CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Lecture 17 – RISC-V Vectors

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152

Last Time in Lecture 16

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

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
- Today, a short tutorial on current draft standard, v0.8/0.9
  - v0.8 is version supported by tools, v0.9 has some small changes highlighted in **red**
  - v0.9 is intended to be close to final version of RISC-V vector extension
  - Still a work in progress, so details might change before standardization
  - **`https://github.com/riscv/riscv-v-spec`**

RISC-V Scalar State

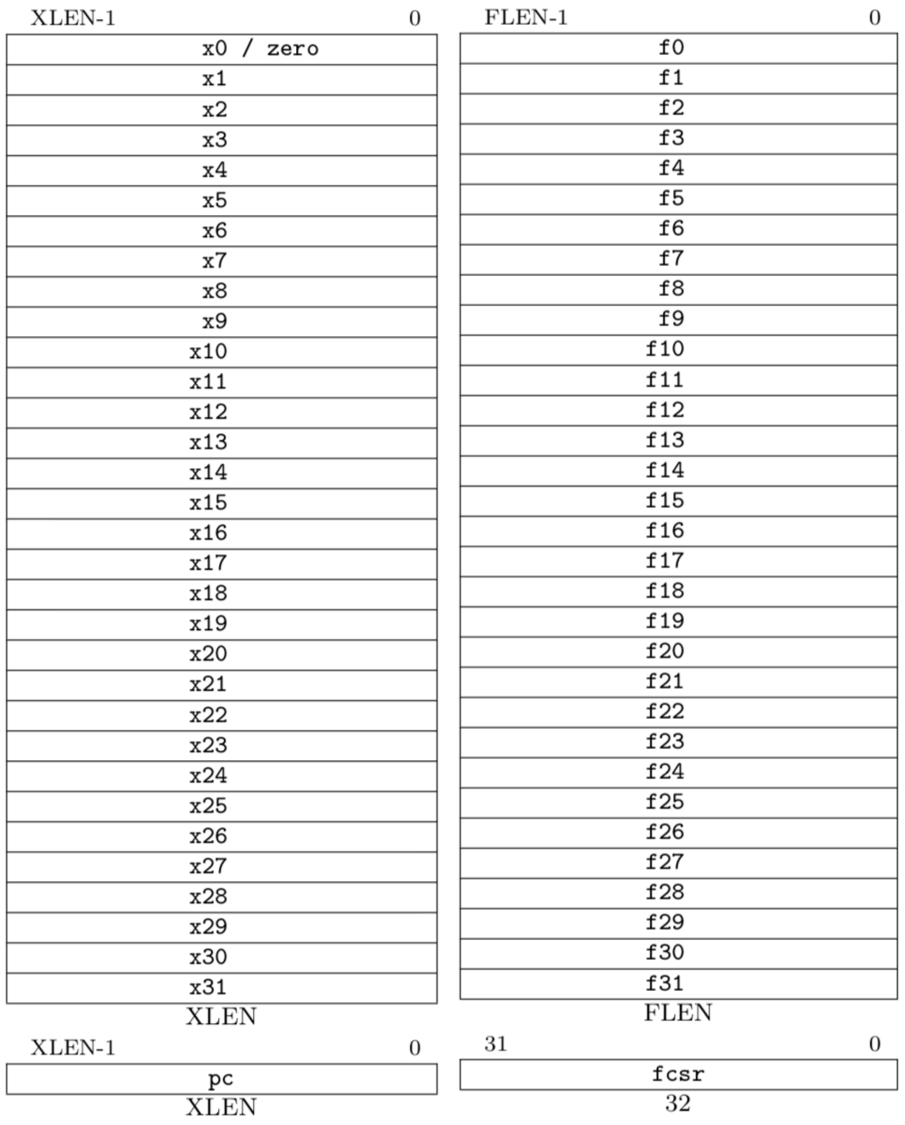
Program counter (**pc**)

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)

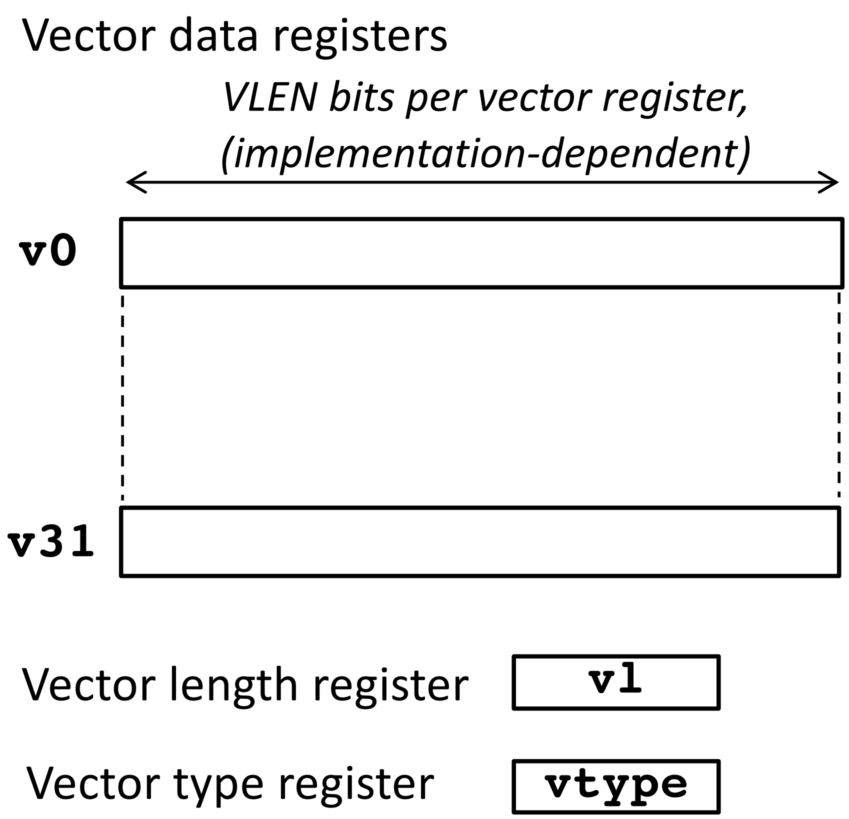
FP status register (**fcsr**), used for FP rounding mode & exception reporting

