



# CS 152/252A Computer Architecture and Engineering



Sophia Shao

## Lecture 19: RISC-V Vectors

### NASA Uses RISC-V Vector Spec to Soup Up Space Computers

With the growing demand for applications that require multiple cores and AI, ML, and computer vision capabilities, faster and power-efficient processing is essential. At the same time, companies are looking to simplify design cycles with more portability and re-use, broader extensibility, and more design scalability. The RISC-V Vector spec (RVV) version 1.0, ratified by RISC-V International last December, was created to meet these market requirements and make it easy to implement vector instructions for modern workloads.



Future NASA human spaceflight and robotic science missions will require more powerful space-based computing for autonomy, high-data-rate instruments, and human-robotic interactions.

**Credits: NASA**



<https://www.eetimes.com/nasa-uses-risc-v-vector-spec-to-soup-up-space-computers/>

[https://www.nasa.gov/directorates/spacetech/ga-me\\_changing\\_development/projects/HPSC](https://www.nasa.gov/directorates/spacetech/ga-me_changing_development/projects/HPSC)

# Last Time in Lecture

## GPU architecture

- Evolved from graphics-only, to more general-purpose computing
- GPUs programmed as attached accelerators, with software required to separate GPU from CPU code, move memory
- Many cores, each with many lanes
  - Thousands of lanes on current high-end GPUs
- SIMT model has hardware management of conditional execution
  - Code written as scalar code with branches, executed as vector code with predication

# Last Time in Lecture

## Vector supercomputers

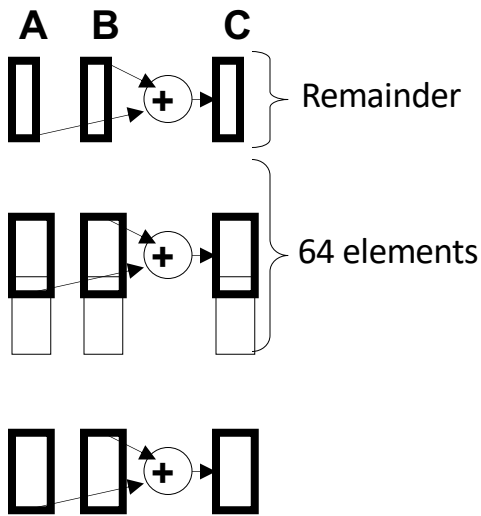
- Vector register versus vector memory
- Scaling performance with lanes
- Stripmining
- Chaining
- Masking
- Scatter/Gather

## Last Time: Vector Stripmining

**Problem:** Vector registers have finite length

**Solution:** Break loops into pieces that fit in registers, “Stripmining”

```
for (i=0; i<N; i++)  
    C[i] = A[i]+B[i];
```



```
andi x1, xN, 63 # N mod 64  
vsetvl x1      # Do remainder  
loop:  
    vld v1, (xA)  
    slli x2, x1, 3 # Multiply by 8  
    add xA, xA, x2 # Bump pointer  
    vld v2, (xB)  
    add xB, xB, x2  
    vadd v3, v1, v2  
    vst v3, (xC)  
    add xC, xC, x2  
    sub xN, xN, x1 # Subtract elements  
    li x1, 64  
    vsetvl x1      # Reset full length  
    bgtz xN, loop # Any more to do?
```

# New RISC-V “V” Vector Extension

- Being added as a standard extension to the RISC-V ISA
  - An updated form of Cray-style vectors for modern microprocessors
  - Appearing in commercial implementations from Alibaba, Andes, Semidynamics, SiFive, ...
  - Basis of European supercomputer initiative (EPI)
- Today, a short tutorial on current standard, v1.0
  - Ratified in December 2021
  - <https://github.com/riscv/riscv-v-spec>

# RISC-V Scalar State

Program counter (**pc**)

32x32/64-bit integer registers (**x0–x31**)

