RISC-V “V” Vector Extension Intrinsics

Introduction

This document introduces the intrinsics for RISC-V vector programming, including the design decision we take, the type system, the general naming rules for intrinsics, and facilities for vector users. It does not list all available intrinsics for vector programming. The full set of intrinsics will be written in another document.

This RFC is based on RISC-V “V” Vector Extension specification version 0.8.

Type System

Data Types

Encode `SEW` and `LMUL` into data types. There are the following data types for `SEW` 64.

Types	LMUL = 1	LMUL = 2	LMUL = 4	LMUL = 8
int64_t	vint64m1_t	vint64m2_t	vint64m4_t	vint64m8_t
uint64_t	vuint64m1_t	vuint64m2_t	vuint64m4_t	vuint64m8_t
int32_t	vint32m1_t	vint32m2_t	vint32m4_t	vint32m8_t
uint32_t	vuint32m1_t	vuint32m2_t	vuint32m4_t	vuint32m8_t
int16_t	vint16m1_t	vint16m2_t	vint16m4_t	vint16m8_t
uint16_t	vuint16m1_t	vuint16m2_t	vuint16m4_t	vuint16m8_t
int8_t	vint8m1_t	vint8m2_t	vint8m4_t	vint8m8_t
uint8_t	vuint8m1_t	vuint8m2_t	vuint8m4_t	vuint8m8_t
vfloat64	vfloat64m1_t	vfloat64m2_t	vfloat64m4_t	vfloat64m8_t
vfloat32	vfloat32m1_t	vfloat32m2_t	vfloat32m4_t	vfloat32m8_t
vfloat16	vfloat16m1_t	vfloat16m2_t	vfloat16m4_t	vfloat16m8_t

Mask Types

Encode `MLEN` into the mask types. There are the following mask types for `SEW` 64.

Types	MLEN = 1	MLEN = 2	MLEN = 4	MLEN = 8	MLEN = 16	MLEN = 32	MLEN = 64
bool	vbool1_t	vbool2_t	vbool4_t	vbool8_t	vbool16_t	vbool32_t	vbool64_t

Types for Segment Load/Store

Under the constraint `LMUL` * `NR` 8.

```
SEGMENT_TYPE ::= 'v' TYPE LMUL 'x' NR '_t'
TYPE ::= ( 'int8' | 'int16' | 'int32' | 'int64' |
           'uint8' | 'uint16' | 'uint32' | 'uint64' |
           'float16' | 'float32' | 'float64' )
LMUL ::= ( m1 | m2 | m4 | m8 )
NR ::= ( 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 )

Example:
```

```
vint32m1x4_t
vfloat32m2x2_t
```

Configuration-Setting

`SEW` and `LMUL` are a part of the naming. They are static information for the intrinsics. There are two variants of configuration setting intrinsics. `vsetvl` is used to set `vl` according to the given AVL. `vsetvlmax` is used to set `vl` to VLMAX.

```
Example:

size_t vsetvl_e8m1 (size_t avl);
size_t vsetvl_e8m2 (size_t avl);
size_t vsetvl_e8m4 (size_t avl);
size_t vsetvl_e8m8 (size_t avl);
size_t vsetvlmax_e8m1 ();
size_t vsetvlmax_e8m2 ();
size_t vsetvlmax_e8m4 ();
size_t vsetvlmax_e8m8 ();
```

Naming Rules

Intrinsics is the interface to the low level assembly in high level programming language. The intrinsic API has the goal to make all the V-ext instructions accessible from C/C++. The intrinsic names are as close as the assembly mnemonics. Besides the basic intrinsics corresponding to assembly mnemonics, there are intrinsics close to semantic naming.

The intrinsic names will encode return type if it is appropriate. It is easier to know the output type of the intrinsics from the name. In addition, if the intrinsic call is used as the operand, having the return type is more immediate. If there is no return value, the intrinsics will encode the input value types. If the return type is the same, use exceptional rules to differentiate them. See Exceptions in Naming.

