
# RISC-V Vector C Intrinsic Specification Document

RISC-V Vector C Intrinsic Task Group

Version v0.13.0-draft, 2024-11-19: Draft

# Table of Contents

# Preamble

*This document is in the Development state*

*Expect potential changes. This draft specification is likely to evolve before it is accepted as a standard. Implementations based on this draft may not conform to the future standard.*

# Copyright and license information

# Contributors

This RISC-V specification has been contributed to directly or indirectly by (in alphabetical order):

Contributors to all versions of the spec in alphabetical order:

Brandon Wu, Craig Topper, Eop Chen, HanKuan Chen, HsiangKai Wang, Jerry Zhang Jian, Kito Cheng, Nick Knight, Olaf Bernstein, Roger Ferrer Ibanez, Yi-Hsiu Hsu, Zakk Chen

# Chapter 1. Introduction

The RISC-V vector C intrinsics provide users interfaces in the C language level to directly leverage the RISC-V "V" extension (*RISC-V "V" Vector Extension*, n.d.) (also abbreviated as "RVV"), with assistance from the compiler in handling instruction scheduling and register allocation. The intrinsics also aim to free users from responsibility of maintaining the correct configuration settings for the vector instruction executions.

This document uses the term "RVV" as an abbreviation for the RISC-V "V" extension. This document uses the term "the RVV specification" to indicate the RISC-V "V" extension specification.

# Chapter 2. Test macro

The `__riscv_v_intrinsic` macro is the C macro to test the compiler's support for the RISC-V "V" extension intrinsics.

This macro should be defined even if the vector extension is not enabled.

The value of the test macro is defined as its version, which is computed using the following formula. The formula is identical to what is defined in the RISC-V C API specification (*RISC-V C API Specification*, n.d.).

```
<MAJOR_VERSION> * 1,000,000 + <MINOR_VERSION> * 1,000 + <REVISION_VERSION>
```

For example, the v1.0 version should define the macro with value `1000000`.

# Chapter 3. Header file inclusion

To leverage the intrinsics in the toolchain, the header `<riscv_vector.h>` needs to be included. We suggest guarding the inclusion with the test macro.

```
#if __riscv_v_intrinsic >= 1000000
#include <riscv_vector.h>
#endif /* __riscv_v_intrinsic */
```

# Chapter 4. Availability

With `<riscv_vector.h>` included, availability of intrinsic variants depends on the required architecture of their corresponding vector instructions. The supported architecture is specified to the compiler using the `-march` option (*User Guide for RISC-V Target*, n.d.; *RISC-V Options*, n.d.).

The standard vector extensions (*RISC-V "V" Vector Extension*, n.d.) provides a set of smaller extensions for embedded use. Please check out the `Zve` extensions (*RISC-V "V" Vector Extension*, n.d.) for the varying degree of support.

For example, RVV type `vint64m1_t` and `__riscv_vle64_v_i64m1` are not available under architecture `rv64gc_zve32x`.

# Chapter 5. Control of the vector extension programming model

The intrinsics allow users to control the fields in `vtype`, as well as the rounding modes for fixed-point (`vxrm`) and floating-point (`frm`) vector computations.

In this chapter, we cover how the intrinsics embed the control of `vtype` fields in the function names. Please see Chapter 6 for the rules described in this chapter.

## 5.1. Control of effective element width (EEW) and effective LMUL (EMUL)

The RISC-V vector intrinsics' data types are strongly-typed. The vector intrinsics encode the EEW (effective-element-width) and EMUL (effective LMUL) of the destination vector register in the suffix of the function name. Users can expect the results of the vector instruction intrinsics are computed under the specified EEW and EMUL.

To see the full list of data types for the intrinsics, please see Chapter 7.

In the following example, the intrinsic will produce the semantic of a `vadd.vv` instruction, with source vector operands of `EEW=32` and `EMUL=1`, which can be observed through provided RVV data type; and produce results for the destination vector operand with `EEW=32` and `EMUL=1`.

```
vint32m1_t __riscv_vadd_vv_i32m1(vint32m1_t vs2, vint32m1_t vs1, size_t vl);
```

In the following example, the intrinsic will produce the semantic of a `vadd.vv` instruction, with source vector operands of `EEW=16` and `EMUL=1/2`, which can be observed through provided RVV data type; and produce results for the destination vector operand with `EEW=32` and `EMUL=1`.

```
vint32m1_t __riscv_vwadd_vv_i32m1(vint16mf2_t vs2, vint16mf2_t vs1, size_t vl);
```

## 5.2. Control of number of elements to be processed

The intrinsics do not directly expose the vector length control register to the intrinsics programmer. The intrinsics programmer specifies an "application vector length (AVL)" using the argument `size_t vl`. The implementation is responsible to set the correct value into the underlying vector length control register (`vl`) given the informed AVL.

*The intrinsics for instructions that behave the same with different* `vl` *settings (e.g.* `vmv.s.x`*) do not have a* `size_t vl` *argument.*

*The actual value written to the* `vl` *control register is an implementation defined behavior and is typically not known until runtime. The actual setting of* `vl`*, given the provided AVL through the parameter, follows the rules in the RVV specification. The number of elements processed can be obtained through the* `_riscv_vsetvl*` *intrinsics Section 8.1.*

## 5.3. Control of vector masking

Instructions that are available for masking have masked variant intrinsics.

The intrinsics fuse the control of vector masking (`vm`) together with the control for policy behavior (`vta`, `vma`) in the same suffix. Please check out Section 5.4 and Section 6.1 for the exact suffix that specifies a masked/unmasked vector operation along with its policy behavior.

## 5.4. Control of behavior of destination tail elements and destination inactive masked-off elements

The behavior of destination tail elements and destination inactive masked-off elements is controlled by the `vta` and `vma` bits.

Given the general assumption that target audience of the intrinsics are high performance cores, and an "undisturbed" policy will generally slow down an out-of-order core, the intrinsics have a default policy scheme of tail-agnostic and mask-agnostic (that is, `vta=1` and `vma=1`).

The intrinsics fuse the control of vector masking (`vm`) together with the control for policy behavior (`vta`, `vma`) in the same suffix. Please checkout Section 5.3 and Section 6.1 for the exact suffix that specifies a masked/unmasked vector operation along with its policy behavior.

## 5.5. Control of fixed-point rounding mode

For the fixed-point intrinsics, representing the fixed-point arithmetic instructions, the `vxrm` argument of the intrinsics indicates the rounding mode (`vxrm`) control.

The `vxrm` argument is required to be a constant integer expression. The implementation should provide the following `enum` that maps to the defined rounding mode values under Table 4 of the RVV specification.

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

*Rounding mode does not affect the computations of* `vsadd`, `vsaddu`, `vssub`, *and* `vssubu`*; therefore, the intrinsics for these instructions do not have the* `vxrm` *argument.*

*The RISC-V psABI (RISC-V Calling Conventions: Vector, n.d.) states that* `vxrm` *is not preserved across calls. Optimization for reducing the number of redundant writes to* `vxrm` *is a compiler and system specific issue.*

*This specification does not provide support for manipulating the* `vxsat` *CSR. Since vxsat is not needed by a large majority of fixed-point code, we believe this specification is broadly useful as-is. Nevertheless, we expect that a future extension will define an additional set of fixed-point intrinsics that update* `vxsat` *in a specified manner, along with intrinsics to*

*explicitly read and write* `vxsat`*. These new intrinsics would be interoperable with the intrinsics in this specification.*

*The value of the* `vxsat` *after a fixed-point intrinsic is UNSPECIFIED.*

# 5.6. Control of floating-point rounding mode

For the floating-point intrinsics, representing the floating-point arithmetic instructions, the intrinsics have two variants: *Implicit FP rounding mode* and *Explicit FP Rounding mode* intrinsics.

*Control of the floating-point accrued exceptions flag fields (*`fflag`*) (RISC-V "F" Vector Extension, n.d.) is not yet covered in the vector intrinsics v1.0. We plan to support it in follow-up versions in a compatible way with existing intrinsics in v1.0.*

## 5.6.1. Implicit FP rounding mode intrinsics

The implicit FP rounding mode intrinsics behave like any C-language floating-point expressions, using the default rounding mode when `FENV_ACCESS` is off, and using the `fenv` dynamic rounding mode when `FENV_ACCESS` is on.

