

From SIMD to RISC-V Vector Assembly

Understanding Data-Level Parallelism

Salman Zaffar

Computer Architecture

March 10, 2026

Why Vector Processing?

Large matrices in...

- Multimedia & graphics
- Machine learning and AI
- Scientific computing
- Signal processing

Why Vector Processing?

Large matrices in...

- Multimedia & graphics
- Machine learning and AI
- Scientific computing
- Signal processing

Scalar processors handle one data item at a time. We need faster ways to process large datasets.

Data Parallelism \Rightarrow SIMD \Rightarrow Vector

Type	Example	Level
Instruction-Level	Pipelining	Within CPU
Thread-Level	Multicore	OS
Data-Level	SIMD/Vector	CPU

What is Data-Level Parallelism?

Perform the **same operation** on multiple data elements simultaneously.

- Array addition
- Image filtering
- Matrix multiplication

RISC-V Scalar: Adding Two Vectors (10 floats)

```
# a0 = &A, a1 = &B, a2 = &C
li    t0, 0
li    t1, 10
loop:
    bge    t0, t1, done
    slli   t2, t0, 2
    add    t3, a0, t2
    flw    fa0, 0(t3)
    add    t3, a1, t2
    flw    fa1, 0(t3)
    fadd.s fa2, fa0, fa1
    add    t3, a2, t2
    fsw    fa2, 0(t3)
    addi   t0, t0, 1
    j      loop
done:
```

Register Map

a0	base of A
a1	base of B
a2	base of C
t0	loop counter i
t1	loop limit (10)
t2	byte offset $i \times 4$
t3	scratch address
fa0/fa1	loaded floats
fa2	sum result

Per-iteration cost

flw $\times 2$	load A[i], B[i]
fadd.s $\times 1$	compute sum
fsw $\times 1$	store C[i]

10 iterations total

Compare: RVV `vfadd.vv`
computes all 10 sums in **1 instruction**.

SIMD&Vector: CPU & GPU Architectures

Intel & AMD (x86/x64)

Extension	Description
MMX	Early 64-bit SIMD for multimedia (Intel)
SSE	128-bit streaming SIMD, widely supported
AVX / AVX2	256-bit advanced vector extensions
AVX-512	512-bit vectors on modern CPUs

ARM (Mobile / Embedded / Server)

Extension	Description
NEON	128-bit SIMD for media on Cortex-A processors
SVE / SVE2	Scalable vector lengths; used in Neoverse HPC cores

SIMD&Vector: Historical & Specialized Architectures

Historical / Scientific Vector Processors

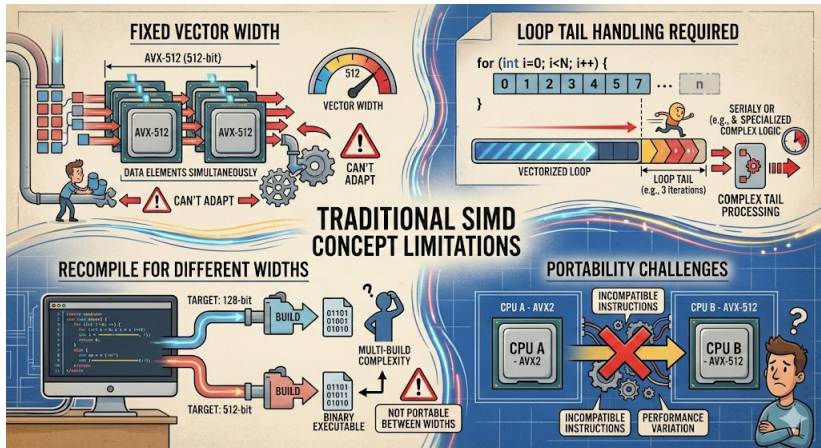
Architecture	Description
Cray-1 / Cray-2	Pioneered vector processing; 1970s–80s supercomputers with massive pipeline registers
NEC SX Series	NEC's high-performance vector supercomputers

Specialized & Other Architectures

Architecture	Description
RISC-V Vector	Open-source; flexible long-vector SIMD op.
AltiVec	PowerPC SIMD; in Apple G4/G5 & IBM Power
Cell Broadband Engine	PlayStation 3 processor: PowerPC core