In general, the naming rule of intrinsics is

```
INTRINSIC ::= MNEMONIC '_' RET_TYPE
MNEMONIC ::= Instruction name in v-ext specification. Replace '.' with '_'.
RET_TYPE ::= SEW LMUL
SEW ::= ( i8 | i16 | i32 | i64 | u8 | u16 | u32 | u64 | f16 | f32 | f64 )
LMUL ::= ( m1 | m2 | m4 | m8 )
```

```
Example:
```

```
vadd.vv vd, vs2, vs1:
vint8m1_t vadd_vv_i8m1(vint8m1_t vs2, vint8m1_t vs1)

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

Exceptions in Naming

If intrinsics have the same return type under different input types, we could not use general naming rules directly on these intrinsics. It will cause the same intrinsic names for different input types.

This section lists all exceptional cases for intrinsic naming.

Vector Stores

It does not encode return type into vector store. There is no return data for store operations. Instead, use the type of store data to name the intrinsics.

```
Example:

vsb.v vs3, (rs1):
void vsb_v_i8m1(int8_t *rs1, vint8m1_t vs3);
```

Comparison Instructions

The result of comparison instructions is mask types. Becuase we use `MLEN` to name the mask types and multiple (`SEW`, `LMUL`) pairs map to the same `MLEN`, in addition to use the return type to name the intrinsics, we also encode the input types to distinguish these intrinsics.

```
Example:

vmseq.vv vd, vs2, vs1:
vbool8_t vmseq_vv_i8m1_b8(vint8m1_t vs2, vint8m1_t vs1);
vbool8_t vmseq_vv_i16m2_b8(vint16m2_t vs2, vint16m2_t vs1);
```

Reduction Instructions

The scalar input and output operands are held in element 0 of a single vector register. Use LMUL = 1 in the return type. To distinguish different intrinsics with different input types, encode the input type and the result type in the name.

```
Example:

vredsum.vs vd, vs2, vs1:
vint8m1_t vredsum_vs_i8m1_i8m1(vint8m1_t vs2, vint8m1_t vs1)
vint8m1_t vredsum_vs_i8m2_i8m1(vint8m2_t vs2, vint8m1_t vs1)
vint8m1_t vredsum_vs_i8m4_i8m1(vint8m4_t vs2, vint8m1_t vs1)
vint8m1_t vredsum_vs_i8m8_i8m1(vint8m8_t vs2, vint8m1_t vs1)
```

`vpopc.m` and `vfirst.m`

The return type of `vpopc.m` and `vfirst.m` is apparently an integer. Do not encode the return type into it. Instead, encode the input type to it.

```
Example:

vpopc.m rd, vs2:
unsigned long vpopc_m_b1(vbool1_t vs2);
unsigned long vpopc_m_b2(vbool2_t vs2);
```

Permutation Instructions

To move the element 0 of a vector to a scalar, encode the input vector type and the output scalar type.

```
Example:

vmv.x.s rd, vs2:
int8_t vmv_x_s_i8m1_i8 (vint8m1_t vs2);
int8_t vmv_x_s_i8m2_i8 (vint8m2_t vs2);
int8_t vmv_x_s_i8m4_i8 (vint8m4_t vs2);
int8_t vmv_x_s_i8m8_i8 (vint8m8_t vs2);
```

Scalar in Vector Operations

In V specification, it defines operations between vector and scalar types. If `XLEN` > `SEW`, the least-significant SEW bits of the scalar register are used. If `XLEN` < `SEW`, the value from the scalar register is sign-extended to SEW bits.

We define arithmetic intrinsics with scalar using SEW types.

