计算机组成与系统结构
Computer Organization & System Architecture

Huang Kejie (黄科杰) 百人计划研究员

Office: 玉泉校区老生仪楼 304

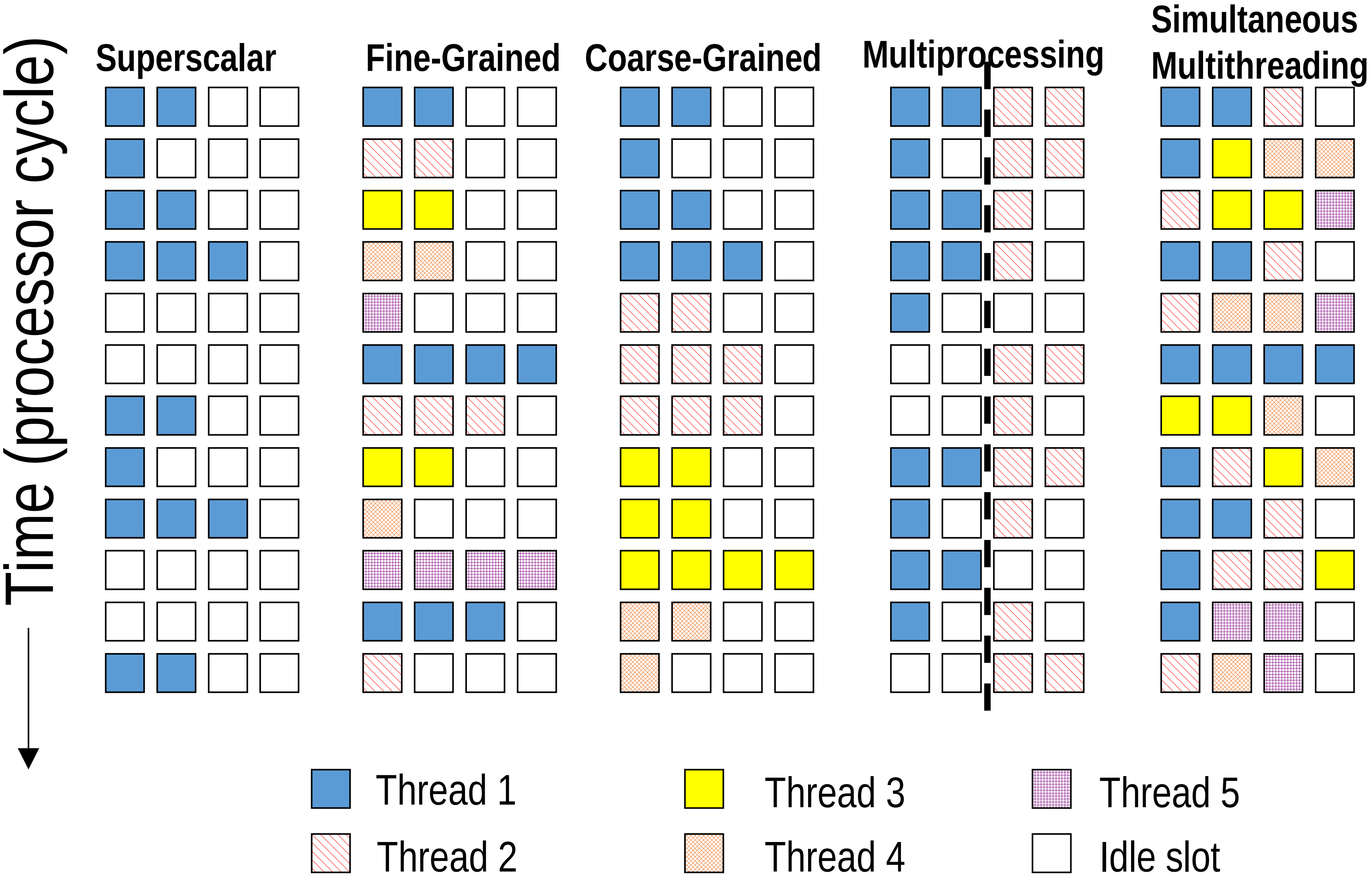
Email address: huangkejie@zju.edu.cn

HP: 17706443800

Project Presentation

- The project presentation. Contents may cover but not limit to
  - The contribution of each member
  - Design review of MP2
    - The implementation of MP2
    - Test benches and simulation results
    - FPGA implementation and timing analysis
  - Proposal for MP3 (no need to write SV code)
    - The design plan to add 5-stage pipeline in MP2
    - The conflicts may meet in this design
    - Your solutions to address issues
    - Optimization in terms of area and speed
- Presentation date: Dec 16 and Dec 18
- 2 person in group, each group will have 12 minutes to present the work
- 1st project: 10 points, 2nd project: 20 points, presentation: 10 points

ReCap: Multithreading



Time (processor cycle)

Superscalar

Fine-Grained

Coarse-Grained

Multiprocessing

Simultaneous Multithreading

Thread 1

Thread 2

Thread 3

Thread 4

Thread 5

Idle slot

Supercomputer Applications

- Typical application areas
  - Military research (nuclear weapons, cryptography)
  - Scientific research
  - Weather forecasting
  - Oil exploration
  - Industrial design (car crash simulation)
  - Bioinformatics
  - Cryptography