- **x0** always contains a 0

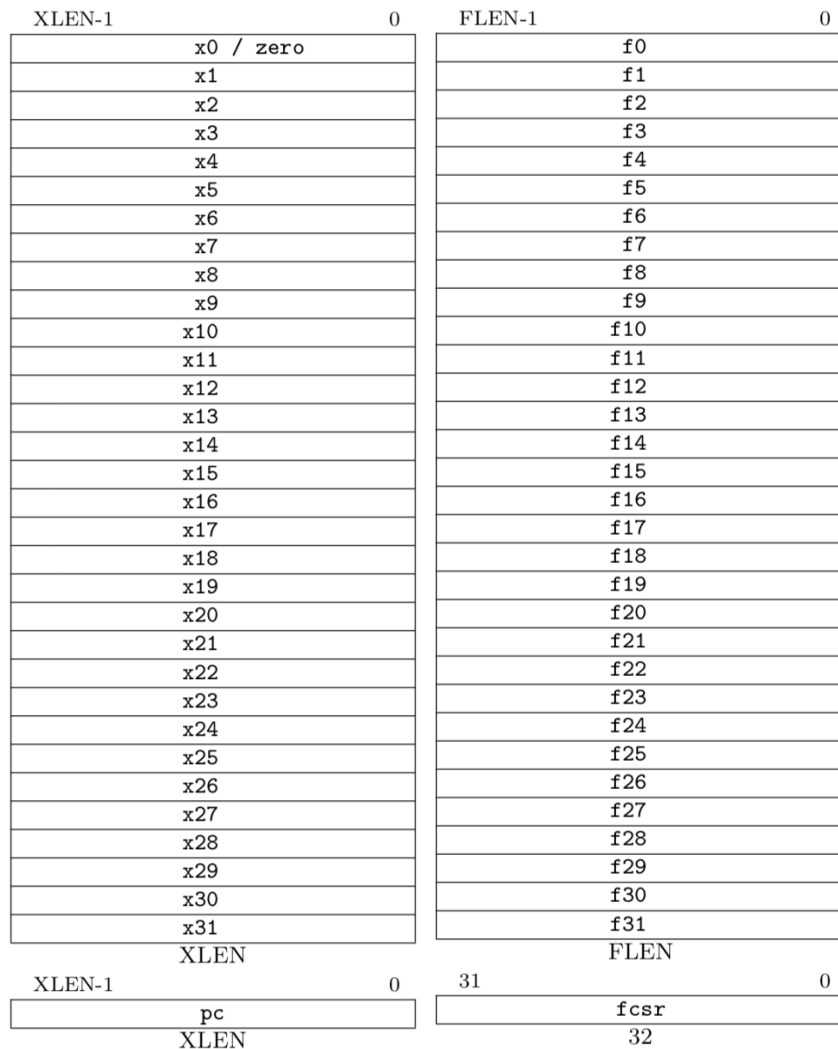
Floating-point (FP), adds 32 registers (**f0–f31**)

- Each can contain a single- or double-precision FP value (32-bit or 64-bit IEEE FP)

FP status register (**fcsr**), used for FP rounding mode & exception reporting

ISA string options:

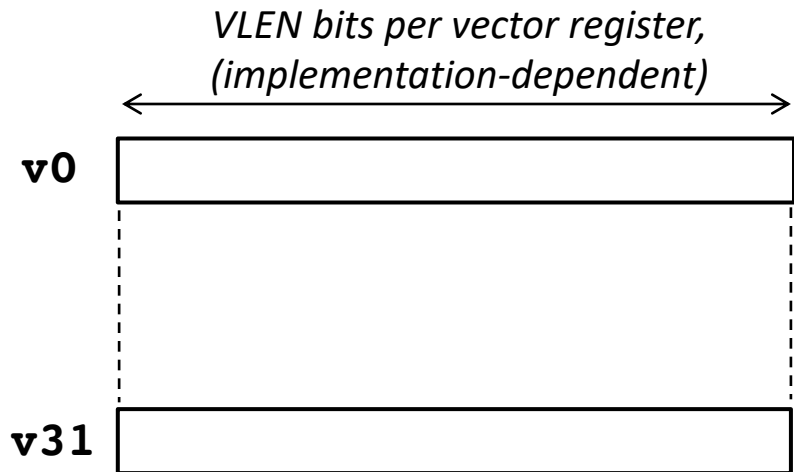
- RV32I (XLEN=32, no FP)
- RV32IF (XLEN=32, FLEN=32)
- RV32ID (XLEN=32, FLEN=64)
- RV64I (XLEN=64, no FP)
- RV64IF (XLEN=64, FLEN=32)
- RV64ID (XLEN=64, FLEN=64)



# Vector Extension Additional State

- 32 vector data registers, **v0–v31**, each VLEN bits long
- Vector length register **v1**
- Vector type register **vtype**
- Other control registers:
  - **vstart**
    - For trap handling
  - **vrn/vxsat**
    - Fixed-point rounding mode/saturation
    - Also appear in separate **vcsr**
  - **vlenb**
    - Gives vector length in bytes (read-only)

Vector data registers



Vector length register

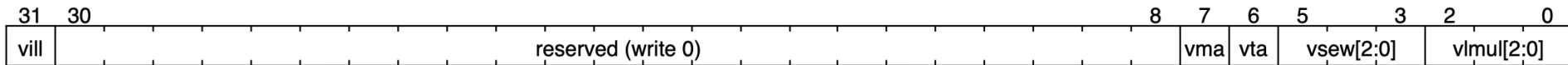
**v1**

Vector type register

**vtype**

# Vector Type Register (vtype)

*Ideally, info would be in instruction encoding, but no space in 32-bit instructions. Planned 64-bit encoding extension would add these as instruction bits.*



**vsew[2:0]** field encodes selected element width (SEW)  
in bits of elements in vector register ( $SEW = 8 * 2^{vsew}$ )

# of elements = VLEN/SEW (when LMUL=1)

**vlmul[2:0]** encodes vector register length multiplier  
(LMUL =  $2^{\text{vlmul}}$  = 1/8 ... 8)

**vta** specifies *tail-agnostic/tail-undisturbed* policy

**vma** specifies *mask-agnostic/mask-undisturbed* policy

vsew[2:0]			SEW
0	0	0	8
0	0	1	16
0	1	1	32
1	0	0	64
1	0	1	128
1	0	1	256
1	1	0	512
1	1	1	1024



