RISC-V Vector programming with intrinsics

Table of Contents

RISC-V Vector programming with intrinsics

1. Introduction

RVV defines a new instruction set extension for the open RISC-V ISA (https://riscv.org/technical/specifications/). This extensions adds 32 vector registers, each VLEN-bit wide. Contrary to SIMD extensions (e.g. **ARM Neon** or **x86 SSE/AVX**) **VLEN** is an implementation parameter **not limited to one value**: multiple implementations can have **different VLEN** value while being fully compliant with the **RVV standard**.
Programs for RVV can be built according to vector length agnostic (VLA) principles, meaning that theoretically they can execute on implementations with different **VLEN** (among the range officially allowed by RVV).

1.1 Vector Length and element type

Most instructions in RVV are affected by **global parameters**: **vl, LMUL, SEW**.
Vl is the **vector length**, it defines on how many elements will the next vector operations be executed. The element before the **vl** mark are part of the vector **head** or vector **body**, The element past **vl** are part of the tail.
SEW is the **Selected Element Width**, that is the elementary size (**in bits**) of an element in the input and output vectors.
LMUL is the **group multiplier**, it defines the size of the vector group for vector operation input and output operands, that is the **number of vector register**(s) forming the group. A vector register group is aligned on the group size, e.g. when **LMUL=4**, only the register indices **v0, v4, v8, v12, v16, v20, v24, v28** are allowed, **v0** encodes for **v0v1v2v3**. Fractional **LMUL**s are legal, and constrain **vl** to only a fraction of a single vector register.
For example if **vl=4, SEW=16b,** a **vfmadd.vv** operation performs a **half precision floating-point multiply** and add on **4-element vectors**, that is 4 **FMA**s. **vl** is limited by **VLMAX=LMUL*VLEN/SEW**.
NOTE: some operations (e.g. widening operations) have different element widths for their inputs and operands. **EEW**, the effective element width, may differ from **SEW**.

1.2 Tail policies

As stated in the previous section, the elements past the vector length **vl** are considered as **part of the tail**: elements which are not affected by the current operation.
RVV defines **two tail policies**: ***undisturbed*** and ***agnostic***. In the ***undisturbed*** tail policy, the elements past **vl** in the destination register are left unmodified (that is keep the same value as the one they had before the operation). In the ***agnostic*** tail policy, the tail elements may be left undisturbed or fill in with all **1s**.
The tail can be used to apply the vector operations to an arbitrary number of elements.

1.3 Masking

Most RVV instructions accepts an extra optional **mask operand**. This **bitmask** defines which of the result element should be actually modified by the operation.
Similarly to the **tail behavior**, **RVV 1.0** defines two mask policies: **mask undisturbed** where mask-off elements in the destination register keep the value they had before the operation and **mask agnostic** where mask-off elements can either be undisturbed or written with all **1s**.

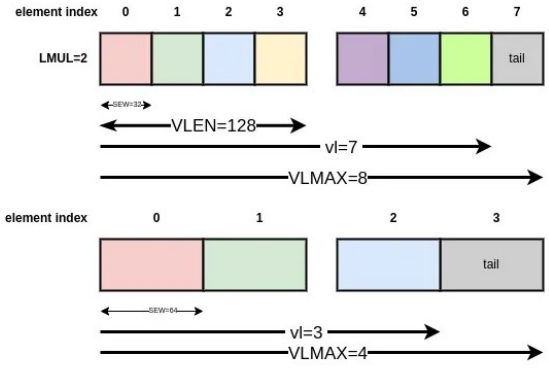


Figure 1.1 The diagram presents an example with a **VLEN=128** bits **LMUL=2** vector group, interpreted as a **vl=7, SEW=32** bits or a **vl=3, SEW=64** bits.

1.4 Example

An assembly program in RVV might look like the following:

```
vsetvli          x1, zero, e64, m1, ta, mu
vfmul.vf         v15, v12, ft5
vfmul.vf         v17, v12, ft6
```

Let us review this snippets in more details.

```
vsetvli x1, zero, e64, m1, ta, mu
```

The first instruction is a **vsetvli** which defines the **SEW:e64 (**64-bit element), the group multiplier **LMUL:m1** (a single register per group), the tail policy: **ta** (**tail agnostic**), the mask policy: **mu** (**mask undisturbed**).

It also request a vector length **vl** and get the actual vector length value. In this specific case, the source register is **zero** (**x0 in the integer register file**) : this means that this instruction requests an **AVL** equal the maximum **vl** value supported for this pair **SEW/LMUL (vlmax)**, and that the actual result **vl** is stored into **x1**.

```
vfmul.vf         v15, v12, ft5
vfmul.vf         v17, v12, ft6
```

The two remaining instructions are vector **floating-point multiplication**, between a **vector** and a **scalar**. The scalar is read in the scalar floating point register file.

The behavior of the instructions is modified by the **vl, LMUL** and **SEW parameter**. Assuming we are executing on an implementation with **VLEN=512 bits**, the **vsetvli** will define **vl** to be **8 (512/64)**, and both **vfmul.vf** will perform double precision (**SEW=64**) vector multiplication between a **8-element vector** and a splat scalar and store the 8-element results in a vector register.

1.5 RVV 1.0: Arithmetic Operations

In this section we review the basic families of RVV 1.0 arithmetic and logic instructions.
Most of those instructions (the most notable exception being the **reductions**) operate **element-wise** as illustrated by the **Figure 1.2** below.
Only the body mask-on elements (for a masked operation) are modified. **Masked-off** and **tail** elements follow mask and tail policies (which are parameters defined in the **`vtype` CSR register**).

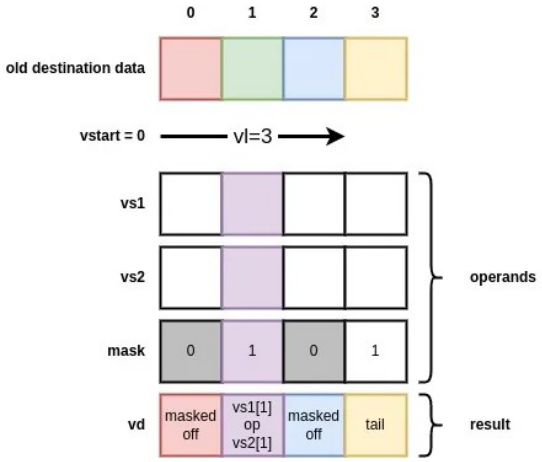


Figure 1.2 Element-wise operation with mask and tail elements

The behavior of those operations is tuned by the global parameters: **`vstart`**, **`vl/LMUL`**, and **`SEW`** (respectively defined by the **`vstart` CSR register** and by subfields of the **`vtype` CSR**).

`vstart` specifies the **first active vector element**, **`vl`** specifies the **number of element**(s) in the **`LMUL`**-wide **vector group** affected by the operation and **`SEW`** specifies what is the **actual format** of the operands.

Optionally those operations **can be masked**: a mask is read from register **`v0`** and the operation is only performed if the corresponding mask bit is set, if the bit is unset, the result value is not defined by the operation but by whether a mask undisturbed or a mask agnostic policy is selected (global parameter).

1.5.1 Vector Integer Arithmetic

https://github.com/riscv/riscv-v-spec/blob/v1.0/v-spec.adoc#11-vector-integer-arithmetic-instructions

RVV 1.0 specifies **standard integer operations**:

- add, sub
- min/max
- zero and sign extensions
- widening add/sub
- add with carry / sub with borrow

1.5.1.1 Operand variants

Most of the integer arithmetic operations admit **three variants**: **`.vv`**, **`.vx`**, and **`.vi`**.

`.vv` describes an **element wise operation between vector registers** while **`.vx`**, (resp. **`.vi`)** describes an **element-wise operation between vector register**(s) and a **splat scalar value** (resp. **scalar immediate**).

The scalar value is read from the scalar register file **`x`** while the immediate value is part of the opcode (generally the **`rs1`** bitfield).

Those variants decrease register pressure for vector operations with at least one uniform operand. Although **`.vx`** requires a register read from the scalar register file.

1.5.1.2 Widening add/sub

Some operations, .e.g. **`vwadd.vv,`** have a **wide result** compared to the size of the operand elements: each operand is interpreted as a vector of **`SEW`**-wide elements, while the result elements are (**`2*SEW`**)-bit wide.

In this case the effective element width (**`EEW`**) of the result differs from the **`SEW`** setting.