- All involve huge computations on large data set

- Supercomputers: CDC6600, CDC7600, Cray-1, …

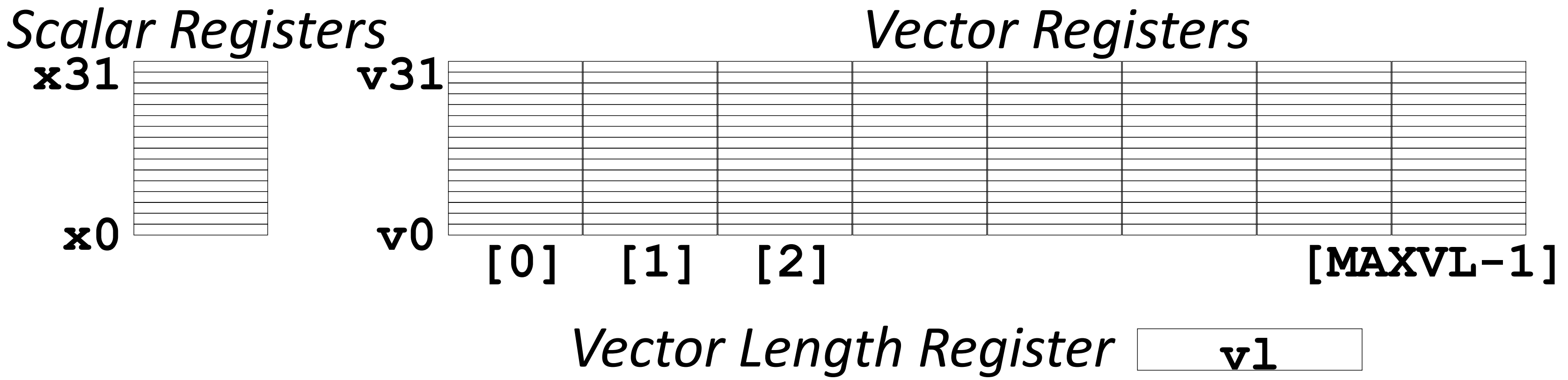
- In 70s-80s, Supercomputer ≡ Vector Machine

Vector Supercomputers

[©Cray Research, 1976]

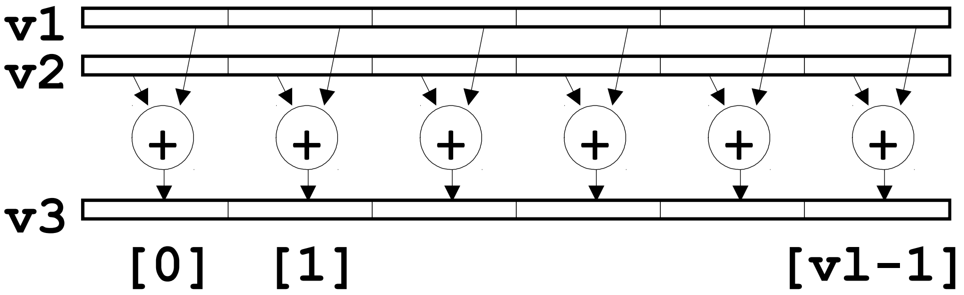
- Epitomized by Cray-1, 1976:
- Scalar Unit
  - Load/Store Architecture
  - Vector Extension
  - Vector Registers
  - Vector Instructions
- Implementation
  - Hardwired Control
  - Highly Pipelined Functional Units
  - Interleaved Memory System
  - No Data Caches
  - No Virtual Memory