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

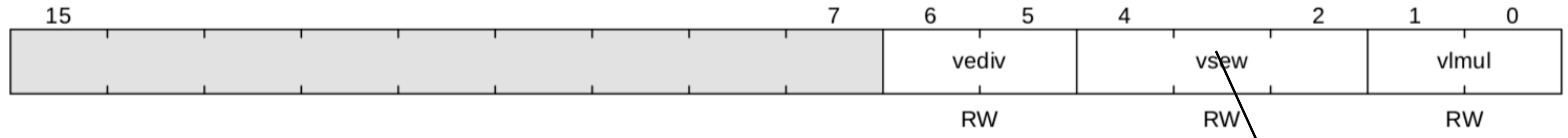
- 32 vector data registers, **`v0-v31`**, each VLEN bits long
- Vector length register **`vl`**
- Vector type register **`vtype`**
- Other control registers:
  - **`vstart`**
    - For trap handling
  - **`vrm/vxsat`**
    - Fixed-point rounding mode/saturation
    - Also appear in **`fcsr`** (0.9: in separate **`vcsr`**)



Vector Type Register

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

vtype register layout



Bits	Contents
1:0	vlmul[1:0]
4:2	vsew[2:0]
6:5	vediv[1:0]
XLEN-1:7	Reserved (write 0)

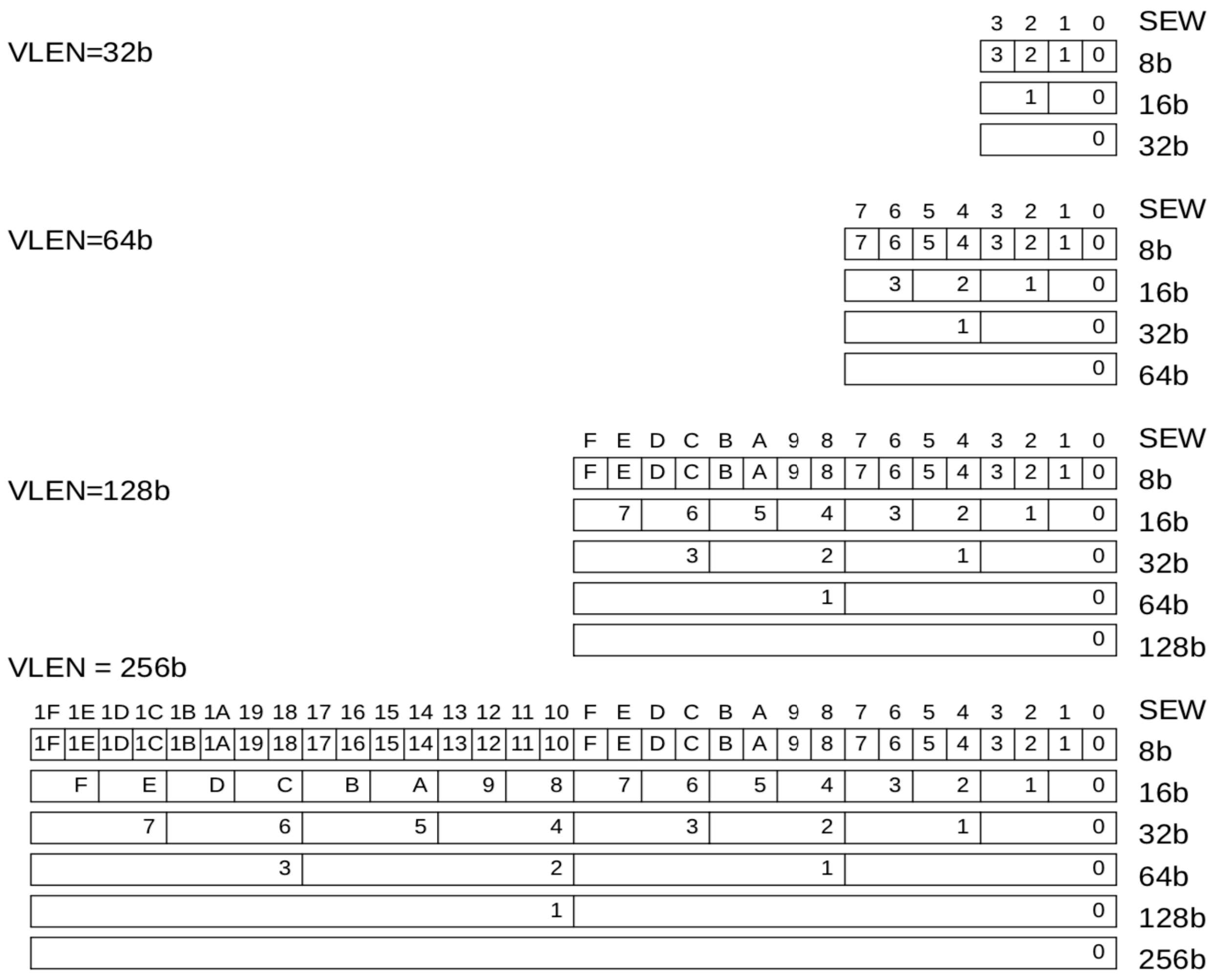
vsew[2:0]			SEW
0	0	0	8
0	0	1	16
0	1	0	32
0	1	1	64
1	0	0	128
1	0	1	256
1	1	0	512
1	1	1	1024

