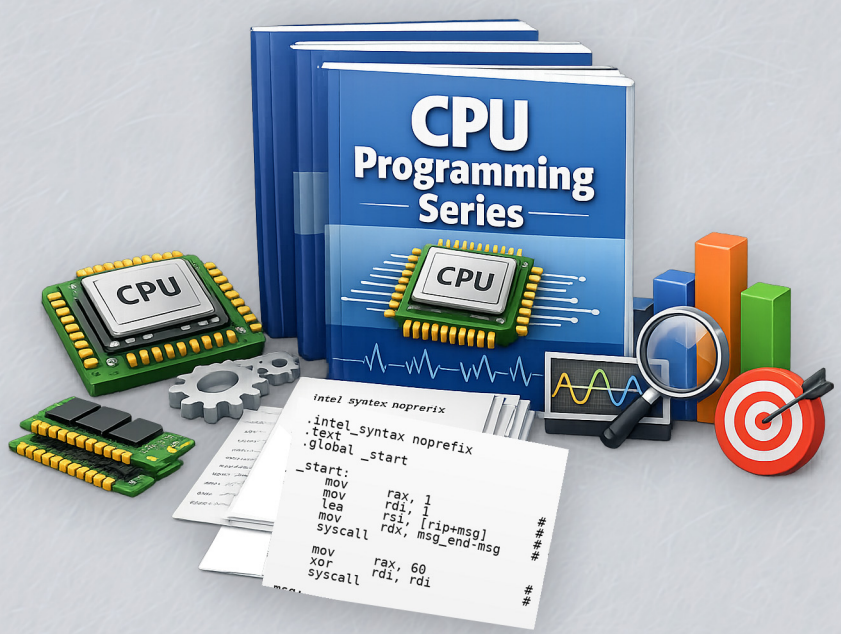
https://simplifycpp.org

CPU Programming Series

RISC-V Vector Extension (RVV)

Vector-Length-Agnostic Programming



22

Prepared by Ayman Alheraki

CPU Programming Series

RISC-V Vector Extension (RVV)

Vector-Length-Agnostic Programming

Prepared by Ayman Alheraki

simplifycpp.org

February 2026

Contents

Preface

Why RVV Exists

RISC-V Vector Extension (RVV) was created to solve a long-standing problem in SIMD programming: **software that hard-codes a vector width ages poorly**. Fixed-width SIMD (e.g., 128/256/512-bit) tends to force programmers to write code that assumes a particular number of lanes, and then maintain multiple versions for different widths. RVV instead standardizes a **vector-length-agnostic (VLA)** execution model:

- The hardware chooses an implementation-defined physical vector register size (`VLEN`).
- The program requests an **active vector length** (`vl`) dynamically at runtime.
- The same binary can scale across implementations with different `VLEN`, without rewriting the algorithm.

RVV also tackles practical engineering constraints that matter in real systems:

- **Portability across cores**: embedded, mobile, server, and accelerator-class implementations can differ widely.
- **Efficient predication and tail handling**: vectorizing loops with non-multiple lengths should be correct and efficient.

- **ISA extensibility**: RVV provides a rich base for compilers and libraries while keeping scalar ISA clean.

The Core RVV Loop Idea: Strip-Mining

At the heart of RVV is the **strip-mined loop**: process "as many elements as the hardware currently supports" each iteration, until the array is exhausted.

```
/* RISC-V GAS syntax (RVV). Example: y[i] = y[i] + x[i] for i in
 ↪  [0..n). */
.text
.align  2
.globl  vadd_f32
vadd_f32:
    /* a0 = x*, a1 = y*, a2 = n (elements) */
    beqz    a2, .Ldone

.Lloop:
    /* Set vl = min(n, MAXVL) for e32,m1 and write actual vl into t0
     ↪  */
    vsetvli t0, a2, e32, m1, ta, ma

    /* Load vl float32 elements from x and y */
    vle32.v v0, (a0)
    vle32.v v1, (a1)

    /* Vector add */
    vfadd.vv v1, v1, v0

    /* Store back to y */
```

```
    vse32.v v1, (a1)

    /* Advance pointers by vl * sizeof(float) = vl * 4 */
    slli    t1, t0, 2
    add     a0, a0, t1
    add     a1, a1, t1

    /* n -= vl */
    sub     a2, a2, t0
    bnez    a2, .Lloop

.Ldone:
    ret
```

This single routine scales to any RVV implementation: the hardware decides how many elements fit per iteration via `vl`.

Why Dynamic `vl` Is a Big Deal

With fixed-width SIMD you often compute "how many lanes" at compile time and peel the loop. With RVV you do not need a special remainder loop: the last iteration naturally runs with a smaller `vl`. This improves correctness, reduces code size, and avoids duplicated logic.

Fixed-Width SIMD vs Vector-Length-Agnostic Design

Fixed-Width SIMD (Traditional Model)

Fixed-width SIMD defines a constant register width (e.g., 128/256/512-bit). The number of lanes depends on element size:

- 256-bit registers hold 8 lanes of 32-bit values.
- 256-bit registers hold 4 lanes of 64-bit values.

This leads to patterns like:

- multiple binaries or runtime dispatch by ISA level,
- width-specialized kernels,
- manual tail handling (scalar remainder loop),
- code duplication and fragile assumptions.

RVV VLA Model (Scalable by Construction)

RVV standardizes a model where:

- **VLEN is not a software contract**; it is an implementation choice.
- software uses `vsetvli` / `vsetivli` to set (`SEW`, `LMUL`, policies) and obtain `vl`.
- all vector operations are implicitly bounded by `vl`.

SEW and LMUL: Expressing Shape Without Hard-Coding Width

RVV lets you select element width (`SEW`) and register grouping (`LMUL`). Conceptually:

- `SEW` chooses the element granularity (e8/e16/e32/e64).
- `LMUL` chooses how many vector registers are grouped to form a larger logical vector (m1, m2, m4, m8; and fractional variants in the ISA).

Practical effect: you tune throughput and register pressure without assuming a fixed number of lanes.

Masking and Tail Handling: Correctness Without Peeling

RVV provides first-class masking and defines tail/mask policies:

- `ta` / `tu`: tail agnostic vs tail undisturbed
- `ma` / `mu`: mask agnostic vs mask undisturbed

A typical use: compute a mask, then do masked operations without branches.

```
/* Masked clamp: for each i, if x[i] < 0 then x[i] = 0 (float32). */
.text
.align  2
.globl  clamp0_f32
clamp0_f32:
    /* a0 = x*, a1 = n */
    beqz    a1, .Ldone

    /* Prepare a vector of +0.0 in v1 */
.Lloop:
    vsetvli t0, a1, e32, m1, ta, ma

    vle32.v v0, (a0)            /* v0 = x */
    vfmv.v.f v1, fa0            /* assume fa0 already holds +0.0, or
     ↪  set it outside */

    /* v0 < 0 ? set mask bit = 1 */
    vmslt.vf v0, v0, fa0, v0.t  /* pseudopattern; actual compare
     ↪  forms a mask in v0 on some assemblers */

    /* Common explicit form: produce mask in v0, then use v0.t as
     ↪  predicate.
```

```
       Depending on assembler, you may prefer vmslt.vf v0, v0, fa0
       and then use v0.t in masked ops below. */

    /* Where mask true, write +0.0 into v0; elsewhere keep original
     ↪  */
    vmerge.vvm v0, v1, v0, v0.t

    vse32.v v0, (a0)

    slli    t1, t0, 2
    add     a0, a0, t1
    sub     a1, a1, t0
    bnez    a1, .Lloop

.Ldone:
    ret
```

A Practical Mental Model

- Fixed-width SIMD: “My machine has W bits; I must match W.”
- RVV: “My algorithm is vector-parallel; hardware tells me how many elements to do now.”

This difference is why RVV code, when written correctly, tends to be **more portable, more maintainable, and less branchy** for tail handling.

Scope, Assumptions, and Audience

Scope

This booklet focuses on the **RVV programming model and practical low-level patterns**:

- how `vl`, `SEW`, and `LMUL` shape code generation,
- how masking and tail policies affect correctness and performance,
- canonical strip-mined loops for arithmetic and memory operations,
- how to reason about performance without relying on a fixed vector width.

It intentionally does **not** mix topics:

- No general RISC-V privilege/traps/syscalls (covered elsewhere in the series).
- No deep compiler IR internals beyond what helps you read generated assembly.
- No advanced algorithm libraries; only patterns you can apply directly in kernels.

Assumptions

You should already be comfortable with:

- basic RISC-V integer ISA concepts (registers, calling convention basics),
- memory layout, alignment, and pointer arithmetic,
- performance fundamentals: bandwidth vs compute, cache locality, loop structure.

If you are new to SIMD-style thinking, you can still follow by treating each example as:

- "set `vl`, load `vl` elements, compute, store, advance pointers, repeat."

Audience

- Systems programmers writing performance-critical kernels.
- Compiler-aware developers who inspect assembly and tune hot loops.
- Engineers porting fixed-width SIMD code to RVV-style scalable vectorization.
- Readers of this CPU Programming Series who want architecture-level clarity.

Notation and Assembly Conventions

All assembly examples are written in GNU assembler style and use:

- `/* ... */` for comments,
- the RVV configuration idiom: `vsetvli rd, rs1, eSEW, mLMUL, ta/tu, ma/mu`.

How to Read This Booklet

Recommended Path

- Read Chapters 1–3 to internalize the RVV model (`vl`, `SEW`, `LMUL`, policies).
- Read Chapter 6 early if your workloads are memory-bound (loads/stores, stride, gather/scatter).
- Use Chapter 8 as your "daily driver": it collects the canonical VLA loop patterns.
- Finish with Chapter 10 for performance trade-offs and common pitfalls.

Two Rules That Prevent 90% of RVV Bugs

1. Never assume a fixed number of lanes. Treat `vl` as a runtime value.
2. Every pointer increment must be derived from the **actual** `vl` returned by `vsetvli`.

How to Validate Your Understanding

For each example:

- identify where `vl` is set and captured (usually into an integer register),
- verify loads/stores cover exactly `vl` elements,
- verify pointer increments are `vl * element_size`,
- verify loop count decrements by `vl`.

A Compact Checklist for Writing RVV Kernels

- Choose (`SEW`, `LMUL`) to match data type and desired throughput.
- Use strip-mining: `while (n > 0) { vsetvli(vl=min(n,MAXVL)); ...; n -= vl; }`
- Prefer contiguous memory access when possible; isolate strided/gather/scatter.
- Use masks to avoid branches for per-element conditions.
- Be explicit about tail/mask policies when correctness depends on inactive lanes.

One More Example: Reduction (Sum of float32 Array)

This pattern shows the RVV-style reduction skeleton. It is a frequent building block.

```
/* Sum reduction: returns sum(x[0..n)) in fa0 (float32).
   Note: actual ABI details may vary by toolchain; treat as a kernel
    ↪  pattern. */
.text
.align  2
.globl  sum_f32
sum_f32:
    /* a0 = x*, a1 = n */
    beqz    a1, .Ldone

    /* v0 = running partial sums (vector) */
    vsetvli t0, zero, e32, m1, ta, ma
    vfmv.v.f v0, fa0            /* assume fa0 is 0.0f on entry or
     ↪  set it before call */

.Lloop:
    vsetvli t0, a1, e32, m1, ta, ma
    vle32.v v1, (a0)
    vfadd.vv v0, v0, v1

    slli    t1, t0, 2
    add     a0, a0, t1
    sub     a1, a1, t0
    bnez    a1, .Lloop

    /* Reduce vector accumulator into a scalar */
```

```
    /* Common pattern: vredsum over v0 into v2 with an initial scalar
     ↪  in v3 */
    vfmv.v.f v3, fa0              /* v3 = 0.0 seed */
    vredsum.vs v2, v0, v3         /* v2[0] = sum(v0) + seed */

    /* Move scalar result to fa0 */
    vfmv.f.s fa0, v2
.Ldone:
    ret
```

Chapter 1

RVV Philosophy and Design Goals

1.1 From SIMD to VLA Computing

Traditional SIMD programming assumes a **fixed hardware vector width**. The software then bakes in a lane count (explicitly or implicitly), and must handle the remainder when the problem size is not a multiple of that lane count. This creates persistent friction:

- multiple code paths (SSE vs AVX2 vs AVX-512, NEON vs wider variants),
- dispatch logic, build-time feature matrices, and duplicated kernels,
- scalar cleanup loops and "tail" complexity that grows with the number of kernels.

RVV adopts **vector-length-agnostic (VLA) computing**: the program does not assume any fixed lane count. Instead, each iteration asks hardware for an appropriate **active vector length** (`vl`) and processes exactly that many elements. This is often called **strip-mining**.

Canonical VLA Loop Skeleton (Strip-Mining)

```
/* RISC-V GAS (RVV). Canonical strip-mined loop:
   process vl elements each iteration until n is exhausted. */

.text
.align  2
.globl  vla_saxpy_f32
vla_saxpy_f32:
    /* a0 = x*, a1 = y*, a2 = n (elements), fa0 = a (scalar float32)
     ↪  */
    beqz    a2, .Ldone

.Lloop:
    /* Configure for e32,m1 and obtain actual vl in t0: vl = min(n,
     ↪  MAXVL). */
    vsetvli t0, a2, e32, m1, ta, ma

    /* Load vl floats from x and y */
    vle32.v v0, (a0)
    vle32.v v1, (a1)

    /* y = a*x + y */
    vfmacc.vf v1, fa0, v0

    /* Store back */
    vse32.v v1, (a1)

    /* Advance pointers by vl * 4 bytes */
    slli    t1, t0, 2
```

```
    add     a0, a0, t1
    add     a1, a1, t1

    /* n -= vl */
    sub     a2, a2, t0
    bnez    a2, .Lloop

.Ldone:
    ret
```

This single loop form is the mental model for RVV: **the same binary scales across implementations** because it adapts to whatever vector capacity exists at runtime.

Why This Changes Everything

With VLA, the "tail" is not a separate algorithmic path. The last iteration simply runs with a smaller `vl`. Correctness becomes simpler, and performance tuning shifts from lane-count engineering to:

- selecting element width (`SEW`) and grouping (`LMUL`),
- managing memory behavior (unit-stride vs strided vs indexed),
- controlling masks and policies when inactive elements matter.

1.2 Hardware Independence and Forward Scalability

RVV was designed so software remains **portable across cores** while allowing implementers freedom to scale the vector unit. Two key principles enable this:

(1) `VLEN` Is Not a Software Contract

The hardware chooses the physical vector register size (`VLEN`) and other limits. Software never assumes it. Instead, software queries `vl` each iteration using `vsetvli` (or `vsetivli`).

(2) The ISA Makes Partial-Vector Execution a First-Class Case

Most real loops have lengths not divisible by any fixed lane count. RVV treats this as normal, not exceptional. That is why `vl` exists as a runtime quantity and why masking is integrated into the ISA.

Forward Scalability in Practice

If a future core has a larger vector unit, the same loop processes more elements per iteration (higher throughput) without changing source or binary.

Example: Same Kernel, Different `vl`

Assume the same code runs on two machines:

- Machine A yields `vl`=8 for `e32,m1` (8 float32 per iteration),
- Machine B yields `vl`=32 for `e32,m1` (32 float32 per iteration).

The loop body is identical; only iteration count changes. This is hardware independence with performance scaling.

Policy Control: Tail and Mask Behavior

Vector configuration also includes tail/mask policies:

- `ta` vs `tu`: tail agnostic vs tail undisturbed
- `ma` vs `mu`: mask agnostic vs mask undisturbed

Use **undisturbed** policies when inactive elements must preserve previous register contents for correctness across dependent operations; use **agnostic** policies when you want maximal freedom for the implementation.

```
/* Demonstrate policy selection.
   tu/mu is a correctness-oriented choice when inactive elements
    ↪  matter. */

vsetvli t0, a2, e32, m1, tu, mu
```

1.3 Why RVV Is Fundamentally Different from SSE / AVX / NEON

RVV is often described as "SIMD", but its programming model is closer to a **scalable vector architecture** than classic fixed-width SIMD. The differences matter at the algorithm level.

Fixed-Width SIMD: Width Is the API

With SSE/AVX/NEON, the register width is fixed, so software naturally evolves around a constant lane count:

- vector types encode width (`__m128`, `__m256`, `float32x4_t`),
- unrolling and remainder strategies depend on that width,
- portability is often solved via multiple kernels or multi-versioning.

RVV: `vl` Is the API

RVV exposes a dynamic `vl` and separates:

- the **shape** you request (`SEW`, `LMUL`),
- from the **capacity** the implementation provides (`vl` result).

This leads to a stable programming discipline:

- loops are written in strip-mined form,
- correctness does not depend on any fixed lane count,
- performance scales with the core's vector resources.

RVV Masking Is Not an Afterthought

Fixed-width SIMD typically handles per-element conditions by blends, masked operations (if available), or branches. RVV integrates masks as a first-class mechanism across arithmetic and memory ops.