Vector Programming Model



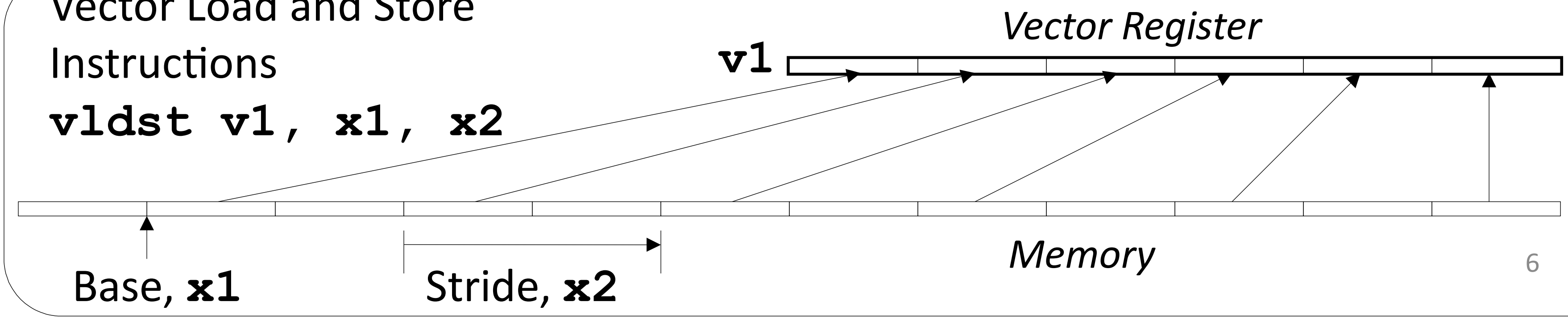
Vector Arithmetic Instructions

vadd v3, v1, v2



Vector Load and Store Instructions

vldst v1, x1, x2



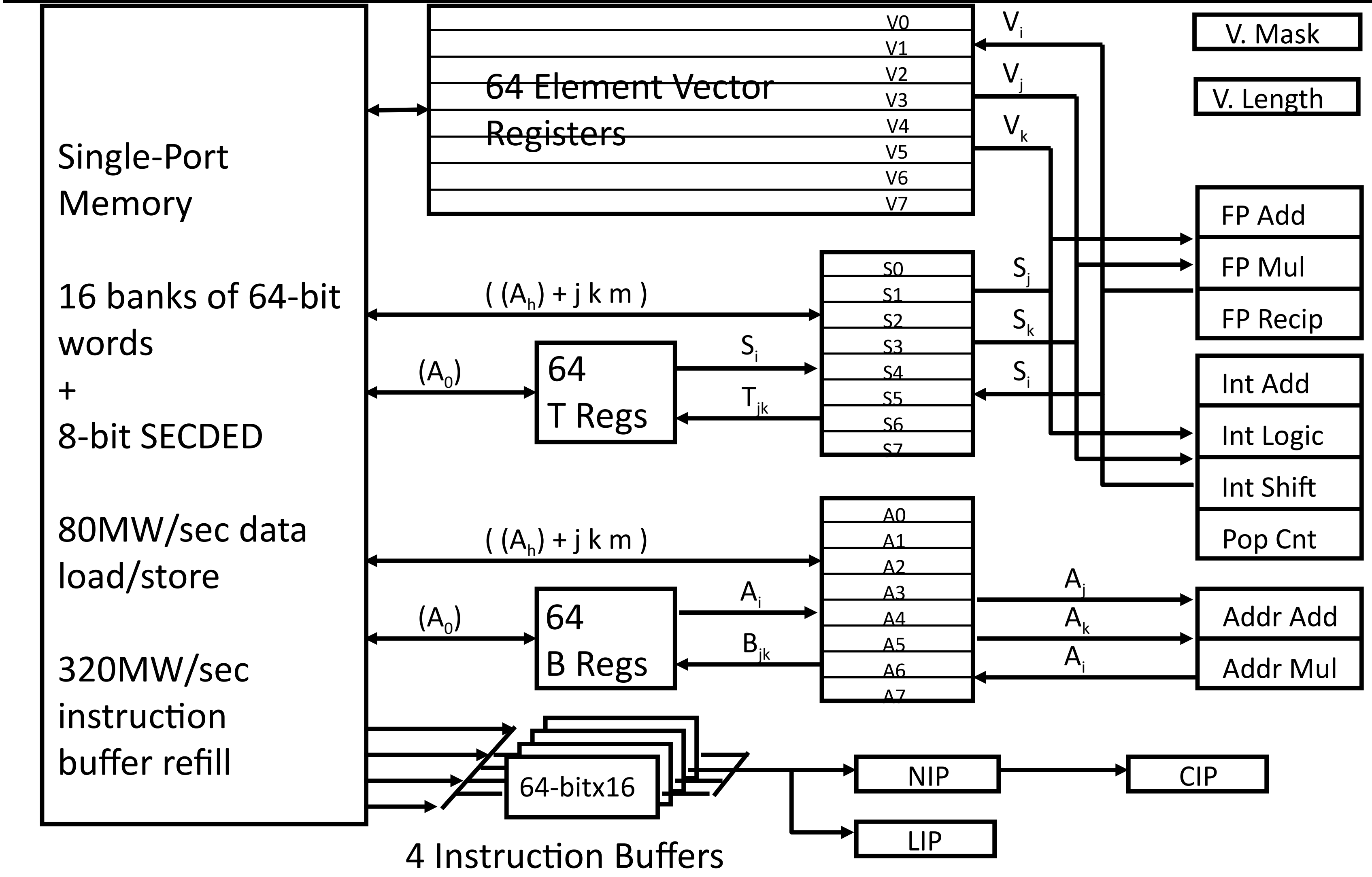
Vector Code Example

```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# Scalar Code
  li x4, 64
loop:
  fld f1, 0(x1)
  fld f2, 0(x2)
  fadd.d f3,f1,f2
  fsd f3, 0(x3)
  addi x1, 8
  addi x2, 8
  addi x3, 8
  subi x4, 1
  bnez x4, loop
```

```
# Vector Code
  li x4, 64
  setvl x4
  vld v1, x1
  vld v2, x2
  vadd v3,v1,v2
  vst v3, x3
```

Cray-1 (1976)



Single-Port Memory

16 banks of 64-bit words
+
8-bit SECDED

80MW/sec data load/store

320MW/sec instruction buffer refill

64 Element Vector Registers

V0
V1
V2
V3
V4
V5
V6
V7

V_i
V_j
V_k

V. Mask

V. Length

((A_h) + j k m)

(A_0)

64
T Regs

S_i
T_{jk}

S0
S1
S2
S3
S4
S5
S6
S7

S_j
S_k
S_i

FP Add
FP Mul
FP Recip

Int Add
Int Logic
Int Shift
Pop Cnt

((A_h) + j k m)

(A_0)

64
B Regs

A_i
B_{jk}

A0
A1
A2
A3
A4
A5
A6
A7

A_j
A_k
A_i

Addr Add
Addr Mul

64-bitx16

4 Instruction Buffers

NIP
CIP
LIP

memory bank cycle 50 ns *processor cycle* 12.5 ns (80MHz)