# Example Vector Register Data Layouts (LMUL=1)

VLEN=32b	3	2	1	0	SEW
	3	2	1	0	8b
	1		0		16b
	0				32b

VLEN=64b	7	6	5	4	3	2	1	0	SEW
	7	6	5	4	3	2	1	0	8b
	3		2		1		0		16b
	1				0				32b
	0								64b

VLEN=128b	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	SEW
	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	8b
	7		6		5		4		3		2		1		0		16b
	3				2				1				0				32b
	1								0								64b
	0																128b

VLEN = 256b																																128b
1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	SEW
1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	8b
F		E		D		C		B		A		9		8		7		6		5		4		3		2		1		0		16b
7				6				5				4				3				2				1				0				32b
3								2								1								0								64b
1																0																128b
0																																256b

# Setting vector configuration, `vsetvli/vsetivli/vsetvl`

The `vset{i}vl{i}` configuration instructions set the **vt** register, and also set the **vl** register, returning the **vl** value in a scalar register

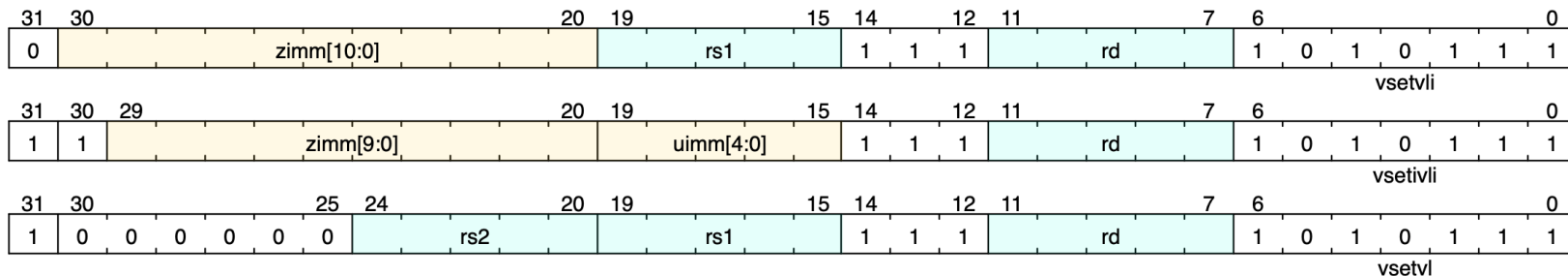
`vsetvli rd, rs1, e8 # Set SEW=8, vl=min(VLEN/SEW,rs1), rd=vl`

Resulting machine vector length setting

Requested application vector length (AVL)

**vt** parameters (SEW, LMUL, VTA, VMA) encoded as immediate in instruction

Instruction encoding



Usually use **register-immediate** form, `vsetvli`, to set **vt** parameters.

**Immediate-immediate** form, `vsetivli`, used when vector length known statically

The **register-register** version `vsetvl` is usually used only for context save/restore

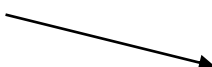
# **vset{i}vl{i} operation**

- The first argument, *rs1* or 5-bit immediate, is the requested application vector length (AVL)
- The type argument (either 10/11-bit immediate or second register *rs2*) indicates how the vector registers should be configured
  - Configuration includes size of each element, SEW, and LMUL value
- The vector length is set to the minimum of requested AVL and **the maximum supported vector length (VLMAX)** in the new configuration
  - $VLMAX = LMUL * VLEN / SEW$
  - **vl** = min(AVL, VLMAX)
- The value placed in **vl** is also written to the scalar destination register *rd*
  - Can be used in scalar ops to control loops


# Simple stripmined vector memcpy example

```
# void *memcpy(void* dest, const void* src, size_t n)
# a0=dest, a1=src, a2=n
#
memcpy:
    mv a3, a0 # Copy destination
loop:
    vsetvli t0, a2, e8,m8,ta,ma    # Vectors of 8b
    vle8.v v0, (a1)                # Load bytes
    add a1, a1, t0                 # Bump pointer
    sub a2, a2, t0                 # Decrement count
    vse8.v v0, (a3)                # Store bytes
    add a3, a3, t0                 # Bump pointer
    bnez a2, loop                  # Any more?
    ret                            # Return
```