Example: Branchless Conditional Update Using a Mask

```
/* If x[i] < 0 then x[i] = 0 for float32 array x[]. */

.text
.align  2
.globl  clamp0_f32_vla
clamp0_f32_vla:
    /* a0 = x*, a1 = n, fa0 = 0.0f */
    beqz    a1, .Ldone
```

```
.Lloop:
    vsetvli t0, a1, e32, m1, ta, ma

    vle32.v v1, (a0)            /* v1 = x */
    vfmv.v.f v2, fa0            /* v2 = 0.0 in all active elements */

    /* Produce mask: v0.t is true where x < 0 */
    vmslt.vf v0, v1, fa0        /* v0 is a mask register (v0) */

    /* Merge: where mask true take 0.0, else keep x */
    vmerge.vvm v1, v2, v1, v0.t

    vse32.v v1, (a0)

    slli    t1, t0, 2
    add     a0, a0, t1
    sub     a1, a1, t0
    bnez    a1, .Lloop

.Ldone:
    ret
```

LMUL and Register Grouping Change the Tuning Game

RVV can group registers (LMUL) to trade register pressure for throughput and reduce overheads for certain data types. Instead of choosing a different fixed-width ISA, you often tune by selecting `LMUL` and `SEW`.

Example: Widening Multiply-Accumulate (Int16 -¿ Int32)

```
/* Dot product style accumulation:
   acc[i] += (int32)a[i] * (int32)b[i], where a,b are int16, acc is
    ↪  int32. */

.text
.align  2
.globl  dot_widen_i16_i32
dot_widen_i16_i32:
    /* a0 = a*, a1 = b*, a2 = acc*, a3 = n (elements) */
    beqz    a3, .Ldone

.Lloop:
    /* Use e16 for inputs; choose m1 here (tune LMUL per
     ↪  microarchitecture). */
    vsetvli t0, a3, e16, m1, ta, ma

    vle16.v v1, (a0)            /* a */
    vle16.v v2, (a1)            /* b */

    /* Widening multiply: produces int32 results in v3 */
    vwmul.vv v3, v1, v2

    /* Reconfigure for e32 to update accumulator with the same vl
     ↪  count semantics. */
    vsetvli t0, t0, e32, m1, ta, ma
    vle32.v v4, (a2)            /* acc */
    vadd.vv  v4, v4, v3
    vse32.v  v4, (a2)
```

```
    /* Advance pointers:
       a,b by vl*2 bytes (int16), acc by vl*4 bytes (int32). */
    slli    t1, t0, 1
    add     a0, a0, t1
    add     a1, a1, t1
    slli    t2, t0, 2
    add     a2, a2, t2

    /* n -= vl */
    sub     a3, a3, t0
    bnez    a3, .Lloop

.Ldone:
    ret
```

This demonstrates an RVV-specific mindset: you may **reconfigure SEW** as the computation widens, while still staying lane-count-agnostic via `vl`.

1.4 Software Longevity as a Design Constraint

RVV treats software longevity as a primary design goal: the ISA aims to avoid forcing a rewrite when hardware changes. That constraint shapes both the **execution model** and the **toolchain contract**.

Longevity Problem in Practice

If a kernel is written against a fixed width, long-lived codebases accumulate:

- specialized kernels per width and per feature level,

- per-platform performance drift as new widths appear,
- growing test matrices and increased risk of rare tail bugs.

RVV Longevity Strategy

1. **Make width variable by definition** (VLA): correctness never depends on width.
2. **Make partial vectors normal**: tails are not a separate algorithmic mode.
3. **Expose configuration, not width**: software asks for `SEW`/`LMUL` and receives `vl`.

A Longevity-Friendly Kernel Checklist

- Never encode lane counts in constants, unroll factors, or indexing.
- Always derive pointer increments from the returned `vl`.
- Prefer unit-stride memory to let implementations scale bandwidth naturally.
- Use masks for per-element conditions; avoid divergence branches in vector loops.
- Choose tail/mask policies intentionally when inactive lanes may affect correctness.

Example: Memory-Bound Copy That Scales Forward

```
/* memcpy-like copy (byte). This is intentionally simple and scales
 ↪  with vl. */

.text
.align  2
.globl  vla_copy_u8
vla_copy_u8:
```

```
    /* a0 = dst*, a1 = src*, a2 = n (bytes) */
    beqz    a2, .Ldone

.Lloop:
    vsetvli t0, a2, e8, m8, ta, ma   /* e8; LMUL chosen for
     ↪  throughput */
    vle8.v  v0, (a1)
    vse8.v  v0, (a0)

    add     a0, a0, t0
    add     a1, a1, t0
    sub     a2, a2, t0
    bnez    a2, .Lloop

.Ldone:
    ret
```

A fixed-width design tends to bake in copy granularity and remainder logic. RVV naturally adapts: the hardware picks the best `vl` each iteration, and the same implementation benefits from future wider vector units without changing the algorithm.

Chapter 2

RVV Architectural Overview

2.1 Vector Registers and Register Groups

RVV defines a dedicated vector register file of **32 architectural vector registers**:

$$\texttt{v0}, \texttt{v1}, \ldots, \texttt{v31}.$$

Each `v` register holds a vector of elements whose **element width** is chosen dynamically (via `SEW`) and whose **active length** is controlled by `vl`. A key RVV design choice is that a single architectural register name (e.g., `v8`) can represent:

- a **single register** when `LMUL=m1`,
- a **register group** when `LMUL>m1` (e.g., `v8` representing `v8--v11` for `m4`),
- a **fractional group** when `LMUL<1` (e.g., `mf2`, `mf4`, `mf8`) for reduced register footprint.

Register Groups: The Practical Meaning

A register group is a logical vector register that spans multiple physical vector registers. For example:

- `LMUL=m2`: one logical destination may occupy `vN` and `vN+1`,
- `LMUL=m4`: one logical destination may occupy `vN` through `vN+3`,
- `LMUL=m8`: one logical destination may occupy `vN` through `vN+7`.

This is a **tuning lever**: larger `LMUL` can increase vector throughput per instruction for some kernels, but it also increases register pressure and reduces how many independent vector values you can keep live.

Mask Register `v0`

Masking in RVV is central. Conventionally, `v0` is used as the mask register for predicated execution (`v0.t`). Many instructions accept an optional mask operand, typically written as `, v0.t`.

```
/* Example: produce a mask in v0, then use it to predicate an
 ↪  operation. */
vsetvli t0, a0, e32, m1, ta, ma
vmslt.vf v0, v1, fa0      /* v0 mask: v1 < scalar */
vadd.vv  v2, v2, v3, v0.t /* only lanes with mask bit=1 are updated
 ↪  */
```

2.2 Vector Length (VLEN) and Element Width (SEW)

VLEN: Implementation Capacity

VLEN is the **implementation-defined** physical size of a vector register (in bits). Software does not assume a specific VLEN. Instead, software requests a vector configuration and obtains the **active vector length** vl at runtime.

SEW: The Element Granularity

SEW (Selected Element Width) defines the element size used by the current vector configuration. Common SEW choices include:

$$\texttt{e8}, \texttt{e16}, \texttt{e32}, \texttt{e64}.$$

Changing SEW changes how many elements fit in the active vector length, and it also affects instruction selection (e.g., vle32.v vs vle16.v).

The Contract: `vsetvli` Produces a Legal `vl`

The instruction vsetvli (or vsetivli) configures vtype (including SEW and LMUL) and returns a legal vl:

- vl is chosen such that **operations are well-defined** for the selected SEW/LMUL,
- vl never exceeds the remaining element count you request (typical strip-mining usage),
- the same code adapts to different VLEN implementations.

```
/* Set vector configuration and get vl in t0. */
vsetvli t0, a2, e32, m1, ta, ma   /* vl = min(a2, MAXVL_for_e32_m1)
↪  */
```

A Correctness Rule That Must Never Be Broken

> **All pointer increments and loop trip updates must be derived from the returned `vl`, not from assumptions.**

```
/* Correct strip-mining pointer arithmetic for float32 arrays. */
vsetvli t0, a2, e32, m1, ta, ma
slli    t1, t0, 2     /* bytes = vl * 4 */
add     a0, a0, t1
sub     a2, a2, t0
```

2.3 Vector Register File Layout

Architectural View

From software's perspective, the vector register file is:

- 32 registers (`v0--v31`),
- each register holds `VLEN` bits of storage (implementation capacity),
- operations interpret those bits as a vector of `vl` active elements of width `SEW`.

Logical View Under `SEW`

For a chosen `SEW`, the vector register is logically partitioned into lanes of that width. For example:

- `SEW=e8`: lanes are bytes,
- `SEW=e32`: lanes are 32-bit words,

- `SEW=e64`: lanes are 64-bit words.

The active lanes are the first `vl` elements. Elements beyond `vl` are **inactive**.

Active vs Inactive Elements: Tail and Mask Policies

Inactive elements may exist for two reasons:

- **Tail**: elements beyond `vl` within the logical register capacity,
- **Mask-off lanes**: elements where the predicate mask bit is 0.

The configuration specifies policies:

- `ta` / `tu`: tail agnostic vs tail undisturbed,
- `ma` / `mu`: mask agnostic vs mask undisturbed.

These policies matter when you build multi-step sequences where you reuse destination registers and rely on inactive lanes preserving earlier values.

```
/* Correctness-oriented: preserve inactive elements in dependent
 ↪  sequences. */
vsetvli t0, a0, e16, m2, tu, mu
```

A Practical Mental Model

Think of the vector register file as **capacity**, and `vl` as **the runtime slice you are allowed to touch**. Correct RVV code only reasons about the slice `[0, vl)`.

2.4 LMUL and Register Grouping Rules

LMUL: Scaling a Logical Vector Register

`LMUL` scales how many architectural vector registers are used to represent a logical vector register. Common integer `LMUL` values:

`m1, m2, m4, m8.`

Fractional values exist to reduce register footprint:

`mf2, mf4, mf8.`

Grouping Rules (What the Hardware Requires)

When `LMUL` forms a group (`m2/m4/m8`), the starting register number must satisfy alignment constraints so the group fits cleanly:

- `m2`: start register must be even (`v0, v2, v4, ...`),
- `m4`: start register must be a multiple of 4 (`v0, v4, v8, ...`),
- `m8`: start register must be a multiple of 8 (`v0, v8, v16, v24`).

If you violate these constraints, the instruction is **not a valid encoding/use** for that `vtype` and must not be generated.

Examples of Legal and Illegal Group Starts

```
/* LMUL=m4 requires group-aligned registers: v0, v4, v8, v12, v16,
 ↪  v20, v24, v28. */
vsetvli t0, a0, e32, m4, ta, ma
```

```
/* Legal: v8 represents the group v8-v11 */
vadd.vv  v8, v12, v16

/* Illegal under m4: v10 would imply a group v10-v13 (misaligned
 ↪  start) */
vadd.vv  v10, v12, v16
```

Register Overlap Hazards: A Real Source of Bugs

With grouping, a single logical register may consume multiple physical registers. You must treat those physical registers as **overlapping storage**. For example, under `m4`:

- writing `v8` writes the whole group `v8--v11`,
- therefore `v9`, `v10`, `v11` are not independent values.

```
/* Overlap hazard example under LMUL=m4. */
vsetvli t0, a0, e32, m4, ta, ma

/* After this, the group v8-v11 is defined. */
vle32.v  v8, (a1)

/* Treating v10 as independent is wrong: it overlaps with v8's group.
 ↪  */
vse32.v  v10, (a2)   /* This is logically inconsistent usage under
 ↪  m4. */
```

Choosing LMUL: A Practical Rule

- Start with `m1` for clarity and maximum register flexibility.

- Increase `LMUL` when you are throughput-limited and can afford fewer live vector values.
- Use fractional `LMUL` when register pressure is high or when vectorizing small kernels that do not benefit from large groups.

Example: Same Loop, Different LMUL Choices

Below are two correct strip-mined loops, identical in structure, differing only by `LMUL`. This is how RVV tuning should look: **change configuration, keep the algorithm stable**.

```
/* Version A: LMUL=m1 (baseline) */
vsetvli t0, a2, e32, m1, ta, ma
vle32.v v1, (a0)
vle32.v v2, (a1)
vfadd.vv v2, v2, v1
vse32.v v2, (a1)

/* Version B: LMUL=m4 (more grouping; fewer independent registers
 ↪  available) */
vsetvli t0, a2, e32, m4, ta, ma
vle32.v v4, (a0)       /* v4 means v4-v7 as a group under m4 */
vle32.v v8, (a1)       /* v8 means v8-v11 as a group under m4 */
vfadd.vv v8, v8, v4
vse32.v v8, (a1)
```

What to Remember

- RVV gives you 32 architectural vector registers, but `LMUL` changes how many *logical* registers you effectively have.
- `SEW` changes the element interpretation; `vl` defines the active slice.

- Correct RVV code is written against `vl`, not against a fixed lane count.
- Group alignment and overlap rules are non-negotiable; violating them produces invalid or logically inconsistent code.

Chapter 3

Vector Configuration and Execution Model

3.1 `vsetvli` and `vsetivli`

RVV uses explicit configuration-setting instructions to define how subsequent vector instructions interpret vector registers and how many elements are *active*.

`vsetvli rd, rs1, vtypei`

`vsetvli` configures the vector unit and sets the **active vector length** `vl` based on a runtime **AVL** (Application Vector Length) value provided in `rs1`. The instruction writes the chosen `vl` to `rd` (often the same register as `rs1` or a temporary).

```
/* Canonical usage: strip-mine n elements with e32,m1. */
.Lloop:
    vsetvli t0, a2, e32, m1, ta, ma   /* t0 = vl chosen for remaining
     ↪  a2 */
```

```
    /* vector work on vl elements */
    sub     a2, a2, t0                /* n -= vl */
    bnez    a2, .Lloop
```

`vsetivli rd, uimm, vtypei`

`vsetivli` is identical in effect except the AVL is a small immediate (useful for short fixed trip-count kernels, micro-kernels, prolog/epilog handling, and constant-sized transforms).

```
/* Process up to 16 elements, regardless of implementation width. */
vsetivli t0, 16, e32, m1, ta, ma   /* t0 = min(16, VLMAX(e32,m1)) */
```

Practical rules for both instructions

- Treat the returned `vl` as the only truth.
- Derive pointer increments from `vl` and the element size.
- Keep the loop structure stable; tune performance by choosing `SEW`/`LMUL` and policies.

```
/* Correct pointer math: float32 arrays (4 bytes per element). */
vsetvli t0, a2, e32, m1, ta, ma
slli    t1, t0, 2         /* bytes = vl * 4 */
add     a0, a0, t1        /* x += vl */
add     a1, a1, t1        /* y += vl */
sub     a2, a2, t0        /* n -= vl */
```

3.2 Vector Type (`vtype`) Encoding

Vector configuration is represented by the `vtype` CSR. Conceptually, `vtype` captures:

- **`vsew`**: selected element width (e8/e16/e32/e64),

- **`vlmul`**: register grouping multiplier (mf8/mf4/mf2/m1/m2/m4/m8),
- **`vta`** and **`vma`**: tail and mask policies,
- **`vill`**: illegal-configuration indicator.

Assembler view: readable `vtype` immediates

Most programmers should *not* encode `vtype` bits manually. The standard practice is to use assembler mnemonics:

`e32, m1, ta, ma` or `e16, m2, tu, mu`.

```
/* Two different vtype configurations (same program, different tuning
 ↪  points). */

/* e32 elements, LMUL=1, tail agnostic, mask agnostic */
vsetvli  t0, a2, e32, m1, ta, ma

/* e16 elements, LMUL=2, tail undisturbed, mask undisturbed
 ↪  (correctness-oriented) */
vsetvli  t0, a2, e16, m2, tu, mu
```

The `vill` rule (illegal settings)

If a requested `vtype` is not supported or is otherwise illegal, hardware indicates this via `vill` and the resulting `vl` becomes unusable for real work (treat it as a hard configuration failure). Robust low-level code should avoid generating illegal combinations by construction:

- only request `SEW` values supported by the enabled vector subsets,
- obey `LMUL` grouping alignment constraints,
- keep `SEW` and `LMUL` within the architectural ranges.

Reading `vtype` (debug / verification)

In bring-up, simulators, or debug builds, it is sometimes useful to read back `vtype` to confirm what the hardware accepted.

```
/* Read back vtype (CSR number is toolchain-defined mnemonic
 ↪  "vtype"). */
csrr    t2, vtype
/* t2 now contains fields such as vill/vma/vta/vsew/vlmul (encoded).
 ↪  */
```

3.3 Tail and Mask Policies

RVV defines two distinct categories of inactive elements:

- **Tail elements**: lanes beyond `vl` up to the maximum capacity for the current `vtype`.
- **Mask-disabled elements**: lanes within `vl` whose predicate bit is 0 for a masked instruction.

Two policy bits control what happens to those inactive elements in the *destination* register:

Tail policy: `ta` vs `tu`

- `ta` (tail agnostic): tail elements may become arbitrary values.
- `tu` (tail undisturbed): tail elements preserve their previous contents.

Mask policy: `ma` vs `mu`

- `ma` (mask agnostic): mask-disabled destination elements may become arbitrary.

- `mu` (mask undisturbed): mask-disabled destination elements preserve their previous contents.