Vector Instruction Set Advantages

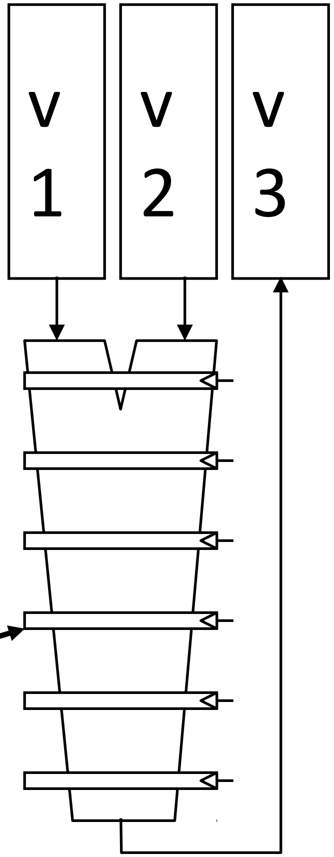
- Compact
  - one short instruction encodes N operations

- Expressive, tells hardware that these N operations:
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in same pattern as previous instructions
  - access a contiguous block of memory (unit-stride load/store)
  - access memory in a known pattern (strided load/store)

- Scalable
  - can run same code on more parallel pipelines (lanes)

Vector Arithmetic Execution

- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)