```
Example:

// Use uint8_t for op2.
vuint8m1_t vadd_vx_u8m1(vuint8m1_t op1, uint8_t op2);
// Use uint64_t for op2.
```

```
vuint64m1_t vadd_vx_u64m1(vuint64m1_t op1, uint64_t op2);
```

The compiler may generate multiple instructions for the intrinsics. For example, it is a little bit complicated to support `uint64_t` operations under `XLEN` = 32.

It breaks the one-to-one mapping between intrinsics and assembly mnemonics in some hardware configurations. However, it makes more sense for users to use the scalar types consistent with the `SEW` of vector types.

There is the same issue for `vmv.x.s`, `vmv.s.x`, `vfmv.f.s`, `vfmv.s.f`, `vslide1up.vx`, `vfslide1up.vf`, `vslide1down.vx`, and `vfslide1down.vx`. Use `SEW` to encode the scalar type.

Example:

```
// Use uint8_t for op2.
vuint8m1_t vslide1up_vx_u8m1(vuint8m1_t op1, uint8_1 op2);
```

Mask in Intrinsics

RISC-V "V" extension only has "merge in output" semantic. Intrinsics with mask has two additional arguments, `mask` and `maskedoff`.

```
vd = vop(mask, maskedoff, arg1, arg2)
vd[i] = maskedoff[i], if mask[i] == 0
vd[i] = vop(arg1[i], arg2[2]), if mask[i] == 1
```

In general, the naming rule of intrinsic with mask is

```
INTRINSIC_WITH_MASK ::= INTRINSIC '_m'
```

Example:

```
vadd.vv vd, vs2, vs1, v0.t:
vint8m1_t vadd_vv_i8m1_m(vbool8_t mask, vint8m1_t maskedoff, vint8m1_t vs2, vint8m1_t vs1)
```

There are two additional masking semantics: *zero in output* semantic and *don't care in output* semantic. Users could leverage *merge in output* intrinsics to simulate these two additional masking semantics.

Example:

```
// Zero in output semantic
vint8m1_t vadd_vv_i8m1_m(vbool8_t mask, vzero_i8m1(), vint8m1_t vs2, vint8m1_t vs1)

// Don't care in output semantic
vint8m1_t vadd_vv_i8m1_m(vbool8_t mask, vundefined_i8m1(), vint8m1_t vs2, vint8m1_t vs1)
```

Masked Intrinsics Without MaskedOff

Vector Stores

There is no `maskedoff` argument for store operations. The value of `maskedoff` already exists in memory.

```
Example:

vsb.v vs3, (rs1), v0.t:
void vsb_v_i8m1_m(vbool8_t mask, int8_t *rs1, vint8m1_t vs3);
```

Reduction Instructions

The result of reductions is put in element 0 of the output vector. There is no `maskedoff` argument for reduction operations.

```
Example:

vredsum.vs vd, vs2, vs1, v0.t:
vint8m1_t vredsum_vs_i8m2_i8m1_m(vbool4_t mask, vint8m2_t vs2, vint8m1_t vs1)
```

Merge Instructions

The result of merge operations comes from their two source operands. Merge intrinsics have no `maskedoff` argument.

```
Example:

vmerge.vvm vd, vs2, vs1, v0:
vint8m1_t vmerge_vvm_i8m1_m(vbool8_t mask, vint8m1_t vs2, vint8m1_t vs1);
```

With or Without the VL Argument

(This design decision is still under dispute. We have no final decision about which one is better for users. This RFC includes both version of intrinsics for users.)

There are two variants of intrinsics regarding to `vl`.

1. Explicit vl intrinsics:

   Pass `vl` as an argument to the intrinsics. The `vl` argument helps users to track `vl` values from previous `vsetvl`. It is strict about using `vl` only coming from an earlier `vsetvl` or `vsetvlmax` intrinsics. It inhibits to do arithmetic on `vl` argument.

The design philosophy of explicit vl intrinsics is

- Operations could be entirely defined by the operands.
- In C programmers' point of view, there is no side-effect of intrinsics with `vl` argument.

2. Implicit vl intrinsics:

   No `vl` argument. Users need to know where the `vl` comes by the program execution flow. Users need to take care by themselves if they want to mix multiple `vl` operations.

   The design philosophy of implicit vl intrinsics is

   - The user is responsible for setting the `vl`.
   - It is more faithful to the underlying assembly code.
   - It seems to open the possibility for additional programming errors through additional vl argument.
   - It is more consistent with C operator syntax.

The semantics between these two variants are the same.

The naming rule is