When `ta,ma` is the right default

For most high-performance kernels where inactive lanes are never observed, `ta,ma` gives implementations freedom to optimize.

```
/* High-throughput default: inactive elements are don't-care. */
vsetvli t0, a2, e32, m1, ta, ma
```

When `tu,mu` is required for correctness

If you intentionally reuse a destination register across multiple masked steps and the subsequent steps depend on preserving inactive elements, you must choose undisturbed policies.

```
/* Correctness pattern: build a result in multiple masked phases. */
vsetvli t0, a2, e32, m1, tu, mu

/* Phase 1: write only where mask1 true */
vmslt.vf v0, v1, fa0
vmerge.vvm v8, v2, v8, v0.t   /* keep old v8 where mask is false (mu
↪  helps) */

/* Phase 2: write only where mask2 true, expecting other lanes
↪  unchanged */
vmsgt.vf v0, v1, fa1
vmerge.vvm v8, v3, v8, v0.t
```

A simple mental model

- Use `ta,ma` when you will never read inactive lanes.
- Use `tu,mu` when you are composing results across masked/tail-partial operations and need inactive lanes preserved.

3.4 VL as a Dynamic Runtime Value

The defining property of RVV is that `vl` is a **dynamic runtime value**. It can change:

- each loop iteration (strip-mining),
- each time you change `SEW` or `LMUL`,
- across different implementations of the same ISA.

The strip-mined loop is the execution model

A correct RVV loop follows three invariants:

1. `vsetvli` computes `vl` for the remaining element count.
2. Every vector instruction in the body operates on exactly `vl` active elements.
3. Pointer increments and trip count updates are derived from the returned `vl`.

Example: vector add with a non-multiple tail (no scalar remainder)

```
/* y[i] += x[i], float32, no scalar tail loop needed. */
.text
.align  2
```

```
.globl  vla_add_f32
vla_add_f32:
    /* a0=x*, a1=y*, a2=n */
    beqz    a2, .Ldone
.Lloop:
    vsetvli t0, a2, e32, m1, ta, ma

    vle32.v v0, (a0)
    vle32.v v1, (a1)
    vfadd.vv v1, v1, v0
    vse32.v v1, (a1)

    slli    t1, t0, 2
    add     a0, a0, t1
    add     a1, a1, t1
    sub     a2, a2, t0
    bnez    a2, .Lloop
.Ldone:
    ret
```

Example: changing `SEW` changes effective `VLMAX` and may change `vl`

Even if the remaining element count is the same, switching `SEW` or `LMUL` can change how many elements fit.

```
/* Same remaining count a2, but different configurations. */
vsetvli t0, a2, e16, m1, ta, ma   /* t0 = vl16 */
vsetvli t1, a2, e32, m1, ta, ma   /* t1 = vl32 (often smaller than
↪  vl16) */
```

Example: safe widening sequence (keep VLA discipline)

```
/* Widening flow: load int16, widen, then operate in e32 while still
 ↪  respecting vl. */
vsetvli t0, a2, e16, m1, ta, ma
vle16.v  v1, (a0)
vsext.vf2 v2, v1           /* sign-extend to 32-bit elements (widen)
 ↪  */

/* Reconfigure for 32-bit ops; re-derive vl for the new vtype and
 ↪  remaining count. */
vsetvli t0, t0, e32, m1, ta, ma
vadd.vx  v2, v2, a3
```

The one rule that prevents most RVV bugs

Never assume lane count. Assume only `vl` returned by `vsetvli`/`vsetivli`.

Chapter 4

Vector Data Types and Element Semantics

4.1 Integer Vector Types

RVV treats vector registers as **untyped storage** whose meaning is defined by the current vector configuration (`vtype`) and the instruction. Integer element semantics are primarily determined by:

- **SEW** (selected element width): `e8/e16/e32/e64`
- **signed vs unsigned** interpretation (instruction-specific)
- **VL** (active element count): only elements `[0, vl)` are active

Common integer instruction families

- arithmetic: `vadd/vsub`, `vmax/vmin` (signed), `vmaxu/vminu` (unsigned)
- shifts: `vsll`, `vsrl` (logical), `vsra` (arithmetic)

- comparisons producing masks: `vmslt/vmsltu`, `vmseq/vmsne`, `vmsle/vmsleu`, `vmsgt/vmsgtu`
- bitwise: `vand/vor/vxor`, `vnot`

Example: signed vs unsigned min/max on the same bits

The same bit patterns can represent different numeric values depending on signedness. RVV provides distinct instructions.

```
/* Compare semantics: vmax (signed) vs vmaxu (unsigned). */
vsetvli t0, a2, e8, m1, ta, ma

vle8.v   v1, (a0)          /* bytes */
vle8.v   v2, (a1)

/* Signed max: treats bytes as int8_t */
vmax.vv  v3, v1, v2

/* Unsigned max: treats bytes as uint8_t */
vmaxu.vv v4, v1, v2
```

Example: mask-producing compare + predicated update

```
/* If x[i] < 0 then x[i] = 0, for int16_t array. */
vsetvli  t0, a1, e16, m1, ta, ma

vle16.v  v1, (a0)          /* v1 = x */
vmv.v.i  v2, 0             /* v2 = 0 */
```

```
/* v0 mask true where x < 0 (signed compare against 0) */
vmslt.vx v0, v1, zero

/* Replace negatives with 0, keep others */
vmerge.vvm v1, v2, v1, v0.t
vse16.v  v1, (a0)
```

Element-size-correct pointer math

Integer loads/stores use EEW (effective element width) implied by the mnemonic:

- `vle8.v` advances by `vl * 1`
- `vle16.v` advances by `vl * 2`
- `vle32.v` advances by `vl * 4`
- `vle64.v` advances by `vl * 8`

```
/* Advance an int16_t pointer by vl elements. */
vsetvli t0, a2, e16, m1, ta, ma
slli    t1, t0, 1       /* bytes = vl * 2 */
add     a0, a0, t1
```

4.2 Floating-Point Vector Types

Floating-point vector operations are available when the relevant vector floating subsets are implemented. Element widths typically include:

- **FP16** (half), **FP32** (single), **FP64** (double) depending on the implementation

Floating semantics follow the platform floating-point model (rounding modes, exceptions, NaNs, signed zeros). In RVV, element width selection still comes from `SEW`, and operations are bounded by `vl`.

Example: vector add for FP32

```
/* y[i] = y[i] + x[i], float32. */
vsetvli t0, a2, e32, m1, ta, ma

vle32.v  v1, (a0)         /* x */
vle32.v  v2, (a1)         /* y */
vfadd.vv v2, v2, v1
vse32.v  v2, (a1)
```

Example: fused multiply-add (FMA) for FP32

```
/* y[i] = a*x[i] + y[i], float32, scalar a in fa0. */
vsetvli t0, a2, e32, m1, ta, ma

vle32.v   v1, (a0)        /* x */
vle32.v   v2, (a1)        /* y */
vfmacc.vf v2, fa0, v1     /* y += a*x */
vse32.v   v2, (a1)
```

Comparisons and masks in FP

```
/* mask = (x <= y) for float32 */
vsetvli   t0, a2, e32, m1, ta, ma
vle32.v   v1, (a0)
vle32.v   v2, (a1)
```

```
vmfle.vv  v0, v1, v2     /* v0 mask */
```

4.3 Widening and Narrowing Operations

A defining RVV capability is **mixed element widths within a computation flow** while still preserving the VLA model. Widening/narrowing is explicit via instruction families that change the destination element width relative to the source.

Widening integer operations

Widening produces **2×SEW** results from SEW inputs (e.g., int16 -¿ int32). Key patterns:

- `vwadd/vwsub` (signed), `vwaddu/vwsubu` (unsigned)
- `vwmul/vwmulu` widening multiply
- sign/zero extend helpers: `vsext.vf2`, `vzext.vf2` (and other factors)

Example: widening multiply int16 → int32 and accumulate

```
/* acc[i] += (int32)a[i] * (int32)b[i]
   a0=a*, a1=b*, a2=acc*, a3=n (elements)
*/
beqz    a3, .Ldone
.Lloop:
    /* Load int16 lanes */
    vsetvli t0, a3, e16, m1, ta, ma
    vle16.v  v1, (a0)          /* a */
    vle16.v  v2, (a1)          /* b */

    /* Widening multiply -> int32 lanes in v3 */
```

```
    vwmul.vv v3, v1, v2

    /* Switch to e32 for accumulator update.
       Use the returned vl for this configuration (do not assume it
        ↪  is identical). */
    vsetvli  t0, t0, e32, m1, ta, ma
    vle32.v  v4, (a2)         /* acc */
    vadd.vv  v4, v4, v3
    vse32.v  v4, (a2)

    /* Advance pointers: a,b by vl*2; acc by vl*4 */
    slli     t1, t0, 1
    add      a0, a0, t1
    add      a1, a1, t1
    slli     t2, t0, 2
    add      a2, a2, t2

    sub      a3, a3, t0
    bnez     a3, .Lloop
.Ldone:
ret
```

Widening floating-point operations

Widening FP produces wider FP results (e.g., FP32 → FP64, or FP16 → FP32 where supported). Typical families:

- `vfwadd/vfwsub` (widening add/sub)
- `vfwmul` (widening multiply)

- `vfwmacc` (widening fused multiply-accumulate)

Example: widening multiply-accumulate FP32 → FP64 accumulator (when supported)

```
/* acc64[i] += (double)x32[i] * (double)y32[i]
   a0=x32*, a1=y32*, a2=acc64*, a3=n
*/
beqz    a3, .Ldone
.Lloop:
    vsetvli  t0, a3, e32, m1, ta, ma
    vle32.v  v1, (a0)
    vle32.v  v2, (a1)

    /* Widening multiply: produces FP64 results in v3 */
    vfwmul.vv v3, v1, v2

    /* Reconfigure to operate on FP64 accumulator */
    vsetvli  t0, t0, e64, m1, ta, ma
    vle64.v  v4, (a2)
    vfadd.vv v4, v4, v3
    vse64.v  v4, (a2)

    /* Advance: x,y by vl*4; acc by vl*8 */
    slli     t1, t0, 2
    add      a0, a0, t1
    add      a1, a1, t1
    slli     t2, t0, 3
    add      a2, a2, t2

    sub      a3, a3, t0
```

```
    bnez     a3, .Lloop
.Ldone:
ret
```

Narrowing operations

Narrowing reduces element width (e.g., int32 -¿ int16, FP32 -¿ FP16). Narrowing commonly appears in:

- storing packed results,
- converting wide accumulators back to a smaller format,
- saturating/rounding down-shifts.

For integers, narrowing often requires explicit shifting/rounding or clipping:

- `vnclip` / `vnclipu`: narrowing with rounding and saturation behavior appropriate for packed fixed-point style flows
- shift-based narrowing patterns: widen compute, then shift-right and narrow

Example: narrow int32 to int16 with shift (fixed-point style)

```
/* Example pattern: out16 = (in32 >> s), then stored as 16-bit.
   This is a conceptual fixed-point flow; choose s per algorithm.
*/
vsetvli  t0, a2, e32, m1, ta, ma
vle32.v  v1, (a0)           /* in32 */

vsra.vx  v1, v1, a3         /* arithmetic shift right by scalar s */
```

```
vsetvli  t0, t0, e16, m1, ta, ma
/* A true narrowing instruction may be preferred where
 ↪  available/appropriate.
   Otherwise, use a pack/narrow step consistent with your data rules.
    ↪  */
vse16.v  v1, (a1)           /* store low 16 bits per element
 ↪  (algorithm-dependent) */
```

Example: narrow FP32 to FP16 (when supported)

```
/* out16[i] = (fp16)in32[i] */
vsetvli    t0, a2, e32, m1, ta, ma
vle32.v    v1, (a0)

/* Narrowing convert (rounding per current FP mode); exact mnemonic
 ↪  depends on subset support. */
vfncvt.f.f.w v2, v1

vsetvli    t0, t0, e16, m1, ta, ma
vse16.v    v2, (a1)
```

4.4 Mixed-Width Computation Rules

Mixed-width computation is not a hack in RVV; it is a normal, architected flow. Correctness and performance both depend on respecting these rules.

Rule 1: `SEW` defines the default element width, but instructions may override EEW

Most vector arithmetic uses the element width implied by the current `SEW`. Some instructions explicitly widen/narrow:

- widening: destination EEW = 2×SEW (or 4× in some extend operations)
- narrowing: destination EEW = SEW/2 (or smaller)

Therefore you must re-check:

- which load/store width matches your data,
- whether you need to reconfigure `SEW` before storing or combining results.

Rule 2: Changing `SEW` or `LMUL` can change `vl`

A common pitfall is assuming `vl` remains constant across configurations. It does not. Always treat `vl` as a value returned by `vsetvli` for the current `vtype`.

```
/* Same remaining element count, two configurations may yield
 ↪  different vl. */
vsetvli t0, a2, e16, m1, ta, ma   /* vl16 */
vsetvli t1, a2, e32, m1, ta, ma   /* vl32 (often <= vl16) */
```

Rule 3: Widening often increases register demand; plan for LMUL/register pressure

Widening results are larger. Practical implications:

- widening ops may require more destination register space (logically larger vectors),

- the number of independent live vectors you can keep decreases,
- you may need to reduce unrolling or adjust `LMUL`.

Rule 4: Explicitly separate phases by data width

Write kernels as **phases**:

1. load narrow data (`e8/e16`),
2. widen to compute width (`e32/e64`),
3. compute/accumulate at wide width,
4. narrow/convert back for storage if needed.

Phase-style example: int8 inputs, int32 accumulation, int8 output

```
/* Conceptual flow: out8[i] = clamp((sum of products) >> s)
   Demonstrates width phases; clamp/narrow rules are
    ↪  algorithm-defined.
*/
beqz    a3, .Ldone
.Lloop:
    /* Phase 1: load int8 */
    vsetvli t0, a3, e8, m1, ta, ma
    vle8.v  v1, (a0)
    vle8.v  v2, (a1)

    /* Phase 2: widen to int16 then int32 as needed */
    vsext.vf2 v3, v1          /* int16 */
    vsext.vf2 v4, v2          /* int16 */
```

```
    vsetvli  t0, t0, e16, m1, ta, ma
    vwmul.vv v5, v3, v4       /* int32 products in v5 */

    /* Phase 3: accumulate in int32 */
    vsetvli  t0, t0, e32, m1, ta, ma
    vle32.v  v6, (a2)         /* acc32 */
    vadd.vv  v6, v6, v5
    vse32.v  v6, (a2)

    /* Phase 4: optional shift + narrow/pack for output
     ↪  (algorithm-specific) */
    vsra.vx  v6, v6, a4       /* shift right by s */

    vsetvli  t0, t0, e8, m1, ta, ma
    vse8.v   v6, (a5)         /* store low bytes (use explicit
     ↪  narrow+clip if required) */

    /* Advance pointers per phase (use the same t0 returned by the
     ↪  current configuration). */
    /* For production code, keep pointer math consistent with each
     ↪  data stream's element size. */
    sub      a3, a3, t0
    bnez     a3, .Lloop
.Ldone:
ret
```

What to remember

- RVV data “types” are the combination of `SEW`, instruction semantics, and `vl`.

- Signed and unsigned integer behavior is instruction-defined; do not assume.
- Widening/narrowing is a core RVV workflow; design kernels in width phases.
- Never assume `vl` is constant across `SEW`/`LMUL` changes.

Chapter 5

Masking, Predication, and Control Flow

5.1 Vector Masks as First-Class Values

RVV treats **vector masks** as first-class values that can be:

- **produced** by compares (integer or floating-point),
- **consumed** by most arithmetic and memory operations (predication),
- **stored/loaded** (mask load/store) and combined (logical ops),
- **used** to express control flow without branching.

The conventional mask register is `v0`. Masked (predicated) execution is written using `v0.t` (mask-true lanes are active for that instruction).

Mask-producing comparisons

Comparisons write a mask value (typically into `v0`):

- integer: `vmseq/vmsne/vmslt/vmsltu/vmsle/vmsleu/vmsgt/vmsgtu`
- floating-point: `vmfeq/vmfne/vmflt/vmfle/vmfgt/vmfge` (subset-dependent)

```
/* Produce a mask: v0.t is true where x[i] < y[i] (signed int32). */
vsetvli   t0, a2, e32, m1, ta, ma
vle32.v   v1, (a0)              /* x */
vle32.v   v2, (a1)              /* y */
vmslt.vv  v0, v1, v2            /* v0 = (x < y) */
```

Mask composition (AND/OR/NOT)

Masks can be combined to form more complex predicates (mnemonics may vary by assembler, but the concept is stable):

```
/* Conceptual: combine two masks into v0.
   Use the mask logical operations supported by your
    ↪ toolchain/assembler. */
vmslt.vv  v0, v1, v2            /* m0 = (x < y) in v0 */
vmseq.vx  v1, v3, zero          /* m1 = (z == 0) written to some mask
 ↪ destination (toolchain-dependent) */
/* Then: m = m0 AND m1 (use mask logical op supported by ISA subset)
 ↪ */
```

Key rule

A mask is **data**. Once created, you can reuse it across multiple instructions to express a whole "if" block without a branch.

5.2 Masked Arithmetic and Memory Ops

Most RVV vector instructions accept an optional mask. The semantics are:

- if mask bit is 1, the element is updated normally;
- if mask bit is 0, the destination element is **inactive for that instruction** and is handled according to the selected **mask policy** (`ma` or `mu`).