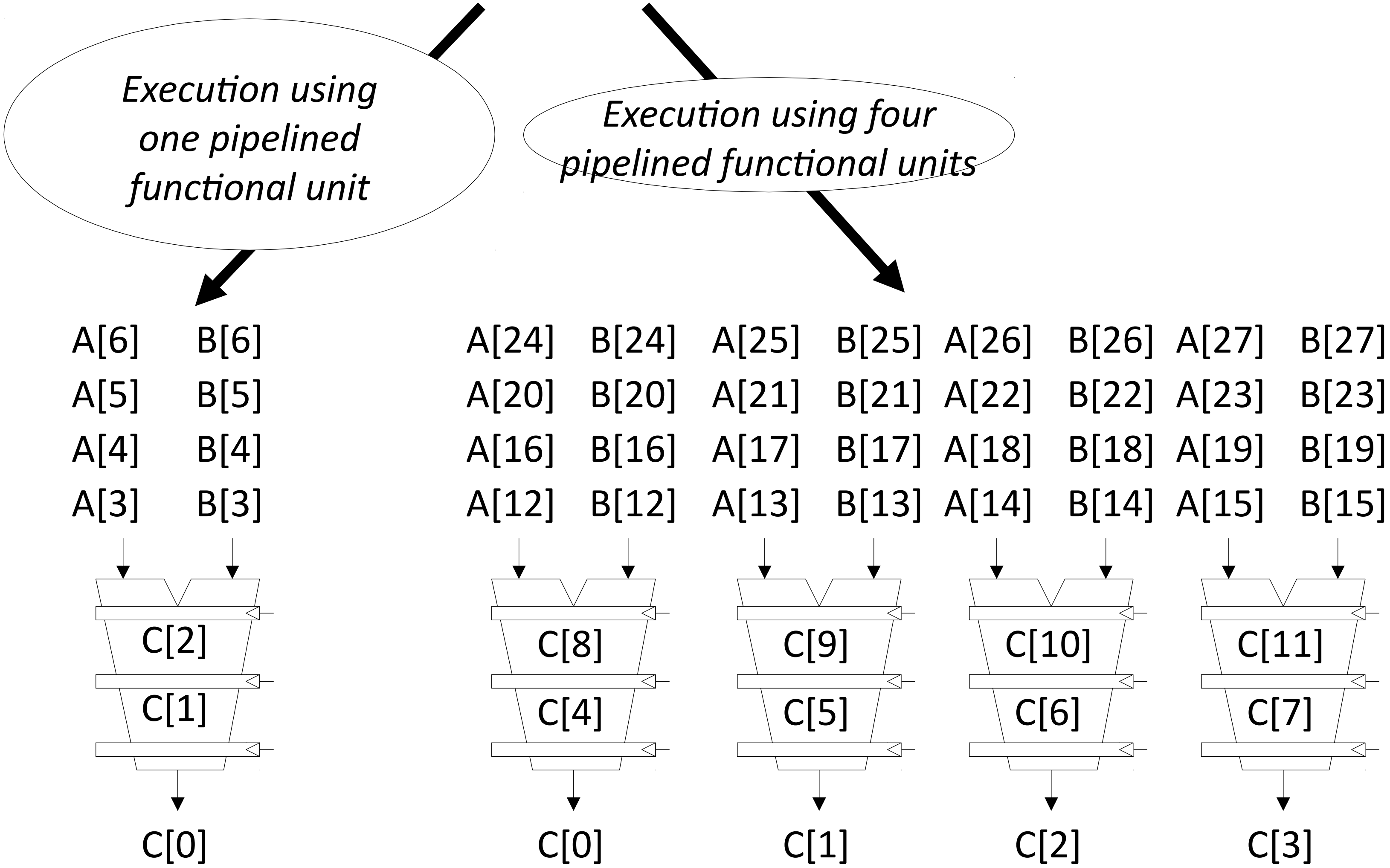


Six-stage multiply pipeline

v3 <- v1 * v2

Vector Instruction Execution

```
vadd vc, va, vb
```



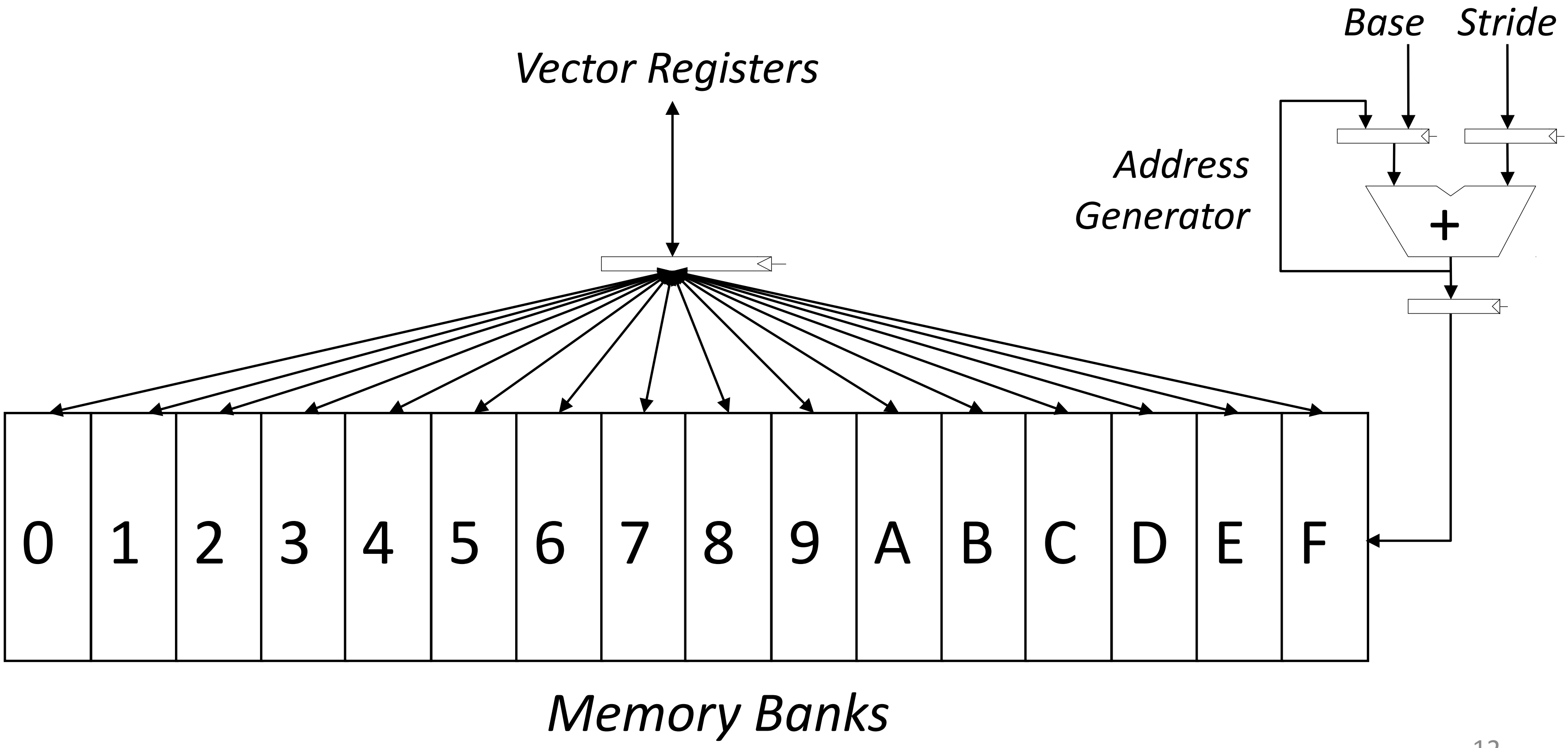
Execution using one pipelined functional unit