Limitations of Traditional SIMD

- Fixed vector width
- Loop tail handling required
- Recompile for different widths
- Portability challenges



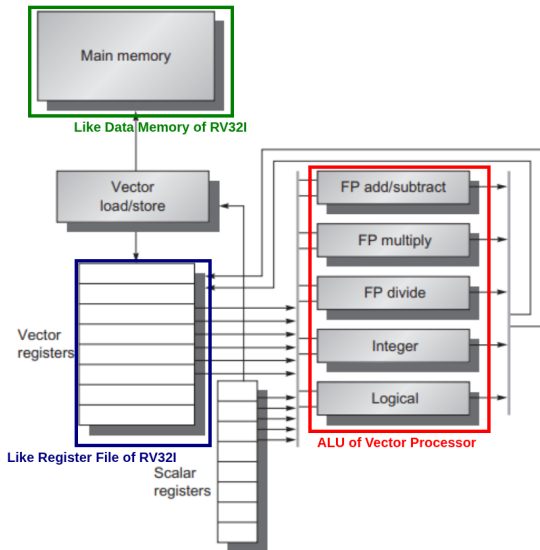
RVV vs Traditional SIMD

Feature	Traditional SIMD	RVV
Vector width	Fixed	Variable
Portability	Limited	High
Tail handling	Manual	Automatic
Scalability	Limited	Excellent

RVV Programming Model

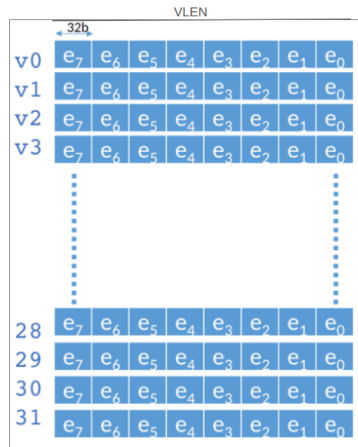
Vector Registers

- v0 – v31
- store multiple data elements
- **Vector Length (VLEN)**
Size of vector registers in terms of bits (128, 256, or 512 bits). Fixed by the chip designer
- **Selected Element Width (SEW)** The size of each element to be processed (8, 16, 32, or 64 bits)



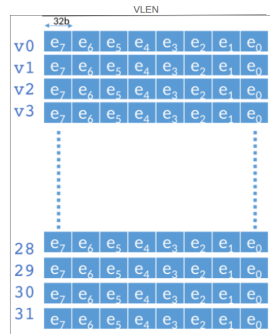
RVV Programming Model

- **Length Multiplier (LMUL)** Grouping of multiple hardware registers together to act as one large register (1, 2, 4, 8 or 1/2, 1/4)
- **Application Vector Length (AVL)**
The total number of elements an application uses in a vector
- **Vector Length (VL)** The number of elements processed by the hardware on runtime or **size of vector registers in terms of elements**



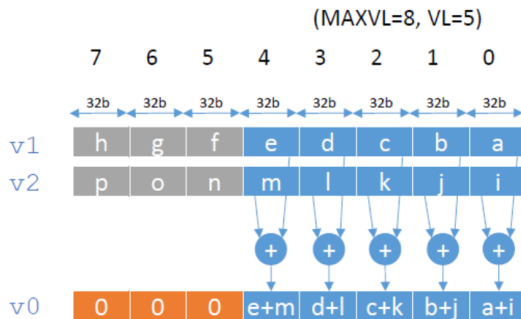
Setting Vector Length-An Example

- Vector Register Length **VLEN=256 bits**
- Set Element Width **SEW=32 bits**
- **LMUL=1** for our purposes
- Configuration of hardware capacity
VLMAX=LMUL x VLEN/SEW
- Vector Length
VL=min(**AVL**, **VLMAX**)
- Application Vector Length AVL (user required in SW)



Setting Vector Length-An Example

- VL = $\min(\text{AVL}, \text{VLMAX}) = \min(5, 8) = 5$



Setting Vector Length

Three Instructions

- `vsetvli rd, rs1, vtypei` (fixed)