*Set configuration,  
calculate vector strip  
length*



*Unit-stride  
vector load  
elements (bytes)*

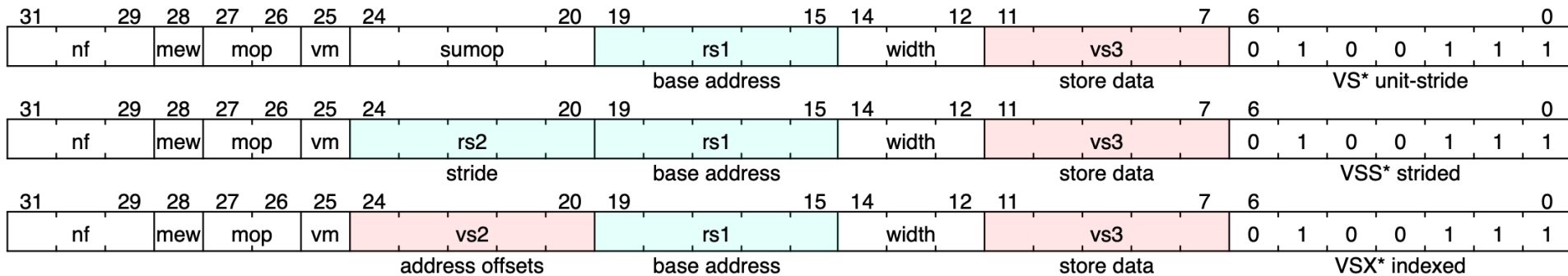
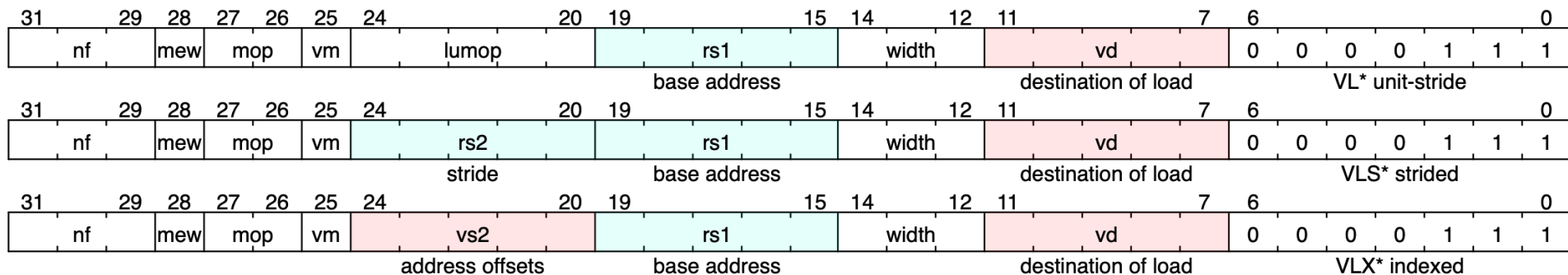


*Unit-stride vector  
store elements  
(bytes)*



*Same binary machine code can run on machines with any VLEN!*

# Vector Load and Store Instructions



# Vector Unit-Stride Loads/Stores

```
# vd destination, rs1 base address, vm is mask encoding (v0.t or <missing>)
vle8.v    vd, (rs1), vm #    8-bit unit-stride load
vle16.v   vd, (rs1), vm #   16-bit unit-stride load
vle32.v   vd, (rs1), vm #   32-bit unit-stride load
vle64.v   vd, (rs1), vm #   64-bit unit-stride load
```

```
# vs3 store data, rs1 base address, vm is mask encoding (v0.t or <missing>)
vse8.v    vs3, (rs1), vm #    8-bit unit-stride store
vse16.v   vs3, (rs1), vm #   16-bit unit-stride store
vse32.v   vs3, (rs1), vm #   32-bit unit-stride store
vse64.v   vs3, (rs1), vm #   64-bit unit-stride store
```

# Vector *Strided* Load/Store Instructions

```
# vd destination, rs1 base address, rs2 byte stride
vlse8.v      vd, (rs1), rs2, vm #    8-bit strided load
vlse16.v     vd, (rs1), rs2, vm #   16-bit strided load
vlse32.v     vd, (rs1), rs2, vm #   32-bit strided load
vlse64.v     vd, (rs1), rs2, vm #   64-bit strided load
```

```
# vs3 store data, rs1 base address, rs2 byte stride
vsse8.v      vs3, (rs1), rs2, vm #    8-bit strided store
vsse16.v     vs3, (rs1), rs2, vm #   16-bit strided store
vsse32.v     vs3, (rs1), rs2, vm #   32-bit strided store
vsse64.v     vs3, (rs1), rs2, vm #   64-bit strided store
```

# Vector *Indexed* Loads/Stores

```
# Vector unordered indexed load instructions
# vd destination, rs1 base address, vs2 indices
vluxei8.v    vd, (rs1), vs2, vm # unordered 8-bit indexed load of SEW data
vluxei16.v   vd, (rs1), vs2, vm # unordered 16-bit indexed load of SEW data
vluxei32.v   vd, (rs1), vs2, vm # unordered 32-bit indexed load of SEW data
vluxei64.v   vd, (rs1), vs2, vm # unordered 64-bit indexed load of SEW data

# Vector ordered indexed load instructions
# vd destination, rs1 base address, vs2 indices
vloxei8.v    vd, (rs1), vs2, vm # ordered 8-bit indexed load of SEW data
vloxei16.v   vd, (rs1), vs2, vm # ordered 16-bit indexed load of SEW data
vloxei32.v   vd, (rs1), vs2, vm # ordered 32-bit indexed load of SEW data
vloxei64.v   vd, (rs1), vs2, vm # ordered 64-bit indexed load of SEW data

# Vector unordered-indexed store instructions
# vs3 store data, rs1 base address, vs2 indices
vsuxei8.v    vs3, (rs1), vs2, vm # unordered 8-bit indexed store of SEW data
vsuxei16.v   vs3, (rs1), vs2, vm # unordered 16-bit indexed store of SEW data
vsuxei32.v   vs3, (rs1), vs2, vm # unordered 32-bit indexed store of SEW data
vsuxei64.v   vs3, (rs1), vs2, vm # unordered 64-bit indexed store of SEW data

# Vector ordered indexed store instructions
# vs3 store data, rs1 base address, vs2 indices
vsoxei8.v    vs3, (rs1), vs2, vm # ordered 8-bit indexed store of SEW data
vsoxei16.v   vs3, (rs1), vs2, vm # ordered 16-bit indexed store of SEW data
vsoxei32.v   vs3, (rs1), vs2, vm # ordered 32-bit indexed store of SEW data
vsoxei64.v   vs3, (rs1), vs2, vm # ordered 64-bit indexed store of SEW data
```