Execution using four pipelined functional units

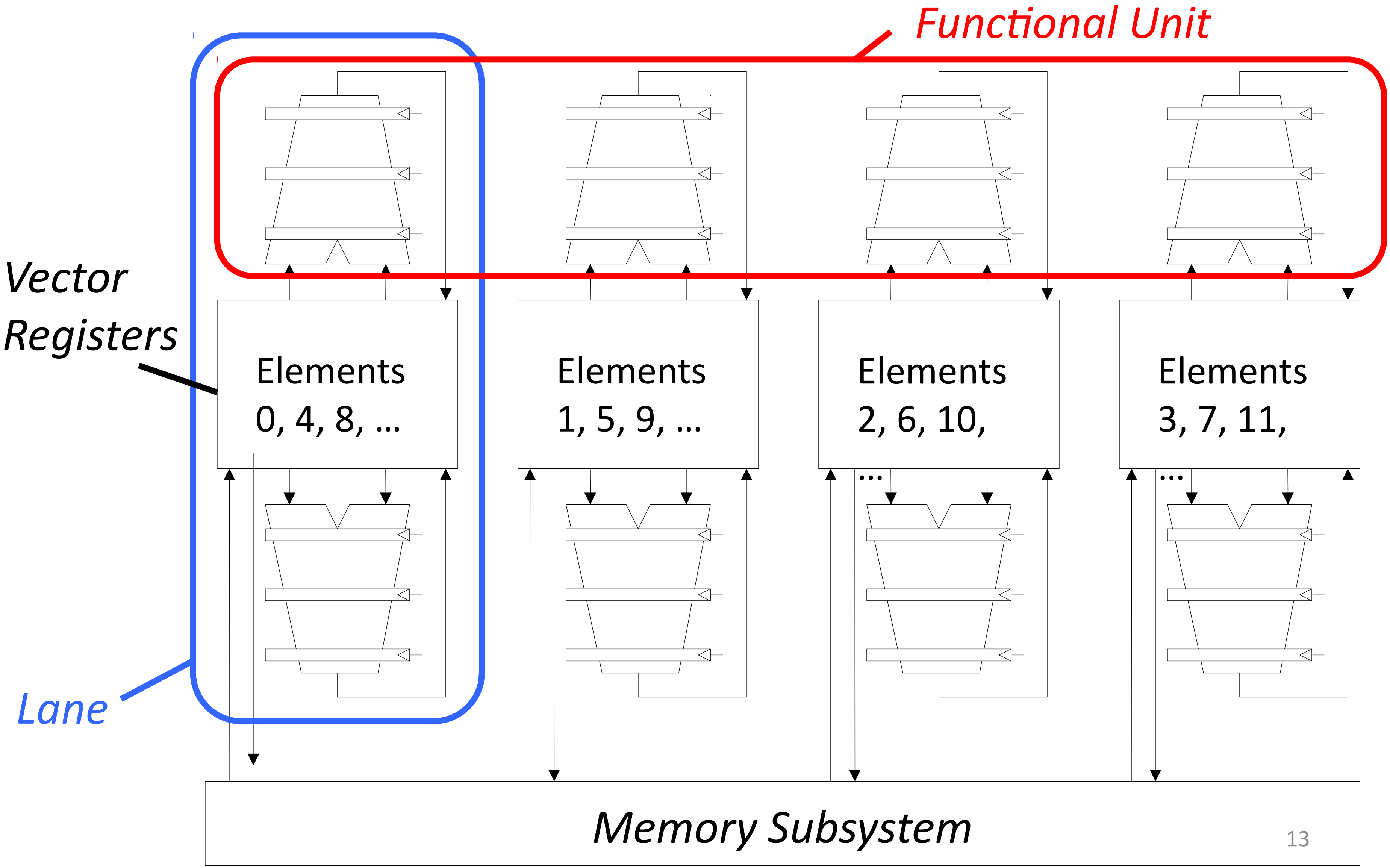
A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]
C[2]
C[1]
C[0]
A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]
C[8] C[9] C[10] C[11]
C[4] C[5] C[6] C[7]
C[0] C[1] C[2] C[3]

Interleaved Vector Memory System

- Bank busy time: Time before bank ready to accept next request
- Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