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

`vlmul[1:0]` encodes vector register length multiplier (LMUL = $2^{\texttt{vlmul}}$ = 1-8) *(v0.9 adds "fractional LMUL" < 1)*

`vediv[1:0]` encodes how vector elements are divided into equal sub-elements (EDIV = $2^{\texttt{vediv}}$ = 1-8)

Example Vector Register Data Layouts (LMUL=1)


VLEN=32b
VLEN=64b
VLEN=128b
VLEN = 256b
SEW
8b
16b
32b
64b
128b
256b
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

Setting vector configuration, **`vsetvli/vsetvl`**

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

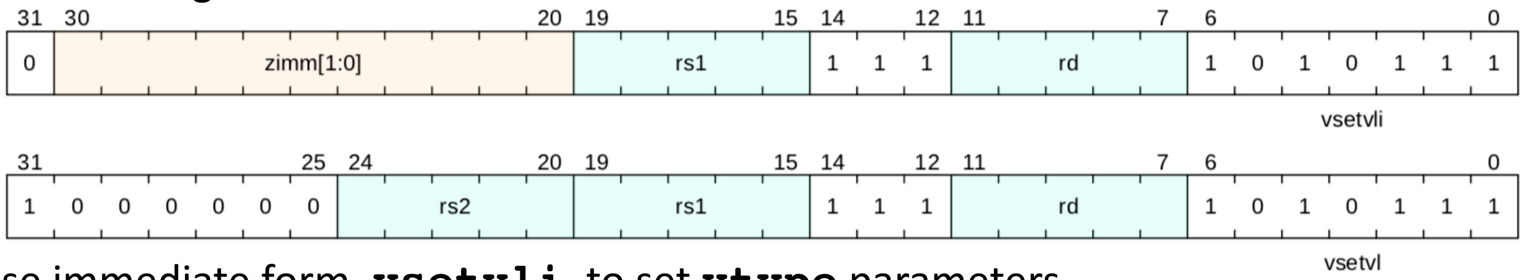
```
vsetvli rd, rs1, e8 # Set SEW=8, vl=min(VLEN/SEW,rs1), rd=vl
```

`vtype` parameters (SEW,LMUL,EDIV) encoded as immediate in instruction

Resulting machine vector length setting

Requested application vector length

Instruction encoding



Usually use immediate form, **`vsetvli`**, to set **`vtype`** parameters.

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

`vsetvl{i}` operation

- The first scalar register argument, *rs1*, is the requested application vector length (AVL)
- The type argument (either 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*

