Three potential gaps in RISC-V Vector

Jose Moreira, IBM

Three potential gaps in RISC-V Vector

1. Incomplete/partial specifications
2. OCP Microscaling Formats
3. Multiple vector multiply-accumulate

Example of partial specification

14.3.1. Vector Ordered Single-Width Floating-Point Sum Reduction

The `vfredosum` instruction must sum the floating-point values in element order, starting with the scalar in `vs1[0]` --that is, it performs the computation:

```
vd[0] = `(((vs1[0] + vs2[0]) + vs2[1]) + ...) + vs2[vl-1]`
```

where each addition operates identically to the scalar floating-point instructions in terms of raising exception flags and generating or propagating special values.

14.3.2. Vector Unordered Single-Width Floating-Point Sum Reduction

The unordered sum reduction instruction, `vfredusum`, provides an implementation more freedom in performing the reduction.

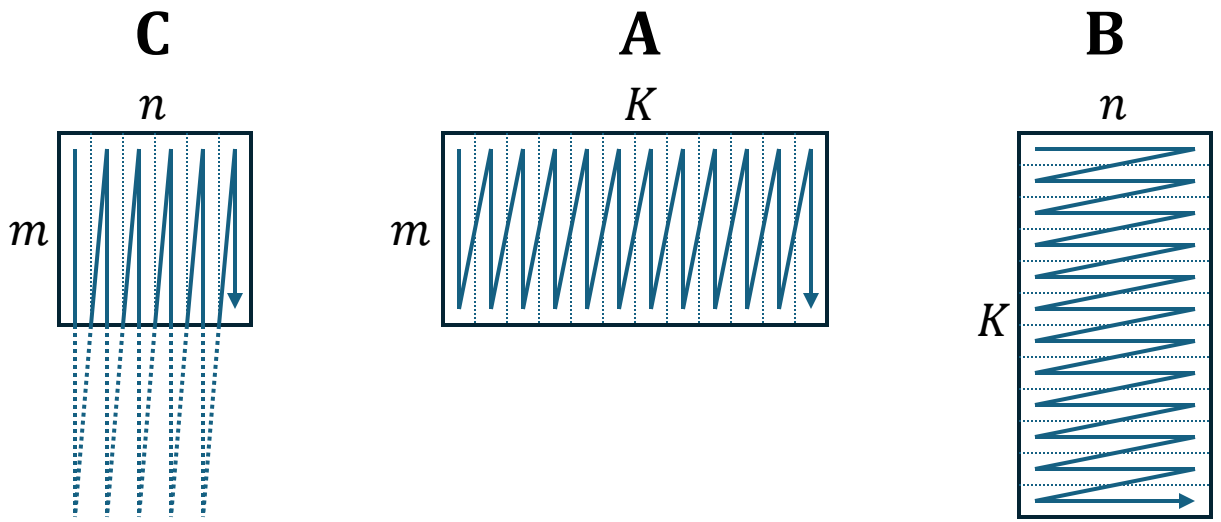
The implementation must produce a result equivalent to a reduction tree composed of binary operator nodes, with the inputs being elements from the source vector register group (`vs2`) and the source scalar value (`vs1[0]`). Each operator in the tree accepts two inputs and produces one result. Each operator first computes an exact sum as a RISC-V scalar floating-point addition with infinite exponent range and precision, then converts this exact sum to a floating-point format with range and precision each at least as great as the element floating-point format indicated by SEW, rounding using the currently active floating-point dynamic rounding mode and raising exception flags as necessary. A different floating-point range and precision may be chosen for the result of each operator. A node where one input is derived only from elements masked-off or beyond the active vector length may either treat that input as the additive identity of the appropriate EEW or simply copy the other input to its output. The rounded result from the root node in the tree is converted (rounded again, using the dynamic rounding mode) to the standard floating-point format indicated by SEW. An implementation is allowed to add an additional additive identity to the final result.