There is often an extra variant for widening operations: **`.wv`**. This variant interprets one of the operand as wide (on top of the result). Such operation requires reading up to **`5*SEW`** bits per element operations: **`3*SEW`** bits for the operands, and an extra **`2*SEW`** for the result old-destination-data (tail or mask undisturbed policy) and it requires writing **`2*SEW`** bits per result element.

1.5.1.3 Carry and borrow

RVV uses the mask register (**`v0`**) to store **carry/borrow inputs**. The bit from this mask is combined with an addition (resp. subtract) in the **`vadc`** (resp. **`vsbc`**) instruction family. This instruction produces the addition result. A variant, whose mnemonic is **`vmadc`** can be used to provide the carry mask result.
Since vector instructions produces a single result, it is not possible to get simultaneously the add-with-carry result and carry out.

> **Note**: A superscalar implementation could perform instruction fusion between these two operations and execute them as a single micro-op producing both result and output carry mask.

Any register can be used to store the **result mask**, but as for most RVV instructions only **`v0`** can be used as input mask.

1.5.1.4 Vector Fixed-Point Arithmetic

https://github.com/riscv/riscv-v-spec/blob/v1.0/v-spec.adoc#12-vector-fixed-point-arithmetic-instructions

RVV specifies:

- **saturating** and **averaging** operations.
- fractional multiplies with **rounding** and **saturation**
- scaling **shifts**
- narrowing fixed-point **clips**.

The rounding is performed according to the rounding mode set in the **`vxrm`** register (a **`csr`** register part of **`vcsr`**).
Saturating operations **saturate to minimal or maximal values on overflow** (and set the status bit **`vxsat`**).
For averaging operation, the operation result is right shifted by one bit (equivalent to a division by 2).
The fixed point formal for fractional multiplies depends on **`SEW`** : it assumes operands with (**`SEW-1`**) fractional bits and produces a result with (**`SEW-1`**) fractional bits too, performing rounding to remove the extra fractional bits.

1.5.1.5 Vector Floating-Point Arithmetic

https://github.com/riscv/riscv-v-spec/blob/v1.0/v-spec.adoc#13-vector-floating-point-instructions

RVV defines a **standard set of floating-point** addition, subtraction, multiply and various variant of **fused multiply-add** which operate with a **single final rounding**.

Operations exist in both single-width and widening forms. Fused multiply and add instructions only exist in **destructive** form: **one of the operand is also the destination register** overwritten by the operation.

One of the most **demanding operations** in terms of operands is **`vfwadd.wv vd,vs2,vs1`**: in undisturbed configuration (tail or mask), when **`LMUL=1`**, this operations may require **5 register inputs** (one for **`vs1`** and two for each of **`vs2`** and **`vd`**) and may produce **2 register outputs** (similarly to the case described for integer widening instructions).

The floating-point set also contains **vector division**, **square root** and estimate operations for reciprocal (**`vfrec7.v`**) and **reciprocal square root** (**`vfrsqrt7.v`**).
These estimates provide approximation with 7-bit accuracy, useful to implement Newton-Raphson based division or square root approximations or argument reductions for elementary functions such as logarithm.

Finally the floating-point set includes floating-point min/max, sign injection, numerous format conversions and compare operations.

Compare operations provide their result as a mask: the boolean result of each element-wise comparison is encoded by the corresponding mask bit.

1.5.1.6 Vector Reductions

`Specification:` **`https://github.com/riscv/riscv-v-spec/blob/v1.0/v-spec.adoc#vector-reduction-operations`**

RVV 1.0 specifies a few reduction operations. Those operations expect a **scalar operand** (**element 0** from the vector register operand **`vs1`**) and a vector group (**`vs2`**) and reduce all the active elements from **`vs2`** with the scalar operand into a single scalar value stored into element 0 of the destination register **vd**.

RVV 1.0 specifies (f)min(u)/(f)max(u), and/or/xor, single-width and widening sum reductions (integer or floating-point).

RVV 1.0 offers two types of f**loating-point sum reductions**: **`ordered`** and **`unordered`**.

`ordered` forces a **sequential order of evaluation** for the intermediary additions, its result is deterministic across implementations and can be applied to the vectorization of reduction loops in most language when a sequential semantic is expected (e.g. C/C++ when **`-ffast-math, -fassociative-math`** or another aggressive optimization option are not applied).

`unordered` offers more room for micro-architectural implementations. As long as the implementation is deterministic it can implement the **unordered** reduction as a multi-level binary tree (even allowing different precisions in the tree).

1.6 RVV 1.0: mask and masked operations

As stated in the previous section, **most operations in RVV 1.0 can be masked**, that is an extra mask operand can be provided through **`v0`** to select which elements will be actually modified by the operation.

Most operation use the **25-th opcode bits** to select if the operation is masked or not.
If the operation is **unmasked**, (**`inst[25]=1`**) **all body elements are considered active**.
If the operation is **masked** (**`inst[25]=0`**), then **each element lane** is associated with a bit in **`v0`**, i**f the bit is set then the lane operation is performed**, else the result is either left unmodified (undisturbed or agnostic mask policy) from its previous value in **`vd`** or it can be filled with **all 1s** (**agnostic** mask policy).

As for **tail**, agnostic policy allows the implementer to chose between **all1s** and undisturbed elements, and this on a per-lane basis.

This choice can be exploited to reduce the number of value a processor using register renaming must used: an old-destination-data read can be saved if the implementation choses to fill in all1s in the agnostic case. In **RVV 1.0**, only **`v0`** can be used as the mask operand for a masked operation.

1.6.1 `VLEN` and mask size

Because the group multiplier, **`LMUL`**, cannot exceed 8, the mask size of a mask, even for the largest possible vector group, is **`VLEN`**. This means that a single vector register is wide enough to fit a mask even for an operation with **`LMUL=8`** and the smallest **`SEW`**, there is no need to combine multiple registers to build a mask register.
For largest element width, only a portion of the **`VLEN`-wide vector register** is necessary.
For example for **`SEW=64-bit`**, an **`LMUL=8`** vector group contains **`VLEN/SEW*LMUL=VLEN/8`** elements: only **1/8 of the vector register** contains a **valid mask bit**.

1.6.2 Operation producing masks

The exist a few **instructions to manipulate masks** that we will review in the next paragraphe, but there are also specific instructions to produce masks for non-mask operands.
These instructions include the integer and floating-point comparisons: those instructions perform element-wise comparisons and produce a bit per comparison, concatenating them into a result mask.
Those instructions can be masked: only the active elements comparison are performed, the result of inactive element depends on the mask policy.

1.6.3 Operation with masks

`https://github.com/riscv/riscv-v-spec/blob/v1.0/v-spec.adoc#sec-vector-mask`

RVV 1.0 also specifies **operations on masks**, such as **`vmand.mm`**, those operations accept mask operands and produce a mask result. Those mask operations do not have restriction on mask register, **any vector register can be used as a mask input or output**. The final mask operation will need to target **`v0`** or the mask result will have to be moved into **`v0`** before being used as a mask operand.

RVV 1.0 specifies standard bitwise logic operations on mask:

`https://github.com/riscv/riscv-v-spec/blob/v1.0/v-spec.adoc#151-vector-mask-register-logical-instructions`.

RVV 1.0 also specifies:

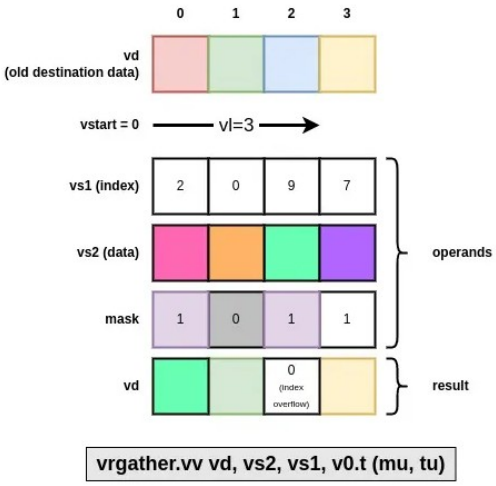
- population count (popcount) on mask: **`vcpop.m`**, useful to count the active bit set in a mask, this instruction writes its result to a scalar register.
- p leading zero count on active element: **`vfirst.m`**, this instruction also write to a scalar register, this operation can be masked it counts inactive elements as zeros in the **`vs2`** source.
- instructions to build mask around the first active 1: **`vmsbf.m`**, **`vmisf.m`**, **`vmsof.m`**
- **`viota`** and **`vid`** to build accumulated **active indices and indices vector**s.