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      # Vectors of 8b
  vle.v v0, (a1)                 # Load bytes
    add a1, a1, t0               # Bump pointer
    sub a2, a2, t0               # Decrement count
  vse.v v0, (a3)                 # Store bytes
    add a3, a3, t0               # Bump pointer
    bnez a2, loop                # Any more?
    ret                          # Return
```

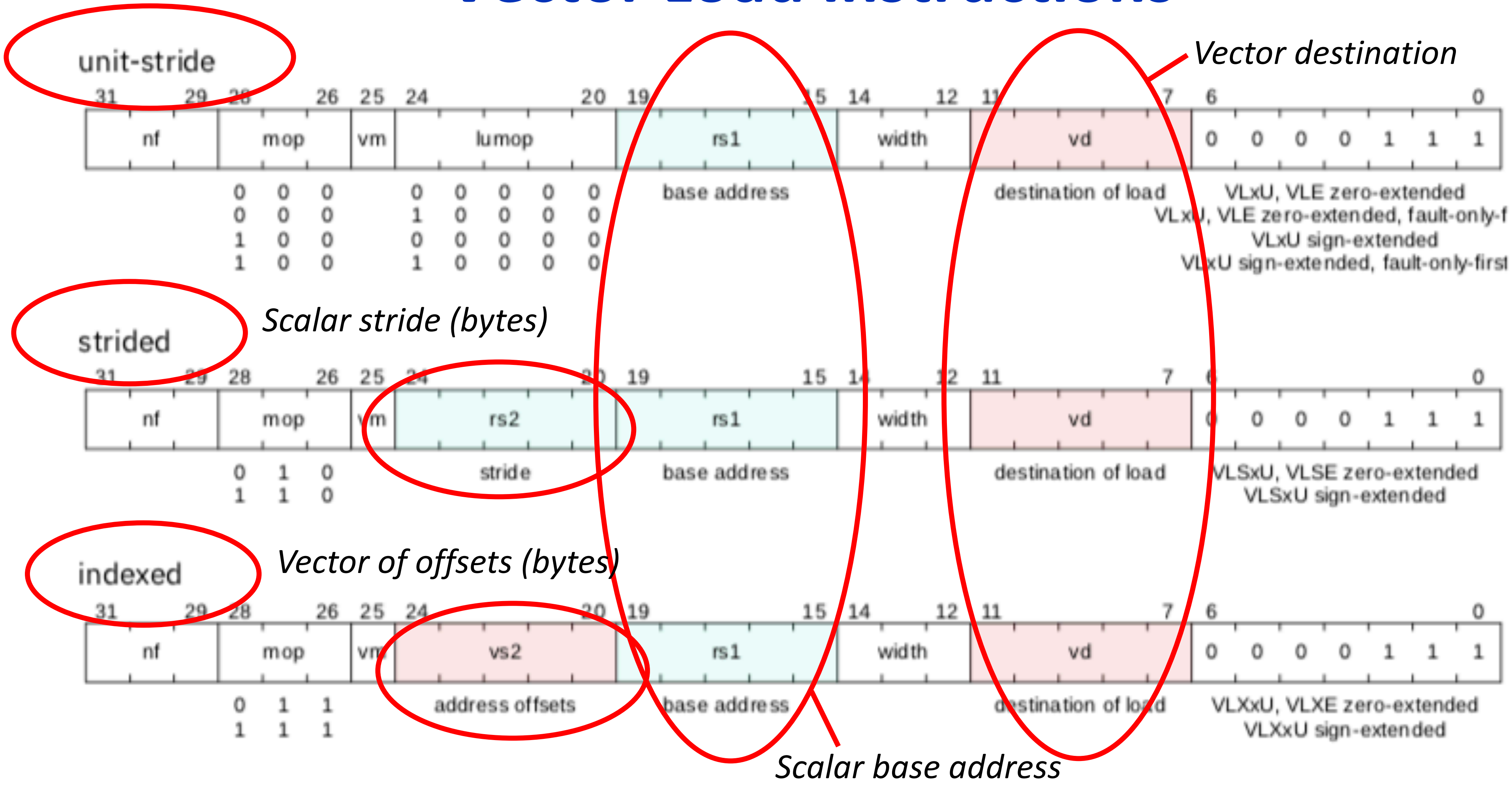
Set configuration, calculate vector strip length → `vsetvli`

Unit-stride vector load elements (bytes) → `vle.v`

Unit-stride vector store elements (bytes) → `vse.v`

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

Vector Load Instructions


Vector destination
Scalar stride (bytes)
Vector of offsets (bytes)
Scalar base address

unit-stride

31–29	28–26	25	24–20	19–15	14–12	11–7	6–0
nf	mop	vm	lumop	rs1	width	vd	0 0 0 0 1 1 1
	0 0 0		0 0 0 0 0	base address		destination of load	VLxU, VLE zero-extended
	0 0 0		1 0 0 0 0				VLxU, VLE zero-extended, fault-only-f
	1 0 0		0 0 0 0 0				VLxU sign-extended
	1 0 0		1 0 0 0 0				VLxU sign-extended, fault-only-first