For floating-point arithmetic (and similarly for saturating integer arithmetic), there is no practical error bound without specifying an order of operations

- Example with IEEE fp32 arithmetic:
  - $\mathbf{x} = [+3 \times 10^{38}, +3 \times 10^{38}, -3 \times 10^{38}, -3 \times 10^{38}, -3 \times 10^{38}, -3 \times 10^{38}]$
  - $\Big(\Big(\big((x_0 + x_1) + x_2\big) + x_3\Big) + x_4\Big) + x_5 = +\infty$
  - $\big((x_0 + x_2) + x_4\big) + \big((x_1 + x_3) + x_5\big) = -\infty$
- Some alternatives to achieve maximum flexibility AND full specification:
  - RISC-V publishes a “general” architecture, implementor finishes the details, including specification of how reductions are performed
  - Implementor provides a reference scalar code that will produce the same bit-for-bit results as the vector/matrix instruction (another way to do a spec) – in particular, support checking of results from machine A in machine B
  - We find a more formal and satisfactory way to specify the possible outcomes of an architected instruction, ideally bounding the differences

What do we mean by “portable binary code”?

- Let $\mathbf{A}$ be an $m \times K$ matrix of single-precision floating-point numbers (fp32), stored in column-major order and with a leading dimension of m – that is, no gap between successive columns of $\mathbf{A}$
- Let $\mathbf{B}$ be a $K \times n$ matrix of single-precision floating-point numbers (fp32), stored in row-major order and with a leading dimension of n – that is, no gap between successive rows of $\mathbf{B}$
- Let $\mathbf{C}$ be an $m \times n$ matrix of single-precision floating-point numbers (fp32), stored in column-major order and with a leading dimension of ldc – that is, $\&[\mathbf{C}(i, j+1)] = \&[\mathbf{C}(i,j)] + \mathrm{ldc}$



Compute: $\mathbf{C} \leftarrow \alpha\mathbf{AB} + \beta\mathbf{C}$

- We must be able to produce a single binary code that will run on every implementation of an architecture extension (IME and AME are different extensions) and result in a bit-by-bit identical $\mathbf{C}$
- We must be able to characterize the performance of this binary code, for various values of m, n, and K in any of the above implementations – we should always get “good” performance

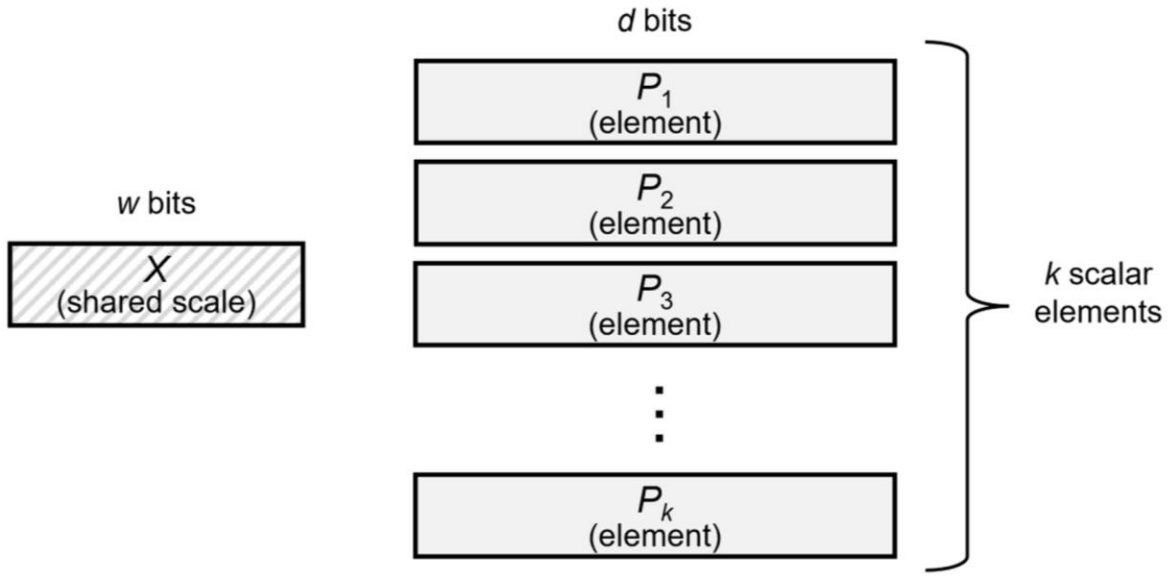
OCP Microscaling Formats

From: Open Compute Project
• OCP Microscaling Formats (MX) Specification

5.1 Microscaling (MX)

An MX-compliant format is characterized by three components:

- Scale (X) data type / encoding
- Private elements (P_i) data type / encoding
- Scaling block size (k)



All k elements (P_i) have the same data type and, therefore, the same bit-width. The scale factor X is shared across all k elements. The data types of the elements and scale are chosen independently. In this sense, MX can be seen as a mechanism to build a vector data type from scalar data types.

Concrete MX-compliant Formats

From: Open Compute Project
• OCP Microscaling Formats (MX) Specification

Format Name	Element Data Type	Element Bits (d)	Scaling Block Size (k)	Scale Data Type	Scale Bits (w)
MXFP8	FP8 (E5M2)	8	32	E8M0	8
	FP8 (E4M3)				
MXFP6	FP6 (E3M2)	6	32	E8M0	8
	FP6 (E2M3)				
MXFP4	FP4 (E2M1)	4	32	E8M0	8
MXINT8	INT8	8	32	E8M0	8

Table 1. Format names and parameters of concrete MX-compliant formats.

Alternatives for supporting MX in RISC-V Vector

1. Directly support arithmetic instructions with data in MX formats
   - Input values require both the scaling factor and the scalar elements (1 scalar register + 1 vector register per input)
   - Output “must” be conventional FP – no guarantee that will fit in MX
   - Need to convert back from conventional to MX
2. Only support conversion instructions (MX→FP and FP →MX)
   - MX→FP: 1 scalar + 1 vector → 1 vector (widening)
   - FP →MX: 1 vector → 1 scalar + 1 vector (narrowing) – not unique!
3. Keep scale factors and elements separate, operate only on the elements, in some intermediate format, apply resulting scaling factor at end, will still need a final conversion at the end back to a MX format

Multiple vector multiply-accumulate

- Existing instruction:

```
vfmacc.vf vd, rs1, vs2, vm  # for (i = 0; i < LMUL × vlene; i++)
                            #   vd[i] = +(f[rs1] * vs2[i]) + vd[i]
```

  - LMUL applies to both vd and vs2, rs1 is always the same scalar
  - LMUL=4 → 8 vector register reads + 4 vector register writes

- Proposed instruction:

```
vmfmacc.vv vd, vs1, vs2, vm # for (j = 0; j < LMUL; j++)
                            #   for (i = 0; i < vlene; i++)
                            #     vd[i] = +(vs1[j] * vs2[j][i]) + vd[i]
```

  - LMUL applies to vs2 only, vd and vs1 are always just one vector register
  - LMUL=4 → 6 vector register reads + 1 vector register write (same # of flops)

Example (VLEN=128, LMUL=4, FP32)

$$[vd[0] \quad vd[1] \quad vd[2] \quad vd[3]] = [vd[0] \quad vd[1] \quad vd[2] \quad vd[3]]$$
$$+ \quad vs1[0] \quad \times \quad [vs2[0][0] \quad vs2[0][1] \quad vs2[0][2] \quad vs2[0][3]]$$
$$+ \quad vs1[1] \quad \times \quad [vs2[1][0] \quad vs2[1][1] \quad vs2[1][2] \quad vs2[1][3]]$$
$$+ \quad vs1[2] \quad \times \quad [vs2[2][0] \quad vs2[2][1] \quad vs2[2][2] \quad vs2[2][3]]$$
$$+ \quad vs1[3] \quad \times \quad [vs2[3][0] \quad vs2[3][1] \quad vs2[3][2] \quad vs2[3][3]]$$