`rd` = new `vl` in HW, `rs1` = AVL, `vtypei` = new `vtype` setting

- `vsetivli rd, uimm, vtypei`

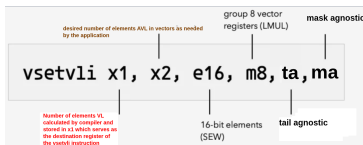
`rd` = new `vl` in HW, `uimm` = AVL, `vtypei` = new `vtype` setting

- `vsetvl rd, rs1, rs2` (dynamic)

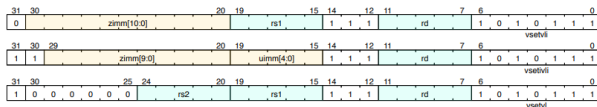
`rd` = new `vl` in HW, `rs1` = AVL, `rs2` = new `vtype` val

Configures:

- element width
- vector length
- hardware capacity



Formats for Vector Configuration Instructions under OP-V major opcode



Tail Agnosticism:

- Say we set $vl = 5$, meaning we only want to process 5 elements. Elements 0–4 are "active," elements 5–7 are the tail.
- Register before: [10, 20, 30, 40, 50, 60, 70, 80]
idx 0 1 2 3 4 5 6 7
—tail—
- `vadd v1, v2, v3` (add two vectors, store in v1), with $vl=5$
- **Tail Undisturbed** Result: [a, b, c, d, e, 60, 70, 80]
- **Tail Agnostic**
 - Result: [a, b, c, d, e, ff, ff, ff] ← hardware filled with 1s
 - Result: [a, b, c, d, e, 60, 70, 80] ← hardware left them (also valid)
 - Result: [a, b, c, d, e, ?, ?, ?] ← anything goes

Tail and Mask

vl = 8 (all 8 elements active), but mask that only enables some lanes:

```
Mask:   [1, 0, 1, 0, 1, 1, 0, 1]
         ^   ^   ^   ^   ^
         ^   ^       ^   ← active (mask=1)
         ^   ^       ^   ← inactive (mask=0)
```

vl = 8 (all 8 elements active), but we have a mask that only enables some lanes:

Mask-undisturbed

Before: [10, 20, 30, 40, 50, 60, 70, 80]

After: [a, 20, c, 40, e, f, 70, h]

 ^ ^ ^
inactive elements preserved

Mask-agnostic

```
After:  [ a, ff,  c, ff,  e, f, ff,  h]   ← hardware wrote 1s
        or  [ a, 20,  c, 40,  e, f, 70,  h]   ← hardware left them (also valid)
        or  [ a,  ?,  c,  ?,  e, f,  ?,  h]   ← anything goes
```

A Coding Example

- 1 Background
- 2 Data & Code
- 3 Instruction Breakdown
- 4 Step-by-Step Execution
- 5 Results & Summary

What is RISC-V Vector Extension (RVV)?

Key Idea: SIMD Processing

Instead of adding one pair of numbers at a time, RVV adds **multiple pairs simultaneously** using vector registers.

.data Section

```
.section .data  
  
x: .word 10,20,30,40,50,60  
y: .word 5,15,25,35,45,55  
z: .word 0, 0, 0, 0, 0, 0  
n: .word 6
```

iteration 1 iteration 2

Advantages:

- Fewer loop iterations
- Higher throughput
- Hardware decides vector length (flexible)
- Same code works on different hardware widths

Scalar vs Vector

Scalar (1 datum at a time):

$z[0] = x[0] + y[0]$

$z[1] = x[1] + y[1]$

...6 operations

Vector (4 data at a time):

$v2 = v0 + v1$

1 vector instruction

iterated twice!