Masked arithmetic (predicated update)

```
/* If x[i] < 0 then x[i] += k (int32), otherwise unchanged. */
vsetvli   t0, a1, e32, m1, tu, mu    /* undisturbed: masked-off lanes
 ↪  preserve old values */

vle32.v   v1, (a0)                   /* x */
vmslt.vx  v0, v1, zero               /* mask: x < 0 */

vadd.vx   v1, v1, a2, v0.t           /* predicated add: only negative
 ↪  lanes updated */
vse32.v   v1, (a0)
```

Masked loads and stores

Masked memory operations are essential for:

- conditional reads/writes (scatter/gather patterns),
- in-place selective updates,
- avoiding branches in sparse workloads.

```
/* Store y[i] only when predicate is true. */
vsetvli   t0, a2, e32, m1, ta, ma

vle32.v   v1, (a0)                  /* y */
vle32.v   v2, (a1)                  /* x */

/* mask: x != 0 (update only where x is non-zero) */
vmsne.vx  v0, v2, zero

/* masked store: write only lanes with v0.t = 1 */
vse32.v   v1, (a0), v0.t
```

Mask policy matters for correctness

- `ma` (mask agnostic): masked-off destination elements may become arbitrary.
- `mu` (mask undisturbed): masked-off destination elements preserve prior contents.

If you perform a masked load into a register and later use the whole register (including lanes that were masked off), you must use `mu` and explicitly initialize or preserve the inactive lanes.

```
/* Correctness: initialize destination, then masked-load into it
 ↪  (mu). */
vsetvli   t0, a2, e32, m1, tu, mu
vmv.v.i   v3, 0                     /* v3 = 0 for all lanes (active)
 ↪  */
vmslt.vx  v0, v1, a3                /* predicate */
vle32.v   v3, (a0), v0.t            /* masked load writes only
 ↪  selected lanes, others stay 0 */
```

5.3 Control Flow without Branching

RVV predication enables “branchless control flow” by turning conditions into masks and using:

- masked arithmetic,
- masked memory,
- merges/selects.

Branchless `if/else` via merge

```
/* y[i] = (x[i] < 0) ? a : b   for int32, branchless */
vsetvli    t0, a2, e32, m1, tu, mu

vle32.v    v1, (a0)                 /* x */
vmv.v.x    v2, a3                   /* broadcast a */
vmv.v.x    v3, a4                   /* broadcast b */

vmslt.vx   v0, v1, zero             /* mask: x < 0 */

/* Merge: where mask true take v2 else take v3 */
vmerge.vvm v4, v2, v3, v0.t

vse32.v    v4, (a1)                 /* store y */
```

Branchless clamp (min/max)

```
/* Clamp int16 values into [lo, hi] without branching. */
vsetvli   t0, a2, e16, m1, ta, ma
```

```
vle16.v   v1, (a0)                  /* x */
vmax.vx   v1, v1, a3                /* x = max(x, lo)   signed */
vmin.vx   v1, v1, a4                /* x = min(x, hi)   signed */
vse16.v   v1, (a0)
```

Why this is better than branches

- avoids branch mispredictions on data-dependent conditions,
- expresses per-element control decisions directly in dataflow,
- naturally composes with strip-mining: last iteration works automatically.

5.4 Safe Partial-Vector Execution

Partial-vector execution happens in two independent ways:

- **VL tail**: when `n` is not a multiple of the implementation's maximum vector capacity, the final iterations run with smaller `vl`.
- **Mask predication**: within `vl`, some lanes may be inactive for a given instruction.

Safety means: **no out-of-bounds memory access** and **no reliance on inactive lanes**.

Rule 1: strip-mine always (no fixed-lane assumptions)

```
/* Safe VLA add: no scalar remainder loop needed. */
.text
.align  2
.globl  vla_add_i32
```

```
vla_add_i32:
    /* a0=x*, a1=y*, a2=n */
    beqz    a2, .Ldone
.Lloop:
    vsetvli t0, a2, e32, m1, ta, ma

    vle32.v  v1, (a0)
    vle32.v  v2, (a1)
    vadd.vv  v2, v2, v1
    vse32.v  v2, (a1)

    slli     t1, t0, 2
    add      a0, a0, t1
    add      a1, a1, t1
    sub      a2, a2, t0
    bnez     a2, .Lloop
.Ldone:
    ret
```

Rule 2: masked memory must not touch inactive lanes

Masked loads/stores are the tool to safely handle conditional memory traffic. When a lane is masked off, it must not perform the memory access.

```
/* Safe selective store: write only when index is in-range, no
 ↪  branch. */
vsetvli   t0, a2, e32, m1, ta, ma

vle32.v   v1, (a0)                  /* values */
vle32.v   v2, (a1)                  /* indices */
```

```
/* mask: (idx < limit) unsigned */
vmsltu.vx v0, v2, a3                /* v0 = (idx < limit) */

/* Indexed store predicated by mask (scatter, subset-dependent). */
vsuxei32.v v1, (a4), v2, v0.t        /* store values to base[a4 +
 ↪  idx*4] where mask true */
```

Rule 3: if you will later use the full destination register, choose `tu`, `mu` and initialize

A common bug pattern is doing a masked load into a register and later reducing or storing the entire register while assuming masked-off lanes are unchanged. Fix it by:

1. initializing the destination register,

2. using `mu` so masked-off lanes are preserved.

```
/* Correct pattern: build a partial vector safely, then reduce. */
vsetvli   t0, a2, e32, m1, tu, mu
vmv.v.i   v8, 0                     /* seed inactive lanes */

vmsgt.vx  v0, v1, zero              /* mask: x > 0 */
vle32.v   v8, (a0), v0.t            /* load only where x>0; others
 ↪  remain 0 */

/* Now v8 is safe to consume as a whole vector (inactive lanes are
 ↪  defined as 0). */
```

Rule 4: policies are part of the correctness contract

- Use `ta,ma` for maximum freedom when inactive lanes are never observed.
- Use `tu,mu` when composing multi-step masked operations or when inactive lanes must remain valid.

A compact checklist

- Create masks with compares; reuse them to express whole conditional blocks.
- Prefer masked ops over branches for per-element conditions.
- Use masked loads/stores for safe conditional memory traffic.
- Initialize destinations and use `tu,mu` when inactive lanes will be observed later.
- Never assume any fixed lane count; trust only `vl`.

Chapter 6

Vector Load and Store Operations

6.1 Unit-Stride Loads and Stores

Unit-stride memory operations are the RVV "fast path": contiguous elements in memory map to contiguous elements in the active vector. They are the foundation for bandwidth-efficient kernels because they:

- maximize spatial locality and cache-line utilization,
- allow hardware to coalesce accesses naturally,
- minimize address-generation overhead (single base pointer).

Basic unit-stride forms

The element width is encoded in the mnemonic:

- loads: `vle8.v`, `vle16.v`, `vle32.v`, `vle64.v`
- stores: `vse8.v`, `vse16.v`, `vse32.v`, `vse64.v`

Example: memcpy-style copy (u8) with strip-mining

```
/* dst[i] = src[i] for i in [0..n_bytes).
   a0=dst*, a1=src*, a2=n_bytes
*/
.text
.align 2
.globl vla_copy_u8
vla_copy_u8:
    beqz    a2, .Ldone
.Lloop:
    vsetvli t0, a2, e8, m8, ta, ma   /* vl = min(n, VLMAX) for bytes
     ↪  */
    vle8.v  v0, (a1)
    vse8.v  v0, (a0)
    add     a0, a0, t0               /* advance by vl bytes */
    add     a1, a1, t0
    sub     a2, a2, t0
    bnez    a2, .Lloop
.Ldone:
    ret
```

Example: AXPY (float32), unit-stride

```
/* y[i] = a*x[i] + y[i], float32
   a0=x*, a1=y*, a2=n, fa0=a
*/
.text
.align 2
.globl vla_axpy_f32
```

```
vla_axpy_f32:
    beqz    a2, .Ldone
.Lloop:
    vsetvli t0, a2, e32, m1, ta, ma

    vle32.v   v1, (a0)          /* x */
    vle32.v   v2, (a1)          /* y */
    vfmacc.vf v2, fa0, v1       /* y += a*x */
    vse32.v   v2, (a1)

    slli      t1, t0, 2         /* bytes = vl * 4 */
    add       a0, a0, t1
    add       a1, a1, t1
    sub       a2, a2, t0
    bnez      a2, .Lloop
.Ldone:
    ret
```

Masked unit-stride store (selective write)

Unit-stride stores can be predicated to avoid branches and prevent unwanted writes.

```
/* If x[i] != 0 then y[i] = x[i], else leave y[i] unchanged.
   a0=x*, a1=y*, a2=n
*/
.text
.align 2
.globl store_if_nonzero_i32
store_if_nonzero_i32:
    beqz    a2, .Ldone
```

```
.Lloop:
    vsetvli  t0, a2, e32, m1, ta, ma
    vle32.v  v1, (a0)           /* x */
    vmsne.vx v0, v1, zero       /* mask: x != 0 */
    vse32.v  v1, (a1), v0.t     /* store only where mask true */

    slli     t1, t0, 2
    add      a0, a0, t1
    add      a1, a1, t1
    sub      a2, a2, t0
    bnez     a2, .Lloop
.Ldone:
    ret
```

Mask load/store (bitmask vectors)

Masks are data. RVV provides mask load/store to move mask bits to/from memory: `vlm.v` (load mask) and `vsm.v` (store mask).

```
/* Load a mask from memory and use it to predicate an add.
   a0=mask_bytes*, a1=x*, a2=y*, a3=n
*/
vsetvli  t0, a3, e32, m1, tu, mu
vlm.v    v0, (a0)               /* load mask bits into v0 */
vle32.v  v1, (a1)
vle32.v  v2, (a2)
vadd.vv  v2, v2, v1, v0.t
vse32.v  v2, (a2)
```

Fault-only-first unit-stride load (data-dependent loop exits)

`vleff.v` is designed for loops that may stop early due to a fault (e.g., page boundary / invalid memory) without doing a separate scalar probe. On a fault, it loads elements up to the first faulting element and sets `vl` to the number successfully loaded.

```
/* Conceptual pattern: safely pull bytes until a fault occurs.
   a0=src*, a1=max_bytes_to_try
   Returns: a1 reduced by bytes loaded; a0 advanced.
*/
.text
.align 2
.globl pull_until_fault_u8
pull_until_fault_u8:
    beqz    a1, .Ldone
.Lloop:
    vsetvli t0, a1, e8, m8, ta, ma
    vle8ff.v v1, (a0)           /* may fault; loads up to first
     ↪  fault, updates vl */
    /* After vle8ff.v, the architectural vl may be reduced to
     ↪  elements loaded */
    csrr    t2, vl              /* t2 = actual loaded count in
     ↪  elements (bytes here) */

    /* Consume v1[0..t2) here (e.g., scan, copy, parse) */

    add     a0, a0, t2
    sub     a1, a1, t2
    beqz    t2, .Ldone          /* if loaded 0, we hit a fault
     ↪  immediately */
```

```
    bnez    a1, .Lloop
.Ldone:
    ret
```

6.2 Strided Memory Access

Strided operations access elements at a fixed byte stride between consecutive elements:

- loads: `vlse8/16/32/64.v`
- stores: `vsse8/16/32/64.v`

The stride is a runtime register and is interpreted in **bytes**. Strided access is useful for:

- columns in a row-major matrix,
- interleaved structures when you cannot (or do not want to) reorganize memory,
- fixed-pattern sampling (e.g., every k-th element).

Example: load a column from row-major float32 matrix

Assume a row-major matrix `A` with `rows` and `cols`. A column `j` has a stride of `cols*4` bytes.

```
/* Load column j of float32 matrix A into a vector and add to y.
   a0=A*, a1=y*, a2=rows_remaining, a3=stride_bytes (cols*4),
    ↪  a4=col_offset_bytes (j*4)
*/
.text
.align 2
.globl add_column_f32
```

```
add_column_f32:
    beqz    a2, .Ldone
    add     t3, a0, a4          /* base = &A[0][j] */
.Lloop:
    vsetvli  t0, a2, e32, m1, ta, ma

    vlse32.v v1, (t3), a3       /* v1[k] = *(base + k*stride) */
    vle32.v  v2, (a1)           /* y contiguous */
    vfadd.vv v2, v2, v1
    vse32.v  v2, (a1)

    slli     t1, t0, 2
    add      a1, a1, t1         /* y += vl */
    /* Advance base by vl*stride for next chunk */
    mul      t2, t0, a3
    add      t3, t3, t2

    sub      a2, a2, t0
    bnez     a2, .Lloop
.Ldone:
    ret
```

Example: store strided (scatter-like, but regular)

```
/* Write x[i] into dst[i*stride] for i in [0..n), stride in bytes.
   a0=x*, a1=dst_base*, a2=n, a3=stride_bytes
*/
.text
.align 2
```

```
.globl store_strided_i32
store_strided_i32:
    beqz    a2, .Ldone
.Lloop:
    vsetvli  t0, a2, e32, m1, ta, ma
    vle32.v  v1, (a0)
    vsse32.v v1, (a1), a3       /* dst_base + i*stride */
    slli     t1, t0, 2
    add      a0, a0, t1
    mul      t2, t0, a3
    add      a1, a1, t2
    sub      a2, a2, t0
    bnez     a2, .Lloop
.Ldone:
    ret
```

Performance reality of strided accesses

- Strides that stay within a cache line behave closer to unit-stride.
- Large strides often become bandwidth-inefficient (many cache lines touched, low reuse).
- If you can transform data into SoA (structure-of-arrays) and use unit-stride, do so.

6.3 Indexed (Gather / Scatter) Operations

Indexed operations use a vector of indices to compute per-element addresses. They are the RVV gather/scatter tools:

- indexed loads: `vluxei32.v`, `vluxei64.v` (unordered), `vloxei32.v`, `vloxei64.v` (ordered)

- indexed stores: `vsuxei32.v`, `vsuxei64.v` (unordered), `vsoxei32.v`, `vsoxei64.v` (ordered)

The indices are interpreted as **byte offsets** added to a base address. The `ei32`/`ei64` suffix specifies the index element width (32-bit or 64-bit offsets).

Unordered vs ordered: when it matters

- **unordered** forms allow the implementation to reorder element accesses for performance.
- **ordered** forms preserve element ordering constraints (important for certain memory-mapped I/O patterns or when ordering has externally visible effects).

Example: gather float32 from base + offsets

```
/* out[i] = *(base + offsets[i]) as float32
   a0=base*, a1=offsets_u32*, a2=out*, a3=n
   offsets are byte offsets.
*/
.text
.align 2
.globl gather_f32_ei32
gather_f32_ei32:
    beqz    a3, .Ldone
.Lloop:
    vsetvli   t0, a3, e32, m1, ta, ma
    vle32.v   v1, (a1)              /* offsets (u32) */
    vluxei32.v v2, (a0), v1         /* unordered gather: v2[i] =
     ↪  *(base + v1[i]) */
```

```
    vse32.v   v2, (a2)

    slli      t1, t0, 2
    add       a1, a1, t1
    add       a2, a2, t1
    sub       a3, a3, t0
    bnez      a3, .Lloop
.Ldone:
    ret
```

Example: scatter int32 with bounds-check mask (memory-safe)

The correct pattern is: compute a predicate mask that ensures every active lane is in-bounds, then scatter under that mask.

```
/* dst[idx[i]] = val[i] for i in [0..n), with idx bounds check.
   a0=dst_base*, a1=idx_u32*, a2=val_i32*, a3=n, a4=limit_elems
   idx are element indices; convert to byte offsets by <<2.
*/
.text
.align 2
.globl scatter_i32_checked
scatter_i32_checked:
    beqz    a3, .Ldone
.Lloop:
    vsetvli   t0, a3, e32, m1, tu, mu

    vle32.v   v1, (a1)              /* idx */
    vle32.v   v2, (a2)              /* val */
```

```
    /* mask = (idx < limit) unsigned */
    vmsltu.vx v0, v1, a4

    /* offsets_bytes = idx << 2 */
    vsll.vi   v1, v1, 2

    /* masked unordered scatter */
    vsuxei32.v v2, (a0), v1, v0.t

    slli      t1, t0, 2
    add       a1, a1, t1
    add       a2, a2, t1
    sub       a3, a3, t0
    bnez      a3, .Lloop
.Ldone:
    ret
```

Indexed access performance notes

- Gathers/scatters are inherently latency-heavy when indices are random (poor locality).
- Use them when necessary, but prefer:
  - data reordering (SoA transforms),
  - blocked algorithms that increase locality,
  - converting irregular patterns into unit-stride/strided patterns.

6.4 Alignment, Faulting, and Memory Safety

This section is about writing kernels that are correct on *all* RVV systems and safe under tail/mask partial execution.

Alignment expectations

- **Natural alignment is the performance-friendly default** (e.g., 4-byte alignment for `vle32.v`, 8-byte for `vle64.v`).
- Misalignment can be slower due to split-line accesses and extra micro-ops.
- Some environments may raise **misaligned address exceptions** for certain vector memory instructions (notably some whole-register load/store forms) when the base is not naturally aligned.

Rule 1: never assume alignment unless you own it

If the caller does not guarantee alignment, either:

- add an alignment prolog (scalar or vector with smaller EEW),
- or use a safe, always-correct path and accept the performance hit.