Index data width encoded in instruction, while data size encoded in **vtype.vsew** field



# Vector Length Multiplier (LMUL)

- Gives fewer but longer vector registers
  - Called “vector register groups” — operate as single vectors
  - Must use even register names only for LMUL=2 (v0, v2, ...), and every fourth register for LMUL=4 (v0, v4, ...), etc.
- Used to 1) accommodate mixed-width operations, and/or 2) to increase efficiency by using longer vectors when fewer separate registers needed
- Set by **vlmul[2:0]** field in **vtype** during **vsetvli**

vlmul[2:0]			LMUL	#groups	VLMAX	Registers grouped with register <i>n</i>
1	0	0	-	-	-	reserved
1	0	1	1/8	32	VLEN/SEW/8	v <i>n</i> (single register in group)
1	1	0	1/4	32	VLEN/SEW/4	v <i>n</i> (single register in group)
1	1	1	1/2	32	VLEN/SEW/2	v <i>n</i> (single register in group)
0	0	0	1	32	VLEN/SEW	v <i>n</i> (single register in group)
0	0	1	2	16	2*VLEN/SEW	v <i>n</i> , v <i>n</i> +1
0	1	0	4	8	4*VLEN/SEW	v <i>n</i> , ..., v <i>n</i> +3
0	1	1	8	4	8*VLEN/SEW	v <i>n</i> , ..., v <i>n</i> +7

# LMUL=8 stripmined vector memcpy example

```
# void *memcpy(void* dest, const void* src, size_t n)
# a0=dest, a1=src, a2=n
#
memcpy:
    mv a3, a0 # Copy destination
loop:
    vsetvli t0, a2, e8,m8,ta,ma # Vectors of 8b
    vle8.v v0, (a1)             # Load bytes
    add a1, a1, t0              # Bump pointer
    sub a2, a2, t0              # Decrement count
    vse8.v v0, (a3)             # Store bytes
    add a3, a3, t0              # Bump pointer
    bnez a2, loop               # Any more?
    ret                         # Return
```

*Combine eight vector registers into group*  
(v0 <- v0,v1,...,v7)

*Set configuration,  
calculate vector strip  
length*

*Unit-stride  
vector load bytes*

*Unit-stride vector  
store bytes*

Binary machine code can run on machines with any VLEN!

# CS152 Administritivia

- Lab 3 due 3/23
  - New fix pushed.
- HW 4 due 3/30
  - You can already start (last week).
  - Ask your question early!
- No real-time lecture this Thursday
  - Watch Turing Award lecture (linked online)
- Enjoy Spring Break!

## CS252 Administrivia

- No paper reading this week
- Project update 4/5

# Vector Integer Add Instructions

# Integer adds.

```
vadd.vv vd, vs2, vs1, vm    # Vector-vector  
vadd.vx vd, vs2, rs1, vm    # vector-scalar  
vadd.vi vd, vs2, imm, vm    # vector-immediate
```

# Integer subtract

```
vsub.vv vd, vs2, vs1, vm    # Vector-vector  
vsub.vx vd, vs2, rs1, vm    # vector-scalar
```

# Integer reverse subtract

```
vrsb.vx vd, vs2, rs1, vm    #  $vd[i] = rs1 - vs2[i]$   
vrsb.vi vd, vs2, imm, vm    #  $vd[i] = imm - vs2[i]$ 
```

# Vector FP Add Instructions

# Floating-point add

vfadd.vv vd, vs2, vs1, vm # Vector-vector

vfadd.vf vd, vs2, rs1, vm # vector-scalar

# Floating-point subtract

vfsb.vv vd, vs2, vs1, vm # Vector-vector

vfsb.vf vd, vs2, rs1, vm # Vector-scalar  $vd[i] = vs2[i] - f[rs1]$

vfrsub.vf vd, vs2, rs1, vm # Scalar-vector  $vd[i] = f[rs1] - vs2[i]$

SEW can be 16b, 32b, 64b, 128b for half/single/double/quad FP

Scalar values come from floating-point *f* registers

# Masking

- Nearly all operations can be optionally under a mask (or predicate) held in vector register **v0**
- A single *vm* bit in instruction encoding selects whether unmasked or under control of **v0**
- Constrained by encoding space in 32-bit instructions
  - Longer 64-bit encoding extension will support predicate in any register
- Integer and FP compare instructions provided to set masks into any vector register
- Can perform mask logical operations between any vector registers