*Both GNU and LLVM compilers generate scalar floating-point instructions using dynamic rounding mode, relying on the environment initialization to set* `frm` *to* `RNE` *(specified as "roundTiesToEven" in IEEE-754 (a.k.a. IEC 60559)) (ANSI/IEEE Std 754-2008, IEEE Standard for Floating-Point Arithmetic, 2008).*

*The implicit FP rounding mode intrinsics are intended to be used regardless of* `FENV_ACCESS`*. They are provided when* `FENV_ACCESS` *is on for the (few) programmers who are already using* `fenv`*; and they are provided when* `FENV_ACCESS` *is off for the (vast majority of) programmers who prefer the default rounding mode.*

## 5.6.2. Explicit FP rounding mode intrinsics

The explicit FP rounding mode intrinsics contain the `frm` argument which indicates the rounding mode (`frm`) (*RISC-V "F" Vector Extension*, n.d.) control. The floating-point intrinsics with the `frm` argument are followed by an `_rm` suffix in the function name.

The `frm` argument is required to be a constant integer expression. The implementation should provide the following `enum` that maps to the defined rounding mode values under RISC-V ISA Manual Table 8.1 (*RISC-V Calling Conventions: Vector*, n.d.).

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

*The explicit FP rounding mode intrinsics are intended to be used when* `FENV_ACCESS` *is off, enabling more aggressive optimization while still providing the programmer with control over the rounding mode. Using explicit FP rounding mode intrinsics when* `FENV_ACCESS` *is on will still work correctly, but is expected to lead to extra saving/restoring of* `frm`*, that*

*could be avoided by using* `fenv` *functionality and implicit FP rounding mode intrinsics.*

# Chapter 6. Naming scheme

The naming scheme of the intrinsics expresses the users' control of fields in `vtype`, `vl`, and rounding modes for fixed-point and floating-point vector computations. For details of these CSR controls, please see [control-of-vector-programming-mode].

As mentioned in Section 5.3 and Section 5.4, the intrinsics fuses the control of `vm`, `vta`, and `vma` into the same suffix. Section 6.1 enumerates the exact suffixes. You may find where these suffixes are appended in Section 6.2.

The intrinsics can be split into two major types, called "explicit (non-overloaded) intrinsics" and "implicit (overloaded) intrinsics".

The explicit (non-overloaded) intrinsics embed the control described in Chapter 5 in the function name. This scheme gives intrinsic codebase more readability as the execution states are explicitly specified in the code.

The implicit (overloaded) intrinsics, on the contrary, omit the explicit specifications for `vtype` control. The implicit (overloaded) intrinsics aim to provide a generic interface to let users put values of different EEW and EMUL as the input argument.

This section covers the general naming rule of the two types of intrinsics accordingly. Then, this section also enumerates the exceptions and the rationales behind them in Section 6.3 and Section 6.5.

## 6.1. Policy and masked naming scheme

With the default policy scheme mentioned under Section 5.4, each intrinsic provides corresponding variants for their available control of `vm`, `vta` and `vma`. The following list enumerates the possible suffixes.

- No suffix: Represents an unmasked (`vm=1`) vector operation with tail-agnostic (`vta=1`)
- `_tu` suffix: Represents an unmasked (`vm=1`) vector operation with tail-undisturbed (`vta=0`) policy
- `_m` suffix: Represents a masked (`vm=0`) vector operation with tail-agnostic (`vta=1`), mask-agnostic (`vma=1`) policy
- `_tum` suffix: Represents a masked (`vm=0`) vector operation with tail-undisturbed (`vta=0`), mask-agnostic (`vma=1`) policy
- `_mu` suffix: Represents a masked (`vm=0`) vector operation with tail-agnostic (`vta=1`), mask-undisturbed (`vma=0`) policy
- `_tumu` suffix: Represents a masked (`vm=0`) vector operation with tail-undisturbed (`vta=0`), mask-undisturbed (`vma=0`) policy

Using `vadd` with EEW=32 and EMUL=1 as an example, the variants are:

```
// vm=1, vta=1
vint32m1_t __riscv_vadd_vv_i32m1(vint32m1_t vs2, vint32m1_t vs1, size_t vl);
// vm=1, vta=0
vint32m1_t __riscv_vadd_vv_i32m1_tu(vint32m1_t vd, vint32m1_t vs2,
                                    vint32m1_t vs1, size_t vl);
// vm=0, vta=1, vma=1
vint32m1_t __riscv_vadd_vv_i32m1_m(vbool32_t vm, vint32m1_t vs2, vint32m1_t vs1,
                                   size_t vl);
```

```
// vm=0, vta=0, vma=1
vint32m1_t __riscv_vadd_vv_i32m1_tum(vbool32_t vm, vint32m1_t vd,
                                     vint32m1_t vs2, vint32m1_t vs1, size_t vl);
// vm=0, vta=1, vma=0
vint32m1_t __riscv_vadd_vv_i32m1_mu(vbool32_t vm, vint32m1_t vd, vint32m1_t vs2,
                                    vint32m1_t vs1, size_t vl);
// vm=0, vta=0, vma=0
vint32m1_t __riscv_vadd_vv_i32m1_tumu(vbool32_t vm, vint32m1_t vd,
                                      vint32m1_t vs2, vint32m1_t vs1,
                                      size_t vl);
```

*When policy is set to "agnostic", there is no guarantee of what will be in the tail/masked-off elements. Under this policy, users should not assume the values within to be deterministic.*

*Pseudo intrinsics mentioned under Chapter 8 do not map to real vector instructions. Therefore these intrinsics are not affected by the policy setting, nor do they have intrinsic variants of the suffixes listed above.*

## 6.2. Explicit (Non-overloaded) naming scheme

In general, the intrinsics are encoded as the following. The intrinsics under this naming scheme are the "non-overloaded intrinsics", which in parallel we have the "overloaded intrinsics" defined under Section 6.4.

The naming rules are as follows.

```
__riscv_{V_INSTRUCTION_MNEMONIC}_{OPERAND_MNEMONIC}_{RETURN_TYPE}_{ROUND_MODE}_{POLICY}{(...)
```

- `OPERAND_MNEMONIC` are like `v`, `vv`, `vx`, `vs`, `vvm`, `vxm`
- `RETURN_TYPE` depends on whether the return type of the vector instruction is a mask register...
  - For intrinsics that represents instructions with a non-mask destination register:
    - `EEW` is one of `i8 | i16 | i32 | i64 | u8 | u16 | u32 | u64 | f16 | f32 | f64`.
    - `EMUL` is one of `m1 | m2 | m4 | m8 | mf2 | mf4 | mf8`.
    - Chapter 7 explains the limited enumeration of EEW-EMUL pairs.
  - For intrinsics that represent intrinsics with a mask destination register:
    - `RETURN_TYPE` is one of `b1 | b2 | b4 | b8 | b16 | b32 | b64`, which is derived from the ratio `EEW/EMUL`.
- `V_INSTRUCTION_MNEMONIC` are like `vadd`, `vfmacc`, `vsadd`.
- `ROUND_MODE` is the `_rm` suffix mentioned in Section 5.6.2. Other intrinsics do not have this suffix.
- `POLICY` are enumerated under Section 6.1.

The general naming scheme is not sufficient to express all intrinsics. The exceptions are enumerated in the proceeding section Section 6.3.

# 6.3. Exceptions in the explicit (non-overloaded) naming scheme

This section enumerates the exceptions in the explicit (non-overloaded) naming scheme.