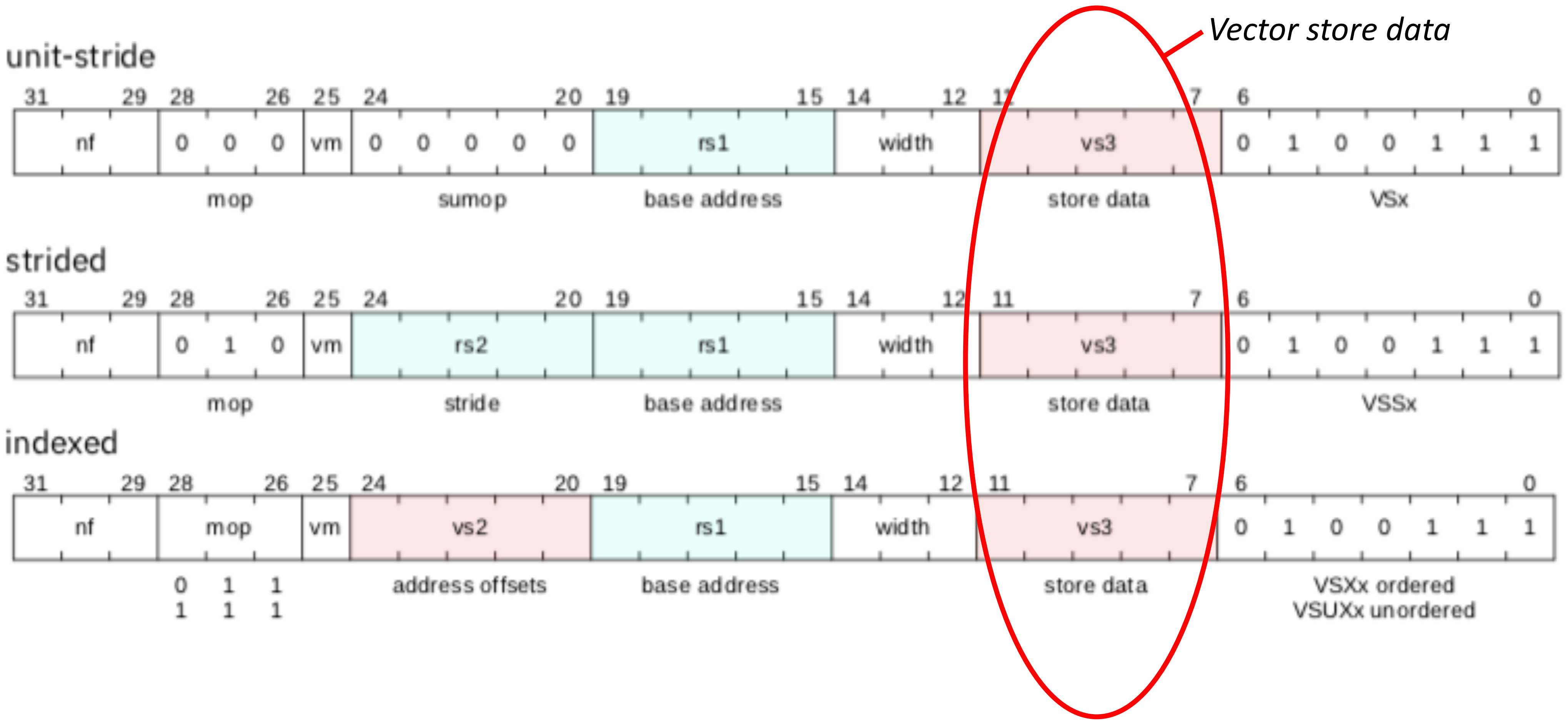
strided

31–29	28–26	25	24–20	19–15	14–12	11–7	6–0
nf	mop	vm	rs2	rs1	width	vd	0 0 0 0 1 1 1
	0 1 0		stride	base address		destination of load	VLSxU, VLSE zero-extended
	1 1 0						VLSxU sign-extended

indexed

31–29	28–26	25	24–20	19–15	14–12	11–7	6–0
nf	mop	vm	vs2	rs1	width	vd	0 0 0 0 1 1 1
	0 1 1		address offsets	base address		destination of load	VLXxU, VLXE zero-extended
	1 1 1						VLXxU sign-extended

Vector Store Instructions

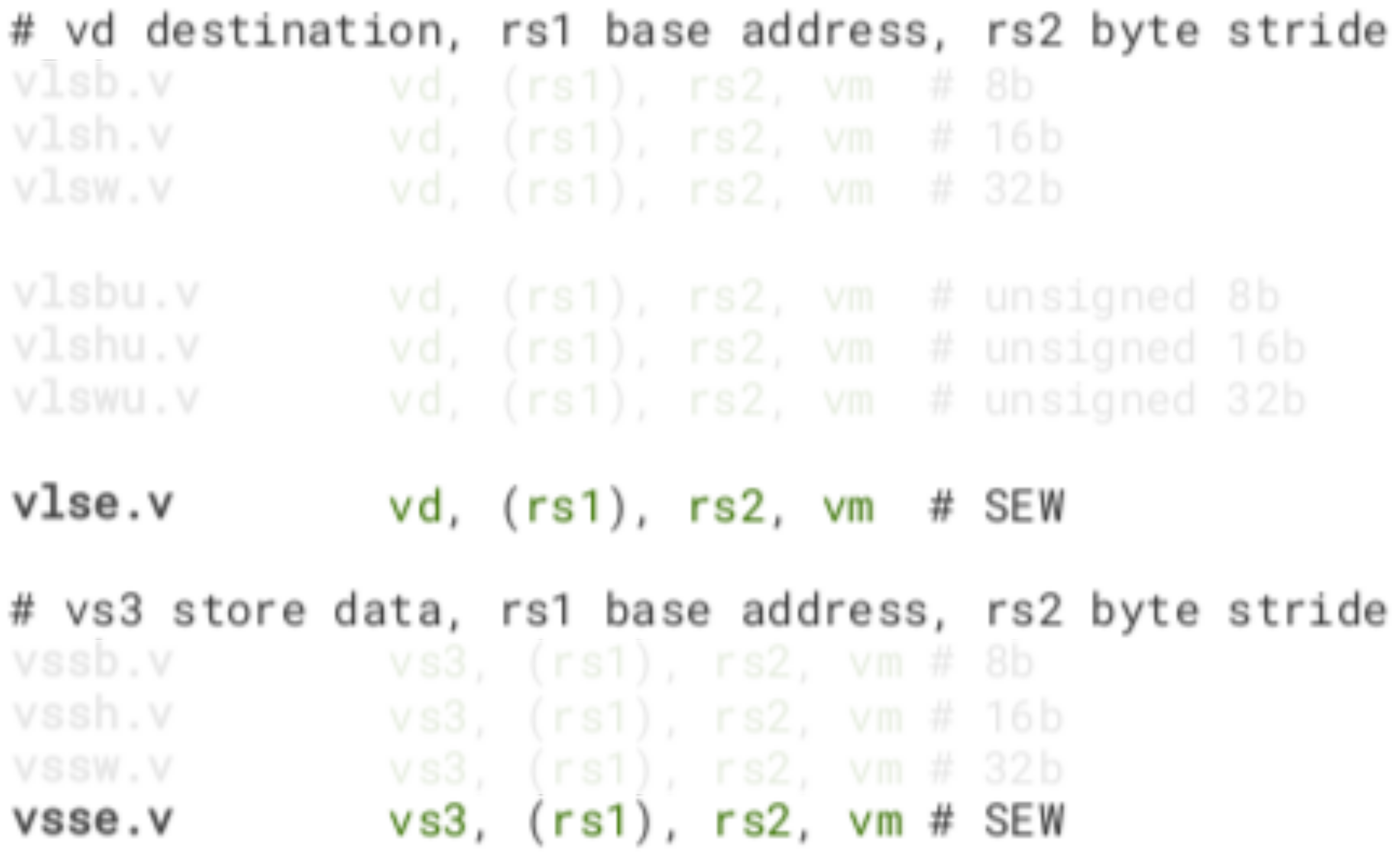

Vector store data
unit-stride
31 29 28 26 25 24 20 19 15 14 12 11 7 6 0
nf | 0 0 0 | vm | 0 0 0 0 0 | rs1 | width | vs3 | 0 1 0 0 1 1 1
mop | sumop | base address | store data | VSx
strided
31 29 28 26 25 24 20 19 15 14 12 11 7 6 0
nf | 0 1 0 | vm | rs2 | rs1 | width | vs3 | 0 1 0 0 1 1 1
mop | stride | base address | store data | VSSx
indexed
31 29 28 26 25 24 20 19 15 14 12 11 7 6 0
nf | mop | vm | vs2 | rs1 | width | vs3 | 0 1 0 0 1 1 1
0 1 1 | address offsets | base address | store data | VSXx ordered
1 1 1 | VSUXx unordered

Vector Unit-Stride Loads/Stores