# Integer Compare Instructions

Comparison	Assembler Mapping	Assembler Pseudoinstruction
va < vb	vmslt{u}.vv vd, va, vb, vm	
va <= vb	vmsle{u}.vv vd, va, vb, vm	
va > vb	vmslt{u}.vv vd, vb, va, vm	vmsgt{u}.vv vd, va, vb, vm
va >= vb	vmsle{u}.vv vd, vb, va, vm	vmsge{u}.vv vd, va, vb, vm
va < x	vmslt{u}.vx vd, va, x, vm	
va <= x	vmsle{u}.vx vd, va, x, vm	
va > x	vmsgt{u}.vx vd, va, x, vm	
va >= x	see below	
va < i	vmsle{u}.vi vd, va, i-1, vm	vmslt{u}.vi vd, va, i, vm
va <= i	vmsle{u}.vi vd, va, i, vm	
va > i	vmsgt{u}.vi vd, va, i, vm	
va >= i	vmsgt{u}.vi vd, va, i-1, vm	vmsge{u}.vi vd, va, i, vm

va, vb vector register groups

x scalar integer register

i immediate



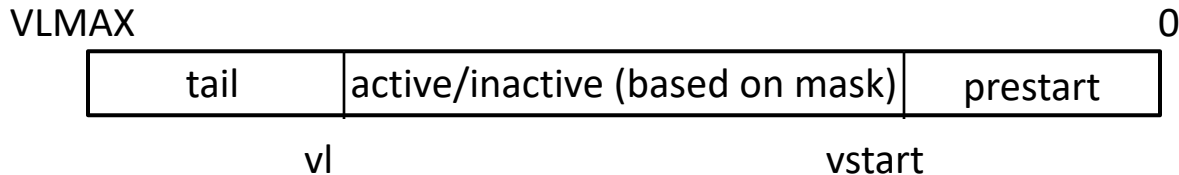
# Mask Logical Operations

```
vmmand.mm vd, vs2, vs1 # vd.mask[i] = vs2.mask[i] && vs1.mask[i]
vmnand.mm vd, vs2, vs1 # vd.mask[i] = !(vs2.mask[i] && vs1.mask[i])
vmmandn.mm vd, vs2, vs1 # vd.mask[i] = vs2.mask[i] && !vs1.mask[i]
vmxor.mm vd, vs2, vs1 # vd.mask[i] = vs2.mask[i] ^ vs1.mask[i]
vmor.mm vd, vs2, vs1 # vd.mask[i] = vs2.mask[i] || vs1.mask[i]
vmnor.mm vd, vs2, vs1 # vd.mask[i] = !(vs2.mask[i] || vs1.mask[i])
vmorn.mm vd, vs2, vs1 # vd.mask[i] = vs2.mask[i] || !vs1.mask[i]
vmxnor.mm vd, vs2, vs1 # vd.mask[i] = !(vs2.mask[i] ^ vs1.mask[i])
```

Several assembler pseudoinstructions are defined as shorthand for common uses of mask logical operations:

```
vmmv.m vd, vs => vmmand.mm vd, vs, vs # Copy mask register
vmclr.m vd => vmxor.mm vd, vd, vd # Clear mask register
vmset.m vd => vmxnor.mm vd, vd, vd # Set mask register
vmnot.m vd, vs => vmnand.mm vd, vs, vs # Invert bits
```

# Prestart, Active, Inactive, Body, and Tail Elements



The destination element indices operated on during a vector instruction's execution can be divided into three disjoint subsets.

- The *prestart* elements are those whose element index is less than the initial value in the `vstart` register. The prestart elements do not raise exceptions and do not update the destination vector register.
- The *body* elements are those whose element index is greater than or equal to the initial value in the `vstart` register, and less than the current vector length setting in `vl`. The body can be split into two disjoint subsets:
  - The *active* elements during a vector instruction's execution are the elements within the body and where the current mask is enabled at that element position. The active elements can raise exceptions and update the destination vector register group.
  - The *inactive* elements are the elements within the body but where the current mask is disabled at that element position. The inactive elements do not raise exceptions and do not update any destination vector register group unless masked agnostic is specified (`vtype.vma=1`), in which case inactive elements may be overwritten with 1s.
- The *tail* elements during a vector instruction's execution are the elements past the current vector length setting specified in `vl`. The tail elements do not raise exceptions, and do not update any destination vector register group unless tail agnostic is specified (`vtype.vta=1`), in which case tail elements may be overwritten with 1s. When  $LMUL < 1$ , the tail includes the elements past VLMAX that are held in the same vector register.