## 6.3.1. Scalar move instructions

Only encoding the return type will cause naming collisions for the permutation instruction intrinsics. The intrinsics encode the input vector type and the output scalar type in the suffix.

```
int8_t __riscv_vmv_x_s_i8m1_i8 (vint8m1_t vs2, size_t vl);
int8_t __riscv_vmv_x_s_i8m2_i8 (vint8m2_t vs2, size_t vl);
int8_t __riscv_vmv_x_s_i8m4_i8 (vint8m4_t vs2, size_t vl);
int8_t __riscv_vmv_x_s_i8m8_i8 (vint8m8_t vs2, size_t vl);
```

## 6.3.2. Reduction instructions

Only encoding the return type will cause naming collisions for the reduction instruction intrinsics. The intrinsics encode the input vector type and the output vector type in the suffix.

```
vint8m1_t __riscv_vredsum_vs_i8m1_i8m1(vint8m1_t vs2, vint8m1_t vs1, size_t vl);
vint8m1_t __riscv_vredsum_vs_i8m2_i8m1(vint8m2_t vs2, vint8m1_t vs1, size_t vl);
vint8m1_t __riscv_vredsum_vs_i8m4_i8m1(vint8m4_t vs2, vint8m1_t vs1, size_t vl);
vint8m1_t __riscv_vredsum_vs_i8m8_i8m1(vint8m8_t vs2, vint8m1_t vs1, size_t vl);
```

## 6.3.3. Add-with-carry / Subtract-with-borrow instructions

Only encoding the return type will cause naming collisions for the reduction instruction intrinsics. The intrinsics encode the input vector type and the output mask vector type in the suffix.

```
vbool64_t __riscv_vmadc_vvm_i8mf8_b64(vint8mf8_t vs2, vint8mf8_t vs1,
                                      vbool64_t v0, size_t vl);
vbool64_t __riscv_vmadc_vvm_i16mf4_b64(vint16mf4_t vs2, vint16mf4_t vs1,
                                       vbool64_t v0, size_t vl);
vbool64_t __riscv_vmadc_vvm_i32mf2_b64(vint32mf2_t vs2, vint32mf2_t vs1,
                                       vbool64_t v0, size_t vl);
vbool64_t __riscv_vmadc_vvm_i64m1_b64(vint64m1_t vs2, vint64m1_t vs1,
                                      vbool64_t v0, size_t vl);
```

## 6.3.4. `vreinterpret`, `vlmul_trunc`/`vlmul_ext`, and `vset`/`vget`

Only encoding the return type will cause naming collisions for these pseudo instructions. The intrinsics encode the input vector type before the return type in the suffix.

The following shows an example with `__riscv_vreinterpret_v` of `vint32m1_t` input vector type.

```
vfloat32m1_t __riscv_vreinterpret_v_i32m1_f32m1 (vint32m1_t src);
vuint32m1_t __riscv_vreinterpret_v_i32m1_u32m1 (vint32m1_t src);
vint8m1_t __riscv_vreinterpret_v_i32m1_i8m1 (vint32m1_t src);
vint16m1_t __riscv_vreinterpret_v_i32m1_i16m1 (vint32m1_t src);
vint64m1_t __riscv_vreinterpret_v_i32m1_i64m1 (vint32m1_t src);
vbool64_t __riscv_vreinterpret_v_i32m1_b64 (vint32m1_t src);
```

```
vbool32_t __riscv_vreinterpret_v_i32m1_b32 (vint32m1_t src);
vbool16_t __riscv_vreinterpret_v_i32m1_b16 (vint32m1_t src);
vbool8_t __riscv_vreinterpret_v_i32m1_b8 (vint32m1_t src);
vbool4_t __riscv_vreinterpret_v_i32m1_b4 (vint32m1_t src);
```

## 6.4. Implicit (Overloaded) naming scheme

The implicit (overloaded) interface aims to provide a generic interface that takes values of different EEW and EMUL as the input. Therefore, the implicit intrinsics omit the EEW and EMUL encoded in the function name. The `_rm` prefix for explicit FP rounding mode intrinsics (Section 5.6) is also omitted. The intrinsics under this scheme are the "overloaded intrinsics", which in parallel we have the "non-overloaded intrinsics" defined under Section 6.2.

Take the vector addition (`vadd`) instruction intrinsics as an example, stripping off the operand mnemonics and encoded EEW, EMUL information, the intrinsics provides the following overloaded interfaces.

```
vint32m1_t __riscv_vadd(vint32m1_t v0, vint32m1_t v1, size_t vl);
vint16m4_t __riscv_vadd(vint16m4_t v0, vint16m4_t v1, size_t vl);
```

Since the main intent is to let the users put different value(s) of EEW and EMUL as input argument(s), the overloaded intrinsics do not omit the policy suffix. That is, the suffix listed under Section 5.4 is not omitted and is still encoded in the function name.

The masked variants with the default policy shares the same interface with the unmasked variants with the default policy. They do not have any trailing suffixes.

Take the vector floating-point add (`vfadd`) as an example, the intrinsics provides the following overloaded interfaces.

```
vfloat32m1_t __riscv_vfadd(vfloat32m1_t vs2, vfloat32m1_t vs1, size_t vl);
vfloat32m1_t __riscv_vfadd(vbool32_t vm, vfloat32m1_t vs2, vfloat32m1_t vs1,
                           size_t vl);
vfloat32m1_t __riscv_vfadd(vfloat32m1_t vs2, vfloat32m1_t vs1, unsigned int frm,
                           size_t vl);
vfloat32m1_t __riscv_vfadd(vbool32_t vm, vfloat32m1_t vs2, vfloat32m1_t vs1,
                           unsigned int frm, size_t vl);
vfloat32m1_t __riscv_vfadd_tu(vfloat32m1_t vd, vfloat32m1_t vs2,
                              vfloat32m1_t vs1, size_t vl);
vfloat32m1_t __riscv_vfadd_tum(vbool32_t vm, vfloat32m1_t vd, vfloat32m1_t vs2,
                               vfloat32m1_t vs1, size_t vl);
vfloat32m1_t __riscv_vfadd_tumu(vbool32_t vm, vfloat32m1_t vd, vfloat32m1_t vs2,
                                vfloat32m1_t vs1, size_t vl);
vfloat32m1_t __riscv_vfadd_mu(vbool32_t vm, vfloat32m1_t vd, vfloat32m1_t vs2,
                              vfloat32m1_t vs1, size_t vl);
vfloat32m1_t __riscv_vfadd_tu(vfloat32m1_t vd, vfloat32m1_t vs2,
                              vfloat32m1_t vs1, unsigned int frm, size_t vl);
vfloat32m1_t __riscv_vfadd_tum(vbool32_t vm, vfloat32m1_t vd, vfloat32m1_t vs2,
                               vfloat32m1_t vs1, unsigned int frm, size_t vl);
vfloat32m1_t __riscv_vfadd_tumu(vbool32_t vm, vfloat32m1_t vd, vfloat32m1_t vs2,
                                vfloat32m1_t vs1, unsigned int frm, size_t vl);
vfloat32m1_t __riscv_vfadd_mu(vbool32_t vm, vfloat32m1_t vd, vfloat32m1_t vs2,
                              vfloat32m1_t vs1, unsigned int frm, size_t vl);
```

The naming scheme to prune everything except the instruction mnemonics is not available for all the intrinsics. Please see Section 6.5 for overloaded intrinsics with irregular naming patterns.

Due to the limitations of the C language (without the aid of features like C++ templates), some intrinsics do not have an overloaded version. Therefore these intrinsics do not possess a simplified, EEW/EMUL-omitted interface. Please see Section 6.6 for more detail.

# 6.5. Exceptions in the implicit (overloaded) naming scheme

The following intrinsics have an irregular naming pattern.

## 6.5.1. Widening instructions

Widening instruction intrinsics (e.g. `vwadd`) have the same return type but different types of arguments. The operand mnemonics are encoded into their overloaded versions to help distinguish them.

```
vint32m1_t __riscv_vwadd_vv(vint16mf2_t vs2, vint16mf2_t vs1, size_t vl);
vint32m1_t __riscv_vwadd_vx(vint16mf2_t vs2, int16_t rs1, size_t vl);
vint32m1_t __riscv_vwadd_wv(vint32m1_t vs2, vint16mf2_t vs1, size_t vl);
vint32m1_t __riscv_vwadd_wx(vint32m1_t vs2, int16_t rs1, size_t vl);
```

## 6.5.2. Type-convert instructions

Type-convert instruction intrinsics (e.g. `vfcvt.x.f`, `vfcvt.xu.f`, `vfcvt.rtz.xu.f`) encode the returning type mnemonics into their overloaded variants to help distinguish them.