1.7 RVV 1.0: permute operations

SIMD and **Vector** instructions sets needs to offer **permutation instructions** to allow the programmer to manipulate and re-organize data within vector registers. **`RVV 1.0`** offers such instructions in multiple flavours, we will review the permutation instruction families in this post and provide some illustrations.

1.7.1 Full Permute

The most generic operation is the **full permute**, which is also the most operand consuming as it often requires a full vector operand for indices and the most expensive in hardware. **RVV 1.0** specifies two flavors of full permutes: **`vrgather.vv`** and **`vrgatherei16.vv`**.
The first one, **`vrgather.vv`**, accepts an **index vector** with indices as wide as the data elements (**`SEW`**).



vrgather.vv vd, vs2, vs1, v0.t (mu, tu)

The second variant, **`vrgatherei16.vv`**, uses **16-bit wide indices** whatever the **`SEW`**. 16-bit is enough to contain an index for the largest possible vector (**`LMUL=8, SEW=8-bit, VLEN=65536`**). This variant is useful to save index space when manipulating large elements and to unify the index management.

1.7.2 Vector Splat

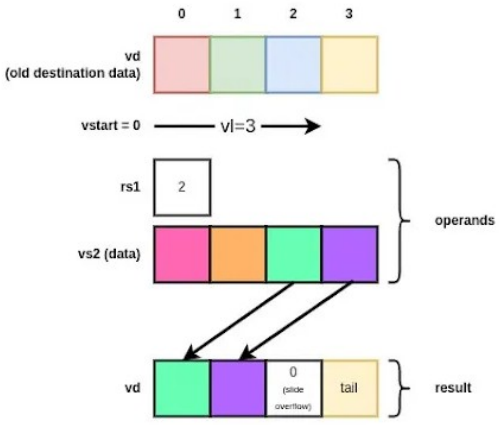
RVV 1.0 defines two **indirect splat** instructions:
`vrgather.vx` and **`vrgather.vi`**, both of them **replicate an element to all the active result elements**, the replicated value is selected in the data source from the index read from a scalar register (**`vrgather.vx`**) or from an immediate in the opcode (**`vrgather.vi`**).
These instructions can be implemented with much less hardware than the full permute.

1.7.3 Vector Slide

https://github.com/riscv/riscv-v-spec/blob/v1.0/v-spec.adoc#163-vector-slide-instructions

RVV 1.0 specifies **simple slide permutes**: the input vector active elements are translated by some amount:

- **`vslide1up.vx`** translates the element by **1 index up** and insert a value read from a scalar integer **`x`** register in the least significant element position (index **0**)
- **`vslide1down.vx`** translates the element by **1 index down** and insert a value read from a scalar integer **`x`** register in the most significant active element position (index **`vl-1`**)
- **`vslideup.v[xi]`** translates the element by **`<n>`** **index up** and leaves the **n lowest elements** undisturbed. **`<n>`** is either read from a scalar register (**`.vx`**) or an immediate (**`.vi`**)
- **`vslidedown.v[xi]`** translates the **`vl`** active element by **`<n>`** **index down**, any element which should have been read outside the source group is replaced with **0s**. **`<n>`** is either read from a scalar register (**`.vx`**) or an immediate (**`.vi`**)

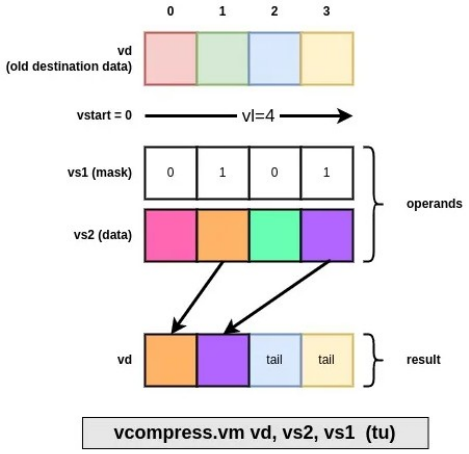


vslidedown.vx vd, vs2, rs1 (tu)

There exists floating-point versions of the **`vslide1*`**instructions: **`vfslide1up.vf`** and **`vfslide1down.vf`**, which read the inserted element from the scalar floating-point register file (**`f`** registers).

1.7.4 Vector Compress

RVV 1.0 defines a compression/group instruction: **`vcompress.vm`**, this instruction extract active elements from a vector source operand **`vs2`** based on a mask in **`vs1`** and group the extracted elements **in the least significant elements of the result**.



RVV 1.0 specification also lists the whole register move operations as part of the **vector permute instructions**.

There are 4 such instructions: **`vmv1r.v, vmv2r.v, vmv4r.v, vmv8r.v`**.

These instruction are not impacted by the **`vtype.vlmul`** parameter value: **`EMUL`** is directly encoded in the opcode. Similarly, the effective **`vl`** value **`evl`** is defined by **`VLEN/SEW*NREG`**, where **`NREG`** is the **`EMUL`** size encoded in the opcode.

Those instructions are used to copy the vector register file efficiently without needing to modify the **`vtype`** parameter values.

All those permutation instructions are really handy to manipulate elements in registers. They can also be used to implement small size **lookup table**, load value to vector register from the scalar register files ...

1.8 Vector loads and store

RVV 1.0 defines several families of memory operations, they can be split across addressing modes:

- unit-strided load/store
- index-unordered load/store
- strided load/store
- index-ordered load/store
-

In this section we will focus on the unit-strided and strided accessing modes then we will review the indexed addressing mode .

For most loads and stores, the effective element width (**`EEW`**) is encoded in the opcode and not driven by **`SEW`**: e.g. **`vle8`** is a unit-strided load of **8-bit elements** and **`vse32`** is a **unit-strided store of 32-bit elements**.

1.8.1 Simple memory operations

1.8.1.1 Unit-strided operations

https://github.com/riscv/riscv-v-spec/blob/v1.0/v-spec.adoc#74-vector-unit-stride-instructions

RVV 1.0 specifies **4 unit stride loads** and **4 unit stride stores**, one operation for each **`SEW` between 8-bit and 64-bit**. They are currently no operations for **SEW=128-bit** and upward. The **unit stride loads** (resp. **stores**) load (resp. store) **`vl` elements from** (resp. **to**) memory.

The operation **can be masked** as most arithmetic operations, when enabled, the mask operand is read from **`v0`** and only the active elements are read from or stored to memory.
Inactive elements follow the mask policy. Unactive accesses are not performed, and cannot raise errors.

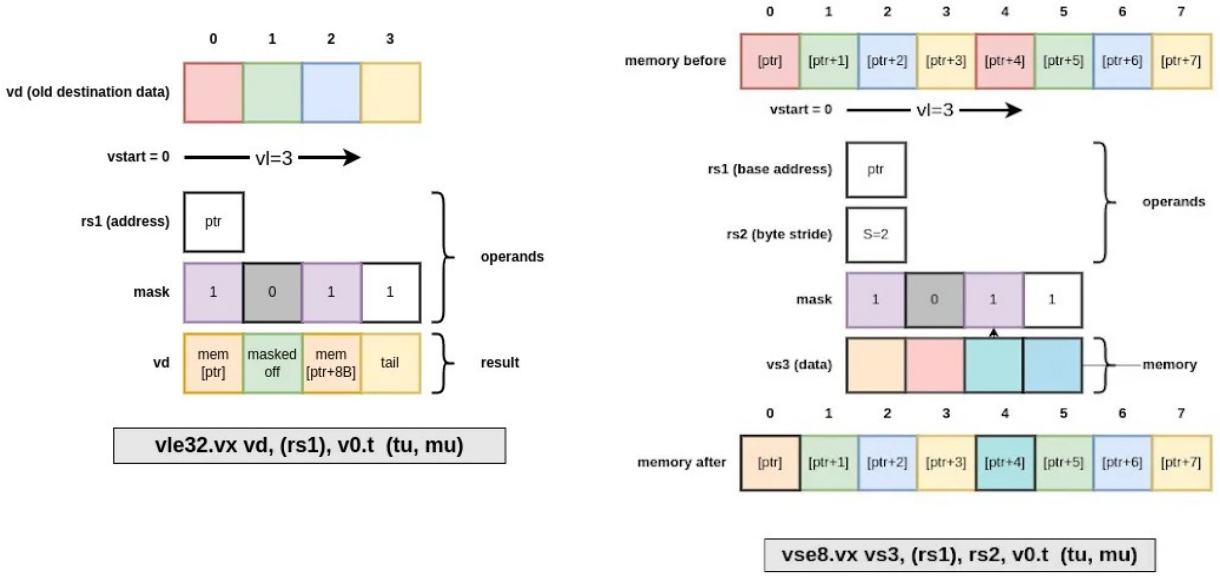


Figure 1.7 Unit-strided (a) and strided (b) load/store operations

1.8.1.2 Strided operations

https://github.com/riscv/riscv-v-spec/blob/v1.0/v-spec.adoc#75-vector-strided-instructions