```
INTRINSIC_WITH_VL ::= INTRINSIC '_vl'
INTRINSIC_WITH_MASK_AND_VL ::= INTRINSIC_WITH_MASK '_vl'
```

To avoid users to manipulate `vl` argument in explicit vl intrinsic, there is an opaque type for `vl`.

```
// A possible implementation but it is implementation-defined and opaque to the user.
typedef struct {
  size_t _vl;
} _VL_T;
```

```
Example:

vadd.vv vd, vs2, vs1:
vint8m1_t vadd_vv_i8m1(vint8m1_t vs2, vint8m1_t vs1)
vint8m1_t vadd_vv_i8m1_vl(vint8m1_t vs2, vint8m1_t vs1, _VL_T vl)

vadd.vv vd, vs2, vs1, v0.t:
vint8m1_t vadd_vv_i8m1_m(vbool8_t mask, vint8m1_t maskedoff, vint8m1_t vs2, vint8m1_t vs1)
vint8m1_t vadd_vv_i8m1_m_vl(vbool8_t mask, vint8m1_t maskedoff, vint8m1_t vs2, vint8m1_t vs1, _
```

In the implementation point of view, we could treat explicit vl intrinsics as wrappers of implicit vl intrinsics.

```
Example:

vint8m1_t vadd_vv_i8m1_vl(vint8m1_t vs2, vint8m1_t vs1, _VL_T vl) {
  vsetvl_i8m1(vl_extract(vl));
  return vadd_vv_i8m1(vs2, vs1);
}
```

SEW and LMUL of Intrinsics

`SEW` and `LMUL` are the static information for the intrinsics. The compiler will generate vsetvli when vtype is changed between operations.

```
Example:

vint8m1_t a, b, c, d;
vint16m2_t a2, b2, c2;
...
a2 = vwadd_vv_i16m2(a, b);
b2 = vwadd_vv_i16m2(c, d);
c2 = vadd_vv_i16m2(a2, b2);
```

It will generate the following instructions.

```
vsetvli x0, x0, e8,m1
vwadd.vv a2, a, b
vwadd.vv b2, c, d
vsetvli x0, x0, e16,m2
vadd.vv c2, a2, b2
```

Be aware that when the ratio of `LMUL/SEW` is changed, users need to ensure the `vl` is correct for the following operations if using *implicit vl intrinsics*.

C Operators on RISC-V Vector Types

The semantic of C builtin operators, other than simple assignment, hasn't been decided yet. Simple assignment keeps the usual C semantics of storing the value on the right operand into the variable of the left operand.

Semantic Intrinsics

This section lists all intrinsics with higher semantic naming. It is usually an alias of a vector instruction or a combination of vector instructions.

Vector Copy

It is an alias of the `vmv.v.v` instruction.

```
vmv.v.v vd, vs1:
vint8m1_t vcopy_v_i8m1(vint8m1_t vs1);
```

Splat

It is an alias of the `vmv.v.x` or `vfmv.v.f` instruction. Use ‘s’ to represent scalar argument, regardless integer or floating point types.

```
vmv.v.x vd, rs1:
vint8m1_t vsplat_s_i8m1(int8_t rs1);
vfmv.v.f vd, fs1:
vfloat32m1_t vsplat_s_f32m1(float32_t fs1);
```

Multiply-Add

`vmacc` and `vmadd` are semantically equivalent operations. Provide `vma` intrinsics so the compiler has freedom to choose the best instruction. Same as `vfmacc` and `vfmadd`.