The following shows how `_x`, `_rtz_x`, `_xu`, and `_rtz_xu` are appended to the suffix for distinction.

```
vint32m1_t __riscv_vfcvt_x (vfloat32m1_t src, size_t vl);
vint32m1_t __riscv_vfcvt_rtz_x (vfloat32m1_t src, size_t vl);
vuint32m1_t __riscv_vfcvt_xu (vfloat32m1_t src, size_t vl);
vuint32m1_t __riscv_vfcvt_rtz_xu (vfloat32m1_t src, size_t vl);
```

## 6.5.3. `vreinterpret`, LMUL truncate/extension, and `vset`/`vget`

These pseudo intrinsics encode the return type (e.g. `__riscv_vreinterpret_b8`) into their overloaded variants to help distinguish them.

The following shows how the return type is appended to the suffix for distinction.

```
vfloat32m1_t __riscv_vreinterpret_f32m1 (vint32m1_t src);
vuint32m1_t __riscv_vreinterpret_u32m1 (vint32m1_t src);
vint8m1_t __riscv_vreinterpret_i8m1 (vint32m1_t src);
vint16m1_t __riscv_vreinterpret_i16m1 (vint32m1_t src);
vint64m1_t __riscv_vreinterpret_i64m1 (vint32m1_t src);
vbool64_t __riscv_vreinterpret_b64 (vint32m1_t src);
vbool32_t __riscv_vreinterpret_b32 (vint32m1_t src);
vbool16_t __riscv_vreinterpret_b16 (vint32m1_t src);
vbool8_t __riscv_vreinterpret_b8 (vint32m1_t src);
vbool4_t __riscv_vreinterpret_b4 (vint32m1_t src);
```

# 6.6. Un-supported intrinsics for implicit (overloaded) naming scheme

Due to the limitation of the C language (without the aid of features like C++ templates), some intrinsics do not have an overloaded version. Intrinsics with characteristics of either of the following do not possess an overloaded version.

- Intrinsics with input arguments that are all scalar types and scalar types alone (e.g. unmasked vector load instruction intrinsics, `vmv.s.x`)
- Intrinsics with `vl` as the only argument (e.g. `vmclr`, `vmset`, `vid`)
- Intrinsics with vector boolean input(s), returning a vector non-boolean vector type (e.g. `viota`)

# Chapter 7. Type system

The intrinsics are designed to be strongly-typed. The intrinsics provide `vreinterpret` intrinsics to help users go across the strongly-typed scheme if necessary.

Non-mask (integer and floating-point) data types have SEW and LMUL encoded.

## 7.1. Integer types

Integer types have EEW and EMUL encoded into the type. The first row describes the EMUL and the first column describes the data type and element width of the scalar type.

Types with an asterisk (*) are available when `ELEN >= 64` (that is, unavailable under `Zve32*` and require at least `Zve64x`).

| Types | EMUL=1/8 | EMUL=1/4 | EMUL=1/ 2 | EMUL=1 | EMUL=2 | EMUL=4 | EMUL=8 |
|---|---|---|---|---|---|---|---|
| int8_t | vint8mf8_t* | vint8mf4_t | vint8mf2_t | vint8m1_t | vint8m2_t | vint8m4_t | vint8m8_t |
| int16_t | N/A | vint16mf4_t* | vint16mf2_t | vint16m1_t | vint16m2_t | vint16m4_t | vint16m8_t |
| int32_t | N/A | N/A | vint32mf2_t* | vint32m1_t | vint32m2_t | vint32m4_t | vint32m8_t |
| int64_t | N/A | N/A | N/A | vint64m1_t* | vint64m2_t* | vint64m4_t* | vint64m8_t* |
| uint8_t | vuint8mf8_t* | vuint8mf4_t | vuint8mf2_t | vuint8m1_t | vuint8m2_t | vuint8m4_t | vuint8m8_t |
| uint16_t | N/A | vuint16mf4_t* | vuint16mf2_t | vuint16m1_t | vuint16m2_t | vuint16m4_t | vuint16m8_t |
| uint32_t | N/A | N/A | vuint32mf2_t* | vuint32m1_t | vuint32m2_t | vuint32m4_t | vuint32m8_t |
| uint64_t | N/A | N/A | N/A | vuint64m1_t* | vuint64m2_t* | vuint64m4_t* | vuint64m8_t* |

*Table 1. Integer types*

## 7.2. Floating-point types

*This specification uses `_Float16` to designate IEEE-754 binary16, `float` to designate IEEE-754 binary32 and `double` to designate IEEE-754 binary64.*

Floating-point types have EEW and EMUL encoded into the type. The first row describes the EMUL and the first column describes the data type and element width of the scalar type.

Floating-point types with element widths of 16 (Types=`_Float16`) require the `zvfh` and `zvfhmin` extension to be specified in the architecture.

Floating-point types with element widths of 32 (Types=`float`) require the `zve32f` extension to be specified in the architecture.

Floating-point types with element widths of 64 (Types=`double`) require the `zve64d` extension to be specified in the architecture.

Types with an asterisk (*) are available when `ELEN >= 64` (that is, unavailable under `Zve32f` and require at least `Zve64f`).

| Types | EMUL=1/8 | EMUL=1/4 | EMUL=1/ 2 | EMUL=1 | EMUL=2 | EMUL=4 | EMUL=8 |
|---|---|---|---|---|---|---|---|
| _Float16 | N/A | vfloat16mf4_t* | vfloat16mf2_t | vfloat16m1_t | vfloat16m2_t | vfloat16m4_t | vfloat16m8_t |
| float | N/A | N/A | vfloat32mf2_t* | vfloat32m1_t | vfloat32m2_t | vfloat32m4_t | vfloat32m8_t |
| double | N/A | N/A | N/A | vfloat64m1_t | vfloat64m2_t | vfloat64m4_t | vfloat64m8_t |

*Table 2. Floating-point types*

# 7.3. Mask types

Mask types have the ratio that is derived from `EEW/EMUL` encoded into the type. The mask types represent mask register values that follows the Mask Register Layout.

Types with an asterisk (*) are available when `ELEN >= 64` (that is, unavailable under `Zve32x` and require at least `Zve64x`).

| Types | n = 1 | n = 2 | n = 4 | n = 8 | n = 16 | n = 32 | n = 64 |
|---|---|---|---|---|---|---|---|
| bool | vbool1_t | vbool2_t | vbool4_t | vbool8_t | vbool16_t | vbool32_t | vbool64_t* |

*Table 3. Mask types*

# 7.4. Tuple type

Tuple types encode `SEW`, `LMUL`, and `NFIELD` into the data type.

These types are utilized through the segment load/store instruction intrinsics along with getters Section 8.5 and setters Section 8.6 to extract/combine them. The types listed in Section 7.1 and Section 7.2 all have tuple types. Types under the combination of `LMUL`, `NFIELD` follows the restriction by the RVV specification, `EMUL * NFIELDS ≤ 8`.

Availability of the tuple types follows the availability of their corresponding non-tuple (`NFIELD=1`) types.

| Non-tuple Types (NFILED=1) | NFIELD=2 | NFIELD=3 | NFIELD=4 | NFIELD=5 | NFIELD=6 | NFIELD=7 | NFIELD=8 |
|---|---|---|---|---|---|---|---|
| vint8mf8_t | vint8mf8x2_t | vint8mf8x3_t | vint8mf8x4_t | vint8mf8x5_t | vint8mf8x6_t | vint8mf8x7_t | vint8mf8x8_t |
| vuint8mf8_t | vuint8mf8x2_t | vuint8mf8x3_t | vuint8mf8x4_t | vuint8mf8x5_t | vuint8mf8x6_t | vuint8mf8x7_t | vuint8mf8x8_t |

*Table 4. Tuple types (EMUL=1/8)*

| Non-tuple Types (NFILED=1) | NFIELD=2 | NFIELD=3 | NFIELD=4 | NFIELD=5 | NFIELD=6 | NFIELD=7 | NFIELD=8 |
|---|---|---|---|---|---|---|---|
| vint8mf4_t | vint8mf4x2_t | vint8mf4x3_t | vint8mf4x4_t | vint8mf4x5_t | vint8mf4x6_t | vint8mf4x7_t | vint8mf4x8_t |
| vuint8mf4_t | vuint8mf4x2_t | vuint8mf4x3_t | vuint8mf4x4_t | vuint8mf4x5_t | vuint8mf4x6_t | vuint8mf4x7_t | vuint8mf4x8_t |
| vint16mf4_t | vint16mf4x2_t | vint16mf4x3_t | vint16mf4x4_t | vint16mf4x5_t | vint16mf4x6_t | vint16mf4x7_t | vint16mf4x8_t |
| vuint16mf4_t | vuint16mf4x2_t | vuint16mf4x3_t | vuint16mf4x4_t | vuint16mf4x5_t | vuint16mf4x6_t | vuint16mf4x7_t | vuint16mf4x8_t |
| vfloat16mf4_t | vfloat16mf4x2_t | vfloat16mf4x3_t | vfloat16mf4x4_t | vfloat16mf4x5_t | vfloat16mf4x6_t | vfloat16mf4x7_t | vfloat16mf4x8_t |

