *c, int n, double
constant) {
  for (int i = 0; i < n; ++i) {
    c[i] = (b[i] != 0.0) ? a[i] / b[i] : constant;
  }
}

void branch_vec (double *a, double *b, double *c, int n, double constant)
{
  // set vlmax and initialize variables
  size_t vlmax = __riscv_vsetvlmax_e64m1();
  vfloat64m1_t vec_constant = __riscv_vfmv_v_f_f64m1(constant, vlmax);
  for (size_t vl; n > 0; n -= vl, a += vl, b += vl, c += vl) {
    vl = __riscv_vsetvl_e64m1(n);

    vfloat64m1_t vec_a = __riscv_vle64_v_f64m1(a, vl);
    vfloat64m1_t vec_b = __riscv_vle64_v_f64m1(b, vl);

    vbool64_t mask = __riscv_vmfne_vf_f64m1_b64(vec_b, 0, vl);

    vfloat64m1_t vec_c = __riscv_vfdiv_vv_f64m1_mu(mask,
                                                   /*maskedoff*/ vec_constant,
                                                   vec_a, vec_b, vl);
    __riscv_vse64_v_f64m1(c, vec_c, vl);
  }
}

int main() {
  const int N = 31;
  const double constant = 7122.0;
  const uint32_t seed = 0xdeadbeef;
  srand(seed);

  // data gen
```

```
  double A[N], B[N];
  gen_rand_1d(A, N);
  gen_rand_1d(B, N);
  for (int i = 0; i < 5; ++i) {
    int pos = rand() % N;
    B[pos] = 0;
  }

  // compute
  double golden[N], actual[N];
  branch_golden(A, B, golden, N, constant);
  branch(A, B, actual, N, constant);


  // compare
  puts(compare_1d(golden, actual, N) ? "pass" : "fail");
}
```

12.3. Index example

```
#include "common.h"
#include <riscv_vector.h>

// index arithmetic
void index_golden(double *a, double *b, double *c, int n) {
  for (int i = 0; i < n; ++i) {
    a[i] = b[i] + (double)i * c[i];
  }
}

void index_vec (double *a, double *b, double *c, int n) {
  size_t vlmax = __riscv_vsetvlmax_e32m1();
  vuint32m1_t vec_i = __riscv_vid_v_u32m1(vlmax);
  for (size_t vl; n > 0; n -= vl, a += vl, b += vl, c += vl) {
    vl = __riscv_vsetvl_e64m2(n);

    vfloat64m2_t vec_i_double = __riscv_vfwcvt_f_xu_v_f64m2(vec_i, vl);

    vfloat64m2_t vec_b = __riscv_vle64_v_f64m2(b, vl);
    vfloat64m2_t vec_c = __riscv_vle64_v_f64m2(c, vl);

    vfloat64m2_t vec_a =
        __riscv_vfmadd_vv_f64m2(vec_c, vec_i_double, vec_b, vl);
    __riscv_vse64_v_f64m2(a, vec_a, vl);

    vec_i = __riscv_vadd_vx_u32m1(vec_i, vl, vl);
  }
}

int main() {
  const int N = 31;
  const uint32_t seed = 0xdeadbeef;
  srand(seed);

  // data gen
  double B[N], C[N];
  gen_rand_1d(B, N);
  gen_rand_1d(C, N);

  // compute
  double golden[N], actual[N];
```

```
  index_golden(golden, B, C, N);
  index_(actual, B, C, N);

  // compare
  puts(compare_1d(golden, actual, N) ? "pass" : "fail");
}
```

12.4. Matmul example