The C Function We Are Implementing

C Code

```
void vvaddint32(  
    size_t n,  
    const int* x,  
    const int* y,  
    int* z)  
{  
    for (size_t i = 0; i < n;  
        z[i] = x[i] + y[i];  
    }  
}
```

Register Mapping

Register	Meaning
a0	n (element count)
a1	pointer to x
a2	pointer to y
a3	pointer to z
t0	vector length

Our Example

n = 6, VL = 4 per iteration

Data Section — Input Arrays

.data Section

```
.section .data
```

```
x: .word 10,20,30,40,50,60
y: .word 5,15,25,35,45,55
z: .word 0, 0, 0, 0, 0, 0
n: .word 6
```

Memory Layout

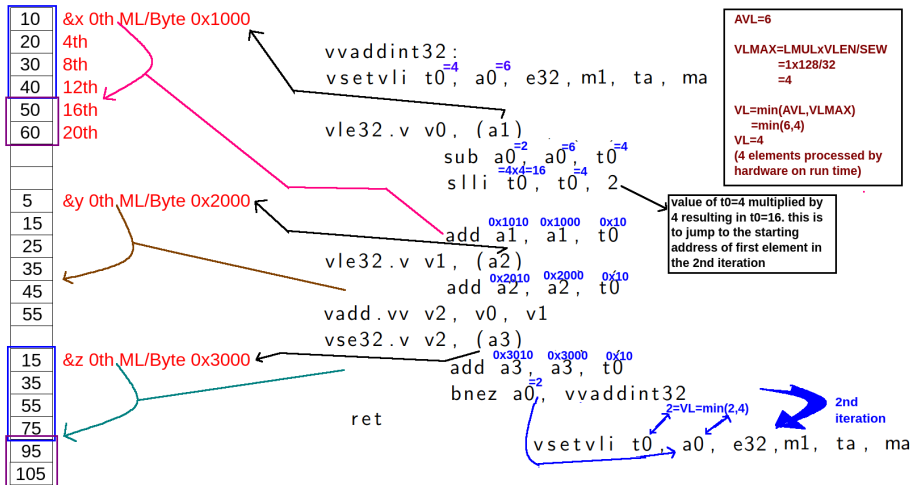
Label	Values	Bytes
x	10,20,30,40,50,60	24
y	5,15,25,35,45,55	24
z	0,0,0,0,0,0	24
n	6	4

Each `.word` = **4 bytes** (32-bit integer)

The Full Assembly Routine

```
vvaddint32:
    vsetvli t0, a0, e32, # Set vector length,
        m1, ta, ma        #32-bit elements
    vle32.v v0, (a1)        # Load chunk of x into v0
    sub a0, a0, t0          # Decrement remaining count
    slli t0, t0, 2          # t0 = t0 * 4
                           #(convert to byte offset)
    add a1, a1, t0          # Advance x pointer
    vle32.v v1, (a2)        # Load chunk of y into v1
    add a2, a2, t0          # Advance y pointer
    vadd.vv v2, v0, v1      # Element-wise add: v2 = v0 + v1
    vse32.v v2, (a3)        # Store result chunk to z
    add a3, a3, t0          # Advance z pointer
    bnez a0, vvaddint32    # Loop if elements remain
    ret                    # Return
```

Code Analysis



Initial State Before Loop

Input Arrays

Index	x[i]	y[i]
0	10	5
1	20	15
2	30	25
3	40	35
4	50	45
5	60	55

Register State

Register	Value
a0	6
a1	&x[0]
a2	&y[0]
a3	&z[0]

Hardware VLEN

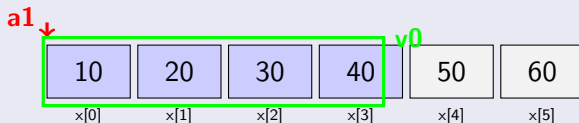
This simulation assumes the hardware can process **4 elements per iteration**

Iteration 1 — Step by Step (Part 1)

```
vsetvli t0, a0, e32, ta, ma
```

$a0=6$, hardware $\text{max}=4 \Rightarrow \mathbf{t0 = 4}$

```
vle32.v v0, (a1) — Load first 4 elements of x
```



$v0 = [10, 20, 30, 40]$

```
sub a0, a0, t0
```

$a0 = 6 - 4 = 2$ (2 elements remaining)

Iteration 1 — Step by Step (Part 2)

`slli t0, t0, 2` — Convert count to bytes

$t0 = 4 \times 4 = 16$ bytes (4 elements \times 4 bytes each)