*Table 5. Tuple types (EMUL=1/4)*

| Non-tuple Types (NFILED=1) | NFIELD=2 | NFIELD=3 | NFIELD=4 | NFIELD=5 | NFIELD=6 | NFIELD=7 | NFIELD=8 |
|---|---|---|---|---|---|---|---|
| vint8mf2_t | vint8mf2x2_t | vint8mf2x3_t | vint8mf2x4_t | vint8mf2x5_t | vint8mf2x6_t | vint8mf2x7_t | vint8mf2x8_t |
| vuint8mf2_t | vuint8mf2x2_t | vuint8mf2x3_t | vuint8mf2x4_t | vuint8mf2x5_t | vuint8mf2x6_t | vuint8mf2x7_t | vuint8mf2x8_t |
| vint16mf2_t | vint16mf2x2_t | vint16mf2x3_t | vint16mf2x4_t | vint16mf2x5_t | vint16mf2x6_t | vint16mf2x7_t | vint16mf2x8_t |
| vuint16mf2_t | vuint16mf2x2_t | vuint16mf2x3_t | vuint16mf2x4_t | vuint16mf2x5_t | vuint16mf2x6_t | vuint16mf2x7_t | vuint16mf2x8_t |
| vint32mf2_t | vint32mf2x2_t | vint32mf2x3_t | vint32mf2x4_t | vint32mf2x5_t | vint32mf2x6_t | vint32mf2x7_t | vint32mf2x8_t |
| vuint32mf2_t | vuint32mf2x2_t | vuint32mf2x3_t | vuint32mf2x4_t | vuint32mf2x5_t | vuint32mf2x6_t | vuint32mf2x7_t | vuint32mf2x8_t |
| vfloat16mf2_t | vfloat16mf2x2_t | vfloat16mf2x3_t | vfloat16mf2x4_t | vfloat16mf2x5_t | vfloat16mf2x6_t | vfloat16mf2x7_t | vfloat16mf2x8_t |
| vfloat32mf2_t | vfloat32mf2x2_t | vfloat32mf2x3_t | vfloat32mf2x4_t | vfloat32mf2x5_t | vfloat32mf2x6_t | vfloat32mf2x7_t | vfloat32mf2x8_t |

*Table 6. Tuple types (EMUL=1/2)*

| Non-tuple Types (NFILED=1) | NFIELD=2 | NFIELD=3 | NFIELD=4 | NFIELD=5 | NFIELD=6 | NFIELD=7 | NFIELD=8 |
|---|---|---|---|---|---|---|---|
| vint8m1_t | vint8m1x2_t | vint8m1x3_t | vint8m1x4_t | vint8m1x5_t | vint8m1x6_t | vint8m1x7_t | vint8m1x8_t |
| vuint8m1_t | vuint8m1x2_t | vuint8m1x3_t | vuint8m1x4_t | vuint8m1x5_t | vuint8m1x6_t | vuint8m1x7_t | vuint8m1x8_t |
| vint16m1_t | vint16m1x2_t | vint16m1x3_t | vint16m1x4_t | vint16m1x5_t | vint16m1x6_t | vint16m1x7_t | vint16m1x8_t |
| vuint16m1_t | vuint16m1x2_t | vuint16m1x3_t | vuint16m1x4_t | vuint16m1x5_t | vuint16m1x6_t | vuint16m1x7_t | vuint16m1x8_t |
| vint32m1_t | vint32m1x2_t | vint32m1x3_t | vint32m1x4_t | vint32m1x5_t | vint32m1x6_t | vint32m1x7_t | vint32m1x8_t |
| vuint32m1_t | vuint32m1x2_t | vuint32m1x3_t | vuint32m1x4_t | vuint32m1x5_t | vuint32m1x6_t | vuint32m1x7_t | vuint32m1x8_t |
| vint64m1_t | vint64m1x2_t | vint64m1x3_t | vint64m1x4_t | vint64m1x5_t | vint64m1x6_t | vint64m1x7_t | vint64m1x8_t |
| vuint64m1_t | vuint64m1x2_t | vuint64m1x3_t | vuint64m1x4_t | vuint64m1x5_t | vuint64m1x6_t | vuint64m1x7_t | vuint64m1x8_t |
| vfloat16m1_t | vfloat16m1x2_t | vfloat16m1x3_t | vfloat16m1x4_t | vfloat16m1x5_t | vfloat16m1x6_t | vfloat16m1x7_t | vfloat16m1x8_t |
| vfloat32m1_t | vfloat32m1x2_t | vfloat32m1x3_t | vfloat32m1x4_t | vfloat32m1x5_t | vfloat32m1x6_t | vfloat32m1x7_t | vfloat32m1x8_t |
| vfloat64m1_t | vfloat64m1x2_t | vfloat64m1x3_t | vfloat64m1x4_t | vfloat64m1x5_t | vfloat64m1x6_t | vfloat64m1x7_t | vfloat64m1x8_t |

*Table 7. Tuple types (EMUL=1)*

| Non-tuple Types (NFILED=1) | NFIELD=2 | NFIELD=3 | NFIELD=4 | NFIELD=5 | NFIELD=6 | NFIELD=7 | NFIELD=8 |
|---|---|---|---|---|---|---|---|
| vint8m2_t | vint8m2x2_t | vint8m2x3_t | vint8m2x4_t | N/A | N/A | N/A | N/A |
| vuint8m2_t | vuint8m2x2_t | vuint8m2x3_t | vuint8m2x4_t | N/A | N/A | N/A | N/A |
| vint16m2_t | vint16m2x2_t | vint16m2x3_t | vint16m2x4_t | N/A | N/A | N/A | N/A |
| vuint16m2_t | vuint16m2x2_t | vuint16m2x3_t | vuint16m2x4_t | N/A | N/A | N/A | N/A |
| vint32m2_t | vint32m2x2_t | vint32m2x3_t | vint32m2x4_t | N/A | N/A | N/A | N/A |
| vuint32m2_t | vuint32m2x2_t | vuint32m2x3_t | vuint32m2x4_t | N/A | N/A | N/A | N/A |
| vint64m2_t | vint64m2x2_t | vint64m2x3_t | vint64m2x4_t | N/A | N/A | N/A | N/A |
| vuint64m2_t | vuint64m2x2_t | vuint64m2x3_t | vuint64m2x4_t | N/A | N/A | N/A | N/A |
| vfloat16m2_t | vfloat16m2x2_t | vfloat16m2x3_t | vfloat16m2x4_t | N/A | N/A | N/A | N/A |
| vfloat32m2_t | vfloat32m2x2_t | vfloat32m2x3_t | vfloat32m2x4_t | N/A | N/A | N/A | N/A |
| vfloat64m2_t | vfloat64m2x2_t | vfloat64m2x3_t | vfloat64m2x4_t | N/A | N/A | N/A | N/A |

*Table 8. Tuple types (EMUL=2)*

| Non-tuple Types (NFILED=1) | NFIELD=2 | NFIELD=3 | NFIELD=4 | NFIELD=5 | NFIELD=6 | NFIELD=7 | NFIELD=8 |
|---|---|---|---|---|---|---|---|
| vint8m4_t | vint8m4x2_t | N/A | N/A | N/A | N/A | N/A | N/A |
| vuint8m4_t | vuint8m4x2_t | N/A | N/A | N/A | N/A | N/A | N/A |
| vint16m4_t | vint16m4x2_t | N/A | N/A | N/A | N/A | N/A | N/A |
| vuint16m4_t | vuint16m4x2_t | N/A | N/A | N/A | N/A | N/A | N/A |
| vint32m4_t | vint32m4x2_t | N/A | N/A | N/A | N/A | N/A | N/A |
| vuint32m4_t | vuint32m4x2_t | N/A | N/A | N/A | N/A | N/A | N/A |
| vint64m4_t | vint64m4x2_t | N/A | N/A | N/A | N/A | N/A | N/A |
| vuint64m4_t | vuint64m4x2_t | N/A | N/A | N/A | N/A | N/A | N/A |
| vfloat16m4_t | vfloat16m4x2_t | N/A | N/A | N/A | N/A | N/A | N/A |
| vfloat32m4_t | vfloat32m4x2_t | N/A | N/A | N/A | N/A | N/A | N/A |
| vfloat64m4_t | vfloat64m4x2_t | N/A | N/A | N/A | N/A | N/A | N/A |

*Table 9. Tuple types (EMUL=4)*

# Chapter 8. Pseudo intrinsics

Pseudo intrinsics provide additional utility functions to assist users in manipulating across intrinsic types. They do not map to any specific RVV instruction. The specific mapping to actual instructions is given in the description of each pseudo intrinsic.