Example: alignment prolog to 16-byte boundary (u8), then vector copy

```
/* Align dst and src to 16 bytes by copying a few bytes first
 ↪  (conceptual prolog). */
andi    t0, a0, 15
beqz    t0, .Lvec
li      t1, 16
```

```
sub    t1, t1, t0              /* t1 = bytes to reach 16B boundary */
bltu   a2, t1, .Ltail_scalar

/* scalar prolog: copy t1 bytes */
.Lprolog:
lb     t2, 0(a1)
sb     t2, 0(a0)
addi   a0, a0, 1
addi   a1, a1, 1
addi   a2, a2, -1
addi   t1, t1, -1
bnez   t1, .Lprolog

.Lvec:
    /* now use the vla_copy_u8 style loop */
```

Faulting behavior essentials

Vector memory operations can fault similarly to scalar loads/stores (page faults, access faults, etc.). The key RVV tool for safe early-stop patterns is **fault-only-first** (`vleff.v`) which:

- loads contiguous elements up to (but not including) the first faulting element,
- updates `vl` to the number of elements successfully loaded,
- enables robust vectorization of while-loops with data-dependent exit and unknown safe length.

Rule 2: memory safety is achieved by `vl` and masks, not by luck

- Use strip-mining so the last chunk uses a smaller `vl` instead of reading past the end.

- For indexed accesses, **always compute a bounds mask** and perform masked gather/scatter.

- If masked-off lanes must retain defined values for later use, initialize the destination and use `tu,mu`.

Example: safe gather with bounds mask + zero fill

```
/* out[i] = (idx[i] < limit) ? src[idx[i]] : 0
   a0=src_base*, a1=idx_u32*, a2=out*, a3=n, a4=limit_elems
*/
.text
.align 2
.globl gather_i32_safe_zerofill
gather_i32_safe_zerofill:
    beqz    a3, .Ldone
.Lloop:
    vsetvli   t0, a3, e32, m1, tu, mu

    vle32.v   v1, (a1)              /* idx */
    vmsltu.vx v0, v1, a4            /* mask: idx < limit */

    /* Prepare out vector with zeros for all lanes (so masked-off
     ↪  lanes become defined). */
    vmv.v.i   v2, 0

    /* offsets = idx << 2 (byte offsets) */
    vsll.vi   v3, v1, 2
```

```
    /* Masked gather into v2: only in-range lanes are loaded, others
     ↪  remain 0. */
    vluxei32.v v2, (a0), v3, v0.t

    vse32.v   v2, (a2)

    slli      t1, t0, 2
    add       a1, a1, t1
    add       a2, a2, t1
    sub       a3, a3, t0
    bnez      a3, .Lloop
.Ldone:
    ret
```

Rule 3: prefer unit-stride whenever possible

From a performance engineering standpoint:

- unit-stride is the baseline target,
- strided is acceptable when the stride is modest and predictable,
- indexed should be treated as a last resort unless your algorithm is naturally sparse/irregular.

What to remember

- Unit-stride (`vle*/vse*`) is the primary high-throughput memory path.
- Strided (`vlse*/vsse*`) is regular but can be cache-inefficient for large strides.

- Indexed gather/scatter (`vluxei*/vsuxei*`, `vloxei*/vsoxei*`) is powerful but expensive; mask it for safety.
- Alignment affects both performance and, in some environments, correctness (possible misaligned traps for some instruction forms).
- Use `vleff.v` when the safe readable length is not known in advance.

Chapter 7

Arithmetic, Logical, and Reduction Operations

7.1 Integer Arithmetic and Saturation

RVV integer operations are primarily **lane-wise**: each element is computed independently for the active lanes `[0, vl)`. Signed vs unsigned behavior is instruction-defined, not data-defined.

Core lane-wise arithmetic and logic

- add/sub: `vadd`, `vsub`
- min/max: `vmin`/`vmax` (signed), `vminu`/`vmaxu` (unsigned)
- shifts: `vsll`, `vsrl` (logical), `vsra` (arithmetic)
- bitwise: `vand`, `vor`, `vxor`, `vnot`

Saturating arithmetic (clamping on overflow)

Saturating ops are essential for DSP, imaging, and packed integer pipelines where overflow must clamp instead of wrap.

- signed saturating add/sub: `vsadd`, `vssub`
- unsigned saturating add/sub: `vsaddu`, `vssubu`

Example: unsigned saturating add for bytes (u8)

```
/* dst[i] = sat_u8(a[i] + b[i])
   a0=a*, a1=b*, a2=dst*, a3=n (bytes)
*/
.text
.align 2
.globl sat_add_u8
sat_add_u8:
    beqz    a3, .Ldone
.Lloop:
    vsetvli  t0, a3, e8, m1, ta, ma
    vle8.v   v1, (a0)
    vle8.v   v2, (a1)
    vsaddu.vv v3, v1, v2        /* unsigned saturating add */
    vse8.v   v3, (a2)

    add      a0, a0, t0
    add      a1, a1, t0
    add      a2, a2, t0
    sub      a3, a3, t0
    bnez     a3, .Lloop
```

```
.Ldone:
    ret
```

Example: signed saturating add for int16

```
/* dst[i] = sat_i16(x[i] + y[i])
   a0=x*, a1=y*, a2=dst*, a3=n (elements)
*/
.text
.align 2
.globl sat_add_i16
sat_add_i16:
    beqz    a3, .Ldone
.Lloop:
    vsetvli   t0, a3, e16, m1, ta, ma
    vle16.v   v1, (a0)
    vle16.v   v2, (a1)
    vsadd.vv  v3, v1, v2        /* signed saturating add */
    vse16.v   v3, (a2)

    slli      t1, t0, 1
    add       a0, a0, t1
    add       a1, a1, t1
    add       a2, a2, t1
    sub       a3, a3, t0
    bnez      a3, .Lloop
.Ldone:
    ret
```

Narrowing with rounding/saturation (packed pipelines)

When you compute in a wider type then pack down, prefer the dedicated narrowing/clip family:

- `vnclip` (signed), `vnclipu` (unsigned): narrowing with rounding and saturation

```
/* Conceptual fixed-point pack:
   out16 = sat( round( in32 >> sh ) ) for unsigned (use vnclipu).
   a0=in32*, a1=out16*, a2=n, a3=sh
*/
.text
.align 2
.globl pack_u32_to_u16_clip
pack_u32_to_u16_clip:
    beqz    a2, .Ldone
.Lloop:
    vsetvli    t0, a2, e32, m1, ta, ma
    vle32.v    v1, (a0)

    vnclipu.vx v2, v1, a3       /* narrow with rounding/saturation */

    vsetvli    t0, t0, e16, m1, ta, ma
    vse16.v    v2, (a1)

    slli       t1, t0, 2
    add        a0, a0, t1
    slli       t2, t0, 1
    add        a1, a1, t2
    sub        a2, a2, t0
```

```
    bnez        a2, .Lloop
.Ldone:
    ret
```

Practical guidance

- Use non-saturating ops for general-purpose arithmetic (wrap semantics).
- Use saturating ops for pixel/audio/packed fixed-point where overflow must clamp.
- Pack/unpack with widening + `vnclip`/`vnclipu` instead of ad-hoc bit tricks.

7.2 Floating-Point Arithmetic and Precision

RVV floating-point operations follow IEEE-style behavior (NaNs, infinities, signed zeros) and obey the current FP environment (rounding mode, exceptions). Key themes:

- **precision** depends on element width (FP16/FP32/FP64 subsets),
- **FMA** changes numerical results compared to separate mul+add,
- **reductions** are order-dependent and can be non-associative in FP.

Core floating ops

- add/sub/mul/div: `vfadd`, `vfsub`, `vfmul`, `vfdiv`
- sqrt: `vfsqrt`
- min/max with FP rules: `vfmin`, `vfmax`
- fused multiply-add: `vfmacc`, `vfnmacc`, etc.

Example: FP32 SAXPY using FMA (single rounding)

```
/* y[i] = a*x[i] + y[i], float32
   a0=x*, a1=y*, a2=n, fa0=a
*/
.text
.align 2
.globl saxpy_f32_fma
saxpy_f32_fma:
    beqz    a2, .Ldone
.Lloop:
    vsetvli   t0, a2, e32, m1, ta, ma
    vle32.v   v1, (a0)          /* x */
    vle32.v   v2, (a1)          /* y */
    vfmacc.vf v2, fa0, v1       /* y += a*x (fused) */
    vse32.v   v2, (a1)

    slli      t1, t0, 2
    add       a0, a0, t1
    add       a1, a1, t1
    sub       a2, a2, t0
    bnez      a2, .Lloop
.Ldone:
    ret
```

Precision pitfalls that matter in kernels

- **FMA vs mul+add**: FMA rounds once; separate operations round twice. Results can differ (often better with FMA).

- **FP reductions**: floating-point addition is not associative; different `vl` and reduction trees can change the last bits.
- **FP16**: useful for bandwidth/throughput, but error grows quickly; accumulate in FP32/FP64 when accuracy matters.

Example: accumulate FP16 inputs into FP32 (when supported)

```
/* sum += (float)h[i]  where h is FP16, accumulate in FP32.
   a0=h*, a1=n, fa0=seed (0.0)
*/
.text
.align 2
.globl sum_f16_to_f32
sum_f16_to_f32:
    beqz    a1, .Ldone

    /* vector accumulator v0 as FP32 */
    vsetvli  t0, zero, e32, m1, ta, ma
    vfmv.v.f v0, fa0

.Lloop:
    vsetvli  t0, a1, e16, m1, ta, ma
    vle16.v  v1, (a0)            /* FP16 lanes in v1 */

    /* Widen convert FP16 -> FP32 (subset-dependent mnemonic). */
    vfwcvt.f.f.v v2, v1

    /* Switch to e32 to accumulate */
    vsetvli  t0, t0, e32, m1, ta, ma
```

```
    vfadd.vv v0, v0, v2

    slli     t1, t0, 1
    add      a0, a0, t1
    sub      a1, a1, t0
    bnez     a1, .Lloop

    /* Reduce vector accumulator to scalar */
    vfmv.v.f   v3, fa0
    vfredsum.vs v4, v0, v3
    vfmv.f.s   fa0, v4
.Ldone:
    ret
```

7.3 Reductions and Horizontal Operations

Reduction semantics

Reductions take a vector and produce a scalar (in a vector element) by applying an associative operator across the active elements `[0, vl)`.

- integer reductions: `vredsum`, `vredmax`, `vredmin`, `vredand`, `vredor`, `vredxor`
- floating reductions: `vfredsum`, `vfredmax`, `vfredmin` (subset-dependent)

Reductions use a **seed** vector operand (often a vector with a scalar value broadcast) and write the result to element 0 of a destination vector.

Example: integer sum reduction (int32)

```
/* Return sum(x[0..n)) in a0 (int32) for conceptual pattern.
```

```
   a0=x*, a1=n
*/
.text
.align 2
.globl sum_i32_reduce
sum_i32_reduce:
    beqz    a1, .Ldone

    /* v0 = partial sums */
    vsetvli  t0, zero, e32, m1, ta, ma
    vmv.v.i  v0, 0

.Lloop:
    vsetvli  t0, a1, e32, m1, ta, ma
    vle32.v  v1, (a0)
    vadd.vv  v0, v0, v1

    slli     t1, t0, 2
    add      a0, a0, t1
    sub      a1, a1, t0
    bnez     a1, .Lloop

    /* Reduce v0 into v2[0] with seed 0 in v3 */
    vmv.v.i    v3, 0
    vredsum.vs v2, v0, v3

    /* Move result to scalar */
    vmv.x.s    a0, v2
.Ldone:
```

```
    ret
```

Example: horizontal bitwise OR (u64)

```
/* Return OR of all u64 elements in a0.
   a0=x*, a1=n
*/
.text
.align 2
.globl or_u64_reduce
or_u64_reduce:
    beqz    a1, .Ldone

    vsetvli  t0, zero, e64, m1, ta, ma
    vmv.v.i  v0, 0

.Lloop:
    vsetvli  t0, a1, e64, m1, ta, ma
    vle64.v  v1, (a0)
    vor.vv   v0, v0, v1

    slli     t1, t0, 3
    add      a0, a0, t1
    sub      a1, a1, t0
    bnez     a1, .Lloop

    vmv.v.i    v3, 0
    vredor.vs  v2, v0, v3
    vmv.x.s    a0, v2
```

```
.Ldone:
    ret
```

Horizontal operations beyond reductions

For operations like prefix-sum, compaction, and permutations, RVV provides cross-lane primitives (next section) that you combine into horizontal algorithms. The key is: **keep the algorithm VLA** (never hard-code lane count).

7.4 Cross-Lane Semantics

Most RVV arithmetic/logical instructions are **element-wise** and have no cross-lane interaction. Cross-lane behavior appears in a distinct set of operations:

1) Permute / gather within a vector

- `vrgather`: gather elements from a source vector using per-lane indices
- `vrgatherei16`: variant with 16-bit indices (subset-dependent)

```
/* Reverse a vector chunk (conceptual):
   idx[i] = (vl-1-i), then vrgather to reverse lanes.
   a0=x*, a1=out*, a2=n
*/
.text
.align 2
.globl reverse_chunk_i32
reverse_chunk_i32:
    beqz    a2, .Ldone
.Lloop:
```

```
    vsetvli  t0, a2, e32, m1, ta, ma
    vle32.v  v1, (a0)

    /* Build indices: idx = [vl-1, vl-2, ...] (conceptual).
       In practice, generate with vid + subtract from (vl-1). */
    vid.v    v2                    /* v2 = [0,1,2,...] */
    addi     t1, t0, -1
    vmv.v.x  v3, t1                /* broadcast (vl-1) */
    vsub.vv  v2, v3, v2            /* idx = (vl-1) - id */

    vrgather.vv v4, v1, v2         /* v4[i] = v1[idx[i]] */
    vse32.v     v4, (a1)

    slli     t2, t0, 2
    add      a0, a0, t2
    add      a1, a1, t2
    sub      a2, a2, t0
    bnez     a2, .Lloop
.Ldone:
    ret
```

2) Slide operations (neighbor lane movement)

- `vslideup`/`vslidedown`: shift lanes up/down inserting a scalar or preserving policy-defined values
- `vslide1up`/`vslide1down`: slide by 1 with scalar insertion

```
/* Build a 1-lane shifted version and add: y[i] = x[i] + x[i-1] (with
↪  x[-1]=0).
```

```
   a0=x*, a1=y*, a2=n
*/
.text
.align 2
.globl add_prev_i32
add_prev_i32:
    beqz    a2, .Ldone
.Lloop:
    vsetvli      t0, a2, e32, m1, tu, mu
    vle32.v      v1, (a0)

    vmv.v.i      v2, 0
    vslide1up.vx v2, v1, zero     /* v2 = [0, x0, x1, ...] within
     ↪  this chunk */

    vadd.vv      v3, v1, v2
    vse32.v      v3, (a1)

    slli         t1, t0, 2
    add          a0, a0, t1
    add          a1, a1, t1
    sub          a2, a2, t0
    bnez         a2, .Lloop
.Ldone:
    ret
```

3) Compress / expand (mask-driven lane movement)

- `vcompress`: pack elements with mask=1 into the low lanes (order-preserving)

```
/* Filter positives: compact x where x>0 into the front of a vector
 ↪  register.
   This is a building block for branchless filtering.
*/
vsetvli   t0, a1, e32, m1, tu, mu
vle32.v   v1, (a0)
vmsgt.vx  v0, v1, zero            /* mask: x>0 */
vcompress.vm v2, v1, v0.t         /* v2 holds packed positives in low
 ↪  lanes */
```

Cross-lane rule of thumb

- Use element-wise arithmetic for throughput (no lane dependencies).
- Use cross-lane ops deliberately: they are powerful but can be more expensive and can reduce ILP.
- When writing cross-lane algorithms, keep them VLA: generate indices via `vid`, use `vl` and masks, never assume a fixed lane count.

What to remember

- Integer ops are lane-wise; saturation and clip ops exist for packed pipelines.
- FP ops follow IEEE behavior; FMA and reductions influence numerical results.
- Reductions produce a scalar in a vector element using a seed operand; FP reductions are order-sensitive.
- Cross-lane semantics are explicit (gather/slide/compress), not accidental; treat them as separate performance tools.

Chapter 8

Writing Vector-Length-Agnostic Code

8.1 The VL-Driven Loop Pattern

The defining rule of RVV programming is simple: **all vector work is driven by the runtime value `vl`**. Correct code never assumes a fixed lane count and never hard-codes vector widths. Instead, each iteration:

1. requests a legal `vl` for the remaining element count,
2. performs vector work on exactly `vl` elements,
3. advances pointers and counters by `vl`.

This is the **strip-mined loop**. It is not an optimization trick; it is the execution model.

Canonical VL-driven skeleton

```
/* Canonical VLA loop skeleton.
   a0 = ptr0, a1 = ptr1, a2 = n (elements)
```

```
*/
beqz    a2, .Ldone
.Lloop:
    vsetvli t0, a2, e32, m1, ta, ma   /* t0 = vl */

    /* vector work on v[0..vl) */

    slli    t1, t0, 2                 /* bytes = vl * sizeof(element)
    ↪  */
    add     a0, a0, t1
    add     a1, a1, t1
    sub     a2, a2, t0
    bnez    a2, .Lloop
.Ldone:
    ret
```

Why this pattern is mandatory

- It guarantees correctness for any `VLEN`.
- The final iteration naturally handles tails without scalar cleanup.
- The same binary scales forward as vector hardware grows.