```
# vd destination, rs1 base address, vm is mask encoding (v0.t or <missing>)
vlb.v          vd, (rs1), vm     # 8b signed
vlh.v          vd, (rs1), vm     # 16b signed
vlw.v          vd, (rs1), vm     # 32b signed

vlbu.v         vd, (rs1), vm     # 8b unsigned
vlhu.v         vd, (rs1), vm     # 16b unsigned
vlwu.v         vd, (rs1), vm     # 32b unsigned

vle.v          vd, (rs1), vm     # SEW

# vs3 store data, rs1 base address, vm is mask encoding (v0.t or <missing>)
vsb.v          vs3, (rs1), vm    # 8b store
vsh.v          vs3, (rs1), vm    # 16b store
vsw.v          vs3, (rs1), vm    # 32b store
vse.v          vs3, (rs1), vm    # SEW store
```

(These other shaded instructions dropped in v0.9)

Vector Strided Load/Store Instructions



```
# vd destination, rs1 base address, rs2 byte stride
vlsb.v       vd, (rs1), rs2, vm  # 8b
vlsh.v       vd, (rs1), rs2, vm  # 16b
vlsw.v       vd, (rs1), rs2, vm  # 32b

vlsbu.v      vd, (rs1), rs2, vm  # unsigned 8b
vlshu.v      vd, (rs1), rs2, vm  # unsigned 16b
vlswu.v      vd, (rs1), rs2, vm  # unsigned 32b

vlse.v       vd, (rs1), rs2, vm  # SEW

# vs3 store data, rs1 base address, rs2 byte stride
vssb.v       vs3, (rs1), rs2, vm # 8b
vssh.v       vs3, (rs1), rs2, vm # 16b
vssw.v       vs3, (rs1), rs2, vm # 32b
vsse.v       vs3, (rs1), rs2, vm # SEW
```

Vector Indexed Loads/Stores

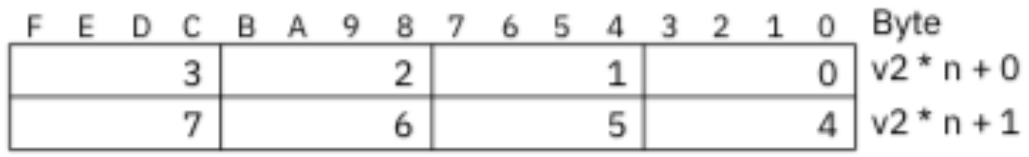
```
# vd destination, rs1 base address, vs2 indices
vlxb.v        vd, (rs1), vs2, vm  # 8b
vlxh.v        vd, (rs1), vs2, vm  # 16b
vlxw.v        vd, (rs1), vs2, vm  # 32b

vlxbu.v       vd, (rs1), vs2, vm  # 8b unsigned
vlxhu.v       vd, (rs1), vs2, vm  # 16b unsigned
vlxwu.v       vd, (rs1), vs2, vm  # 32b unsigned

vlxe.v        vd, (rs1), vs2, vm  # SEW

# Vector ordered-indexed store instructions
# vs3 store data, rs1 base address, vs2 indices
vsxb.v        vs3, (rs1), vs2, vm # 8b
vsxh.v        vs3, (rs1), vs2, vm # 16b
vsxw.v        vs3, (rs1), vs2, vm # 32b
vsxe.v        vs3, (rs1), vs2, vm # SEW

# Vector unordered-indexed store instructions
vsuxb.v       vs3, (rs1), vs2, vm # 8b
vsuxh.v       vs3, (rs1), vs2, vm # 16b
vsuxw.v       vs3, (rs1), vs2, vm # 32b
vsuxe.v       vs3, (rs1), vs2, vm # SEW
```

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 for 1) accommodate mixed-width operations, and/or 2) to increase efficiency by using longer vectors when fewer separate registers needed

- Set by **`vlmul[1:0]`** field in **`vtype`** during **`setvli`**

LMUL=2



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

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  # Vectors of 8b
  vle.v v0, (a1)                # Load bytes
    add a1, a1, t0              # Bump pointer
    sub a2, a2, t0              # Decrement count
  vse.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 bytes

Unit-stride vector store bytes

Combine eight vector registers into group

(v0,v1,…,v7)

Binary machine code can run on machines with any VLEN!

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
vrsub.vx vd, vs2, rs1, vm   # vd[i] = rs1 - vs2[i]
vrsub.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
vfsub.vv vd, vs2, vs1, vm   # Vector-vector
vfsub.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

CS152 Administrivia

- Per campus directions, CS152 will be graded P/NP by default
  - Instructors will maintain full grading information
  - Students can request letter grade if required, up to May 6 (RRR week)

- PS 4 due Friday April 3
- Lab 4 out on Friday April 3
- Lab 3 due Monday April 6

- Students can request extensions on PS and Labs
- Midterm 2 and final format TBD, date unlikely to change
- Krste's office hours now on request (likely 8am-9am)

CS252 Administrivia

- Grad students can modify grade to Satisfactory/Unsatisfactory (S/U) until Friday, May 8, 2020.
  - Dept/College relaxing rulings on course requirements (still TBD)

- Next week readings: Cray-1, VLIW & Trace Scheduling

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
```