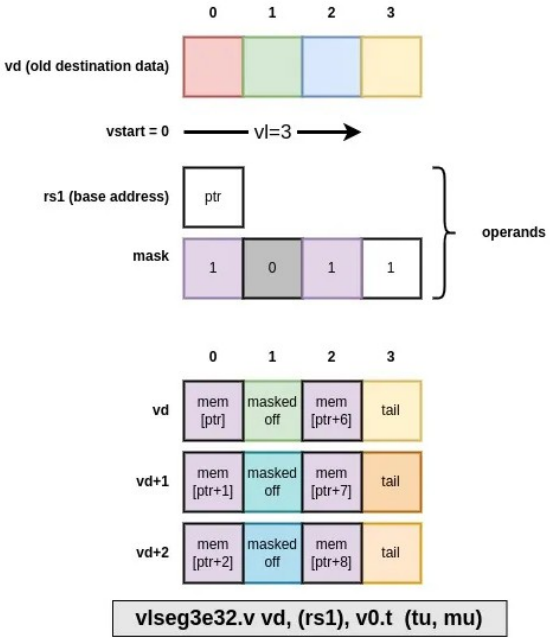
RVV 1.0 offers **constant-stride load and stores**. The signed byte stride is read from the **`rs2`** register and encodes the unit address increment between two vector elements. **If the stride is 0** the implementation is allowed to optimize the memory access and not realize all of them: in that case all source elements will be written at the same position. If the stride is non-zero, all memory access must "appear" as done separately, although they can appear **out-of-order** within a vector operation.

1.8.1.3 Segmented memory operations

https://github.com/riscv/riscv-v-spec/blob/v1.0/v-spec.adoc#sec-aos

RVV 1.0 defines segmented loads **`vlseg<nf>e<eew>.v`** and stores **`vsseg<nf>e<eew>.v`** to manage loading and storing of structures **layed out as array of structure** (**AoS**) in memory and transform them into **structure of arrays** (**SoA**) **layout in vector registers**.
As most memory operations the element-width **`eew`** is part of the mnemonic/opcode, the number of fields **`nf`** is also part of the mnemonic/opcode. **`nf`** encodes the **number of (homogeneous) fields/elements** per structure/segment and **can vary from 1 to 8**.

The effective group multiplier **`EMUL`** must verify **`EMUL*nf <= 8`**.



A segmented load will store **each field in a different destination register**, mask applies to the segments (and not to the fields) and **`vl`** indicates the number of segments to be loaded.

There also exist strided versions of segmented loads and stores:

`vlsseg<nf>e<eew>.v` and **`vssseg<nf>e<eew>.v`**.

An extra **`rs2`** operand provides a byte stride between two segments (the byte stride can be zero or negative).

1.8.2 Indexed memory operations

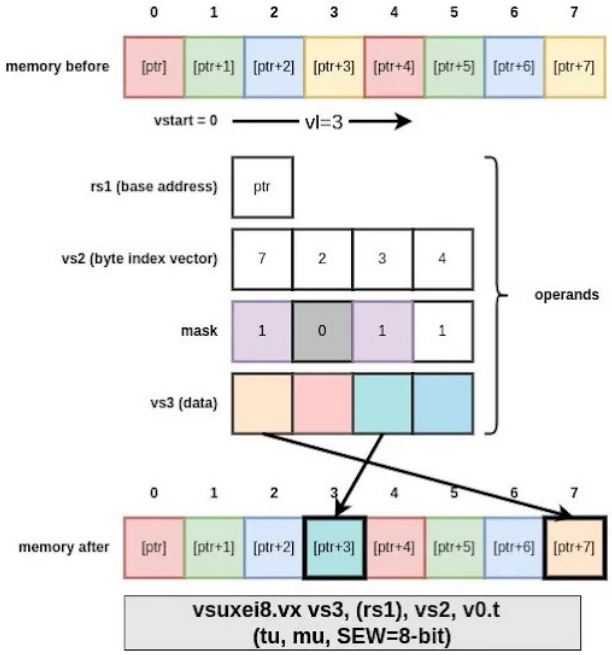
https://github.com/riscv/riscv-v-spec/blob/v1.0/v-spec.adoc#76-vector-indexed-instructions

The indexed memory operations are the memory equivalent to the **`vrgather`** family of instructions: they implement load gathers and store scatters.

Contrary to the **segmented loads and stores**, the indexed memory operations do not assume a regular memory layout. This lack of regularity often means they expose lower throughput and higher latency than other memory operations.There exist two variants for the indexed operations: **unordered** and **ordered**.

While **unordered** can appear to be executed in any order, **ordered** instructions "preserve element ordering on memory accesses".
The former lets the micro-architecture re-order accesses, for example favoring access coalescing, even if this ordering is visible, while the latter forces the micro-architecture to ensure everything eventually happens as if the accesses were executed in element order.

Indexed vector loads and stores read the memory address from the **`rs1`** operand, the index vector from the **`vs2`** operand, indexed stores read the data from the **`vs3`** operand and indexed loads write the result to the **`vd`** operand.
The vector loads and stores use two different element widths: one for the data, read from the global **`vtype SEW`** configuration, and one for the **indices**, **encoded in the opcode**.



1.8.3 Whole register instructions

https://github.com/riscv/riscv-v-spec/blob/v1.0/v-spec.adoc#79-vector-loadstore-whole-register-instructions

There exist load and store versions of the whole vector register move operations (**`vmv<n>r`**): whole vector register loads **vl<n>re<ew>** and whole vector register stores: **vs<n>r**.

As for whole register moves, the group multiplier (**`EMUL`**) is encoded in the instruction opcode (though the **`nf`** field which encodes a **`NFIELD`** value, supported values are 1, 2, 4 and 8), **the register group must be aligned** as any register group (e.g. **`v0`** is a valid source for **`vl8re16`** but **`v4`** is not).

2. How to read a RISC-V Vector assembly instructions

In this chapter we detail how to interpret the various components of a RVV assembly instruction.

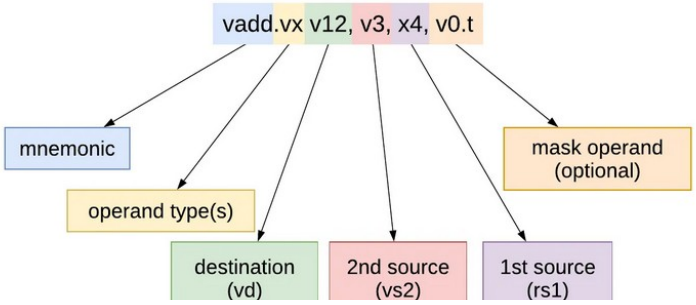
2.1 Overview

We will draw some generalities from a practical example **a masked integer vector-scalar addition:**

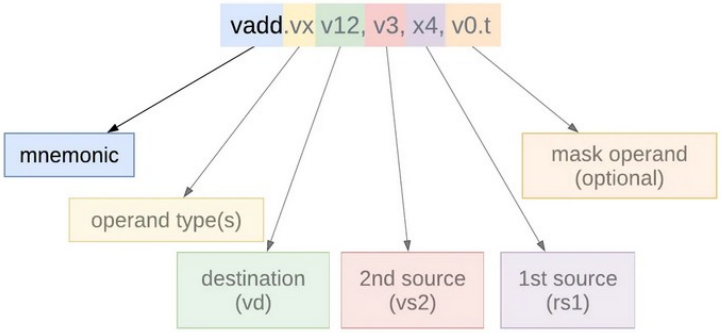
```
vadd.vx v12,v3,x4,v0.t
```

The following diagram illustrates the **6 main components** of any **RVV assembly instruction**.
Most of those components have numerous variants and some of them are optional (e.g. the mask operand).