Any deviation (fixed unrolling, assumed lane count, manual remainder handling) breaks portability.

8.2 Portable Loop Decomposition

Real kernels often mix loads, arithmetic, masking, and stores. The key to portability is decomposing the loop into **phases** that all obey the same `vl`-driven structure.

Phase-based decomposition

A robust RVV loop typically has these phases:

1. Configure vector state and obtain `vl`.
2. Load inputs for `vl` elements.
3. Compute (possibly with masks).
4. Store results for `vl` elements.
5. Advance pointers and counters by `vl`.

Example: portable vector add with a conditional

```
/* y[i] = (x[i] > 0) ? x[i] : y[i]
   a0 = x*, a1 = y*, a2 = n
*/
.text
.align 2
.globl vla_cond_add_i32
vla_cond_add_i32:
    beqz    a2, .Ldone
.Lloop:
    /* Phase 1: configure */
    vsetvli   t0, a2, e32, m1, tu, mu

    /* Phase 2: load */
    vle32.v   v1, (a0)              /* x */
    vle32.v   v2, (a1)              /* y */
```

```
    /* Phase 3: compute mask and update */
    vmsgt.vx  v0, v1, zero          /* mask: x > 0 */
    vadd.vv   v2, v2, v1, v0.t      /* y += x where mask true */

    /* Phase 4: store */
    vse32.v   v2, (a1)

    /* Phase 5: advance */
    slli      t1, t0, 2
    add       a0, a0, t1
    add       a1, a1, t1
    sub       a2, a2, t0
    bnez      a2, .Lloop
.Ldone:
    ret
```

Portability rule

If a loop can be explained as “repeat these phases while `n > 0`”, it is portable. If it relies on knowing how many lanes fit, it is not.

8.3 Avoiding Fixed-Width Assumptions

Most RVV bugs come from accidentally importing SIMD habits from fixed-width ISAs. These assumptions must be avoided.

Common incorrect assumptions

- Assuming `vl` is constant across iterations.

- Assuming `vl` is the same for different `SEW` or `LMUL`.
- Assuming inactive lanes are zero or preserved without policy control.
- Assuming a vector register maps to a single architectural register under all `LMUL`.

Incorrect pattern (do not do this)

```
/* Incorrect: assumes vl does not change after reconfiguration. */
vsetvli t0, a2, e16, m1, ta, ma
/* ... */
vsetvli t0, t0, e32, m1, ta, ma   /* vl may change here */
/* Using old pointer math based on earlier vl is wrong */
```

Correct pattern

Always treat the value returned by `vsetvli` as authoritative for the current configuration.

```
/* Correct: re-derive vl after changing SEW. */
vsetvli t0, a2, e16, m1, ta, ma
/* load/narrow work */
vsetvli t0, t0, e32, m1, ta, ma
/* compute/store using the new vl */
```

Avoiding implicit lane assumptions

Never write code like:

- “process 8 elements per iteration”,
- “unroll by 4 because vectors are 256-bit”,
- “handle remainder with scalar loop”.

RVV already provides the remainder handling via `vl`.

8.4 Correctness Across Implementations

RVV correctness means the same program produces correct results on:

- small embedded cores with minimal vector resources,
- large server cores with wide vector units,
- future implementations with larger `VLEN`.

Correctness invariants

A correct RVV kernel satisfies all of the following:

1. All memory accesses are bounded by `vl` or masked.
2. All pointer increments are derived from the returned `vl`.
3. Mask and tail policies are chosen intentionally.
4. No inactive lane is read unless it is explicitly initialized or preserved.

Example: safe partial-vector execution with initialization

```
/* out[i] = (idx[i] < limit) ? src[idx[i]] : 0
   a0=src*, a1=idx*, a2=out*, a3=n, a4=limit
*/
.text
.align 2
.globl vla_safe_gather_i32
```

```
vla_safe_gather_i32:
    beqz    a3, .Ldone
.Lloop:
    vsetvli   t0, a3, e32, m1, tu, mu

    /* Initialize destination so masked-off lanes are defined */
    vmv.v.i   v2, 0

    /* Load indices and compute bounds mask */
    vle32.v   v1, (a1)
    vmsltu.vx v0, v1, a4

    /* Convert indices to byte offsets */
    vsll.vi   v1, v1, 2

    /* Masked gather */
    vluxei32.v v2, (a0), v1, v0.t

    /* Store full vector safely */
    vse32.v   v2, (a2)

    slli      t1, t0, 2
    add       a1, a1, t1
    add       a2, a2, t1
    sub       a3, a3, t0
    bnez      a3, .Lloop
.Ldone:
    ret
```

Why this is correct everywhere

- Out-of-range indices are masked before memory access.
- Masked-off lanes are initialized to zero.
- No assumption is made about how many lanes exist.

A minimal correctness checklist

- Did every vector loop start with `vsetvli`?
- Are all pointer updates derived from the returned `vl`?
- Are masked-off lanes either ignored or explicitly initialized?
- Would the code still work if `vl` changed every iteration?

If the answer to all four is yes, the code is genuinely vector-length-agnostic.

Chapter 9

Compiler Interaction and Toolchain Behavior

9.1 How Compilers Lower RVV Code

Compilers typically lower RVV code into a small set of recurring assembly patterns. If you can recognize these patterns, you can quickly validate correctness (VLA discipline) and reason about performance.

Pattern A: Strip-mined loop with `vsetvli`

The compiler emits a loop where each iteration:

1. sets `vl` using `vsetvli`,
2. performs loads/computation/stores for `vl` elements,
3. advances pointers by `vl * sizeof(T)`,

4. decrements the remaining count by `vl`.

```
/* Typical lowered form for: for(i) y[i] += x[i] (float32)
   a0=x*, a1=y*, a2=n
*/
.Lloop:
    vsetvli t0, a2, e32, m1, ta, ma   /* t0 = vl */
    vle32.v v1, (a0)
    vle32.v v2, (a1)
    vfadd.vv v2, v2, v1
    vse32.v v2, (a1)

    slli    t1, t0, 2                 /* bytes = vl * 4 */
    add     a0, a0, t1
    add     a1, a1, t1
    sub     a2, a2, t0
    bnez    a2, .Lloop
```

Pattern B: Masked tail without scalar remainder

The compiler prefers RVV-native tail handling. The last iteration simply has a smaller `vl`. No scalar cleanup is required for regular unit-stride loops.

Pattern C: Predicated execution for data-dependent conditions

When vectorizing conditionals, compilers emit:

- a compare that produces a mask (often in `v0`),
- a masked arithmetic op, masked load/store, or a `vmerge`.

```
/* Typical for: if(x[i] > 0) y[i] += x[i]
   a0=x*, a1=y*, a2=n
*/
.Lloop:
    vsetvli   t0, a2, e32, m1, tu, mu
    vle32.v   v1, (a0)                /* x */
    vle32.v   v2, (a1)                /* y */
    vmsgt.vx  v0, v1, zero            /* mask: x > 0 */
    vadd.vv   v2, v2, v1, v0.t        /* predicated add */
    vse32.v   v2, (a1)
    slli      t1, t0, 2
    add       a0, a0, t1
    add       a1, a1, t1
    sub       a2, a2, t0
    bnez      a2, .Lloop
```

Pattern D: Reconfiguration when element width changes

Widening/narrowing flows often require changing `SEW`. Compilers insert additional `vsetvli` when moving between phases (e.g., load int16, compute in int32, store int16).

```
/* Typical widen: int16 -> int32 compute -> store int32
   a0=in16*, a1=out32*, a2=n
*/
.Lloop:
    vsetvli   t0, a2, e16, m1, ta, ma
    vle16.v   v1, (a0)

    /* Widen convert/extend to 32-bit lanes (exact opcode depends on
     ↪  intent) */
```

```
    vsext.vf2 v2, v1

    vsetvli   t0, t0, e32, m1, ta, ma
    /* compute in e32 */
    vse32.v   v2, (a1)

    slli      t1, t0, 1               /* in16 advance: vl*2 */
    add       a0, a0, t1
    slli      t2, t0, 2               /* out32 advance: vl*4 */
    add       a1, a1, t2
    sub       a2, a2, t0
    bnez      a2, .Lloop
```

Lowering sanity checks

- **Always** see `vsetvli` (or `vsetivli`) inside vector loops.
- Pointer increments must be derived from the `vl` that was actually returned.
- If `SEW`/`LMUL` changes, expect a new `vsetvli`.
- Masked operations should match your data-dependent semantics (no accidental use of garbage inactive lanes).

9.2 Intrinsics vs Auto-Vectorization

Auto-vectorization

Auto-vectorization is ideal when:

- loops are simple, straight-line, with predictable memory,

- aliasing is controlled (`restrict`-style assumptions),
- types and loop bounds are clear to the compiler,
- the kernel is not dominated by irregular gathers/scatters.

What the compiler needs (practical)

- no hidden dependencies across iterations,
- contiguous memory when possible (unit-stride),
- explicit alignment hints when valid,
- clear trip counts and no complex control flow.

```
/* Auto-vectorization friendly shape (conceptual C). */
void axpy_f32(float* __restrict y,
              const float* __restrict x,
              float a, unsigned long n)
{
    for (unsigned long i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
```

Intrinsics

Intrinsics are ideal when:

- you need exact control of masks, `SEW`/`LMUL`, or specific instructions,
- the compiler fails to vectorize or generates suboptimal code,
- you implement specialized gather/scatter or mixed-width pipelines,
- you want predictable instruction selection across versions.

Trade-offs

- Auto-vectorization improves portability across compilers but may be brittle to code shape.
- Intrinsics improve control but increase code complexity and tie you to a specific intrinsic API.
- In practice: start with auto-vectorization; use intrinsics for the hot 5% where it matters.

A disciplined hybrid strategy

1. Write a clean scalar reference loop.
2. Make it vectorization-friendly (restrict, simple control flow, separate tails).
3. Inspect emitted assembly.
4. If needed, replace only the innermost kernel with intrinsics or inline assembly.

9.3 ABI Considerations

Vector code correctness is not only about ISA semantics; it is also about the calling convention and how toolchains define preservation of vector state.

General ABI realities for RVV kernels

- Do **not** assume vector registers survive a function call unless your ABI guarantees it.
- Treat `vl` and `vtype` as **part of the vector state**: a call can change them.
- Therefore, robust kernels **always** execute `vsetvli` in the function (and often inside the loop) and never rely on prior configuration.

Recommended kernel discipline

- For **leaf** hot loops: keep them leaf; avoid calls inside the vector loop.
- For **non-leaf** code: set `vtype`/`vl` again after a call if vector work continues.
- If you must keep vector temporaries live across a call, explicitly spill/reload (rare; usually avoid).

Function boundaries and `vsetvli`

Even when a caller already configured vectors, a callee should not assume it. The safe rule is: **configure at the point of use**.

```
/* ABI-safe rule: configure in the callee before vector work. */
.globl  kernel_add_f32
kernel_add_f32:
    /* ... */
.Lloop:
    vsetvli t0, a2, e32, m1, ta, ma
    /* vector work */
    /* ... */
    ret
```

Mask state

- Treat `v0` mask contents as volatile unless you control the full region of code.
- If a mask must be reused after a sequence that might clobber it, recompute it or store/load it with mask load/store.

Mixed object files and ISA attributes

If you mix objects built with different `-march` settings, you must ensure:

- all objects that contain RVV instructions are built with the appropriate vector ISA enabled,
- the final binary targets a consistent baseline (or uses multiversion dispatch intentionally).

9.4 Debugging and Inspection Strategies

Toolchain behavior must be validated by inspecting **what was emitted**, not what you intended.

1) Inspect generated assembly early

Compile to assembly and verify:

- the loop is strip-mined with `vsetvli`,
- pointer math matches `vl` and element sizes,
- masks are used where you expect,
- no accidental scalar remainder loop exists unless intended.

```
/* Typical inspection commands (conceptual). */
clang -O3 -S -march=rv64gcv -mabi=lp64d -fverbose-asm kernel.c
gcc   -O3 -S -march=rv64gcv -mabi=lp64d -fverbose-asm kernel.c
```

2) Ask the compiler why it did or did not vectorize

Compilers can emit vectorization diagnostics:

- **LLVM/Clang**: vectorization remarks (loop vectorizer / SLP)
- **GCC**: vectorization reports (`opt-info-vec`)

```
/* Conceptual vectorization diagnostics. */
clang -O3 -march=rv64gcv -mabi=lp64d -Rpass=loop-vectorize
 ↪  -Rpass-missed=loop-vectorize kernel.c
gcc   -O3 -march=rv64gcv -mabi=lp64d -fopt-info-vec-optimized
 ↪  -fopt-info-vec-missed kernel.c
```

3) Disassemble the final binary

Always inspect the linked binary because:

- LTO and inlining can change code shape,
- scheduling and relaxation can alter instruction placement,
- the final result may differ from the standalone `-S` output.

```
/* Conceptual disassembly. */
objdump -drwC a.out
llvm-objdump -d --no-show-raw-insn a.out
```

4) Validate `vl` handling in tricky cases

Watch for these common bugs in emitted or handwritten code:

- using a stale `vl` after changing `SEW`/`LMUL`,
- advancing pointers with a constant instead of `vl * sizeof(T)`,

- masked loads without initialization when masked-off lanes are later consumed,
- assuming fixed unroll factors match a particular `VLEN`.

5) Microbenchmark the memory path

RVV performance is often dominated by memory. To understand whether you are:

- compute-bound: ALU/FMA throughput dominates,
- memory-bound: load/store bandwidth dominates,
- latency-bound: gathers/scatters or cache misses dominate,

benchmark variants:

- unit-stride vs strided vs indexed,
- different `SEW` and `LMUL`,
- masked vs unmasked.

6) Keep kernels single-purpose and leaf when possible

The best debugging strategy is architectural: keep RVV hot loops:

- short and self-contained,
- free of function calls inside the strip-mined loop,
- explicit about configuration (`vsetvli`) and policies (`ta/tu`, `ma/mu`).

What to remember

- Compilers lower RVV into a small set of recognizable VLA patterns; learn to spot them.
- Auto-vectorization is excellent for regular loops; intrinsics are for control and irregularity.
- ABI boundaries can clobber vector state; configure vectors at the point of use.
- Always verify by inspection: assembly output, linked disassembly, and vectorization diagnostics.

Chapter 10

Performance Characteristics and Pitfalls

10.1 Throughput vs Latency in RVV

RVV performance is shaped by the same two forces as any vector engine:

- **Throughput**: how many vector operations can retire per cycle (steady-state).
- **Latency**: how long a dependency chain takes (critical path).

The VLA model does not change these fundamentals, but it changes how you *write* loops so they scale across implementations.

Throughput-driven kernels (good RVV candidates)

These are loops with abundant independent work and predictable memory:

- vector adds/muls/FMA on large arrays,
- simple stencils with unit-stride loads,

- image/audio kernels with regular access patterns,
- reduction-like loops where the compiler can build wide trees.

Latency-driven kernels (harder to accelerate)

These are loops where each step depends on the previous:

- pointer chasing / linked structures,
- serial prefix algorithms without enough parallelism,
- heavy gathers/scatters with cache-miss dominated latency,
- branchy scalar control that cannot be expressed as masks cleanly.

Example: increasing ILP via unrolling (while staying VLA)

You can raise throughput by keeping multiple independent accumulators. This hides latency without assuming lane count.

```
/* Dot-like accumulation with 2 accumulators to reduce dependency
 ↪  chains.
   a0=x*, a1=y*, a2=n (float32), fa0 = scalar multiplier
*/
.text
.align 2
.globl axpy_2acc_f32
axpy_2acc_f32:
    beqz     a2, .Ldone

    /* Initialize accumulators (conceptual) */
```

```
    vsetvli  t0, zero, e32, m1, ta, ma
    vfmv.v.f v8, fa0              /* keep scalar broadcasted if
     ↪  useful */
    /* For real kernels, accumulators hold partial sums or
     ↪  temporaries. */

.Lloop:
    vsetvli  t0, a2, e32, m1, ta, ma
    vle32.v  v1, (a0)
    vle32.v  v2, (a1)

    /* Independent ops to help hide latency (conceptual scheduling
     ↪  freedom) */
    vfmul.vf v3, v1, fa0
    vfadd.vv v2, v2, v3

    vse32.v  v2, (a1)

    slli     t1, t0, 2
    add      a0, a0, t1
    add      a1, a1, t1
    sub      a2, a2, t0
    bnez     a2, .Lloop
.Ldone:
    ret
```

Practical signals

- If your loop is limited by a chain of dependent operations, RVV helps less unless you restructure it.
- If your loop is limited by independent arithmetic and regular loads, RVV can scale very well.

10.2 Register Pressure and LMUL Trade-offs

`LMUL` changes the effective number of available logical registers:

- bigger `LMUL` can increase per-instruction data width (more lanes per op),
- but it reduces how many independent vector values you can keep live,
- and it increases the chance of spills or forced recomputation.