`add a1, a1, t0` and `vle32.v v1, (a2)` — Load first 4 of `y`

`v1 = [5, 15, 25, 35]` then `a1` and `a2` advance by 16 bytes

`vadd.vv v2, v0, v1` — Element-wise Addition

<code>v0</code>	=	[10	20	30	40]
<code>v1</code>	=	[5	15	25	35]
<code>v2</code>	=	15	35	55	75

`vse32.v v2, (a3)` — Store result

`z[0..3] = [15, 35, 55, 75]` then `a3` advances by 16 bytes

`bnez a0, vvaddint32`

`a0 = 2 \neq 0` \Rightarrow **LOOP BACK**

Iteration 2 — Remaining 2 Elements

```
vsetvli t0, a0, e32, ta, ma
```

$a0=2$, hardware $\text{max}=4 \Rightarrow \mathbf{t0 = 2}$ (only 2 left!)

```
vle32.v v0, (a1) and vle32.v v1, (a2)
```

$v0 = [50, 60]$ $v1 = [45, 55]$

```
sub a0, a0, t0
```

$a0 = 2 - 2 = 0$ (done!)

```
vadd.vv v2, v0, v1
```

$v0$	$=$	$[50$	$60]$
$v1$	$=$	$[45$	$55]$
$v2$	$=$	95	115

```
vse32.v v2, (a3)
```

$z[4..5] = [95, 115]$

Array		Elements					
x	=	10	20	30	40	50	60
y	=	5	15	25	35	45	55
z	=	15	35	55	75	95	115

Iteration	Elements Computed
1	$z[0..3] = [15, 35, 55, 75]$
2	$z[4..5] = [95, 115]$

Iteration Summary Table

Property	Iteration 1	Iteration 2
a0 at start	6	2
t0 (batch size)	4	2
v0 loaded	[10,20,30,40]	[50,60]
v1 loaded	[5,15,25,35]	[45,55]
v2 result	[15,35,55,75]	[95,115]
a0 after sub	2	0
Byte offset (slli)	16 bytes	8 bytes
Loop back?	YES (a0=2)	NO (a0=0)

Key Insight

`vsetvli` automatically adjusts the batch size in iteration 2 — no out-of-bounds access, no extra code needed!

Why is `slli t0, t0, 2` Needed?

The Problem

Pointers move in **bytes**, but we count elements. We must convert:

$$\text{byte offset} = \text{elements} \times 4$$

Because each `int32` = **4 bytes**

Shift Left Logical = Multiply by 2^2

```
slli t0, t0, 2  
=  $t0 \times 2^2 = t0 \times 4$ 
```

Example

t0 (elements)	byte offset
4	$4 \times 4 = 16$
2	$2 \times 4 = 8$

Pointer Arithmetic

```
add a1, a1, t0
```

moves `a1` forward by exactly the right number of bytes to point to the next chunk

Summary

What We Learned

- RISC-V RVV uses **variable-length vector registers** — hardware decides how many elements fit
- `vsetvli` is the key instruction: it sets the batch size and **handles leftover elements automatically**
- The loop runs only **2 times** for $n=6$ (vs 6 times in scalar code)
- Pointer advancement requires `slli` to convert element counts to byte offsets
- The same code works on hardware with $VLEN=4, 8, 16$, or more — **no recompilation needed**

Performance Benefit

With $VLEN=4$: **2 iterations** instead of 6

With $VLEN=8$: **1 iteration** for $n=6$!

Speedup scales with hardware vector width.

Overview: Four Memory Addressing Modes

Unit-Stride

Contiguous
Fastest

Strided

Fixed gap
Medium

Indexed

Arbitrary
Slowest

Special

Whole-reg
Fault-first

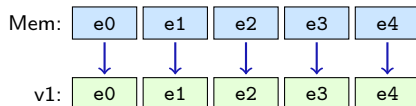
Mode	Instruction	Speed	Use case
Unit-stride	<code>vle32.v</code>	Fastest	Arrays, most loops
Strided	<code>vlse32.v</code>	Medium	Matrix columns
Indexed	<code>vluxe32.v</code>	Slowest	Sparse, gather
Whole-register	<code>vllre32.v</code>	Fast	Save/restore
Fault-only-first	<code>vle32ff.v</code>	Fast	Unknown-length