```
vint32m1_t vma_vv_i32m1(vint32m1_t acc, vint32m1_t op1, vint32m1_t op2)
vfloat32m1_t vfma_vv_f32m1(vfloat32m1_t acc, vfloat32m1_t op1, vfloat32m1_t op2)
```

Utility Functions

This section lists all utility functions to help users program in V intrinsics easier.

Bump pointers Through Opaque `vl`

In order to bump pointers from the opaque `vl` value, we define an utility function to extract the value of `vl` from the opaque type.

```
size_t vl_extract(_VL_T vl);

Example:

char *a;

_VL_T vl = vsetvl_i8m1(avl);
a += vl_extract(vl);
```

Vector Initialization

These utility functions are used to initialize vector values. They could be used in masking intrinsics with *zero in output* and *don’t care in output* semantics.

```
Example:

vint8m1_t vzero_i8m1()
```

```
vint8m1_t vundefined_i8m1()
```

Reinterpret between floating point and integer types

These utility functions help users to convert types between floating point and integer types.

`Example:`

```
// Convert floating point to signed integer types.
vint64m1_t vreinterpret_v_f64m1_i64m1(vfloat64m1_t src)
// Convert floating point to unsigned integer types.
vuint64m1_t vreinterpret_v_f64m1_u64m1(vfloat64m1_t src);
```

Reinterpret between signed and unsigned types

These utility functions help users to convert types between signed and unsigned types.

`Example:`

```
// Convert signed to unsigned types.
vuint8m1_t vreinterpret_v_i8m1_u8m1(vint8m1_t src)
```

Reinterpret between different SEW under the same LMUL

These utility functions help users to convert types between `SEW`s under the same `LMUL`, e.g., convert vint32m1_t to vint64m1_t.

Users need to be aware that these reinterpretation functions are not portable when `LMUL` > 1 and `SEW` > `SLEN`, where `SEW` denotes the larger of the two `SEW`s in the conversion. The compiler should show a warning message for users. For example, if users write data using SEW = 32-bits under the following hardware configurations, users will get the following order of data arrangement. If users convert the type to SEW = 64-bits to read out under the same LMUL, the data will be in different order. The data users get will be (4, 0), (5, 1), (6, 2), (7, 3), …, (F, B) under SEW = 64-bits.

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

Byte          F E D C B A 9 8 7 6 5 4 3 2 1 0
v4*n                C       8       4       0   32b elements
v4*n+1              D       9       5       1
v4*n+2              E       A       6       2
v4*n+3              F       B       7       3
```

```
Example:

// Convert SEW under the same LMUL.
vint64m1_t vreinterpret_v_i32m1_i64m1(vint32m1_t src)
```

Utility Functions for Segment Load/Store Types

These utility functions help users to create segment load/store types and insert/extract elements to/from segment load/store types.

```
Example:

// Create segment load/store types.
vint32m2x2_t vcreate_i32m2x2(vint32m2_t val1, vint32m2_t val2)
vint32m2x3_t vcreate_i32m2x3(vint32m2_t val1, vint32m2_t val2, vint32m2_t val3)

// Insert an element to segment load/store types.
vint32m2x2_t vset_i32m2x2(vint32m2x2_t tuple, size_t index, vint32m2_t value)

// Extract an element from segment load/store types.
vint32m2_t vget_i32m2x2_i32m2(vint32m2x2_t tuple, size_t index)
vint32m2_t vget_i32m2x3_i32m2(vint32m2x3_t tuple, size_t index)
```

C11 Generic Interface

Use C11 `_Generic` keyword to choose one of these intrinsics at compile time, based on the types of input arguments.

```
Example:

vint8m1_t vadd(vint8m1_t op1, vint8m1_t op2);
// The compiler will choose the following intrinsic
vint8m1_t vadd_vv_i8m1(vint8m1_t op1, vint8m1_t op2);

vint8m2_t vadd(vint8m2_t op1, vint8m2_t op2);
// The compiler will choose the following intrinsic
vint8m2_t vadd_vv_i8m2(vint8m2_t op1, vint8m2_t op2);
```