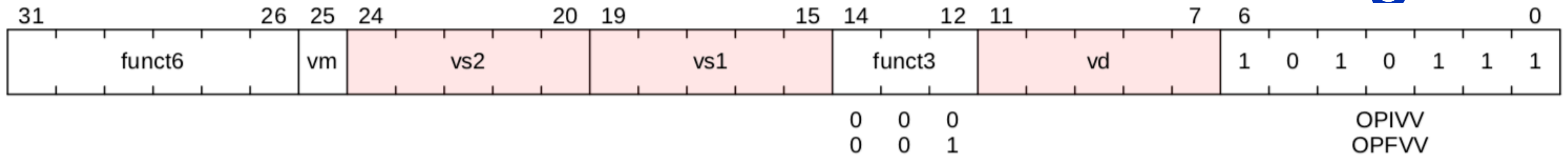
Mask Logical Operations

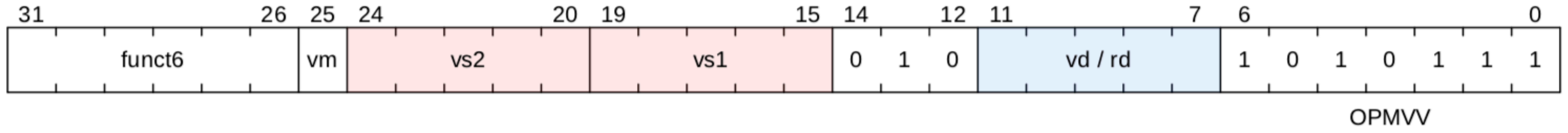
```
    vmand.mm vd, vs2, vs1     # vd[i] =   vs2[i].LSB &&  vs1[i].LSB
    vmnand.mm vd, vs2, vs1    # vd[i] = !(vs2[i].LSB &&  vs1[i].LSB)
    vmandnot.mm vd, vs2, vs1  # vd[i] =   vs2[i].LSB && !vs1[i].LSB
    vmxor.mm  vd, vs2, vs1    # vd[i] =   vs2[i].LSB ^^  vs1[i].LSB
    vmor.mm  vd, vs2, vs1     # vd[i] =   vs2[i].LSB ||  vs1[i].LSB
    vmnor.mm  vd, vs2, vs1    # vd[i] = !(vs2[i[.LSB ||  vs1[i].LSB)
    vmornot.mm  vd, vs2, vs1  # vd[i] =   vs2[i].LSB || !vs1[i].LSB
    vmxnor.mm vd, vs2, vs1    # vd[i] = !(vs2[i].LSB ^^  vs1[i].LSB)
```

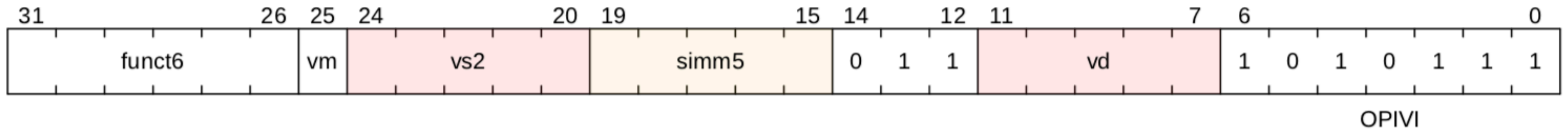
Several assembler pseudoinstructions are defined as shorthand for common uses of mask logical operations:

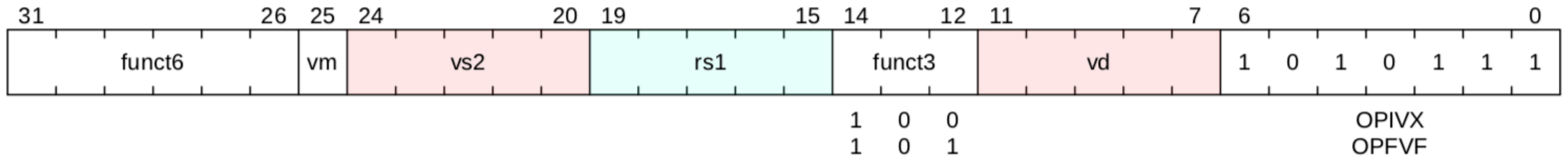
```
    vmcpy.m vd, vs  => vmand.mm vd, vs, vs  # Copy mask register
    vmclr.m vd     => vmxor.mm vd, vd, vd   # Clear mask register
    vmset.m vd     => vmxnor.mm vd, vd, vd  # Set mask register
    vmnot.m vd, vs => vmnand.mm vd, vs, vs  # Invert bits
```

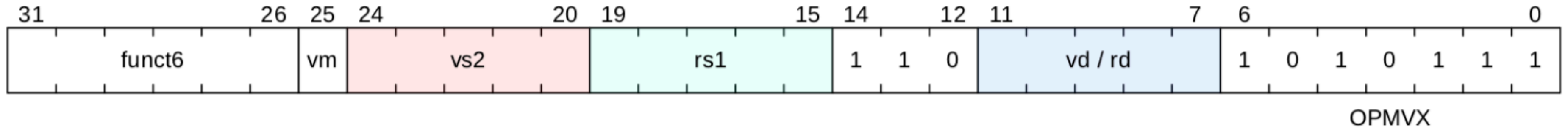
Vector Arithmetic Instruction Encodings











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
vfwmsac.vv vd, vs1, vs2, vm    # vd[i] = +(vs1[i] * vs2[i]) - vd[i]
vfwmsac.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*16b -> 32b product
- Want same number of elements in each vector register, even if different bits/element
- Solution: Keep SEW/LMUL constant