## 8.1. vsetvl

The `vsetvl` intrinsics return the number of elements processed in a stripmining loop when provided with the element width and LMUL in the intrinsic suffix. This pseudo intrinsic is typically mapped to `vsetvli` or `vsetivli` instructions.

*The implementation must respect the ratio between SEW and LMUL given to the intrinsic. On the other hand, as mentioned in Section 5.2, the* `vsetvl` *intrinsics do not necessarily map to the emission a* `vsetvli` *or* `vsetivli` *instruction of that exact SEW and LMUL provided. The actual value written to the* `vl` *control register is an implementation defined behavior and typically not known until runtime.*

## 8.2. vsetvlmax

The `vsetvlmax` intrinsics return `VLMAX` when provided with the element width and LMUL in the intrinsic suffix. This pseudo intrinsic is typically mapped to the `vsetvli` instruction.

*As mentioned in Section 5.2, the* `vsetvlmax` *intrinsics do not necessarily map to the emission a* `vsetvli` *instruction of that exact SEW and LMUL provided. The actual value written to the* `vl` *control register is an implementation defined behavior and typically not known until runtime.*

## 8.3. vreinterpret

The `vreinterpret` intrinsics are provided for users to transition across the strongly-typed scheme. The intrinsic is limited to conversion between types operating upon the same number of registers. These intrinsics are not mapped to any instruction because reinterpretation of registers is a no-operation.

These pseudo intrinsics do not alter the bits held by a register. Please use `vfcvt/v(f)wcvt/v(f)ncvt` intrinsics if you seek to extend, narrow, or perform real float/interger type conversions for the values.

## 8.4. vundefined

The `vundefined` intrinsics are placeholders to represent unspecified values in variable initialization, or as arguments of `vset` and `vcreate`. These pseudo intrinsics are not mapped to any instruction.

## 8.5. vget

The `vget` intrinsics allow users to obtain small LMUL values from larger LMUL ones. The `vget` intrinsics also allows users to extract non-tuple (`NFIELD=1`) types from tuple (`NFIELD>1`) types after segment load intrinsics. The index provided must be a constant known at compile time.

These pseudo intrinsics do not map to any real instruction. The compiler may emit zero or more instructions to implement the semantics of these pseudo intrinsics. The precise set of instructions emitted is a compiler optimization issue.

## 8.6. vset

The `vset` intrinsics allow users to combine small LMUL values into larger LMUL ones. The `vset` intrinsics also allows users to combine non-tuple (`NFIELD=1`) types to tuple (`NFIELD>1`) types for segment store intrinsics. The index provided must be a constant known at compile time.

These pseudo intrinsics do not map to any real instruction. The compiler may emit zero or more instructions to implement the semantics of these pseudo intrinsics. The precise set of instructions emitted is a compiler optimization issue.

## 8.7. vlmul_trunc

The `vlmul_trunc` intrinsics are syntactic sugar for RVV vector types other than tuples and have the same semantic as `vget` with the `index` operand having the value `0`.

## 8.8. vlmul_ext

The `vlmul_ext` intrinsics are syntactic sugar for RVV vector types other than tuples and have the same semantic as `vset` with the `index` operand having the value `0`.

## 8.9. vcreate

The `vcreate` intrinsics are syntactic sugar for the creation of values of RVV types. They have the same semantic as multiple `vset` pseudo intrinsics filling in values accordingly.

*Example 1. Pseudo intrinsic* `vcreate` *used to build a SEW=32, LMUL=4* `float` *vector (*`vfloat32m4`*) from two SEW=32, LMUL=2* `float` *vectors (*`vfloat32m2`*).*

```
// Given the following declarations
vfloat32m4_t dest;
vfloat32m2_t v0 = ...;
vfloat32m2_t v1 = ...;

// this pseudo intrinsic
dest = __riscv_vcreate_v_f32m2_f32m4(v0, v1);

// is semantically equivalent to
dest = __riscv_vset_v_f32m2_f32m4(__riscv_vundefined_f32m4(), 0, v0);
dest = __riscv_vset_v_f32m2_f32m4(dest, 1, v1);
```

## 8.10. vlenb

The `vlenb` intrinsic returns what is held inside the read-only CSR `vlenb`, which is the vector register length in bytes. This pseudo intrinsic is mapped to a `csrr` instruction that reads from the CSR `vlenb`.

```
unsigned __riscv_vlenb();
```

# Chapter 9. Programming Notes

## 9.1. Agnostic value in the RVV C intrinsics

An agnostic value is in the granularity of "element" contained in an RVV type. Agnostic values are either:

- Tail element(s) of an RVV value produced through a `ta` instruction
- Masked-off element(s) of an RVV value produced through a `ma` instruction
- All elements in an uninitialized value
- A value assigned with the `vundefined` intrinsics

An agnostic value is an indeterminate value and evaluation of an agnostic value is undefined behavior. Users should not rely on any evaluation to an agnostic value.

## 9.2. Copying vector register group contents

There is no intrinsic that directly maps to the whole vector register move instructions (`vmv<nr>r.v`).

For copying of the vector contents in whole, we encourage the users to use the assignment operator (`=`).

The assignment operator (`=`) represents the semantic of a whole vector register (group) copy for the expression on the right hand side to the RVV type object on the left hand side. The semantic will still maintain a whole vector register content copy for fractional LMUL types. This enables the compiler to coalesce register usage when possible.

Users may leverage the vector move intrinsics (`vmv_v_v`) intrinsics if they hope to copy vector register groups with `vl != VLMAX`.

## 9.3. The passthrough (`vd`) argument in the intrinsics

Intrinsics whose computation is relevant to the value held in destination register (assembly mnemonics `vd`) have a `vd` argument in them. The following list enumerates the intrinsics that have a `vd` argument. Please see the appendix for the exact prototypes of these intrinsics.

- Intrinsics with tail-undisturbed (`vta=0`)
- Intrinsics with mask-undisturbed (`vma=0`)
- Intrinsics representing Vector Multiply-Add Operations
- Intrinsics representing Vector Slideup Instructions

For intrinsics with no `vd` argument, the implementation is free to pick any register as the destination register.

## 9.4. Assumption of `vstart=0` for intrinsics users.

The `vstart` CSR is currently not exposed to the intrinsics programmer, and the intrinsics have the semantics of `vstart = 0`. Support for positive `vstart` values is implementation -defined; thus,

portable application software should not set `vstart > 0`.

## 9.5. Assembly generated from the intrinsics

Some users may expect the intrinsics to directly translate and appear in the assembly; however, the intrinsics are the interfaces that expose the vector instruction semantics. The implementation is free to optimize them out if there is an opportunity.

## 9.6. Bookkeeping of configurations

Control of `vl`, `vtype`, `vxrm`, and `frm` is not directly exposed to the user. The implementation is responsible for setting the correct values into these CSRs to achieve the expected semantics of the intrinsic functions with respect to the conventions defined in the ISA specification (*RISC-V "V" Vector Extension*, n.d.) and ABI specification (*RISC-V Calling Conventions: Vector*, n.d.).

## 9.7. Strided load/store with stride of 0

The RVV specification mentions that the strided load/store instruction with a stride of 0 could have different behaviors, performing all memory accesses or fewer memory operations. Since needing all memory accesses isn't likely to be common, the implementation is allowed to generate fewer memory operations with strided load/store intrinsics.