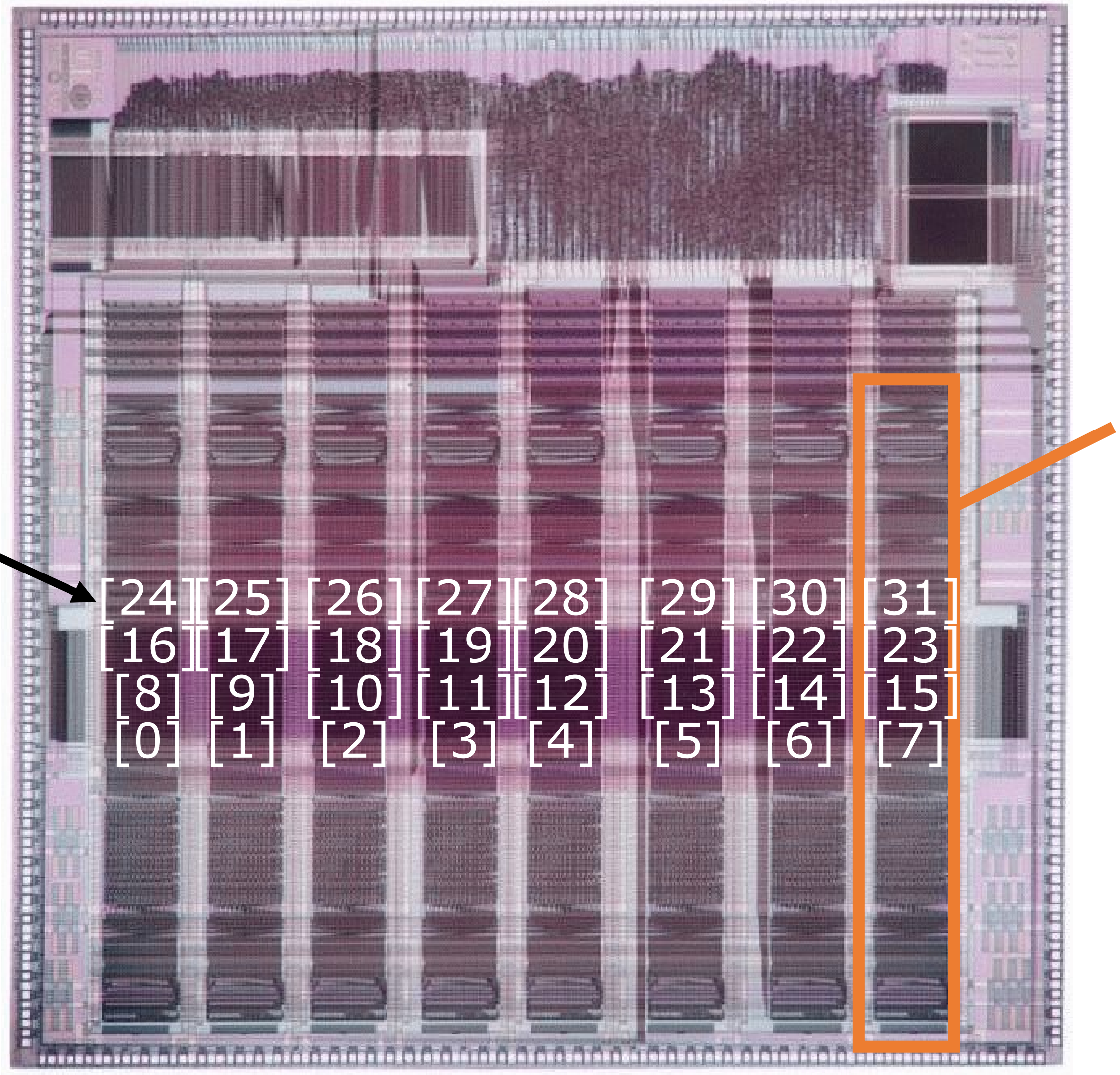


Vector Unit Structure



T0 Vector Microprocessor (UCB/ICSI, 1995)

Vector register elements striped over lanes

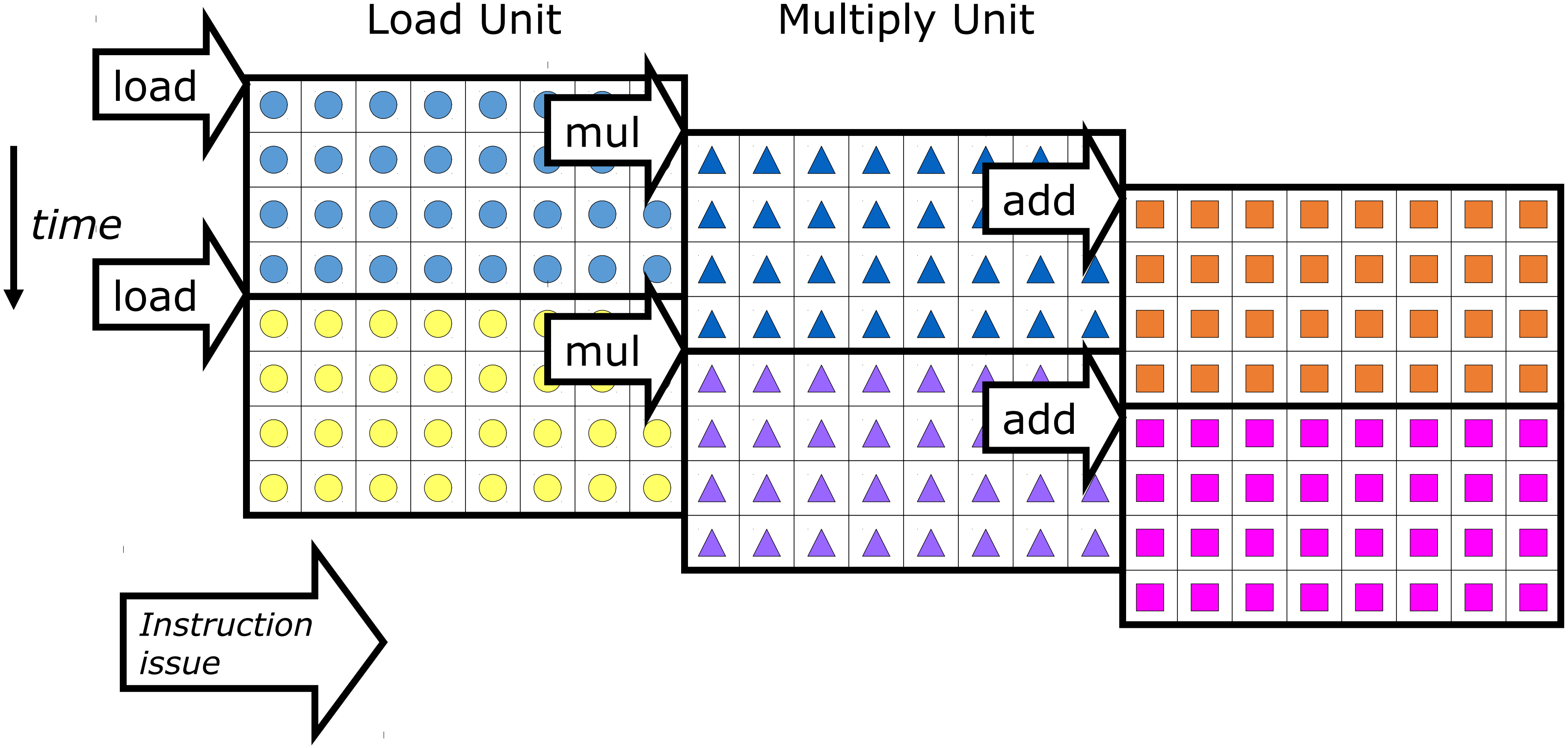


[24]	[25]	[26]	[27]	[28]	[29]	[30]	[31]
[16]	[17]	[18]	[19]	[20]	[21]	[22]	[23]
[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Lane

Vector Instruction Parallelism

- Can overlap execution of multiple vector instructions
  - example machine has 32 elements per vector register and 8 lanes