2.1.1 Elements:



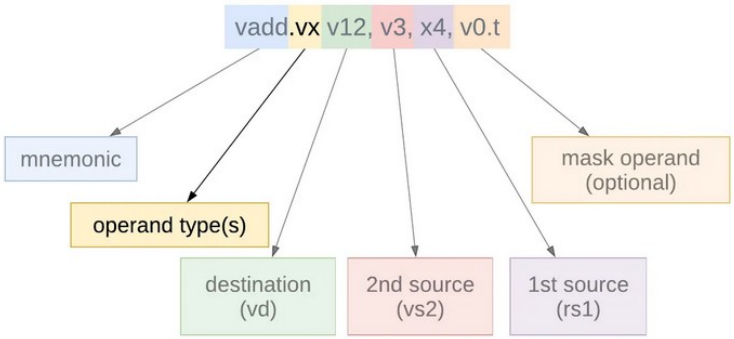
2.1.2 Mnemonic:



The first component is the **mnemonic** which describes which operation should be performed by the instruction (e.g. in our case **`vadd`** will perform a **vector add**).

The mnemonic often describes the destination type: for example **`vadd`** is a vector add with a **single-width destination**, while **`vwadd`** is a **widening vector add** (the destination elements are wider than the main input operands) and **`vmadc`** is a vector **addition with carry** returning a **mask of output carries**.

2.1.3 Operand types



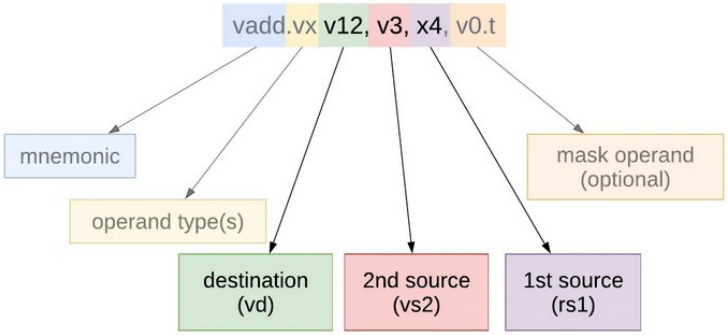
The second component is the **operand type(s)**. It describes on what type of operand(s) the operation is performed. The most common is **`.vv`** (vector-vector) which often means that the instruction admits at least two inputs and both are single-width vectors (non widening operation).

The list of various possibilities includes:

- **`.vv`** operation between two (or three) single-width vectors, e.g. **`vmul.vv`**
- **`.wv`** operation **between two (or three) vectors**, **`vs1`** is single-width while **`vs2`** and **`vd`** are wide operands/destinations (**`EEW=2*SEW`**), e.g. **`vwmacc.wv`**
- **`.vx/.vf`** operation between **one or multiple vector**(s) and a scalar (read from a general purpose register: **`x`** or floating-point register **`f`**), e.g. **`vfadd.vf`**, the s**calar operand is splat** to build a vector operand and added to each active vector element
- **`.wx/.wf`**: operation **between a scalar and a wide second vector** operand, e.g. **`vfwadd.wf`**
- **`.vi`** operation **between one or multiple vector**(s) and an **immediate**. The immediate is a 5-bit [1] wide value encoded in the opcode **`rs1/vs1`** field, **e.g. `vsub.vi v2,v3,7`**
- **`.vs`**: operation **between a vector and a single element** (scalar) contained in a vector register, e.g. **`vredsum.vs`** (the single scalar is used to carry the reduction accumulator). In vector crypto, **`.vs`** defines an operation between a vector and a single element **group**[2].
- **`.vm`**: operation between a **vector and a mask**, e.g. **`vcompress.vm`**
- **`.v`**: operation **with a single vector input**, e.g. **`vmv2r.v`**, may also be used for vector loads (which have scalar operands for the address on top of a single data vector operand: **`vle16.v`**.
- **`.vvm/.vxm /.vim`** operation **vector-vector/vector-scalar/vector-immediate** with a **mask** operand (e.g. **`vadc.vvm`**, addition between two vectors with an extra mask operand used as an input carry vector)

The conversions constitutes a category of their own for the operand types, because the mnemonic suffix describes: the destination format, the source format, and the type of operand.
For example **`vfcvt.x.f.v`** is a vector (**`.v`**) conversion from floating-point element (**`.f`**) to signed integer (**`.x`**) result elements. **`.xu`** is used to indicate unsigned integers, **`.rtz`** is used to indicate a static round-towards-zero rounding mode.

2.1.4 Destination and sources



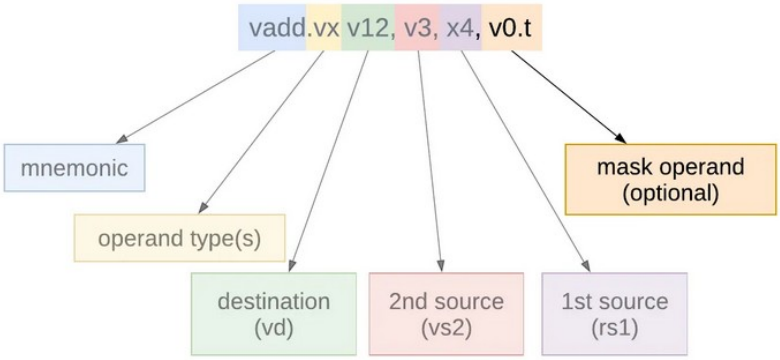
In the assembly instruction, **destination and sources follows the mnemonic**. The destination is the first register to appears, followed by one or multiple sources.

Each of those element encodes a **register group**. The destination and source operands register groups are represented by the first register in the group (for example if **`LMUL=4`**, then **`v12`** represents the 4-wide register group **`v12v13v14v15`**). Thus the actual register group depends on the assembly opcode but also on the value of **`vtype`:** it is context sensitive. Most **RVV** operations have a vector destination, denoted by **`vd`**, some may have a scalar destination (e.g. **`vmv.x.s`** with a **`x`** register destination or **`vfmv.f.s`** with a **`f`** register destination) and others have a memory destination such as the vector stores, e.g. **`vse32.v`**.

There can be one or two sources: **`vs2`** and **`vs1`** for vector-vector instructions. If the operations admits a scalar operand, or an immediate operand then **`vs1`** is replaced by **`rs1`** (respectively ***imm***), e.g. **`vfadd.vf v4,v2, ft3`**.
Vector loads have a memory source, e.g. **`vloxei8.v vd,(rs1),vs2 [, vm]`** which has a scalar register as address source and a vector register as destination source.

RVV defines **3-operand instruction**, e.g. **`vmacc.vv`**. For those operations the destination register **`vd`** is both a source and a destination: the operation is destructive: one of the source operand is going to be overwritten by the result.

2.1.5 Mask operand



Most **RVV** operation **can be masked**. When an operation is masked only a subset of body elements are active and operated upon.
In such case an extra vector register operand appended at the end of the assembly instruction is used as a mask to determine which elements are active and which elements of the result will be copied from the old destination value or filled with a predetermined pattern (the behavior is selected by the **`vconfig.vta`** tail policy flag).
RVV 1.0 only supports the register **`v0`** as a mask operand and true bit as active mask value: the element is considered active if the bit at the corresponding index is set to **`1`** in **`v0`**, and inactive if it is **`0`**.
This is what is encoded by the last operand of our example: **`v0.t`** (for **`v0`** "true as active"). If this last operand is missing, then the operation is unmasked (all body elements are considered active). **`v0.t`** is usually encoded by clearing bit 25 of the opcode, when this bit is set, the operation is unmasked.

2.2 Vector Element Groups

RISC-V Vector extension has defined multiple ways to organize vector data over one or multiple vector registers. Vector register groups defined by **RVV 1.0**: the capacity to group together multiple vector registers into a meta (larger) vector register which can be operated on as a single operand by most instructions and thus extend the size of vector operands and results.
Recently a new concept was introduced: **vector element groups:** considering multiple contiguous elements as a single larger element and operate on a group as if it was a single element.

2.2.1 Definition

A vector element group is defined by an effective element width (**`EEW`**) and an element group size (**`EGS`**), it is a group of **`EGS`** elements, each **`EEW`-bit wide**. The total group width (in bits) is called the **E**lement **G**roup **W**idth (or **`EGW, EGW = EEW * EGS`**).

The element group is useful to manipulate **multiple data elements** which make sense as a block (e.g. a **128-bit ciphertext** for the AES cipher algorithm) without the need to define large element widths and implement their support in hardware.
An element group can be fully contained in one vector register or can overlap multiple registers. In the former case, a single vector register can contain multiple element groups.
An element group can also have **`EGW`** larger than an implementation **`VLEN`**; in this case a multi-register group is required to fit a single element group.
The same constraints as any vector register group apply: the register group is encoded by its first register whose index must be a multiple of the group size.