In other words, the compiler does not guarantee generating the instruction for all memory accesses in strided load/store intrinsics with a stride of 0. If the user needs all memory accesses to be performed, they should use an indexed load/store intrinsics with all zero indices.

## 9.8. Leveraging instructions with operand mnemonics of `vi`

The intrinsics provide variants with operand mnemonics of `vv` and `vx`, but not `vi`. This was an intentional design to reduce the total amount of out-going intrinsics.

It is an optimization issue for the implementation to emit instructions with operand mnemonics of `vi` when an immediate that can be expressed within 5-bit is provided to the intrinsics.

## 9.9. Mixing inline assembly and intrinsics

The compiler will be conservative to registers (`vtype`, `vxrm`, `frm`) when encountering inline assembly. Users should be aware that mixing uses of intrinsics and inline assembly will result in extra save and restore.

## 9.10. The `new_vl` argument in fault-only-first load intrinsics

The fault-only-first load intrinsics write the new value inside the `vl` register into the address of the `new_vl` argument. Providing an illegal memory location is undefined behavior. == Intrinsics for BFloat16 (Brain Float 16) instruction set extensions

The RISC-V vector C intrinsics supports intrinsics that exposes the control of BFloat16 (Brain Float 16) instruction set extensions [31].

## 9.11. Naming scheme

The BFloat16 intrinsics follows the naming scheme defined under Chapter 6, with `bf` as the abbreviation for BFloat16 types in the function suffix.

## 9.12. Control of the vector extension programming model

The BFloat16 intrinsics follows provides the same control of the vector programming model defined under Chapter 5. Intrinsics that represents BFloat16 instructions that are affected by `frm` (`vfncvtbf16.f.f.w` and `vfwmaccbf16`) follow what is defined under Section 5.6 and provides variants of [implicit-frm] and Section 5.6.2.

## 9.13. Type system

Floating-point types have EEW and EMUL encoded into the type. The first row describes the EMUL and the first column describes the data type and element width of the scalar type.

Floating-point types with element widths of 16 (Types=`__bf16`) require the `zfbfmin` and `zvfbfmin` extension to be specified in the architecture.

*Although C++23 introduces `<stdfloat>` for fixed-width floating-point types, this latest standard is not yet supported in the upstream RISC-V compiler. The specification (along with the prototype lists in appendix) uses `__bf16` to represent the BFloat16 floating-point type.*

| Types | EMUL=1/8 | EMUL=1/4 | EMUL=1/ 2 | EMUL=1 | EMUL=2 | EMUL=4 | EMUL=8 |
|---|---|---|---|---|---|---|---|
| __bf16 | N/A | vbfloat16mf4_t | vbfloat16mf2_t | vbfloat16m1_t | vbfloat16m2_t | vbfloat16m4_t | vbfloat16m8_t |

*Table 10. BFloat16 types*

| Non-tuple Types (NFILED=1) | NFIELD=2 | NFIELD=3 | NFIELD=4 | NFIELD=5 | NFIELD=6 | NFIELD=7 | NFIELD=8 |
|---|---|---|---|---|---|---|---|
| vbfloat16mf4_t | vbfloat16mf4x2_t | vbfloat16mf4x3_t | vbfloat16mf4x4_t | vbfloat16mf4x5_t | vbfloat16mf4x6_t | vbfloat16mf4x7_t | vbfloat16mf4x8_t |
| vbfloat16mf2_t | vbfloat16mf2x2_t | vbfloat16mf2x3_t | vbfloat16mf2x4_t | vbfloat16mf2x5_t | vbfloat16mf2x6_t | vbfloat16mf2x7_t | vbfloat16mf2x8_t |
| vbfloat16m1_t | vbfloat16m1x2_t | vbfloat16m1x3_t | vbfloat16m1x4_t | vbfloat16m1x5_t | vbfloat16m1x6_t | vbfloat16m1x7_t | vbfloat16m1x8_t |
| vbfloat16m2_t | vbfloat16m2x2_t | vbfloat16m2x3_t | vbfloat16m2x4_t | N/A | N/A | N/A | N/A |
| vbfloat16m4_t | vbfloat16m4x2_t | N/A | N/A | N/A | N/A | N/A | N/A |

*Table 11. Tuple types*

## 9.14. Psuedo intrinsics

The RISC-V vector BFloat16 types (provided under Section 9.13) also have pseudo intrinsics variants from Chapter 8 to help variable declaration and manipulation across intrinsic types.

# Chapter 10. References

[0]Github - riscv/riscv-v-spec/v-spec.adoc

*Standard extensions are merged into* `riscv/riscv-isa-manual` *after ratification. There is an on-going pull request [26] for the "V" extension to be merged. At this moment this intrinsics specification still references the frozen draft [0]. This reference will be updated in the future once the pull request has been merged.*

[1]Github - riscv-non-isa/riscv-c-api-doc/riscv-c-api.md

[2]User Guide for RISC-V Target

[3]RISC-V Options (Using the GNU Compiler Collection (GCC))

[4]Section 3.4.1 (Vector selected element width `vsew[2:0]`) in the specification [0]

[5]Section 3.4.2 (Vector Register Grouping (`vlmul[2:0]`)) in the specification [0]

[6]Section 3.4.3 (Vector Tail Agnostic and Vector Mask Agnostic `vta` and `vma`) in the specification [0]

[7]Section 5.3 (Vector Masking) in the specification [0]

[8]Section 3.8 (Vector Fixed-Point Rounding Mode Register `vxrm`) in the specification [0]

[9]psABI: Vector Register Convention

[10]The RISC-V Instruction Set Manual: 8.2 Floating-Point Control and Status Register

[11]Section 3.5 (Vector Length Register) in the specification [0]

[12]Section 3.4.2 in the specification [0]

[13]Section 11.13, 11.14, 13.6, 13.7 in the specification [0]

[14]Section 4.5 (Mask Register Layout) in the specification [0]

[15]Section 7.5 in the specification [0]

[16]Section 7.8 in the specification [0]

[17]Section 5.2 (Vector Operands) in the specification [0]

[18]Section 6 (Configuration-Setting Instructions) in the specification [0]

[19]Section 18 (Standrad Vector Extensions) in the specification [0]

[20]Section 18.2 (Zve*: Vector Extensions for Embedded Processors) in the specification [0]

[21]Section 12 (Vector Fixed-Point Arithmetic Instructions) in the specification [0]

[22]Section 3.9 (3.9. Vector Fixed-Point Saturation Flag vxsat) in the specification [0]

[23]Section 13 (Vector Floating-Point Instructions) in the specification [0]

[24]Section 16.3.1 (Vector Slideup Instructions) in the specification [0]

[25]Section 3.7 (Vector Start Index CSR `vstart`) in the specification [0]

[26]riscv/riscv-isa-manual#1088

[27]Section 6.3 (Constraints on Setting `vl`) in the specficiation [0]

[28]Section 6.4 (Example of stripmining and changes to SEW) in the specification [0]

[29]Section 3.6 (Vector Byte Length `vlenb`) in the specification [0]

[30]Section 16.6 (Whole Vector Register Move) in the specification [0]

[31]RISC-V BFloat16 Specification

# Chapter 11. Examples

*This section is non-normative.*

*No claims about efficiency are made the examples presented in this section.*

This section presents examples that use the RVV intrinsics specified in this document. The examples are in C and assume `#include <riscv_vector.h>` has appeared earlier in the source code.

