Discovering the RVV C intrinsics API v1.0

eop Chen

Kito Cheng

Credits

The development of RVV intrinsics owes a great debt to Nick Knight, Craig Topper, and Roger Ferrer Ibáñez for their valuable comments and reviews.

Abstract

- Introduction
- Availability and usage
  - Test macro and header inclusion
  - Current status of support in the upstream compiler toolchain
- Control to the vector extension programming model
  - Overview
  - Explicit (non-overloaded) intrinsics
  - Implicit (overloaded) intrinsics
  - Other variants
- Current status and planning

Availability and usage

Current status of support in the upstream compiler toolchain

LLVM

- LLVM 16 [0] supports the v0.11 intrinsics
    - Lack the tuple-type segment load/store, the floating-point rounding mode intrinsics, and the fixed-point rounding mode intrinsics.
- LLVM 17 [1] supports the v0.12 intrinsics
    - Expecting no more change to the specification and identical to what is in v1.0

GCC

- The next GCC release (GCC 14) [2] is expected to support v1.0

[0] https://releases.llvm.org/16.0.0/tools/clang/docs/ReleaseNotes.html

[1] https://github.com/llvm/llvm-project/tree/release/17.x

[2] https://gcc.gnu.org/releases.html

Availability and usage

Test macro and header file inclusion

Users can check the compiler support through the testing macro __riscv_v_intrinsic.

Please include <riscv_vector.h> to access the RISC-V vector intrinsics.

```
#ifdef __riscv_v_intrinsic
#include <riscv_vector.h>
#endif /* __riscv_v_intrinsic */
```

Control to the vector extension programming model

Overview - Starting from the vsetvl configuration (1)