What register pressure looks like in RVV

- Too many live vectors (inputs, temporaries, accumulators, masks) cause spills or force the compiler to lower unrolling.
- Larger `LMUL` makes each live value "more expensive" because it occupies multiple architectural registers.

Rule of thumb for LMUL selection

- Start with `m1`: best baseline, most flexible.
- Move to `m2/m4` when you are throughput-limited and the kernel has few live vectors.

- Avoid `m8` unless the kernel is extremely simple (few registers) and clearly benefits.
- Consider fractional LMUL (`mf2/mf4/mf8`) when register pressure is high.

Example: LMUL can silently break a "works by accident" register plan

Under `m4`, `v8` occupies `v8--v11`. If you also try to use `v10` as an independent temporary, you overlap.

```
/* Demonstrate overlap hazard: do NOT structure register allocation
 ↪  like this. */
vsetvli  t0, a2, e32, m4, ta, ma

vle32.v  v8, (a0)        /* v8 means v8-v11 */
vle32.v  v10, (a1)       /* overlap with v8 group: invalid plan under
 ↪  m4 */
```

Performance pitfall: spilling vector groups is expensive

Spilling a grouped register means spilling multiple vector registers. This can turn a compute-bound loop into a memory-bound loop.

Practical mitigation

- Reduce live ranges (store early, recompute cheap values, split kernels).
- Reduce unrolling if it triggers spills.
- Use `m1` or fractional LMUL for complex kernels.
- Prefer mask-based control flow over multiple temporaries when possible.

10.3 Memory Bandwidth vs Compute Balance

Most RVV kernels fall into one of two categories:

- **Memory-bound**: performance limited by load/store bandwidth (e.g., simple add, copy).
- **Compute-bound**: performance limited by arithmetic throughput (e.g., heavy FMA per byte).

Arithmetic intensity intuition

A rough way to reason about this without hardware counters:

- If you do only a couple ops per element and you move many bytes, you are likely memory-bound.
- If you do many ops per element per byte loaded, you may become compute-bound.

Example: memory-bound kernel (vector copy)

```
/* Copy u32: limited by bandwidth more than ALU. */
.text
.align 2
.globl copy_u32
copy_u32:
    /* a0=dst*, a1=src*, a2=n */
    beqz    a2, .Ldone
.Lloop:
    vsetvli  t0, a2, e32, m1, ta, ma
    vle32.v  v1, (a1)
    vse32.v  v1, (a0)
```

```
    slli      t1, t0, 2
    add       a0, a0, t1
    add       a1, a1, t1
    sub       a2, a2, t0
    bnez      a2, .Lloop
.Ldone:
    ret
```

Example: more compute per byte (FMA-heavy)

```
/* y[i] = a*x[i] + b*y[i] + c*z[i] (float32), more compute per byte.
 ↪  */
.text
.align 2
.globl tri_fma_f32
tri_fma_f32:
    /* a0=x*, a1=y*, a2=z*, a3=n, fa0=a, fa1=b, fa2=c */
    beqz    a3, .Ldone
.Lloop:
    vsetvli   t0, a3, e32, m1, ta, ma
    vle32.v   v1, (a0)           /* x */
    vle32.v   v2, (a1)           /* y */
    vle32.v   v3, (a2)           /* z */

    vfmul.vf  v4, v1, fa0        /* a*x */
    vfmacc.vf v4, fa1, v2        /* + b*y */
    vfmacc.vf v4, fa2, v3        /* + c*z */

    vse32.v   v4, (a1)
```

```
    slli      t1, t0, 2
    add       a0, a0, t1
    add       a1, a1, t1
    add       a2, a2, t1
    sub       a3, a3, t0
    bnez      a3, .Lloop
.Ldone:
    ret
```

Memory pitfalls specific to RVV usage

- **Strided** and **indexed** accesses reduce effective bandwidth and increase latency.
- Masked stores can reduce bandwidth if the predicate is sparse and prevents write-combining.
- Larger `LMUL` can increase the working set per iteration; if it exceeds cache, performance can drop.

Actionable tuning checklist

- Prefer unit-stride loads/stores.
- Make data layout SoA when possible.
- Block computations to reuse cache lines.
- Use prefetch-like strategies via loop blocking (software structure), not fixed-width tricks.

10.4 When RVV Helps — When It Hurts

When RVV helps

RVV tends to help most when:

- loops are long and regular (amortize `vsetvli` and loop overhead),
- memory is contiguous or predictably strided,
- there is enough independent arithmetic to hide latency,
- tails are frequent (RVV handles them naturally without scalar cleanup),
- code must remain portable across a wide range of hardware widths.

When RVV hurts (or helps less than expected)

RVV may hurt or deliver limited wins when:

- the loop is tiny (configuration overhead dominates),
- the access pattern is random (gather/scatter, cache misses),
- the kernel is register-heavy (high pressure triggers spills),
- the algorithm has strong loop-carried dependencies (latency-bound),
- misalignment or poor layout causes frequent split accesses.

Pitfall: configuration overhead in tiny loops

If `n` is very small, repeated `vsetvli` and loop control can outweigh vector benefits. Fixes:

- handle very small `n` in a scalar or short-vector micro-path,
- use `vsetivli` for constant small blocks,
- fuse tiny loops to increase work per configuration.

Pitfall: assuming larger LMUL is always faster

Larger `LMUL` can:

- increase throughput for simple kernels,
- but reduce scheduling freedom and increase spills for complex kernels.

Always validate with measurement.

Pitfall: masked operations are not free

Masks avoid branches, but they can:

- reduce effective utilization if most lanes are masked off,
- add overhead for predicate computation,
- complicate memory behavior (sparse stores).

If the mask density is extremely low, scalar may be faster.

Practical performance workflow

1. Start with the correct VLA kernel (`m1`, unit-stride).
2. Measure: determine memory-bound vs compute-bound.
3. Adjust `LMUL` and unrolling to balance register pressure and throughput.
4. Prefer layout changes over fancy instruction tricks when memory dominates.
5. Re-measure on multiple implementations (small and large `VLEN`) to validate portability.

What to remember

- Throughput wins require enough independent work; latency-bound code needs restructuring.
- `LMUL` is a throughput lever but increases register pressure; spills are expensive.
- Most simple RVV loops are memory-bound; fix memory layout before chasing ALU tweaks.
- RVV excels at portable tails and scalable performance; it struggles with tiny loops and irregular memory.

Appendices

Appendix A — Minimal RVV Assembly Patterns

Scalar-to-Vector Transition

The fastest way to "enter" RVV correctly is:

1. keep scalar calling convention and scalar loop counters,
2. configure vectors at point-of-use with `vsetvli`,
3. broadcast scalars when needed,
4. never assume a fixed lane count.

Broadcast a scalar integer into a vector

```
/* v1 = (int32)scalar a0 replicated across active lanes */
vsetvli  t0, a1, e32, m1, ta, ma   /* a1 = element count (AVL), t0 =
↪  vl */
vmv.v.x  v1, a0                    /* broadcast scalar into vector */
```

Broadcast a scalar float into a vector

```
/* v1 = (float32)scalar fa0 replicated across active lanes */
```

```
vsetvli   t0, a0, e32, m1, ta, ma
vfmv.v.f  v1, fa0
```

Scalar to vector load: “first chunk”

```
/* Load first chunk from memory into a vector */
vsetvli  t0, a2, e32, m1, ta, ma
vle32.v  v1, (a0)                 /* a0 points to int32/float32 array
 ↪  */
```

Vector to scalar extract (element 0)

```
/* Extract lane 0 to scalar register */
vmv.x.s  a0, v1                   /* integer */
vfmv.f.s fa0, v1                  /* floating-point */
```

Scalar remainder is usually unnecessary

For unit-stride loops, the final iteration naturally runs with a smaller `vl`. You only need scalar code for:

- extremely small `n` (micro-path),
- special alignment prologs when required,
- non-vectorizable corner semantics.

Canonical Vector Loops

Loop template: unit-stride load/compute/store

```
/* Template:
```

```
   a0=in0*, a1=in1*, a2=out*, a3=n (elements), element size depends
    ↪  on vle/vse
*/
beqz    a3, .Ldone
.Lloop:
    vsetvli t0, a3, e32, m1, ta, ma   /* t0 = vl */

    /* Load */
    vle32.v v1, (a0)
    vle32.v v2, (a1)

    /* Compute (example: out = in0 + in1) */
    vadd.vv v3, v1, v2

    /* Store */
    vse32.v v3, (a2)

    /* Advance pointers by vl * sizeof(elem) */
    slli    t1, t0, 2                 /* bytes = vl * 4 for e32 */
    add     a0, a0, t1
    add     a1, a1, t1
    add     a2, a2, t1

    /* Remaining */
    sub     a3, a3, t0
    bnez    a3, .Lloop
.Ldone:
    ret
```

Loop template: scalar broadcast + vector FMA

```
/* y[i] = a*x[i] + y[i], float32
   a0=x*, a1=y*, a2=n, fa0=a
*/
beqz    a2, .Ldone
.Lloop:
    vsetvli  t0, a2, e32, m1, ta, ma
    vle32.v  v1, (a0)
    vle32.v  v2, (a1)
    vfmacc.vf v2, fa0, v1
    vse32.v  v2, (a1)

    slli     t1, t0, 2
    add      a0, a0, t1
    add      a1, a1, t1
    sub      a2, a2, t0
    bnez     a2, .Lloop
.Ldone:
    ret
```

Loop template: reduction (sum) using `vredsum`

```
/* Sum int32 array into a0 (conceptual pattern)
   a0=ptr*, a1=n
*/
beqz      a1, .Ldone

vsetvli   t0, zero, e32, m1, ta, ma
vmv.v.i   v0, 0                    /* vector accumulator */
```

```
.Lloop:
    vsetvli  t0, a1, e32, m1, ta, ma
    vle32.v  v1, (a0)
    vadd.vv  v0, v0, v1

    slli     t1, t0, 2
    add      a0, a0, t1
    sub      a1, a1, t0
    bnez     a1, .Lloop

vmv.v.i     v2, 0                   /* seed */
vredsum.vs  v3, v0, v2              /* result in v3[0] */
vmv.x.s     a0, v3
.Ldone:
ret
```

Correctness checklist for every loop

- `vsetvli` is inside the loop (or `vsetivli` for fixed micro-blocks).
- Pointer increments use the returned `vl` and correct element size.
- No fixed-lane assumptions or remainder code unless intentionally added.

Mask-Driven Control Examples

Mask-driven control replaces branches with:

- compare → mask,
- masked arithmetic or `vmerge`,

- masked loads/stores for safe conditional memory.

Branchless `if/else` via `vmerge`

```
/* y[i] = (x[i] < 0) ? a : b   int32
   a0=x*, a1=y*, a2=n, a3=a, a4=b
*/
beqz     a2, .Ldone
.Lloop:
    vsetvli    t0, a2, e32, m1, tu, mu
    vle32.v    v1, (a0)           /* x */

    vmv.v.x    v2, a3             /* a broadcast */
    vmv.v.x    v3, a4             /* b broadcast */

    vmslt.vx   v0, v1, zero       /* mask: x < 0 */
    vmerge.vvm v4, v2, v3, v0.t   /* select */

    vse32.v    v4, (a1)

    slli       t1, t0, 2
    add        a0, a0, t1
    add        a1, a1, t1
    sub        a2, a2, t0
    bnez       a2, .Lloop
.Ldone:
    ret
```

Masked update (in-place) without disturbing other lanes

```
/* if (x[i] > 0) x[i] += k, else unchanged
   a0=x*, a1=n, a2=k
*/
beqz     a1, .Ldone
.Lloop:
    vsetvli   t0, a1, e32, m1, tu, mu
    vle32.v   v1, (a0)

    vmsgt.vx  v0, v1, zero       /* mask: x > 0 */
    vadd.vx   v1, v1, a2, v0.t   /* predicated add */

    vse32.v   v1, (a0)

    slli      t1, t0, 2
    add       a0, a0, t1
    sub       a1, a1, t0
    bnez      a1, .Lloop
.Ldone:
    ret
```

Bounds-checked scatter (memory-safe) using a mask

```
/* dst[idx[i]] = val[i] only when idx[i] < limit (no out-of-bounds
↪  stores)
   a0=dst_base*, a1=idx_u32*, a2=val_i32*, a3=n, a4=limit_elems
*/
beqz      a3, .Ldone
.Lloop:
```

```
    vsetvli    t0, a3, e32, m1, tu, mu
    vle32.v    v1, (a1)           /* idx (elements) */
    vle32.v    v2, (a2)           /* values */

    vmsltu.vx  v0, v1, a4         /* mask: idx < limit */
    vsll.vi    v3, v1, 2          /* offsets_bytes = idx << 2 */

    vsuxei32.v v2, (a0), v3, v0.t /* masked scatter */

    slli       t1, t0, 2
    add        a1, a1, t1
    add        a2, a2, t1
    sub        a3, a3, t0
    bnez       a3, .Lloop
.Ldone:
    ret
```

Mask load/store (persist predicate decisions)

```
/* Store a computed mask to memory, then reload and reuse it.
   a0=mask_mem*, a1=x*, a2=n
*/
vsetvli   t0, a2, e32, m1, tu, mu
vle32.v   v1, (a1)
vmsgt.vx  v0, v1, zero        /* mask: x > 0 */
vsm.v     v0, (a0)            /* store mask bits */

vlm.v     v0, (a0)            /* reload mask bits */
vadd.vx   v1, v1, 1, v0.t     /* increment only where x > 0 */
```

Minimum safety rules for mask-driven code

- Masked memory ops must be used for bounds-checked gathers/scatters.
- If masked-off lanes will be observed later, initialize destination and use `tu,mu`.
- Do not assume mask density; extremely sparse masks may favor scalar paths.

Appendix B — RVV vs Traditional SIMD

RVV vs AVX-512

AVX-512 is a **fixed-width SIMD** model:

- The vector width is architecturally fixed per ISA level (e.g., 512-bit ZMM).
- Code is often written around a known lane count (e.g., 16 lanes of FP32 in 512-bit).
- Portability across widths is typically handled by:
  - multiple code paths (SSE/AVX2/AVX-512),
  - runtime dispatch,
  - scalar tails and remainder loops.

RVV is **vector-length-agnostic (VLA)**:

- The lane count is not fixed; it is determined at runtime via `vl`.
- Correct code is naturally tail-safe and scales with future hardware.
- One kernel can run on many widths without recompilation (subject to ISA subset support).

Conceptual difference: iteration structure

```
/* Fixed-width SIMD thinking (conceptual, do not do this for RVV):
   process 16 floats per iteration, then scalar remainder.
*/
 /* for (i=0; i+16<=n; i+=16) { ... } */
 /* for (; i<n; ++i) { ... } */

/* RVV VLA thinking:
   vl = min(remaining, VLMAX) each iteration; no separate remainder
    ↪  loop.
*/
.Lloop:
    vsetvli  t0, a2, e32, m1, ta, ma   /* t0 = vl for remaining a2 */
    /* compute on vl lanes */
    sub      a2, a2, t0
    bnez     a2, .Lloop
```

Mask model comparison

- AVX-512 uses dedicated mask registers (`k0--k7`) for predication and blends.
- RVV uses vector masks as first-class values (`v0` convention) that are produced by compares and used to predicate most ops.

Practical performance implications

- AVX-512: peak throughput can be extremely high, but tuning often becomes width-specific.
- RVV: peak throughput depends on implementation width, but the same kernel is forward scalable and tail-handling is structurally efficient.

Where AVX-512 can be simpler

- When you can hard-code lane count and tightly schedule around a fixed width.
- When the deployment hardware is known and uniform.

Where RVV wins structurally

- Mixed deployments with unknown vector widths.
- Long-lived binaries meant to scale with newer cores.
- Kernels where scalar tail handling is frequent and costly.

RVV vs ARM SVE

ARM SVE is also **vector-length-agnostic**:

- The architectural vector length is implementation-defined.
- Code uses predicates and VLA-style loops to avoid fixed-width assumptions.

So RVV and SVE share the core VLA philosophy, but they differ in **how state and configuration are expressed**.

Key conceptual similarities

- Both encourage strip-mined loops.
- Both use predication to avoid scalar tails.
- Both aim for forward scalability as vector width grows.

Key conceptual differences (programmer-facing)

- RVV uses explicit configuration via `vsetvli`/`vtype` to choose `SEW` and `LMUL`.
- SVE uses a different model where element size selection is encoded in the instruction forms and predicates drive active lanes; scalable vectors are part of the architectural model.

A shared idea: predicated last-iteration

```
/* RVV idiom: last iteration handled by vl, no scalar tail. */
vsetvli  t0, a2, e32, m1, ta, ma
/* ... work on [0..vl) ... */
```

Practical consequence

Because both are VLA, the **algorithmic structure** you write (strip-mining + predication) transfers well between RVV and SVE, even though the instruction sets are different.

Where SVE differs in daily practice

- SVE often encourages a predicate-driven style (explicit per-iteration predicate for "remaining lanes").
- RVV often expresses the active count as `vl` and then optionally uses masks for data-dependent control.