## 11.1. Memory copy

*Example 2. An implementation of the `memcpy` function of the C Standard library using RVV intrinsics.*

```
void *memcpy_rvv(void *restrict destination, const void *restrict source,
    size_t n) {
  unsigned char *dst = destination;
  const unsigned char *src = source;
  // copy data byte by byte
  for (size_t vl; n > 0; n -= vl, src += vl, dst += vl) {
    vl = __riscv_vsetvl_e8m8(n);
    // Load src[0..vl)
    vuint8m8_t vec_src = __riscv_vle8_v_u8m8(src, vl);
    // Store dst[0..vl)
    __riscv_vse8_v_u8m8(dst, vec_src, vl);
    // src is incremented vl (bytes)
    // dst is incremented vl (bytes)
    // n is decremented vl
  }
  return destination;
}
```

## 11.2. SAXPY

Consider the following function that implements a SAXPY-like kernel.

```
void saxpy_reference(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];
  }
}
```

*Example 3. An implementation of SAXPY using RVV intrinsics.*

```
void saxpy_rvv(size_t n, const float a, const float *x, float *y) {
  for (size_t vl; n > 0; n -= vl, x += vl, y += vl) {
    vl = __riscv_vsetvl_e32m8(n);
    // Load x[i..i+vl)
    vfloat32m8_t vx = __riscv_vle32_v_f32m8(x, vl);
    // Load y[i..i+vl)
    vfloat32m8_t vy = __riscv_vle32_v_f32m8(y, vl);
    // Computes vy[0..vl) + a*vx[0..vl)
    // and stores it in y[i..i+vl)
    __riscv_vse32_v_f32m8(y, __riscv_vfmacc_vf_f32m8(vy, a, vx, vl), vl);
  }
```

```
}
```

# 11.3. Matrix multiplication

Consider the following function that implements a naive matrix multiplication.

```
// matrix multiplication
//  C[0..n)[0..m) = A[0..n)[0..p) x B[0..p)[0..m)
void matmul_reference(double *a, double *b, double *c, int n, int m, int p) {
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j) {
      c[i * m + j] = 0;
      for (int k = 0; k < p; ++k) {
        c[i * m + j] += a[i * p + k] * b[k * m + j];
      }
    }
}
```

The following example is a version of the matrix multiplication. The accumulation on `c[i * m + j]` is implemented using partial accumulations followed by a single final accumulation.

*Example 4. An implementation of a naive matrix multiplication using RVV intrinsics.*

```
void matmul_rvv(double *a, double *b, double *c, int n, int m, int p) {
  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 * p];
      double *ptr_b = &b[j];
      int k = p;
      // Set accumulator to  zero.
      vfloat64m1_t vec_s = __riscv_vfmv_v_f_f64m1(0.0, vlmax);
      vfloat64m1_t vec_zero = __riscv_vfmv_v_f_f64m1(0.0, vlmax);
      for (size_t vl; k > 0; k -= vl, ptr_a += vl, ptr_b += vl * m) {
        vl = __riscv_vsetvl_e64m1(k);

        // Load row a[i][k..k+vl)
        vfloat64m1_t vec_a = __riscv_vle64_v_f64m1(ptr_a, vl);
        // Load column b[k..k+vl)[j]
        vfloat64m1_t vec_b =
          __riscv_vlse64_v_f64m1(ptr_b, sizeof(double) * m, vl);

        // Accumulate dot product of row and column. If vl < vlmax we need to
        // preserve the existing values of vec_s, hence the tu policy.
        vec_s = __riscv_vfmacc_vv_f64m1_tu(vec_s, vec_a, vec_b, vl);
      }

      // Final accumulation.
      vfloat64m1_t vec_sum =
        __riscv_vfredusum_vs_f64m1_f64m1(vec_s, vec_zero, vlmax);
      double sum = __riscv_vfmv_f_s_f64m1_f64(vec_sum);
      c[i * m + j] = sum;
    }
}
```

## 11.4. String copy

*Example 5. An implementation of the* `strcpy` *function of the C Standard Library using RVV intrinsics.*

```
char *strcpy_rvv(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;

  // This loop stops when among the loaded bytes we find the null byte
  // of the string i.e., when first_set_bit >= 0
  for (size_t vl; first_set_bit < 0; src += vl, dst += vl) {
    // Load up to vlmax elements if possible.
    // vl is set to the maximum number of elements, no more than vlmax, that
    // could be loaded without causing a memory fault.
    vuint8m8_t vec_src = __riscv_vle8ff_v_u8m8(src, &vl, vlmax);

    // Mask that states where null bytes are in the loaded bytes.
    vbool1_t string_terminate = __riscv_vmseq_vx_u8m8_b1(vec_src, 0, vl);

    // If the null byte is not in the loaded bytes the resulting mask will
    // be all ones, otherwise only the elements up to and including the
    // first null byte of the resulting will be enabled.
    vbool1_t mask = __riscv_vmsif_m_b1(string_terminate, vl);

    // Store the enabled elements as determined by the mask above.
    __riscv_vse8_v_u8m8_m(mask, dst, vec_src, vl);

    // Determine if we found the null byte in the loaded bytes.
    // If not found, first_set_bit is set to all ones (i.e., -1), otherwise
    // first_set_bit will be the number of the first element enabled in the
    // mask.
    first_set_bit = __riscv_vfirst_m_b1(string_terminate, vl);
  }
  return destination;
}
```

## 11.5. Control flow

Consider the following function that computes the division of two arrays elementwise but sets the result to a given value when the element of the divisor array is zero.

```
void branch_ref(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;
  }
}
```

The following example applies if-conversion using masks to implement the semantics of the conditional operator.

*Example 6. An implementation of* `branch_ref` *using RVV intrinsics.*

```
void branch_rvv(double *a, double *b, double *c, int n, double constant) {
  // set vlmax and initialize variables
  size_t vlmax = __riscv_vsetvlmax_e64m1();
```

```
  // "Broadcast" the value of constant to all (vlmax) the elements in
  // vec_constant.
  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);

    // Load a[i..i+vl)
    vfloat64m1_t vec_a = __riscv_vle64_v_f64m1(a, vl);
    // Load b[i..i+vl)
    vfloat64m1_t vec_b = __riscv_vle64_v_f64m1(b, vl);

    // Compute a mask whose enabled elements will correspond to the
    // elements of b that are not zero.
    vbool64_t mask = __riscv_vmfne_vf_f64m1_b64(vec_b, 0.0, vl);

    // Use mask undisturbed policy to compute the division for the
    // elements enabled in the mask, otherwise set them to the given
    // constant above (maskedoff).
    vfloat64m1_t vec_c = __riscv_vfdiv_vv_f64m1_mu(
        mask, /*maskedoff*/ vec_constant, vec_a, vec_b, vl);

    // Store into c[i..i+vl)
    __riscv_vse64_v_f64m1(c, vec_c, vl);
  }
}
```

# 11.6. Reduction and counting

Consider the following function that computes the dot product of two arrays excluding elements of the first array (along with the correspondign element of the second array) where the value is 42. The function also counts how many pairs of elements took part in the dot-product.

```
void reduce_reference(double *a, double *b, double *result_sum,
                      int *result_count, int n) {
  int count = 0;
  double s = 0.0;
  for (int i = 0; i < n; ++i) {
    if (a[i] != 42.0) {
      s += a[i] * b[i];
      count++;
    }
  }

  *result_sum = s;
  *result_count = count;
}
```

The following example implements the accumulation of the `s` variable doing several partial accumulations followed by a final accumulation.

*Example 7. An implementation of* `reduce_reference` *using RVV intrinsics.*

```
void reduce_rvv(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(0.0, vlmax);
  vfloat64m1_t vec_s = __riscv_vfmv_v_f_f64m1(0.0, vlmax);
```

```
  for (size_t vl; n > 0; n -= vl, a += vl, b += vl) {
    vl = __riscv_vsetvl_e64m1(n);

    // Load a[i..i+vl)
    vfloat64m1_t vec_a = __riscv_vle64_v_f64m1(a, vl);
    // Load b[i..i+vl)
    vfloat64m1_t vec_b = __riscv_vle64_v_f64m1(b, vl);

    // Compute a mask whose enabled elements will correspond to the
    // elements of a that are not 42.
    vbool64_t mask = __riscv_vmfne_vf_f64m1_b64(vec_a, 42.0, vl);

    // for all e in [0..vl)
    //  vec_s[e] ← vec_s[e] + vec_a[e] * vec_b[e], if mask[e] is enabled
    //             vec_s[e]                      , otherwise (mask undisturbed)
    vec_s = __riscv_vfmacc_vv_f64m1_tumu(mask, vec_s, vec_a, vec_b, vl);

    // Adds to count the number of elements in mask that are enabled.
    count += __riscv_vcpop_m_b64(mask, vl);
  }

  vfloat64m1_t vec_sum;
  // Final accumulation.
  vec_sum = __riscv_vfredusum_vs_f64m1_f64m1(vec_s, vec_zero, vlmax);
  double sum = __riscv_vfmv_f_s_f64m1_f64(vec_sum);

  // Return values.
  *result_sum = sum;
  *result_count = count;
}
```