Mode 1: Unit-Stride

Instructions

```
vle32.v  v1, (a0)  # load
vse32.v  v1, (a0)
# store
```

Memory layout



Key points

- Elements packed back-to-back in memory
- Hardware can use full memory bandwidth
- Variants: vle8, vle16, vle64
- **Default choice** for array processing

Example: load 8 floats from array A

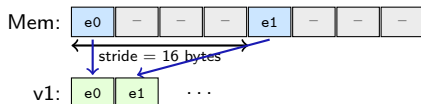
```
vsetvli t0, a3, e32, m1, ta
vle32.v v1, (a0)
# loads VL floats
```

Mode 2: Strided

Instructions

```
li          t0, 16
# stride=16 bytes
vlse32.v v1, (a0), t0
vsse32.v v1, (a0), t0
```

Memory layout (stride = 16 bytes)



- Classic use: **column of a row-major matrix**

Key points

- Stride in **bytes**, stored in a register
- Stride = 0 \Rightarrow **broadcast** one value to all lanes
- Stride = element size \Rightarrow same as unit-stride

Example: load column 0 of 4×4 float matrix

```
# row width = 4 floats = 16
li          t0, 16
vlse32.v v1, (a0), t0
```

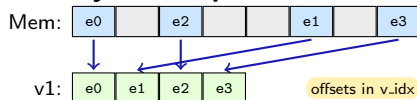
$$v1 = [M[0][0], M[1][0], M[2][0], M[3][0]]$$

Mode 3: Indexed (Gather / Scatter)

Instructions

```
# v_idx holds byte offsets
vluxei32.v v1, (a0), v_idx
# gather
vsuxei32.v v1, (a0), v_idx
# scatter
# ordered variants:
vloxei32.v v1, (a0), v_idx
vsoxei32.v v1, (a0), v_idx
```

Memory access pattern



Ordered vs Unordered

vlux Hardware may re-order — **faster**

vlox Accesses in element order — needed for side-effects

Key points

- Flexible — any element from anywhere
- **Slowest** — cache issues
- Use in sparse matrices, permutations, hash table lookups

$v_{idx} = [0, 20, 8, 28]$ (bytes)

`vluxei32.v v1, (a0), v_idx` # gathers 4 arbitrary floats

Special Modes: Whole-Register & Fault-Only-First

Whole-Register Load/Store

`vl1re32.v`

`v1, (a0) # load 1 reg`

`vl4re32.v`

`v4, (a0) # load 4 regs`

`# (v4–v7)`

`vs1r.v`

`v1, (a0) # store 1 reg`

- Bypasses VL and masking entirely
- Transfers exactly $N \times \text{VLEN}$ bits
- Used for **context save/restore** (e.g. OS task switch)

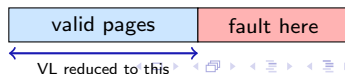
Fault-Only-First

`vle32ff.v v1, (a0)`

`# t0 (vl) silently reduced`

`# if fault occurs mid-load`

- Like unit-stride, but **no exception on fault**
- Hardware truncates VL to elements loaded before fault
- Use in **unknown-length data** (e.g. null-terminated strings)
- Check returned VL to know how many elements arrived

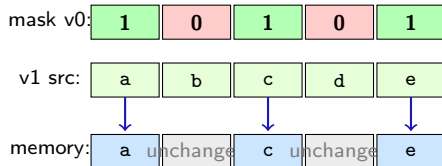


Masking on Memory Instructions

All load/store modes (except whole-register) support an optional **mask register** `v0`.

Masked store — only write active lanes **Use cases**

```
# v0 = mask register  
vse32.v v1, (a0), v0.t  
# only stores where v0[i] = 1  
# other memory locations untouched
```



- **Boundary conditions** — don't write past end of array
- **Conditional updates** — only store where predicate is true
- **Sparse writes** — scatter to selected positions

Masked load fills inactive lanes per mask policy (`mu/ma`):

```
vle32.v v1, (a0), v0.t
```

only loads where `v0[i]=1` inactive lanes: agnostic or undisturbed (set by `mu/ma`)