```
#include "common.h"
#include <riscv_vector.h>

// matrix multiplication
// A[n][o], B[m][o] --> C[n][m];
void matmul_golden(double **a, double **b, double **c, int n, int m, int
o) {
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j) {
      c[i][j] = 0;
      for (int k = 0; k < o; ++k)
        c[i][j] += a[i][k] * b[j][k];
    }
}

void matmul(double **a, double **b, double **c, int n, int m, int o) {
  size_t vlmax = __riscv_vsetvlmax_e64m1();
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
      double *ptr_a = &a[i][0];
      double *ptr_b = &b[j][0];
      int k = o;
      vfloat64m1_t vec_s = __riscv_vfmv_v_f_f64m1(0, vlmax);
      vfloat64m1_t vec_zero = __riscv_vfmv_v_f_f64m1(0, vlmax);
      for (size_t vl; k > 0; k -= vl, ptr_a += vl, ptr_b += vl) {
        vl = __riscv_vsetvl_e64m1(k);

        vfloat64m1_t vec_a = __riscv_vle64_v_f64m1(ptr_a, vl);
        vfloat64m1_t vec_b = __riscv_vle64_v_f64m1(ptr_b, vl);

        vec_s = __riscv_vfmacc_vv_f64m1(vec_s, vec_a, vec_b, vl);
      }

      vfloat64m1_t vec_sum;
      vec_sum = __riscv_vfredusum_vs_f64m1_f64m1(vec_s, vec_zero, vlmax);
      double sum = __riscv_vfmv_f_s_f64m1_f64(vec_sum);
      c[i][j] = sum;
    }
  }
}
```

```
int main() {
  const int N = 8;
  const int M = 8;
  const int O = 7;
  uint32_t seed = 0xdeadbeef;
  srand(seed);

  // data gen
  double **A = alloc_array_2d(N, O);
  double **B = alloc_array_2d(M, O);
  gen_rand_2d(A, N, O);
  gen_rand_2d(B, M, O);

  // compute
  double **golden = alloc_array_2d(N, M);
  double **actual = alloc_array_2d(N, M);
  matmul_golden(A, B, golden, N, M, O);
  matmul(A, B, actual, N, M, O);

  // compare
  puts(compare_2d(golden, actual, N, M) ? "pass" : "fail");
}
```

12.5. Memcpy example

```
#include "common.h"
#include <riscv_vector.h>
#include <string.h>

void *memcpy_vec(void *dst, void *src, size_t n) {
  void *save = dst;
  // copy data byte by byte
  for (size_t vl; n > 0; n -= vl, src += vl, dst += vl) {
    vl = __riscv_vsetvl_e8m8(n);
    vuint8m8_t vec_src = __riscv_vle8_v_u8m8(src, vl);
    __riscv_vse8_v_u8m8(dst, vec_src, vl);
  }
  return save;
}

int main() {
  const int N = 127;
  const uint32_t seed = 0xdeadbeef;
  srand(seed);

  // data gen
  double A[N];
  gen_rand_1d(A, N);

  // compute
  double golden[N], actual[N];
  memcpy(golden, A, sizeof(A));
  memcpy_vec(actual, A, sizeof(A));

  // compare
  puts(compare_1d(golden, actual, N) ? "pass" : "fail");
}
```

12.6. Reduce example

```
#include "common.h"
#include <riscv_vector.h>

// accumulate and reduce
void reduce_golden(double *a, double *b, double *result_sum,
                   int *result_count, int n) {
  int count = 0;
  double s = 0;
  for (int i = 0; i < n; ++i) {
    if (a[i] != 0.0) {
      s += a[i] * b[i];
      count++;
    }
  }

  *result_sum = s;
  *result_count = count;
}

void reduce_vec (double *a, double *b, double *result_sum,
              int *result_count,
              int n) {
  int count = 0;
  // set vlmax and initialize variables
  size_t vlmax = __riscv_vsetvlmax_e64m1();
  vfloat64m1_t vec_zero = __riscv_vfmv_v_f_f64m1(0, vlmax);
  vfloat64m1_t vec_s = __riscv_vfmv_v_f_f64m1(0, vlmax);
  for (size_t vl; n > 0; n -= vl, a += vl, b += vl) {
    vl = __riscv_vsetvl_e64m1(n);

    vfloat64m1_t vec_a = __riscv_vle64_v_f64m1(a, vl);
    vfloat64m1_t vec_b = __riscv_vle64_v_f64m1(b, vl);

    vbool64_t mask = __riscv_vmfne_vf_f64m1_b64(vec_a, 42, vl);

    vec_s = __riscv_vfmacc_vv_f64m1_tumu(mask, vec_s, vec_a, vec_b, vl);
    count = count + __riscv_vcpop_m_b64(mask, vl);
  }
  vfloat64m1_t vec_sum;
  vec_sum = __riscv_vfredusum_vs_f64m1_f64m1(vec_s, vec_zero, vlmax);
  double sum = __riscv_vfmv_f_s_f64m1_f64(vec_sum);
```