Decision on the level of abstraction… Starting from the essential instruction in the vector extension

```
# Example: Load 16-bit values, widen multiply to 32b, shift 32b result
# right by 3, store 32b values.
# On entry:
#  a0 holds the total number of elements to process
#  a1 holds the address of the source array
#  a2 holds the address of the destination array

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

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

Control to the vector extension programming model

Overview - Starting from the vsetvl configuration (2)

Decision on the level of abstraction… Starting from the essential instruction in the vector extension

Intrinsics brings assembly instruction control to the C language level, taking care of tedious jobs like

- Instruction scheduling
- Register allocation
- vsetvl configuration
- Rounding mode (frm and vxrm) configuration

Control to the vector extension programming model

Overview - Type system

Encodes

- Data type
- Element width
- Length multiplier

Intrinsic types are like

- vfloat16m4_t
- vint32m1_t
- vuint8m8_t

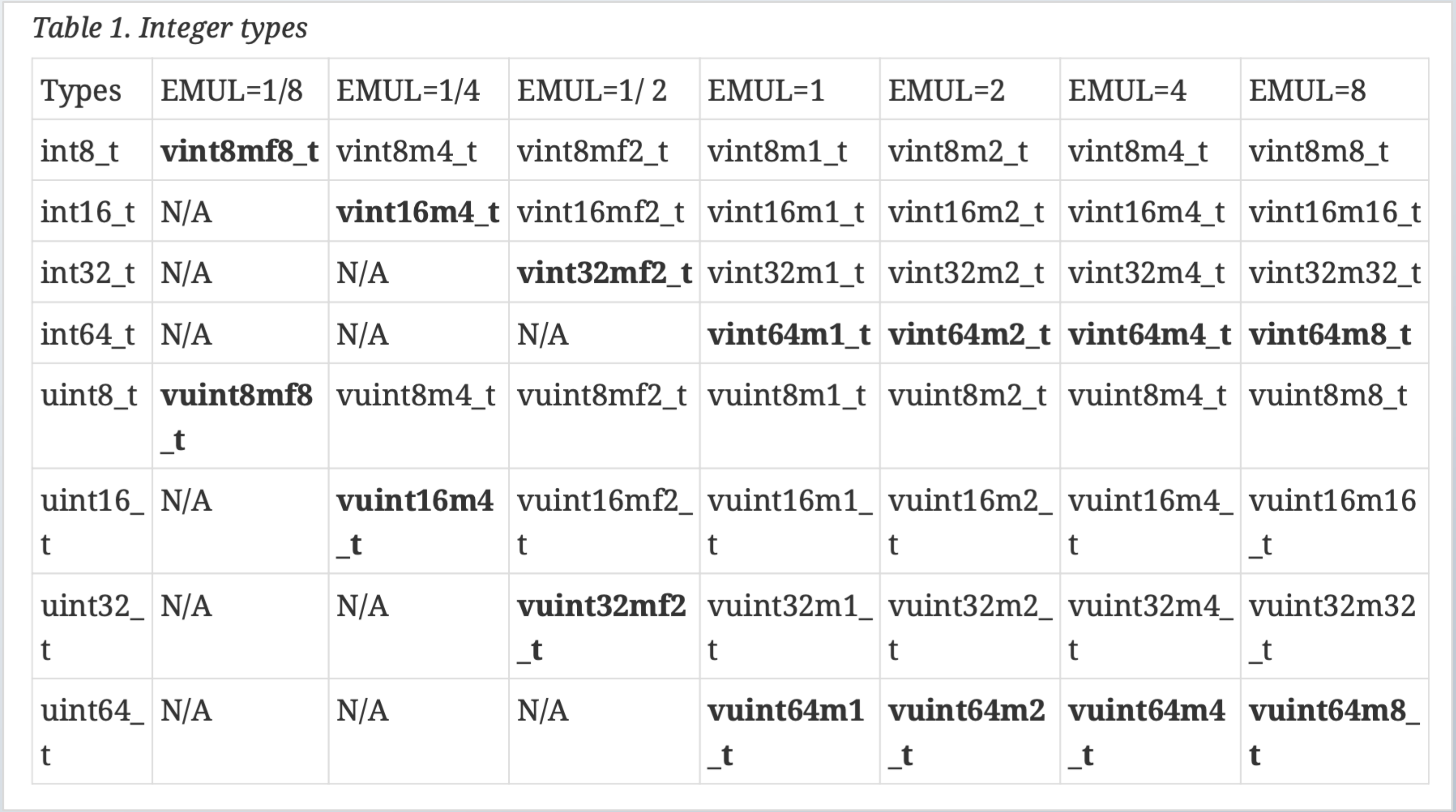


Table 1. Integer types

Types	EMUL=1/8	EMUL=1/4	EMUL=1/ 2	EMUL=1	EMUL=2	EMUL=4	EMUL=8
int8_t	**vint8mf8_t**	vint8m4_t	vint8mf2_t	vint8m1_t	vint8m2_t	vint8m4_t	vint8m8_t
int16_t	N/A	**vint16m4_t**	vint16mf2_t	vint16m1_t	vint16m2_t	vint16m4_t	vint16m16_t
int32_t	N/A	N/A	**vint32mf2_t**	vint32m1_t	vint32m2_t	vint32m4_t	vint32m32_t
int64_t	N/A	N/A	N/A	**vint64m1_t**	**vint64m2_t**	**vint64m4_t**	**vint64m8_t**
uint8_t	**vuint8mf8_t**	vuint8m4_t	vuint8mf2_t	vuint8m1_t	vuint8m2_t	vuint8m4_t	vuint8m8_t
uint16_t	N/A	**vuint16m4_t**	vuint16mf2_t	vuint16m1_t	vuint16m2_t	vuint16m4_t	vuint16m16_t
uint32_t	N/A	N/A	**vuint32mf2_t**	vuint32m1_t	vuint32m2_t	vuint32m4_t	vuint32m32_t
uint64_t	N/A	N/A	N/A	**vuint64m1_t**	**vuint64m2_t**	**vuint64m4_t**	**vuint64m8_t**

Control to the vector extension programming model

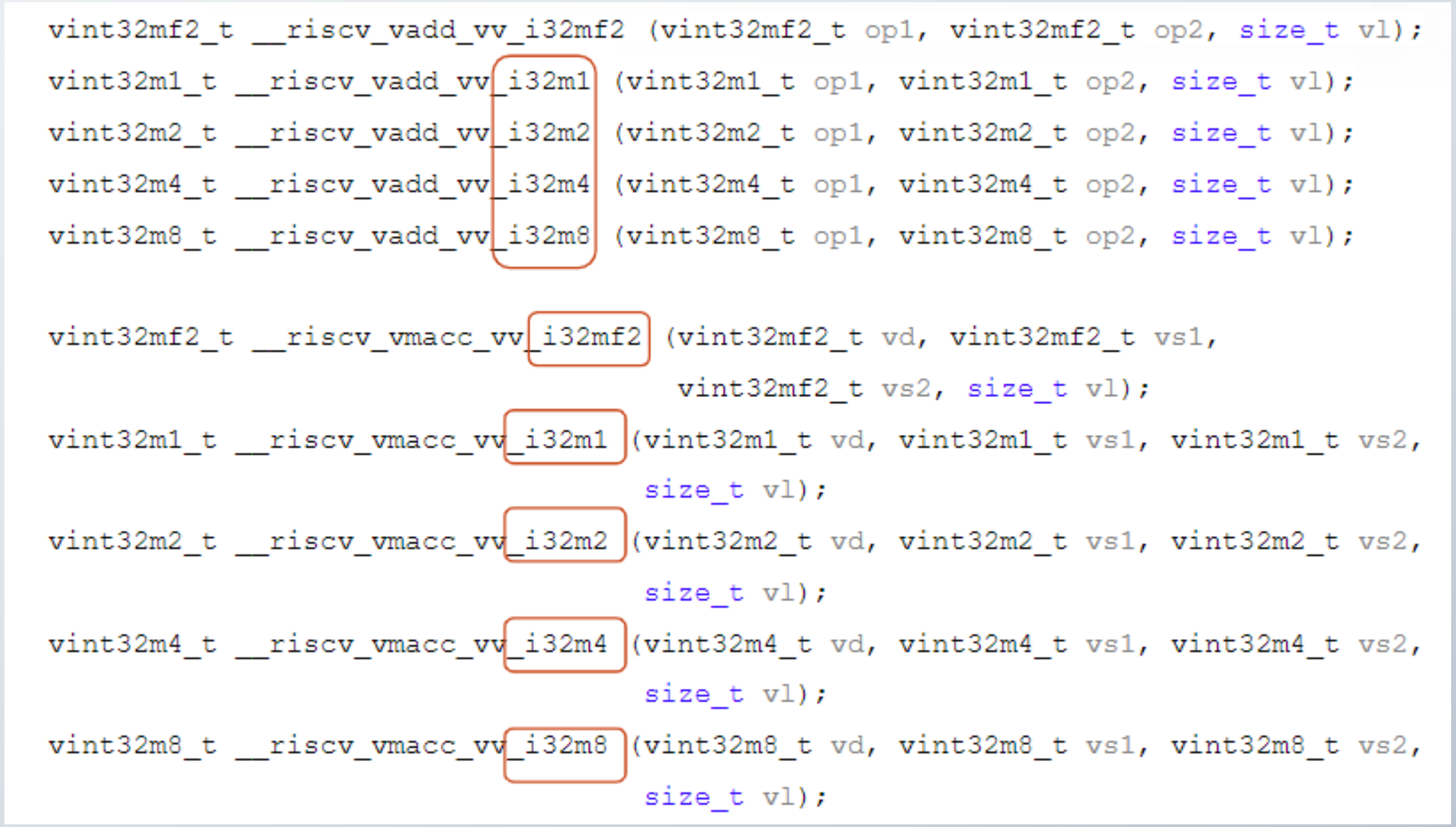
Explicit (non-overloaded) intrinsics (1)

Encodes

- Data type
- Element width
- Length multiplier

Intrinsic types are like

- vfloat16m4_t
- vint32m1_t
- vuint8m8_t

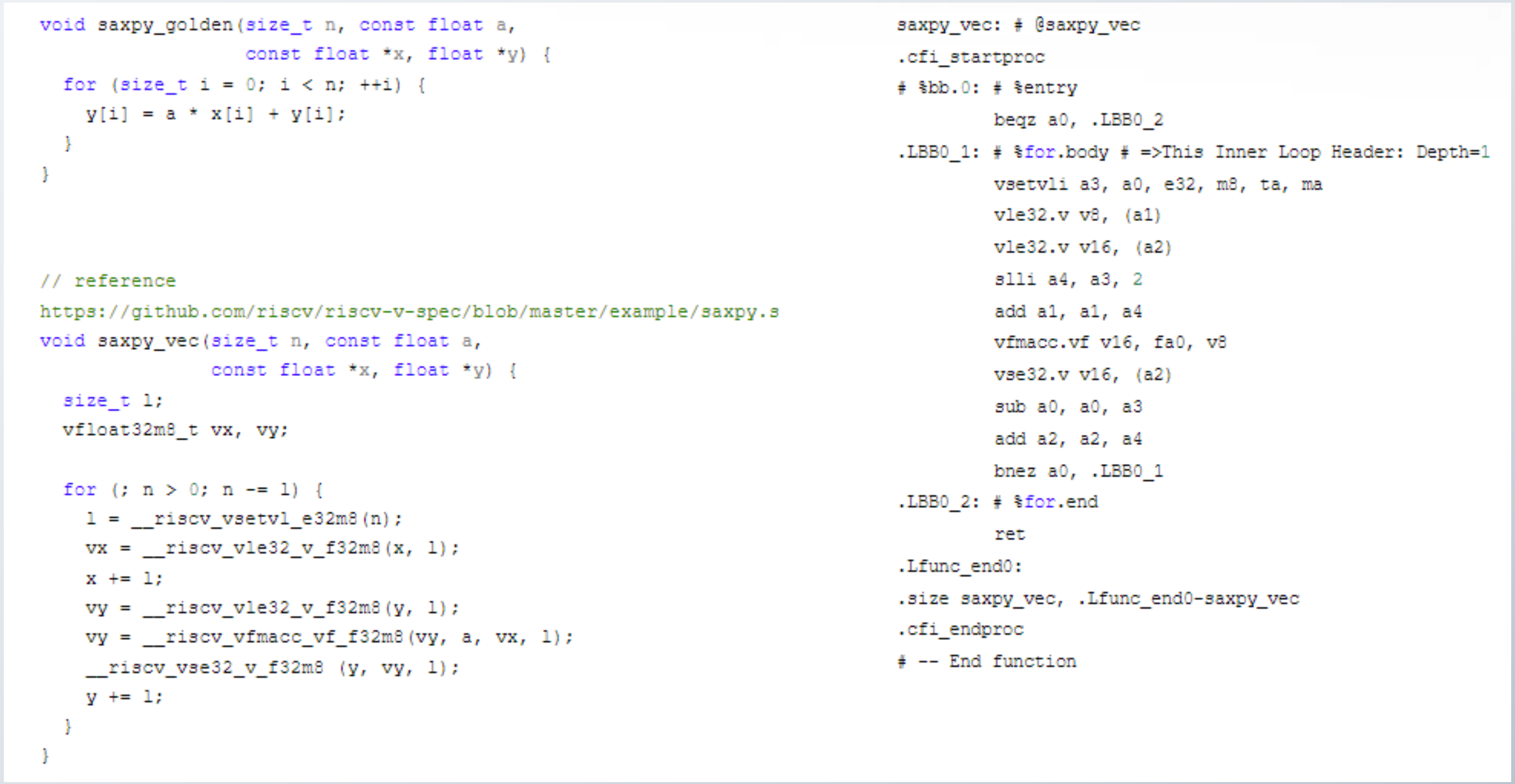