VLEN=128b

SEW/LMUL=8

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

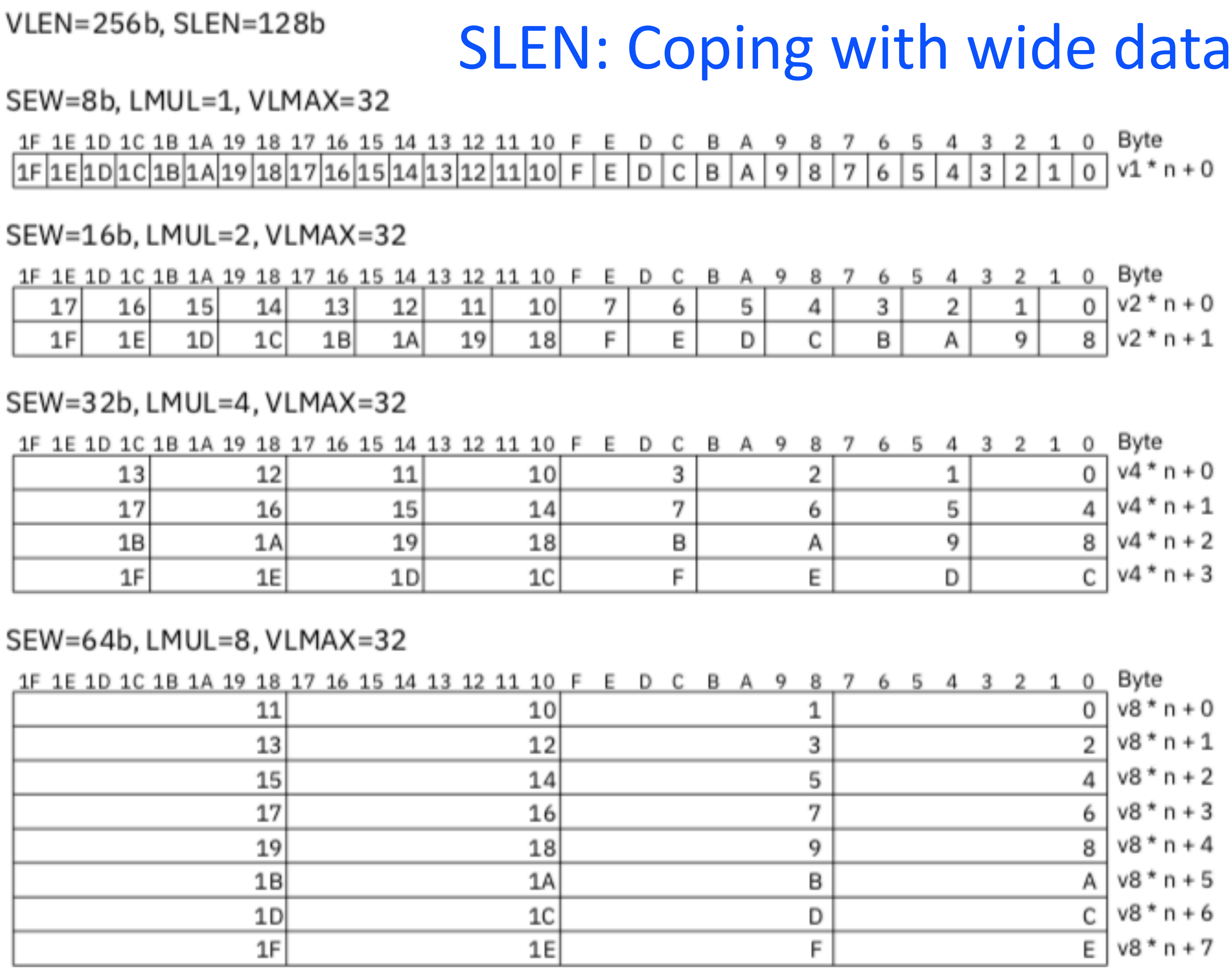
SEW=32b, LMUL=4, VLMAX=16

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

SLEN: Coping with wide datapaths



VLEN=256b, SLEN=128b

SEW=8b, LMUL=1, VLMAX=32

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	Byte
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	v1 * n + 0

SEW=16b, LMUL=2, VLMAX=32

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	Byte
17	16	15	14	13	12	11	10	7	6	5	4	3	2	1	0	v2 * n + 0
1F	1E	1D	1C	1B	1A	19	18	F	E	D	C	B	A	9	8	v2 * n + 1

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

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	Byte
13	12	11	10	3	2	1	0	v4 * n + 0
17	16	15	14	7	6	5	4	v4 * n + 1
1B	1A	19	18	B	A	9	8	v4 * n + 2
1F	1E	1D	1C	F	E	D	C	v4 * n + 3

SEW=64b, LMUL=8, VLMAX=32

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	Byte
11	10	1	0	v8 * n + 0
13	12	3	2	v8 * n + 1
15	14	5	4	v8 * n + 2
17	16	7	6	v8 * n + 3
19	18	9	8	v8 * n + 4
1B	1A	B	A	v8 * n + 5
1D	1C	D	C	v8 * n + 6
1F	1E	F	E	v8 * n + 7

- SLEN is design parameter, so implementers can reduce wiring in their design when SLEN<VLEN
- Unless datapath very wide (>128b) will set SLEN=VLEN

Mask Register Layout

- Masks always held in a single vector register
- All bits written on compare, only LSB considered as mask
- Size of each field, MLEN, is SEW/LMUL
  - E.g.1, SEW=8b, LMUL=8, MLEN=1b
  - E.g.2, SEW=64b, LMUL=1, MLEN=64b
- For mixed-precision loops with constant SEW/LMUL, mask values always ”line up” at each element

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
    vle.v v0, (a1)
    sub a0, a0, a4
    slli a4, a4, 2
    add a1, a1, a4
    vle.v v8, (a2)
    vfmacc.vf v8, fa0, v0
    vse.v v8, (a2)
    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   # Use 8b elements.
    vle.v v0, (a1)          # Get x[i]
      sub a0, a0, t0        # Decrement element count
      add a1, a1, t0        # x[i] Bump pointer
    vmslt.vi v0, v0, 5      # Set mask in v0
      slli t0, t0, 1        # Multiply by 2 bytes
    vsetvli t0, a0, e16,m2  # Use 16b elements.
    vle.v v1, (a2), v0.t    # z[i] = a[i] case
    vmnot.m v0, v0          # Invert v0
      add a2, a2, t0        # a[i] bump pointer
    vle.v v1, (a3), v0.t    # z[i] = b[i] case
      add a3, a3, t0        # b[i] bump pointer
    vse.v v1, (a4)          # Store z
      add a4, a4, t0        # z[i] bump pointer
      bnez a0, loop
```

Creative Commons Licence