```
  *result_sum = sum;
  *result_count = count;
}

int main() {
  const int N = 31;
  uint32_t seed = 0xdeadbeef;
  srand(seed);

  // data gen
  double A[N], B[N];
  gen_rand_1d(A, N);
  gen_rand_1d(B, N);

  // compute
  double golden_sum, actual_sum;
  int golden_count, actual_count;
  reduce_golden(A, B, &golden_sum, &golden_count, N);
  reduce_vec(A, B, &actual_sum, &actual_count, N);

  // compare
  puts(golden_sum - actual_sum < 1e-6 &&
                      golden_count == actual_count ?
"pass"
  : "fail");
}
```

12.7. Saxpy example

```
#include <riscv_vector.h>
#include <stddef.h>
#include <stdio.h>
#include <math.h>

#define N 31

float input[N] = {-0.4325648115282207, -1.6655843782380970, 0.1253323064748307,
                  0.2876764203585489,  -1.1464713506814637, 1.1909154656429988,
                  1.1891642016521031,  -0.0376332765933176, 0.3272923614086541,
                  0.1746391428209245,  -0.1867085776814394, 0.7257905482933027,
                  -0.5883165430141887, 2.1831858181971011,  -0.1363958830865957,
                  0.1139313135208096,  1.0667682113591888,  0.0592814605236053,
                  -0.0956484054836690, -0.8323494636500225, 0.2944108163926404,
                  -1.3361818579378040, 0.7143245518189522,  1.6235620644462707,
                  -0.6917757017022868, 0.8579966728282626,  1.2540014216025324,
                  -1.5937295764474768, -1.4409644319010200, 0.5711476236581780,
                  -0.3998855777153632};

float output_golden[N] = {
    1.7491401329284098,  0.1325982188803279,  0.3252281811989881,
    -0.7938091410349637, 0.3149236145048914,  -0.5272704888029532,
    0.9322666565031119,  1.1646643544607362,  -2.0456694357357357,
    -0.6443728590041911, 1.7410657940825480,  0.4867684246821860,
    1.0488288293660140,  1.4885752747099299,  1.2705014969484090,
    -1.8561241921210170, 2.1343209047321410,  1.4358467535865909,
    -0.9173023332875400, -1.1060770780029008, 0.8105708062681296,
    0.6985430696369063,  -0.4015827425012831, 1.2687512030669628,
    -0.7836083053674872, 0.2132664971465569,  0.7878984786088954,
    0.8966819356782295,  -0.1869172943544062, 1.0131816724341454,
    0.2484350696132857};

float output[N] = {
    1.7491401329284098,  0.1325982188803279,  0.3252281811989881,
    -0.7938091410349637, 0.3149236145048914,  -0.5272704888029532,
    0.9322666565031119,  1.1646643544607362,  -2.0456694357357357,
    -0.6443728590041911, 1.7410657940825480,  0.4867684246821860,
    1.0488288293660140,  1.4885752747099299,  1.2705014969484090,
    -1.8561241921210170, 2.1343209047321410,  1.4358467535865909,
    -0.9173023332875400, -1.1060770780029008, 0.8105708062681296,
    0.6985430696369063,  -0.4015827425012831, 1.2687512030669628,
```

```
    -0.7836083053674872, 0.2132664971465569,  0.7878984786088954,
    0.8966819356782295,  -0.1869172943544062, 1.0131816724341454,
    0.2484350696132857};

void saxpy_golden(size_t n, const float a, const float *x, float *y) {
  for (size_t i = 0; i < n; ++i) {
    y[i] = a * x[i] + y[i];
  }
}