`EEW` can either be specific to the opcode or defined through **`SEW`**. For example most vector crypto instructions defines **`EEW`** from **`SEW`** (even if only a small subset of values are legal): it is required to define **`vtype`** properly before executing a vector crypto instruction.
This is in particular useful to reuse the same instruction for different algorithm: e.g. setting **`SEW`=32 bits, `vl`=4** and executing a **`vsha2c`** will perform a **SHA-256 message compression**, while setting **`SEW`=64** bits and **`vl`=4** and executing a **`vsha2c`** will perform a **SHA-512 message compression**.

2.2.2 Constraints on `vl` and `vstart`

The vector length (**`vl`**) of an element-group operand is counted in elements and not in terms of groups.
And the vector length **must be a multiple of the element group size `EGS`**, the same constraint applies to **`vstart`**. Other cases are reserved. This mean that operand on a single element group with **`EGS=8`**, requires setting **`vl`** to **8**, operating on **3 elements groups** requires setting **`vl`** to **24** and so forth.

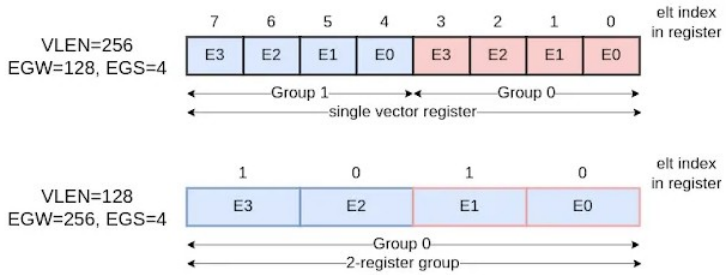
The case when **`vl`** (or **`vstart`**) is **not a multiple of `EGS`** is reserved and executing an instruction working on element group may result in an illegal instruction exception being signaled.

2.2.3 Masking and element groups

The specification of element groups leaves a lot of room when it comes to masking: masking support is defined on a per operation basis.
The concept allow for per-element masking and mask setting or per element-group (if any or all mask bits corresponding to elements in the group are set).
The concept does not seem to cover the cases where a single mask bit corresponds to a full group regardless of the actual number of element in the group: mask **bit `0`** would correspond to **group 0**, mask **bit `1`** to group 1 ...
This case would behave similarly to masking with **`SEW=EGW`**.
In the only existing use case (the vector cryptography extension), none of the instructions defined as operating on element groups support masking, so the problem of element group masking implementation will be delayed to a future extension.

2.2.4 Examples and use cases

The following diagram illustrates two examples of element groups. The top element group has **`EGW`** half as wide as **`VLEN`** and so two element groups fit in a single vector registers. The bottom example has **`EGW`** twice as wide as **`VLEN`** and so a 2-register group is required to fit a single element group.



The element group concept has been first used by the vector cryptography extension proposal (draft under architectural review at the time of writing).
Different element groups configurations are used:

- 4x 32-bit elements for AES, SM4, GHMAC and SHA-256
- 8x 32-bit elements for SM3
- 4x 64-bit elements for SHA-512
-

As mentioned earlier, the vector crypto extension requires SEW to be set to an element width value supported by the instruction being execution, and considers all other cases as reserved.

2.2.5 Difference between element groups and larger SEW

One of the relevant question one may ask regarding element groups is: what is the difference between an element group with **`EGW=128`** (e.g. **`EEW=32`** and **`EGS=4`**) and a single element with **`SEW=128`** **?**

The **first fact** is that currently **`SEW`** (as defined by the **`vsew`** field of the **`vtype`** register) cannot exceed 64 bits (RVV 1.0 spec section).
Although the **`vsew`** field is large enough to accommodate larger values, those encodings are currently reserved. So element groups bring support for larger data blocks without requiring new **`vsew`** encodings.

A **second fact** is that support for element groups is possible even if **`EGW`** exceeds **`ELEN`**, the maximal element width supported by the architecture.
In fact **`EGW`** can even exceed **`VLEN`**: this case is part of the element group concept and is supported by laying down an element group across a multi-register vector register group.

This is not currently supported for single element.

This translate a fact that operations using element groups do not need an actual datapath larger than **`ELEN`**, most operation are performed on **`ELEN`**-bit wide data or less.

A **third fac**t is that supporting element groups can be done mostly transparently from the micro-architecture point of view: there is not much difference between a single element group of **EGS=4** and **`EEW=32`** and a 4-element vector with **`EEW=32`**: internal element masking and tail processing can reuse the same datapaths,

3. RISC-V Vector Programming in C with Intrinsics

In the previous chapters, we introduced the basics of RISC-V vector extension (RVV). If you are not familiar with this extension, you should consult these chapters first.
Once you know the **basic of `RVV`**, the next step is to try it out. The most basic way is assembly programming.

There is a somewhat easier and more modern way to program using RISC-V Vector **directly in the C/C++**: **RVV intrinsics**. RVV instructions can be called within a C/C++ program directly through **intrinsics**: **low-level functions** exposed by the compiler.
Each of those l**ow-level functions** has almost a **one-to-one mapping** with the corresponding RVV instruction making low-level RVV programming accessible without assembly expertise.

Current GCC and clang toolchains can easily assemble asm1 programs which make use of RVV instructions (**as long as support for RVV is enabled**, often by listing the **`v`** extension in the **`-march rv64gcv`** ISA string).

3.1 A first example of RVV intrinsic

The following is an example of **RVV intrinsics** to perform an **integer vector addition**, **`vadd.vv`**, between **two vector register groups** of one vector register (**`m1`**), of **32-bit elements** (**`i32`**).

```
vint32m1_t __riscv_vadd_vv_i32m1(vint32m1_t,vint32m1_t,size_t);
```

Intrinsic naming follow a regular scheme summarized in the Figure 3.1 below:

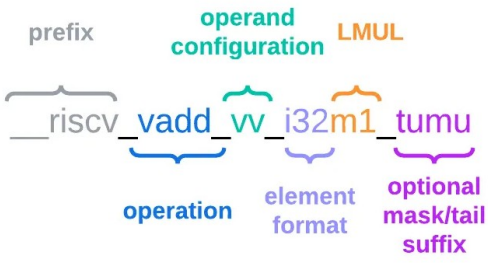


Figure 3.1 Integer vector addition, **`vadd.vv`** intrinsic instruction

If you are familiar with RVV you will already have noted that the function name also contains a description of the **vector configuration** (element size, group multiplier **`LMUL`**) which are not generally encoded in a RVV instruction opcode. Moreover, **most intrinsics expect a vector length parameter**.

This simplify the programming model: all the information about the vector configuration for an operation are embedded in the intrinsics. This include the tail and mask policies: the intrinsic suffix encodes this piece of information. For example no suffix means unmasked and tail agnostic and `_tu` means unmasked and tail undisturbed policies.

```
__riscv_vadd_vv_i32m1_tu(vint32m1_t,vint32m1_t,size_t);
```

Embedding all those configuration items puts more burden on the compiler which has to generate and optimize the sequence of vector configuration instructions (**`vset*`**) and vector operations: factorizing local vector configuration when possible.

There are a **lot of RVV intrinsics**, **too many to count**. The specification and documentation of RVV intrinsics is an on-going effort by RVIA (RISC-V International), with **Yueh-Ting (eop) Chen being one of the main contributors**.

The project can be found here: **https://github.com/riscv-non-isa/rvv-intrinsic-doc/tree/main**.
The intrinsics require compiler support: **`LLVM 17`** and **`gcc`** trunk (dev branch) supports the latest version, **v0.12**, of **RVV** intrinsic specification.

3.2 Why so many intrinsics ?

3.2.1 Many vector types

First of all, there are a lot of possible types for intrinsics operands/destinations (documented in the section Type System of the intrinsics documentation).
These possibilities correspond to the valid cross combination of:

- **element type** (**floating-point, signed integer, unsigned integer, boolean,** …)
- **element-width** (**8, 16, 32,** or **64 bits**)
- **vector group multiplier** (**1, 2, 4, 8** or the fractional **1**/**2, 1**/**4, 1**/**8**)

Here are a few examples:

```
// type for one-register groups of signed 8-bit integers
vint8m1_t
// type for 4-register groups of unsigned 8-bit integers
vuint8m4_t
// type for 2-register groups of 16-bit floating-point values (half)
vfloat16m2_t
```

3.2.2 One intrinsic for each pair (operation, type)

For **each vector operation**, one explicit RVV intrinsic is defined for **each specific set of input**/**destination** types for inputs and destination.
The number of possible signature types creates a very large set of intrinsics for every single RVV instruction.