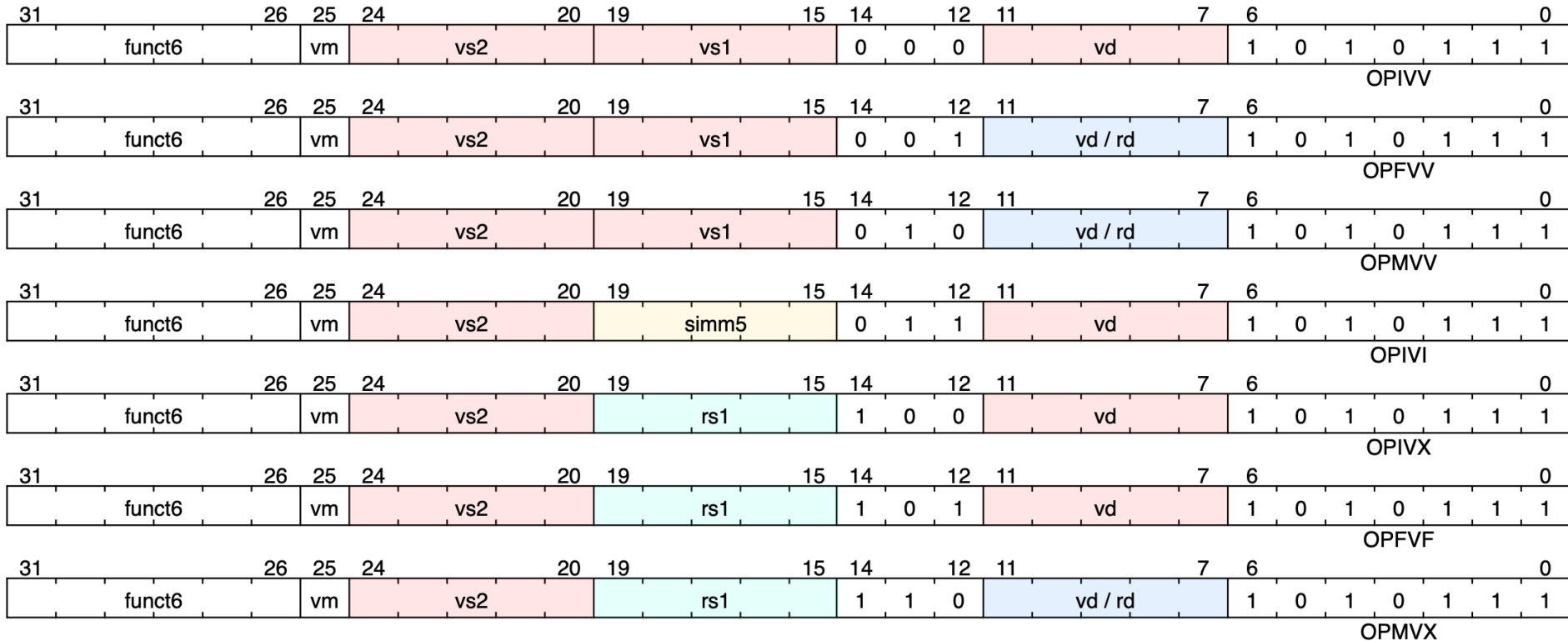
## Vector Tail Agnostic / Vector Mask Agnostic

Affects the behavior of destination tail and inactive masked-off elements during execution of vector instructions

- **Undisturbed:** previous values are preserved
- **Agnostic:** either left undisturbed or overwritten with 1s

```
vsetvli rd, rs1, e32, m2, ta, ma    # tail agnostic, mask agnostic
vsetvli rd, rs1, e32, m2, tu, ma    # tail undisturbed, mask agnostic
vsetvli rd, rs1, e32, m2, ta, mu    # tail agnostic, mask undisturbed
vsetvli rd, rs1, e32, m2, tu, mu    # tail undisturbed, mask undisturbed
```