```
vint32mf2_t __riscv_vadd_vv_i32mf2 (vint32mf2_t op1, vint32mf2_t op2, size_t vl);
vint32m1_t __riscv_vadd_vv_i32m1 (vint32m1_t op1, vint32m1_t op2, size_t vl);
vint32m2_t __riscv_vadd_vv_i32m2 (vint32m2_t op1, vint32m2_t op2, size_t vl);
vint32m4_t __riscv_vadd_vv_i32m4 (vint32m4_t op1, vint32m4_t op2, size_t vl);
vint32m8_t __riscv_vadd_vv_i32m8 (vint32m8_t op1, vint32m8_t op2, size_t vl);

vint32mf2_t __riscv_vmacc_vv_i32mf2 (vint32mf2_t vd, vint32mf2_t vs1,
                                     vint32mf2_t vs2, size_t vl);
vint32m1_t __riscv_vmacc_vv_i32m1 (vint32m1_t vd, vint32m1_t vs1, vint32m1_t vs2,
                                   size_t vl);
vint32m2_t __riscv_vmacc_vv_i32m2 (vint32m2_t vd, vint32m2_t vs1, vint32m2_t vs2,
                                   size_t vl);
vint32m4_t __riscv_vmacc_vv_i32m4 (vint32m4_t vd, vint32m4_t vs1, vint32m4_t vs2,
                                   size_t vl);
vint32m8_t __riscv_vmacc_vv_i32m8 (vint32m8_t vd, vint32m8_t vs1, vint32m8_t vs2,
                                   size_t vl);
```

Control to the vector extension programming model

Explicit (non-overloaded) intrinsics (2)



```c
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;
  }
}
```

```asm
saxpy_vec: # @saxpy_vec
.cfi_startproc
# %bb.0: # %entry
         beqz a0, .LBB0_2
.LBB0_1: # %for.body # =>This Inner Loop Header: Depth=1
         vsetvli a3, a0, e32, m8, ta, ma
         vle32.v v8, (a1)
         vle32.v v16, (a2)
         slli a4, a3, 2
         add a1, a1, a4
         vfmacc.vf v16, fa0, v8
         vse32.v v16, (a2)
         sub a0, a0, a3
         add a2, a2, a4
         bnez a0, .LBB0_1
.LBB0_2: # %for.end
         ret
.Lfunc_end0:
.size saxpy_vec, .Lfunc_end0-saxpy_vec
.cfi_endproc
# -- End function
```

Control to the vector extension programming model

Explicit (non-overloaded) intrinsics (3)

General naming convention is

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

Example:

```
vint32m1_t __riscv_vadd_vv_i32m1 (vint32m1_t op1, vint32m1_t op2, size_t vl);

vint32m1_t __riscv_vadd_vv_i32m1_tum (vbool32_t mask, vint32m1_t maskedoff,
                                      vint32m1_t op1, vint32m1_t op2, size_t vl);
```

Control to the vector extension programming model

Explicit (non-overloaded) intrinsics (4)

General naming convention is

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

Example:

```
vfloat32m2_t __riscv_vfadd_vf_f32m2          (vfloat32m2_t op1, float32_t op2, size_t vl);

vfloat32m8_t __riscv_vfadd_vf_f32m8_rm       (vfloat32m8_t op1, float32_t op2,
                                              unsigned int frm, size_t vl);

vfloat32m8_t __riscv_vfadd_vf_f32m8_rm_tumu (vbool4_t mask, vfloat32m8_t maskedoff,
                                              vfloat32m8_t op1, float32_t op2, unsigned int frm,
                                              size_t vl);
```

For exceptions, please checkout the intrinsics specification.

Control to the vector extension programming model

Implicit (overloaded) intrinsics (1)

Main intention: Provide type agnostic interface to users

General naming convention is

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

Example:

```
vint32m1_t __riscv_vadd (vint32m1_t op1, vint32m1_t op2, size_t vl);
vint32m2_t __riscv_vadd (vint32m2_t op1, vint32m2_t op2, size_t vl);
```

Control to the vector extension programming model

Implicit (overloaded) intrinsics (2)

```
vint8m2_t foo(vint8m1_t op1, vint8m1_t op2, vint8m1_t op3, vint8m1_t op4,
              vint8m1_t op5, vint8m1_t op6, vint8m1_t op7, vint8m1_t op8,
              size_t vl) {
  vint8m1_t op12 = __riscv_vadd(op1, op2, vl);
  vint8m1_t op34 = __riscv_vadd(op3, op4, vl);

  vint8m2_t op1234 = __riscv_vset_v_i8m1_i8m2(op1234, 0, op12);
  op1234 = __riscv_vset_v_i8m1_i8m2(op1234, 1, op34);

  vint8m1_t op56 = __riscv_vadd(op5, op6, vl);
  vint8m1_t op78 = __riscv_vadd(op7, op8, vl);

  vint8m2_t op5678 = __riscv_vset_v_i8m1_i8m2(op5678, 0, op56);
  op5678 = __riscv_vset_v_i8m1_i8m2(op1234, 1, op78);

  return __riscv_vadd(op1234, op5678, vl);
}
```

```
foo:                                        # @foo
.cfi_startproc
# %bb.0:                                    # %entry
vsetvli zero, a0, e8, m1, ta, ma
vadd.vv v8, v8, v9
vadd.vv v9, v10, v11
vadd.vv v10, v14, v15
vmv2r.v v12, v8
vmv.v.v v13, v10
vsetvli zero, a0, e8, m2, ta, ma
vadd.vv v8, v8, v12
ret
.Lfunc_end0:
.size foo, .Lfunc_end0-foo
.cfi_endproc
                                            # -- End
function
```

Due to limitation of the C language, the overloaded intrinsics does not cover 100% of the RVV instructions. For more detail please checkout the intrinsics specification.

Control to the vector extension programming model

Other variants

Other variants allows users to control the vector programming model like:

- Masked variants to control vm

```
vfloat32m2_t __riscv_vfadd_vf_f32m2_m       (vbool16_t mask, vfloat32m2_t op1, float32_t op2, size_t vl);
```

- Policy variants to control vta and vma

```
vfloat32m2_t __riscv_vfadd_vv_f32m2_tumu    (vbool16_t mask, vfloat32m2_t maskedoff,
                                             vfloat32m2_t op1 vfloat32m2_t op2,
                                             size_t vl);
```

- Rounding mode variants to control frm and vxrm

```
vfloat32m2_t __riscv_vfadd_vv_f32m2_rm_tumu (vbool16_t mask, vfloat32m2_t maskedoff,
                                             vfloat32m2_t op1 vfloat32m2_t op2,
                                             unsigned int frm, size_t vl);
```

Current status and planning

Current status: Under first round of internal review among RVI Tech Chairs

Future planning

- C++ templates
- Exception handling intrinsics for fflag and vxsat
- Pick up intrinsics of other vector extensions
  - Vector crypto
  - Non Temporal Local Hint
  - BF16

Feel free to checkout riscv-non-isa/rvv-intrinsic-doc for the latest draft specification.

https://github.com/riscv-non-isa/rvv-intrinsic-doc

Empowering innovators

www.sifive.com