// reference https://github.com/riscv/riscv-v-
spec/blob/master/example/saxpy.s
void saxpy_vec(size_t n, const float a, const float *x, float *y) {
  size_t l;

  vfloat32m8_t vx, vy;

  for (; n > 0; n -= l) {
    l = __riscv_vsetvl_e32m8(n);
    vx = __riscv_vle32_v_f32m8(x, l);
    x += l;
    vy = __riscv_vle32_v_f32m8(y, l);
    vy = __riscv_vfmacc_vf_f32m8(vy, a, vx, l);
    __riscv_vse32_v_f32m8 (y, vy, l);
    y += l;
  }
}

int fp_eq(float reference, float actual, float relErr)
{
  // if near zero, do absolute error instead.
  float absErr = relErr * ((fabsf(reference) > relErr) ?
fabsf(reference) : relErr);
  return fabsf(actual - reference) < absErr;
}

int main() {
  saxpy_golden(N, 55.66, input, output_golden);
  saxpy_vec(N, 55.66, input, output);
  int pass = 1;
  for (int i = 0; i < N; i++) {
    if (!fp_eq(output_golden[i], output[i], 1e-6)) {
      printf("fail, %f=!%f\n", output_golden[i], output[i]);
```

```
      pass = 0;
    }
  }
  if (pass)
    printf("passed\n");
  return (pass == 0);
}
```

12.8. Strcmp example

```
#include "common.h"
#include <riscv_vector.h>
#include <string.h>

// reference https://github.com/riscv/riscv-v-
spec/blob/master/example/strcmp.s
int strcmp_vec(const char *source1, const char *source2) {
  const unsigned char *src1 = (const void *) source1;
  const unsigned char *src2 = (const void *) source2;
  size_t vlmax = __riscv_vsetvlmax_e8m2();
  long first_set_bit = -1;
  size_t vl;
  for (; first_set_bit < 0; src1 += vl, src2 += vl) {
    vuint8m2_t vec_src1 = __riscv_vle8ff_v_u8m2(src1, &vl, vlmax);
    vuint8m2_t vec_src2 = __riscv_vle8ff_v_u8m2(src2, &vl, vl);

    vbool4_t string_terminate = __riscv_vmseq_vx_u8m2_b4(vec_src1, 0, vl);
    vbool4_t no_equal = __riscv_vmsne_vv_u8m2_b4(vec_src1, vec_src2, vl);
    vbool4_t vec_terminate = __riscv_vmor_mm_b4(string_terminate,
no_equal, vl);

    first_set_bit = __riscv_vfirst_m_b4(vec_terminate, vl);
  }
  src1 -= vl - first_set_bit;
  src2 -= vl - first_set_bit;
  return *src1 - *src2;
}

int main() {
  const int N = 1023;
  const uint32_t seed = 0xdeadbeef;
  srand(seed);

  // data gen
  char s0[N], s1[N];
  gen_string(s0, N);
  gen_string(s1, N);

  // compute
  int golden, actual;
  golden = strcmp(s0, s1);
```

```
  actual = strcmp_vec(s0, s1);

  // compare
  puts(golden == actual ? "pass" : "fail");
}
```

12.9. Strcpy example

```
#include "common.h"
#include <assert.h>
#include <riscv_vector.h>
#include <string.h>

// reference https://github.com/riscv/riscv-v-
spec/blob/master/example/strcpy.s
char *strcpy_vec(char *destination, const char *source) {
  unsigned char *dst = (unsigned char*)destination;
  unsigned char *src = (unsigned char*)source;
  size_t vlmax = __riscv_vsetvlmax_e8m8();
  long first_set_bit = -1;
  for (size_t vl; first_set_bit < 0; src += vl, dst += vl) {
    vuint8m8_t vec_src = __riscv_vle8ff_v_u8m8(src, &vl, vlmax);

    vbool1_t string_terminate = __riscv_vmseq_vx_u8m8_b1(vec_src, 0, vl);
    vbool1_t mask = __riscv_vmsif_m_b1(string_terminate, vl);