# Vector Arithmetic Instruction Encodings



# Widening Integer Add Instructions

# Widening unsigned integer add/subtract,  $2*SEW = SEW +/- SEW$

`vwaddu.vv vd, vs2, vs1, vm` # vector-vector

`vwaddu.vx vd, vs2, rs1, vm` # vector-scalar

`vwsubu.vv vd, vs2, vs1, vm` # vector-vector

`vwsubu.vx vd, vs2, rs1, vm` # vector-scalar

# Widening signed integer add/subtract,  $2*SEW = SEW +/- SEW$

`vwadd.vv vd, vs2, vs1, vm` # vector-vector

`vwadd.vx vd, vs2, rs1, vm` # vector-scalar

`vwsub.vv vd, vs2, vs1, vm` # vector-vector

`vwsub.vx vd, vs2, rs1, vm` # vector-scalar

# Widening unsigned integer add/subtract,  $2*SEW = 2*SEW +/- SEW$

`vwaddu.wv vd, vs2, vs1, vm` # vector-vector

`vwaddu.wx vd, vs2, rs1, vm` # vector-scalar

`vwsubu.wv vd, vs2, vs1, vm` # vector-vector

`vwsubu.wx vd, vs2, rs1, vm` # vector-scalar

# Widening signed integer add/subtract,  $2*SEW = 2*SEW +/- SEW$

`vwadd.wv vd, vs2, vs1, vm` # vector-vector

`vwadd.wx vd, vs2, rs1, vm` # vector-scalar

`vwsub.wv vd, vs2, vs1, vm` # vector-vector

`vwsub.wx vd, vs2, rs1, vm` # vector-scalar

# Widening FP Mul-Add

# FP widening multiply-accumulate, overwrites addend

`vfwmacc.vv vd, vs1, vs2, vm`    #  $vd[i] = +(vs1[i] * vs2[i]) + vd[i]$

`vfwmacc.vf vd, rs1, vs2, vm`    #  $vd[i] = +(f[rs1] * vs2[i]) + vd[i]$

# FP widening negate-(multiply-accumulate), overwrites addend

`vfwnmacc.vv vd, vs1, vs2, vm`    #  $vd[i] = -(vs1[i] * vs2[i]) - vd[i]$

`vfwnmacc.vf vd, rs1, vs2, vm`    #  $vd[i] = -(f[rs1] * vs2[i]) - vd[i]$

# FP widening multiply-subtract-accumulator, overwrites addend

`vfwsac.vv vd, vs1, vs2, vm`    #  $vd[i] = +(vs1[i] * vs2[i]) - vd[i]$

`vfwsac.vf vd, rs1, vs2, vm`    #  $vd[i] = +(f[rs1] * vs2[i]) - vd[i]$

# FP widening negate-(multiply-subtract-accumulator), overwrites addend

`vfwnmsac.vv vd, vs1, vs2, vm`    #  $vd[i] = -(vs1[i] * vs2[i]) + vd[i]$

`vfwnmsac.vf vd, rs1, vs2, vm`    #  $vd[i] = -(f[rs1] * vs2[i]) + vd[i]$

# Mixed-Width Loops

- Have different element widths in one loop, even in one instruction
  - e.g., widening multiply,  $16b \times 16b \rightarrow 32b$  product
- Want same number of elements in each vector register, even if different bits/element
- Solution: Keep SEW/LMUL constant

VLEN=128b

SEW=8b, LMUL=1, VLMAX=16

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	$v1 * n + 0$

SEW=16b, LMUL=2, VLMAX=16

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
7	6	5	4	3	2	1	0	$v2 * n + 0$								
F	E	D	C	B	A	9	8	$v2 * n + 1$								

SEW/LMUL=8

SEW=32b, LMUL=4, VLMAX=16

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

SEW=64b, LMUL=8, VLMAX=16

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
1							0									$v8 * n + 0$
3							2									$v8 * n + 1$
5							4									$v8 * n + 2$
7							6									$v8 * n + 3$
9							8									$v8 * n + 4$
B							A									$v8 * n + 5$
D							C									$v8 * n + 6$
F							E									$v8 * n + 7$



# SAXPY Example

```
# void
# saxpy(size_t n, const float a, const float *x, float *y)
# {
#     size_t i;
#     for (i=0; i<n; i++)
#         y[i] = a * x[i] + y[i];
# }
#
# register arguments:
#     a0      n
#     fa0     a
#     a1      x
#     a2      y
```

saxpy:

```
vsetvli a4, a0, e32, m8, ta,ma # a4 = vl
vle32.v v0, (a1)                # load x
sub a0, a0, a4                  # n = n - vl
slli a4, a4, 2                  # multiply by 4 bytes
add a1, a1, a4
vle32.v v8, (a2)                # load y
vfmacc.vf v8, fa0, v0
vse32.v v8, (a2)                # store y
add a2, a2, a4
bnez a0, saxpy
ret
```

# Conditional/Mixed Width Example

```
# (int16) z[i] = ((int8) x[i] < 5) ? (int16) a[i] : (int16) b[i];  
#
```

```
loop:
```

```
    vsetvli t0, a0, e8,m1,ta,ma # Use 8b elements.  
    vle8.v v0, (a1)             # Get x[i]  
    add a1, a1, t0              # x[i] Bump pointer  
    vmslt.vi v0, v0, 5          # Set mask in v0  
    vsetvli t0, a0, e16,m2,ta,mu # Use 16b elements.  
    sub a0, a0, t0             # Decrement element count  
    slli t0, t0, 1             # Multiply by 2 bytes  
    vle16.v v2, (a2), v0.t      # z[i] = a[i] case  
    vmnot.m v0, v0              # Invert v0  
    add a2, a2, t0              # a[i] bump pointer  
    vle16.v v2, (a3), v0.t      # z[i] = b[i] case  
    add a3, a3, t0              # b[i] bump pointer  
    vse16.v v2, (a4)            # Store z  
    add a4, a4, t0              # z[i] bump pointer  
    bnez a0, loop
```

# Creative Commons License

- These lecture slides are made available under a CC BY-SA 4.0 license
- <https://creativecommons.org/licenses/by-sa/4.0/>
- Attribution Title: “RISC-V Vectors, CS152, Spring 2023”
- Attribution Author: Krste Asanović
  - Modified by Albert Ou
- Original content link:  
<https://inst.eecs.berkeley.edu/~cs152/sp23/>