For example, here is a very small subset of intrinsics to perform integer vector addition between two vectors:

```
// addition of register groups of one vector reg of 32-bit elements
vint32m1_t __riscv_vadd_vv_i32m1(vint32m1_t, vint32m1_t, size_t);
// addition of register groups of two vector regs of 32-bit elements
vint32m2_t __riscv_vadd_vv_i32m2(vint32m2_t, vint32m2_t, size_t);
// addition of register groups of two vector regs of 64-bit elements
vint64m2_t __riscv_vadd_vv_i64m2(vint64m2_t, vint64m2_t, size_t);
```

All those intrinsics map to a single vector instructions, **`vadd.vv`**.

3.2.3 Implicit (overloaded) API

Fortunately the intrinsic API also provides an **implicit** (overloaded) **name scheme** which allows the programmer to use a **single overloaded function** (e.g. **`__riscv_vadd`**) to call **all the `EEW`/`LMUL` variants**.
They are some limitations to this scheme; for example there is no overloaded function for intrinsics with only scalar types, which means there is **no overloaded function for unmasked unit-strided load**.

3.2.4 One intrinsic for each mask/tail configurations

As we have seen previously, base intrinsics can be extended by an **optional suffix** to indicate if the operation is **masked**/**unmasked** and with which **policy for unactive elements** and what is the tail policy. The available suffixes are detailed here.
They are **6 possible suffixes** (including the default empty suffix).

For example the following is the intrinsic for a 16-bit element masked unit-strided vector load:

```
vfloat16mf4_t __riscv_vle16_v_f16mf4_m(vbool64_t vm,
                                       const _Float16 *rs1,
                                       size_t vl);
```

These suffixes also exist in the implicit naming scheme.

3.2.5 Are all the intrinsics actual RVV instructions ?

Some of the intrinsics do not necessarily map to real RVV instructions. For example selecting a single register out of a multi-register vector group:

```
vint8m1_t __riscv_vget_v_i8m8_i8m1(vint8m8_t src, size_t index);
```

Similarly re-interpreting a vector of unsigned 32-bit elements as a vector of single precision 32-bit elements requires an intrinsics:

```
vfloat32m1_t __riscv_vreinterpret_v_f32m1_u32m1 (vuint32m1_t);
```

The underlying data does not change within the register group, it is just re-interpreted differently for the next operations. This is due to the fact that the RVV C intrinsics type system distinguish multiple types of 32-bit element, which is not the case in assembly: **`vadd.vv`** and **`vfadd.vv`** can be executed seamlessly on the same inputs or one on the result of the other without requiring any extra operation in between (even if the cases where it actually make sense may be few).

3.3 Example: Vector-Add with RVV Intrinsics

Let's implement the basic vector example, a **32-bit floating-point vector-add**, using the intrinsics. In this vector-add we will define the function:

```
/** vector addition
 *
 * @param dst address of destination array
 * @param lhs address of left hand side operand array
 * @param rhs address of right hand side operand array
 * @param avl application vector length (array size)
 */
void vector_add(float *dst,
                float *lhs,
                float *rhs,
                size_t avl);
```

`vector_add` performs the **element-wise addition** of two arrays, **`lhs`** and **`rhs`**, each with **`avl`** single precision (**`float`**) elements; finally the results are stored in the array **`dst`**.

```
void vector_add(float *dst,
                float *lhs,
                float *rhs,
                size_t avl)
{
    for (size_t vl; avl > 0; avl -= vl, lhs += vl, rhs += vl, dst += vl)
    {
        // compute the number of elements which are going to be
        // processed in this iteration of loop body.
        // this number corresponds to the vector length (vl)
        // and is evaluated from avl (application vector length)
        vl = __riscv_vsetvl_e32m1(avl);
        // loading operands
        vfloat32m1_t vec_src_lhs = __riscv_vle32_v_f32m1(lhs, vl);
        vfloat32m1_t vec_src_rhs = __riscv_vle32_v_f32m1(rhs, vl);
        // actual vector addition
        vfloat32m1_t vec_acc = __riscv_vfadd_vv_f32m1(vec_src_lhs,
                                                      vec_src_rhs,
                                                      vl);
        // storing results
        __riscv_vse32_v_f32m1(dst, vec_acc, vl);
    }
}
```

The method used here is straightforward:

- The **main loop iterates over the input vectors** to compute the vector addition of **`avl`** lements. **`avl`** is used as the **counter of remaining elements**.
- In each iteration:
    - We stop if we detect there are no more elements to compute (said otherwise we start a new iteration if and only if **`avl>0`**)
    - We start by computing the number of elements, **`vl`**, which will be processed during this iteration.

      ```
      vl = __riscv_vsetvl_e32m1(avl);
      ```

    - We load `vl` elements from both `lhs` and `rhs`

      ```
      vfloat32m1_t vec_src_lhs = __riscv_vle32_v_f32m1(lhs, vl);
      vfloat32m1_t vec_src_rhs = __riscv_vle32_v_f32m1(rhs, vl);
      ```

- We perform element-wise additions of **vl** elements

```
vfloat32m1_t vec_acc = __riscv_vfadd_vv_f32m1(vec_src_lhs,
                                              vec_src_rhs,
                                              vl);
```

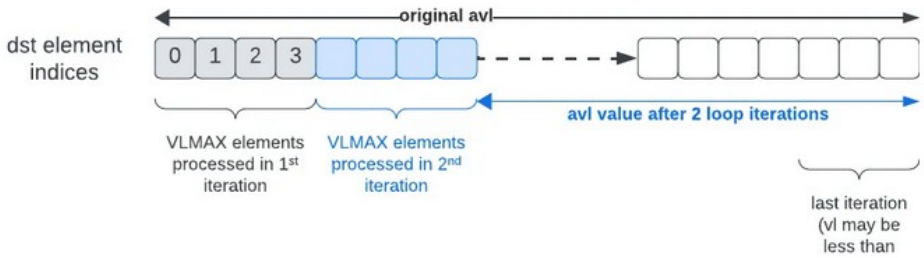
- We store the **vl** results into **dst**

```
__riscv_vse32_v_f32m1(dst, vec_acc, vl);
```

- We update avl by subtracting vl from it, and we update the source and destination pointers

```
avl -= vl, lhs += vl, rhs += vl, dst += vl
```

The likely behavior is depicted by the figure below:



VLMAX elements will be processed in each iterations (**VLMAX** elements from **lhs** and **rhs** will be added to form **VLMAX** elements in **dst**), except the last one which will processed either **VLMAX** elements if the original **avl** value was a multiple of **VLMAX** or **avl%VLMAX** (modulo operation).

3.3.1 Impact of VLEN on the result of vsetvl

Let us come back on the evaluation of the local vector length at the start of the loop iteration:

```
vl = __riscv_vsetvl_e32m1(avl);
```

The value returned by **__riscv_vsetvl_e32m1** depends on two things **avl** but also **VLMAX** (which is directly related to **VLEN**): if **avl** is greater than **VLMAX** then a truncated value is returned (the truncated value is less than or equal to **VLMAX**, else **avl** is returned. RVV 1.0 ensures that **vl=0** can not be returned if **avl >= VLMAX**, so forward progress is ensure but the actual amount of progress is implementation dependent.

```
VLMAX = VLEN * LMUL / SEW
```

In our case, **SEW=32** (**e32**) and **LMUL=1** (**m1**), so we get **VLMAX=VLEN/32**. The actual bound on the value returned by **vsetvl** depends on **VLEN**: the larger the **VLEN** the more elements are computed in each loop iterations. This is the definition of vector length architecture with a vector length agnostic program: the execution will adapt to the actual architectural value of **VLEN**.