    __riscv_vse8_v_u8m8_m(mask, dst, vec_src, vl);
    first_set_bit = __riscv_vfirst_m_b1(string_terminate, vl);
  }
  return destination;
}

int main() {
  const int N = 2000;
  const uint32_t seed = 0xdeadbeef;
  srand(seed);

  // data gen
  char s0[N];
  gen_string(s0, N);

  // compute
  char golden[N], actual[N];
  strcpy(golden, s0);
  strcpy_vec(actual, s0);

  // compare
  puts(strcmp(golden, actual) == 0 ? "pass" : "fail");
}
```

12.10. Strlen example

```
#include "common.h"
#include <riscv_vector.h>
#include <string.h>

// reference https://github.com/riscv/riscv-v-
spec/blob/master/example/strlen.s
size_t strlen_vec(char *source) {
  size_t vlmax = __riscv_vsetvlmax_e8m8();
  unsigned char *src = (unsigned char*)source;
  long first_set_bit = -1;
  size_t vl;
  for (; first_set_bit < 0; src += vl) {
    vuint8m8_t vec_src = __riscv_vle8ff_v_u8m8(src, &vl, vlmax);
    vbool1_t string_terminate = __riscv_vmseq_vx_u8m8_b1(vec_src, 0, vl);
    first_set_bit = __riscv_vfirst_m_b1(string_terminate, vl);
  }
  src -= vl - first_set_bit;
  return (size_t)((char*)src - source);
}

int main() {
  const uint32_t seed = 0xdeadbeef;
  srand(seed);

  int N = rand() % 2000;

  // data gen
  char s0[N];
  gen_string(s0, N);

  // compute
  size_t golden, actual;
  golden = strlen(s0);
  actual = strlen_vec(s0);

  // compare
  puts(golden == actual ? "pass" : "fail");
}
```

12.11. Strncpy example

```
#include "common.h"
#include <riscv_vector.h>
#include <string.h>

//reference https://github.com/riscv/riscv-v-spec/blob/master/example/strncpy.s
char *strncpy_vec(char *destination, char *source, size_t count) {
  unsigned char *dst = (unsigned char*)destination;
  unsigned char *src = (unsigned char*)source;
  long first_set_bit = -1;
  size_t vl;
  for (; first_set_bit < 0; count -= vl, src += vl, dst += vl) {
    if (count == 0)
      return destination;

    vl = __riscv_vsetvl_e8m1(count);
    vuint8m1_t vec_src = __riscv_vle8ff_v_u8m1(src, &vl, vl);

    vbool8_t string_terminate = __riscv_vmseq_vx_u8m1_b8(vec_src, 0, vl);
    vbool8_t mask = __riscv_vmsif_m_b8(string_terminate, vl);

    __riscv_vse8_v_u8m1_m(mask, dst, vec_src, vl);

    first_set_bit = __riscv_vfirst_m_b8(string_terminate, vl);
  }

  size_t tail = vl - first_set_bit;
  count += tail;
  dst -= tail;
  size_t vlmax = __riscv_vsetvlmax_e8m1();
  vuint8m1_t vec_zero = __riscv_vmv_v_x_u8m1(0, vlmax);
  do {
    size_t vl = __riscv_vsetvl_e8m1(count);
    __riscv_vse8_v_u8m1(dst, vec_zero, vl);
    count -= vl;
    dst += vl;
  } while (count > 0);

  return destination;
}

int main() {
```

```
  const int N = 1320;
  const uint32_t seed = 0xdeadbeef;
  srand(seed);

  // data gen
  char s0[N];
  gen_string(s0, N);
  char s1[] = "the quick brown fox jumps over the lazy dog";
  size_t count = strlen(s1) + (rand() % 500);

  // compute
  char golden[N], actual[N];
  strcpy(golden, s0);
  strcpy(actual, s0);
  strncpy(golden, s1, count);
  strncpy_vec(actual, s1, count);

  // compare
  puts(compare_string(golden, actual, N) ? "pass" : "fail");
}
```