Portability and Maintenance Trade-offs

Portability axes

When comparing RVV to traditional SIMD, consider three portability dimensions:

1. **Width portability**: does the same binary scale across implementations with different vector widths?
2. **ISA portability**: can the same source support multiple architectures (x86, ARM, RISC-V) with minimal duplication?
3. **Compiler portability**: does the code survive toolchain differences (GCC vs Clang, version changes)?

RVV strength: width portability by construction

- RVV VLA loops are width-portable: no fixed-lane assumptions, `vl`-driven pointer math, tail-safe structure.
- This reduces the need for multiple width-specific kernels (the classic SSE/AVX/AVX-512 stack).

Traditional SIMD strength: mature ecosystems

- x86 SIMD has long-established tooling, profilers, and optimization folklore.
- Many libraries and compilers have extensive x86 tuning knowledge.

Maintenance reality: one kernel vs many kernels

A typical fixed-width SIMD maintenance pattern:

- scalar fallback,
- SSE/AVX2 kernel,
- AVX-512 kernel,
- runtime dispatch + testing matrix.

A typical RVV maintenance pattern:

- one VLA kernel for all RVV-capable widths,
- optional micro-paths for very small `n` or special alignment needs,
- optional dispatch only across **features** (not widths), e.g., presence/absence of certain subsets.

The trade-off

- RVV reduces width-specialization burden but demands discipline: always compute from `vl`, treat masks/tails carefully, and avoid width-based mental models.
- Fixed-width SIMD can yield excellent peak results on known hardware, but tends to accumulate code paths and testing cost as widths and ISAs grow.

A practical decision checklist

- If you ship to a single known x86 fleet: fixed-width kernels can be justified.
- If you ship broadly and want long-lived binaries: VLA (RVV/SVE-style) reduces width-specific maintenance.
- If your workload is irregular (gather/scatter heavy): performance may be dominated by memory latency on all ISAs; focus on algorithmic locality first.

Minimal portable kernel principle

Regardless of ISA, the most maintainable high-performance code tends to be:

- short, single-purpose kernels,
- explicit about assumptions (alignment, aliasing, data layout),

- validated by inspection (generated assembly) and microbenchmarks,
- backed by scalar reference tests.

Appendix C — Practical Rules of Thumb

Choosing SEW and LMUL

SEW: choose the algorithm's natural element width first

- Use the element width that matches your data format: `e8/e16/e32/e64`.
- If the algorithm needs higher precision or wider intermediates, **widen for compute**, then narrow/pack explicitly.
- For floating-point, prefer:
  - FP32 for general numeric kernels,
  - FP16 for bandwidth/throughput when accuracy tolerates it, often with FP32 accumulation,
  - FP64 only when required by accuracy or dynamic range.

LMUL: start small, grow only when you measure a win

- Default: `m1`. It maximizes scheduling freedom and minimizes spills.
- Increase to `m2/m4` only for **simple kernels** with few live vectors.
- Avoid `m8` unless the loop is extremely simple (copy, add) and clearly benefits.
- Use fractional LMUL (`mf2/mf4/mf8`) to reduce register pressure in complex kernels.

Live-vector budgeting (quick mental model)

Count how many vector values you keep live at once:

- inputs (1–3),
- outputs (1),
- temporaries (1–4),
- accumulators (1–N),
- masks (often 1),
- constants (broadcasts).

If this number is large, **stay at `m1` or use fractional LMUL**. If it is small, try `m2`/`m4`.

Example: safe default for most kernels

```
/* Most portable baseline: e32, m1, tails/masks agnostic unless you
 ↪  need preservation. */
vsetvli  t0, a2, e32, m1, ta, ma
```

Example: widening compute implies reconfiguration

```
/* Load int16, compute in int32: expect a second vsetvli. */
vsetvli   t0, a2, e16, m1, ta, ma
vle16.v   v1, (a0)
vsext.vf2 v2, v1

vsetvli   t0, t0, e32, m1, ta, ma
/* compute/store at e32 */
```

Common trap: LMUL overlap

Under larger LMUL, each logical register occupies a group:

```
/* Under m4: v8 aliases v8-v11. Using v10 independently overlaps (bad
 ↪  plan). */
vsetvli  t0, a2, e32, m4, ta, ma
vle32.v  v8, (a0)
vle32.v  v10, (a1)     /* overlaps v8 group */
```

Writing Future-Proof RVV Code

Future-proof means: correct on any `VLEN`, and robust across toolchains and microarchitectures.

Rule 1: strip-mine everything

Every vector loop must be `vl`-driven:

```
/* Always: vl = min(remaining, VLMAX) and pointers advance by vl. */
beqz    a2, .Ldone
.Lloop:
    vsetvli t0, a2, e32, m1, ta, ma
    /* work on [0..vl) */
    slli    t1, t0, 2
    add     a0, a0, t1
    sub     a2, a2, t0
    bnez    a2, .Lloop
.Ldone:
    ret
```

Rule 2: never assume `vl` stays constant

If you change `SEW` or `LMUL`, `vl` can change. Recompute it with `vsetvli` and use the returned value.

Rule 3: be explicit about tail and mask policies

Use policies as part of your correctness contract:

- `ta,ma`: fastest when inactive lanes are never observed.
- `tu,mu`: required when inactive lanes must preserve values across masked sequences or will be read later.

```
/* Correct when masked-off lanes will be observed later: tu,mu +
 ↪  init. */
vsetvli  t0, a2, e32, m1, tu, mu
vmv.v.i  v2, 0
/* masked load/compute into v2 */
```

Rule 4: mask every potentially unsafe memory access

For indexed gather/scatter, **always** compute a bounds mask:

```
/* Store dst[idx]=val only when idx < limit. */
vsetvli    t0, a3, e32, m1, tu, mu
vle32.v    v1, (a1)          /* idx */
vle32.v    v2, (a2)          /* val */
vmsltu.vx  v0, v1, a4        /* mask: idx < limit */
vsll.vi    v3, v1, 2         /* byte offsets */
vsuxei32.v v2, (a0), v3, v0.t
```

Rule 5: avoid calls inside vector loops

Calls can clobber vector state. Keep hot kernels leaf when possible, or reconfigure vectors after calls.

Rule 6: prefer data layout fixes over instruction tricks

If performance is limited by memory:

- convert AoS to SoA,
- block loops to increase locality,
- reduce gathers/scatters,
- make unit-stride the common case.

Debugging Common Mistakes

Mistake 1: wrong pointer increments

Symptom: correct on small sizes, corrupts on large sizes or different hardware. Fix: pointer increments must use the returned `vl` and correct byte scaling.

```
/* Correct pointer math for e16 */
vsetvli t0, a2, e16, m1, ta, ma
slli    t1, t0, 1      /* bytes = vl * 2 */
add     a0, a0, t1
```

Mistake 2: consuming inactive lanes after masked ops

Symptom: nondeterministic results that change with `VLEN`, compiler version, or optimization. Fix: if masked-off lanes are later read, use `tu,mu` and initialize destination.

```
/* Safe masked gather with zero-fill for masked-off lanes */
vsetvli   t0, a3, e32, m1, tu, mu
vmv.v.i   v2, 0
vle32.v   v1, (a1)
vmsltu.vx v0, v1, a4
vsll.vi   v3, v1, 2
vluxei32.v v2, (a0), v3, v0.t
```

Mistake 3: stale `vl` after reconfiguration

Symptom: pointer math mismatches when moving between phases (e16 loads, e32 compute). Fix: treat every `vsetvli` return value as authoritative for that phase.

```
/* Correct: capture new vl after switching to e32 (do not reuse old
 ↪  count blindly). */
vsetvli t0, a2, e16, m1, ta, ma
/* ... */
vsetvli t0, t0, e32, m1, ta, ma
/* pointers/loop control use this new t0 */
```

Mistake 4: LMUL overlap and accidental register aliasing

Symptom: assembler errors or subtle clobbering in hand-written assembly. Fix: allocate registers as groups under the chosen LMUL; never treat overlapped numbers as independent.

Mistake 5: assuming strided/indexed access will be fast

Symptom: RVV kernel is slower than scalar. Fix: measure unit-stride baseline; if irregular memory dominates, fix locality first.

Fast verification routine (do this every time)

- Inspect assembly: is there a strip-mined `vsetvli` loop?
- Check pointer increments: do they scale by `vl * sizeof(T)`?
- Check masks: are unsafe accesses predicated and are inactive lanes handled by policy/initialization?
- Benchmark: unit-stride vs strided vs indexed variants to locate the true bottleneck.

Appendix D — Conceptual Cross-References

RISC-V Base ISA Interaction

RVV is not a separate "mode"; it is an extension that integrates with the base RISC-V execution model. The practical implications for programmers:

Scalar registers still drive control

- Loop counters, pointers, and bounds checks are usually scalar (`x` registers).
- Vector instructions consume scalar registers for:
  - AVL (application vector length) into `vsetvli`,
  - base addresses for loads/stores,
  - scalar operands in `.vx` and `.vf` forms (vector-scalar).

Vector state is part of architectural state

- `vl` and `vtype` define how vector registers are interpreted.

- Robust code configures vectors at the point of use; do not assume a caller left a useful configuration.

```
/* Base ISA + RVV typical mix: scalar loop + RVV inner body. */
beqz    a2, .Ldone
.Lloop:
    vsetvli  t0, a2, e32, m1, ta, ma
    /* vector body */
    sub      a2, a2, t0
    bnez     a2, .Lloop
.Ldone:
    ret
```

Addressing and pointer math remain scalar

All pointer updates are scalar arithmetic derived from returned `vl`:

```
/* Advance float32 pointer by vl elements */
vsetvli  t0, a2, e32, m1, ta, ma
slli     t1, t0, 2        /* bytes = vl*4 */
add      a0, a0, t1
```

Exceptions and faults follow the same model

- Vector loads/stores can raise the same classes of faults as scalar memory ops.
- Masked memory ops prevent accesses for masked-off lanes; this is a primary safety mechanism for indexed access.
- Fault-only-first loads (`vleff.v`) provide a controlled partial-load mechanism for certain patterns.

```
/* Bounds mask before indexed store: base ISA ensures safe control +
 ↪  RVV safe memory traffic. */
vsetvli    t0, a3, e32, m1, tu, mu
vle32.v    v1, (a1)          /* idx */
vle32.v    v2, (a2)          /* val */
vmsltu.vx  v0, v1, a4        /* idx < limit */
vsll.vi    v3, v1, 2         /* offsets */
vsuxei32.v v2, (a0), v3, v0.t
```

CSR and privilege interaction (conceptual)

- The OS saves/restores vector state according to its ABI and context-switch policy.
- User code should not depend on vector state persisting across calls or traps; always reconfigure as needed.

Memory Model Considerations

Vectorization does not weaken the memory model; it changes how many memory operations occur and how they may be observed. Correctness in concurrent code depends on ordering rules and atomicity guarantees.

What to assume in general

- Ordinary vector loads/stores are **non-atomic** at the multi-element level.
- A single vector store updates many elements; other threads may observe partial progress unless synchronization is used.
- Masked stores can make the visibility pattern more irregular.

Rule 1: do not use plain RVV loads/stores for shared synchronization

If data is shared across threads and requires ordering, use the platform's atomic primitives and fences (scalar ISA atomics and memory-ordering constructs). RVV is primarily for data-parallel computation on properly synchronized regions.

Rule 2: partition data to avoid false sharing

Even without atomics, performance and correctness improve when each thread owns disjoint ranges. RVV amplifies this:

- wider stores touch more bytes per iteration,
- cache-line ping-pong can dominate if two threads write adjacent elements.

Example: avoid overlap by chunking

```
/* Conceptual: thread t processes [start, end) disjoint range. */
void worker(float* y, const float* x, unsigned long start, unsigned long
↪  end)
{
    for (unsigned long i = start; i < end; ++i)
        y[i] += x[i];
}
```

Rule 3: reductions require explicit parallel structure

Reductions are naturally associative for integers (modulo overflow), but concurrency still needs explicit design:

- per-thread partial sums,
- then a synchronized combine step.

```
/* Conceptual parallel reduction structure. */
double total = 0;
#pragma parallel
{
    double local = 0;
    /* compute local on disjoint chunk */
    #pragma critical
    total += local;
}
```

Practical warning for floating-point

Parallel reductions are order-dependent for FP. Different thread scheduling and different `vl` trees can change last bits. If reproducibility matters:

- use deterministic reduction trees,
- or accumulate in higher precision,
- or use compensated summation strategies.

Relationship to Parallel Programming Models

RVV expresses **data parallelism within a core**. Parallel programming models express **task/data parallelism across cores**. They compose naturally when you follow a simple hierarchy:

The hierarchy

- **Thread-level parallelism**: split the global workload into chunks per thread/core.
- **Vector-level parallelism**: within each chunk, use RVV strip-mined loops.

Model mapping (conceptual)

- OpenMP/TBB/pthreads: distribute outer loop iterations.
- RVV: accelerates the inner loop over contiguous elements inside each thread's chunk.

Example: outer parallel loop + inner RVV kernel (conceptual)

```
/* Outer parallelism + inner vectorization. */
#pragma parallel for
for (unsigned long block = 0; block < N; block += BLOCK)
{
    unsigned long end = (block + BLOCK < N) ? (block + BLOCK) : N;
    /* call an RVV kernel that processes [block, end) */
    rvv_kernel(y + block, x + block, end - block);
}
```

Practical composition rules

- Keep RVV kernels **leaf** (no calls in the hot loop) when possible.
- Use **disjoint ranges** per thread to avoid false sharing.
- Synchronize between phases, not between elements.
- Prefer unit-stride inside each thread; the thread partition should preserve locality.

Where RVV is not a substitute

RVV does not replace:

- atomic operations for synchronization,
- fences for ordering between threads,

- locks/barriers for coordination.

It accelerates the computation performed *between* synchronization points.

A final mental model

- RVV: SIMD lanes inside one core, controlled by `vl` and masks.
- Parallel models: multiple cores, controlled by scheduling and synchronization.
- Correct programs use both: synchronize at coarse granularity, compute with RVV at fine granularity.

References

Primary Source Map

RISC-V Vector Architecture (Conceptual)

- **RISC-V Vector Extension (V) Specification** (RVV): definition of `vl`, `vtype`, `SEW`, `LMUL`, masking, tail/mask policies, vector memory ops (unit-stride/strided/indexed), reductions, permutes, and privileged/CSR interactions relevant to V.
- **RISC-V Unprivileged ISA Specification** (RV32/RV64): base integer ISA, floating-point ISA, instruction encodings, and the architectural ground rules RVV builds upon.
- **RISC-V Privileged Architecture Specification**: trap/exception model, CSR conventions, context switching implications, and OS-visible state management relevant to vector enablement and preservation.
- **RISC-V Memory Model / RVWMO documentation**: ordering guarantees for ordinary loads/stores, fences, and the concurrency rules that define what is (and is not) safe in multi-threaded vectorized programs.
- **Vector ISA compatibility notes**: guidance on VLA loop structure, the meaning of `VLMAX`, and why correct programs must be `vl`-driven rather than fixed-width.

ABI and Toolchain Specifications

- **RISC-V psABI (Procedure Call Standard)**: calling convention, register classification, stack rules, ELF ABI details, and the toolchain contract for interoperable binaries.
- **RISC-V ELF psABI / Toolchain ABI supplements**: object format, relocation rules, and platform ABI profiles (e.g., LP64, LP64D) used in real deployments.
- **GNU Binutils (as, ld) RISC-V documentation**: assembler syntax for RVV mnemonics, encoding options, disassembly conventions, and relocation/link behavior.
- **GCC RISC-V port documentation**: `-march`/`-mabi` conventions, vector codegen behavior, vectorization reports, and tuning flags.
- **LLVM/Clang RISC-V backend documentation**: RVV code generation, vectorization remarks, intrinsic mappings, and disassembly/MC layer behavior.

Minimal toolchain sanity patterns (conceptual)

```
/* Build baseline RVV objects with consistent ISA + ABI across
 ↪  translation units.
   -march should include 'v' (and typically 'zve*' / floating subsets
    ↪  as required by your target profile).
   -mabi must match the platform ABI (e.g., lp64d for RV64 with
    ↪  double-precision FP ABI).

   Example intent:
     compile all RVV objects with the same -march/-mabi
     avoid mixing objects with and without vector ISA unless you use
      ↪  explicit dispatch
*/
```

Academic and Industry Vector Research

- **Vector processor foundations**: classic vector architecture literature (strip-mining, vector-length agnostic programming, memory bandwidth vs compute balance, gather/scatter costs, and reduction trees).
- **SIMD vs VLA research**: comparative work that contrasts fixed-width SIMD (SSE/AVX/AVX-512, NEON) with scalable vector models (VLA), focusing on portability, maintenance, and forward scalability.
- **Predication and masking research**: work on predicate registers, mask-driven control flow, branch-avoidance, and the performance trade-offs of predication density.
- **Memory-system and locality research**: cache-line utilization, prefetching effects, TLB behavior, strided/indexed access penalties, and data-layout transformations (AoS$\rightarrow$SoA, blocking/tiling).
- **Parallel programming + vectorization**: studies and guidance on composing thread-level parallelism with vector-level parallelism, including reduction reproducibility and false-sharing avoidance.
- **RISC-V vector implementation case studies**: public microarchitecture talks/papers describing vector pipelines, register-file organization, LMUL implications, and practical throughput/latency bottlenecks in real RVV cores.