3.4 Building the code and looking at the assembly

Now let us take the **`vector_add()`** function with the vector intrinsics:

```
void vector_add(float *dst,
                float *lhs,
                float *rhs,
                size_t avl)
{
    for (size_t vl; avl > 0; avl -= vl, lhs += vl, rhs += vl, dst += vl)
    {
        // compute the number of elements which are going to be
        // processed in this iteration of loop body.
        // this number corresponds to the vector length (vl)
        // and is evaluated from avl (application vector length)
        vl = __riscv_vsetvl_e32m1(avl);
        // loading operands
        vfloat32m1_t vec_src_lhs = __riscv_vle32_v_f32m1(lhs, vl);
        vfloat32m1_t vec_src_rhs = __riscv_vle32_v_f32m1(rhs, vl);
        // actual vector addition
        vfloat32m1_t vec_acc = __riscv_vfadd_vv_f32m1(vec_src_lhs,
                                                      vec_src_rhs,
                                                      vl);
        // storing results
        __riscv_vse32_v_f32m1(dst, vec_acc, vl);
    }
}
```

3.4.1 Compiling the function with vector intrinsics

The following command compiles the C/C++ code with intrinsics into assembly code:

```
bako@k1:~/Programs/lab0$ gcc vector_add_fun.c -march=rv64gcv -O2 -S
```

Lets us look only at **`for()`** loop section:

```
.L3:
        vsetvli a5,a3,e32,m1,ta,ma
        slli    a4,a5,2
        vle32.v v24,0(a1)                   // a1: first source and destination: dst and lhs
        vle32.v v25,0(a2)                   // a2: second source: rhs
        sub     a3,a3,a5                    // a3: avl
        vfadd.vv        v24,v24,v25
        vse32.v v24,0(a0)
        add     a1,a1,a4
        add     a2,a2,a4
        add     a0,a0,a4
        bne     a3,zero,.L3
```

3.4.2 Compiling and testing the complete program

```
// file: vector_add.c
#include <stdio.h>
#include <stdlib.h>
#include <riscv_vector.h>
#include <stddef.h>

void vector_add(float *dst, float *lhs, float *rhs, size_t avl)
{
    for (size_t vl; avl > 0; avl -= vl, lhs += vl, rhs += vl, dst += vl)
    {
        // compute loop body vector length from avl
        // (application vector length)
        vl = __riscv_vsetvl_e32m1(avl);
        // loading operands
        vfloat32m1_t vec_src_lhs = __riscv_vle32_v_f32m1(lhs, vl);
        vfloat32m1_t vec_src_rhs = __riscv_vle32_v_f32m1(rhs, vl);
        // actual vector addition
        vfloat32m1_t vec_acc = __riscv_vfadd_vv_f32m1(vec_src_lhs, vec_src_rhs, vl); // storing
results
        printf("storing results\n");
        __riscv_vse32_v_f32m1(dst, vec_acc, vl);
    }
}
```

```
// Defining a default size fot the inputs and output array
#define ARRAY_SIZE 8
float lhs[ARRAY_SIZE];
float rhs[ARRAY_SIZE];
float dst[ARRAY_SIZE] = {0.f};

int main(void) {
    int i;
    // random initialization of the input arrays
    for (i = 0; i < ARRAY_SIZE; ++i) {
        lhs[i] = rand() / (float) RAND_MAX;
        rhs[i] = rand() / (float) RAND_MAX;
        printf("lhs=%f,rhs=%f\n",lhs[i],rhs[i]);
    }
    vector_add(dst, lhs, rhs, ARRAY_SIZE);
    for (i = 0; i < ARRAY_SIZE; ++i)
        printf("dst=%f\n",dst[i]);
    return 0;
}
```

```
bako@k1:~/Programs/lab0$ gcc vector_add.c –march=rv64gcv –O2 –o vector_add
```

Running the program with **vector size=8**

```
bako@k1:~/Programs/lab0$ ./vector_add
lhs=0.840188,rhs=0.394383
lhs=0.783099,rhs=0.798440
lhs=0.911647,rhs=0.197551
lhs=0.335223,rhs=0.768230
lhs=0.277775,rhs=0.553970
lhs=0.477397,rhs=0.628871
lhs=0.364784,rhs=0.513401
lhs=0.952230,rhs=0.916195
storing results
dst=1.234571
dst=1.581539
dst=1.109199
dst=1.103452
dst=0.831745
dst=1.106268
dst=0.878185
dst=1.868425
```

As we see the loop in **`add_vector`** function was running only once. All 8 elements of the vector has been calculated in **one step** within **256-bit** (8 floating point data) **vector**.

```
// file: vector_add_int8.c
#include <stdio.h>
#include <stdlib.h>
#include <riscv_vector.h>
#include <stddef.h>

void vector_add(int8_t *dst, int8_t *lhs, int8_t *rhs, size_t avl)
{
    for (size_t vl; avl > 0; avl -= vl, lhs += vl, rhs += vl, dst += vl)
    {
        // compute loop body vector length from avl
        // (application vector length)
        vl = __riscv_vsetvl_e8m1(avl);
        printf("vl= %lu\n",vl);
        // loading operands
        vint8m1_t vec_src_lhs = __riscv_vle8_v_i8m1(lhs, vl);
        vint8m1_t vec_src_rhs = __riscv_vle8_v_i8m1(rhs, vl);
        // actual vector addition
        vint8m1_t vec_acc = __riscv_vadd_vv_i8m1(vec_src_lhs, vec_src_rhs, vl); // adding sources
        printf("storing results\n");
        __riscv_vse8_v_i8m1(dst, vec_acc, vl);
    }
}

// Defining a default size fot the inputs and output array
#define ARRAY_SIZE 32
int8_t lhs[ARRAY_SIZE];
```

```
int8_t rhs[ARRAY_SIZE];
int8_t dst[ARRAY_SIZE] = {0};

int main(void) {
    int i;
    // random initialization of the input arrays
    for (i = 0; i < ARRAY_SIZE; ++i) {
        lhs[i] = rand() / (int8_t) RAND_MAX;
        rhs[i] = rand() / (int8_t) RAND_MAX;
        printf("lhs=%d,rhs=%d\n",lhs[i],rhs[i]);
    }
    vector_add(dst, lhs, rhs, ARRAY_SIZE);
    for (i = 0; i < ARRAY_SIZE; ++i)
        printf("dst=%d\n",dst[i]);
    return 0;
}
```

The compilation:

```
bako@k1:~/Programs/lab0$ gcc vector_add_int8.c –march=rv64gcv –O2 –o vector_add_int8
```

Execution result:

```
bako@k1:~/Programs/lab0$ ./vector_add_int8
lhs=-103,rhs=58
lhs=-105,rhs=-115
lhs=-81,rhs=1
..
vl= 32
storing results

dst=-45
dst=36
dst=-80
..
```

The same program with **unsigned 8-bit integers**:

```
// file: vector_add_byte.c
#include <stdio.h>
#include <stdlib.h>
#include <riscv_vector.h>
#include <stddef.h>

void vector_add(uint8_t *dst, uint8_t *lhs, uint8_t *rhs, size_t avl)
{
    for (size_t vl; avl > 0; avl -= vl, lhs += vl, rhs += vl, dst += vl)
    {
        // compute loop body vector length from avl
        // (application vector length)
        vl = __riscv_vsetvl_e8m1(avl);
        printf("vl= %lu\n",vl);
        // loading operands
        vuint8m1_t vec_src_lhs = __riscv_vle8_v_u8m1(lhs, vl);
        vuint8m1_t vec_src_rhs = __riscv_vle8_v_u8m1(rhs, vl);
        // actual vector addition
        vuint8m1_t vec_acc = __riscv_vadd_vv_u8m1(vec_src_lhs, vec_src_rhs, vl); // adding sources
        printf("storing results\n");
        __riscv_vse8_v_u8m1(dst, vec_acc, vl);
    }
}

// Defining a default size fot the inputs and output array
#define ARRAY_SIZE 32
uint8_t lhs[ARRAY_SIZE];
uint8_t rhs[ARRAY_SIZE];
uint8_t dst[ARRAY_SIZE] = {0};

int main(void) {
    int i;
    // random initialization of the input arrays
    for (i = 0; i < ARRAY_SIZE; ++i) {
        lhs[i] = rand() / (uint8_t) RAND_MAX;
        rhs[i] = rand() / (uint8_t) RAND_MAX;
```

```
        printf("lhs=%u,rhs=%u\n",lhs[i],rhs[i]);
    }
    vector_add(dst, lhs, rhs, ARRAY_SIZE);
    for (i = 0; i < ARRAY_SIZE; ++i)
        printf("dst=%u\n",dst[i]);
    return 0;
}

bako@k1:~/Programs/lab0$ gcc -march=rv64gcv -O2 vector_add_byte.c -o vector_add_byte
bako@k1:~/Programs/lab0$ ./vector_add_byte


bako@k1:~/Programs/lab0$ ./vector_add_byte
lhs=60,rhs=209
lhs=57,rhs=226
lhs=2,rhs=191
lhs=167,rhs=16
..
vl= 32
storing results
dst=13
dst=27
dst=193
dst=183
..
```