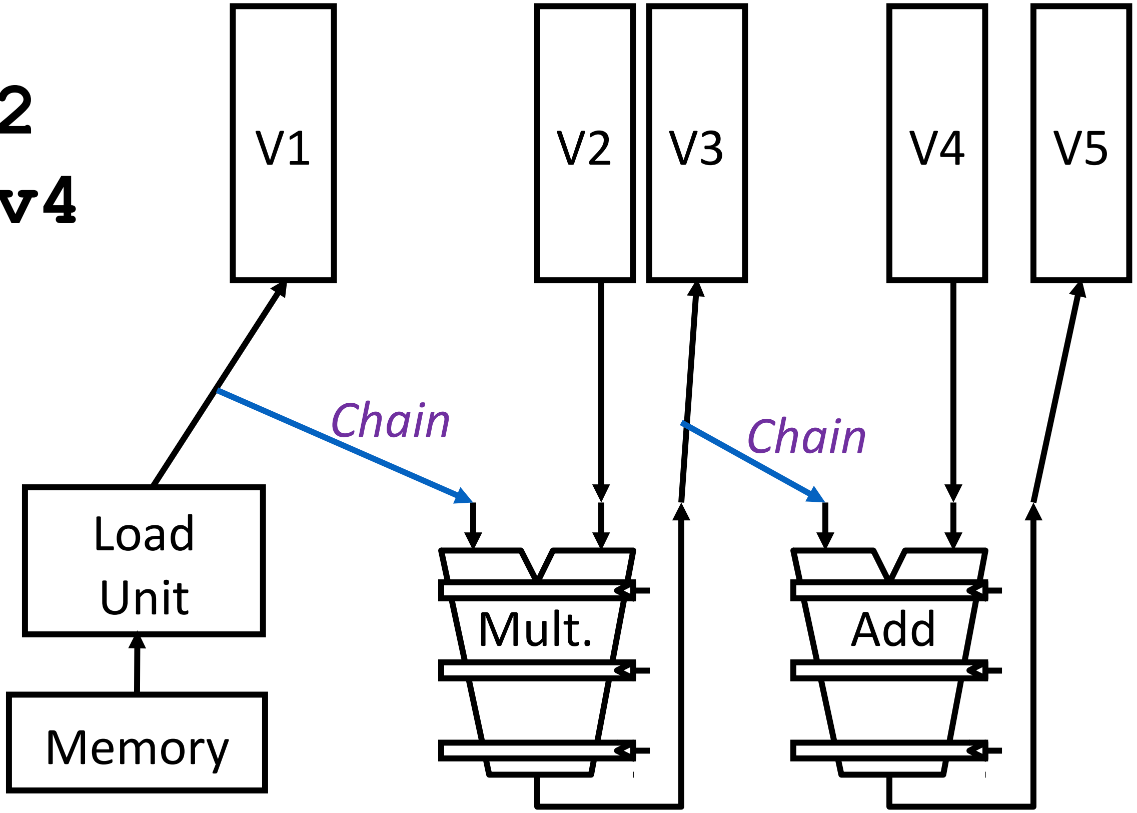


Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Chaining

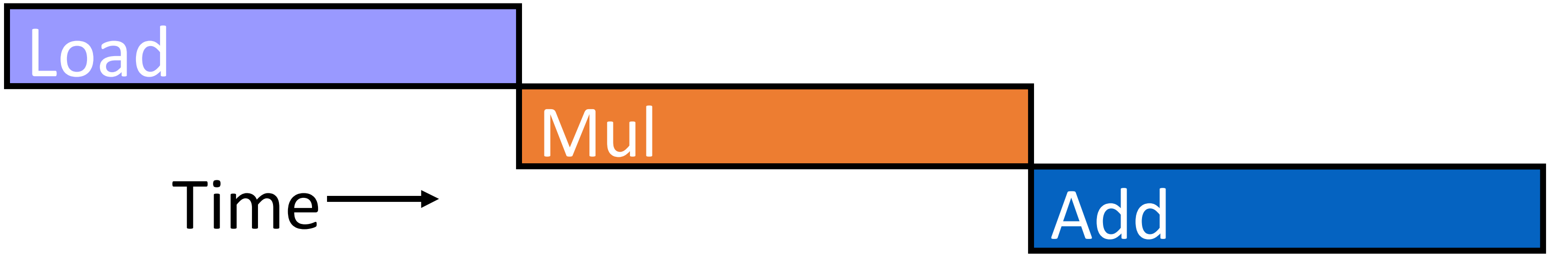
- Vector version of register bypassing
  - introduced with Cray-1

```
vld  v1
vmul v3,v1,v2
vadd v5,v3,v4
```

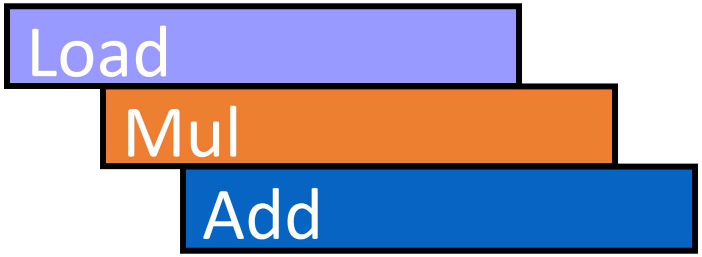


Vector Chaining Advantage

- Without chaining, must wait for last element of result to be written before starting dependent instruction

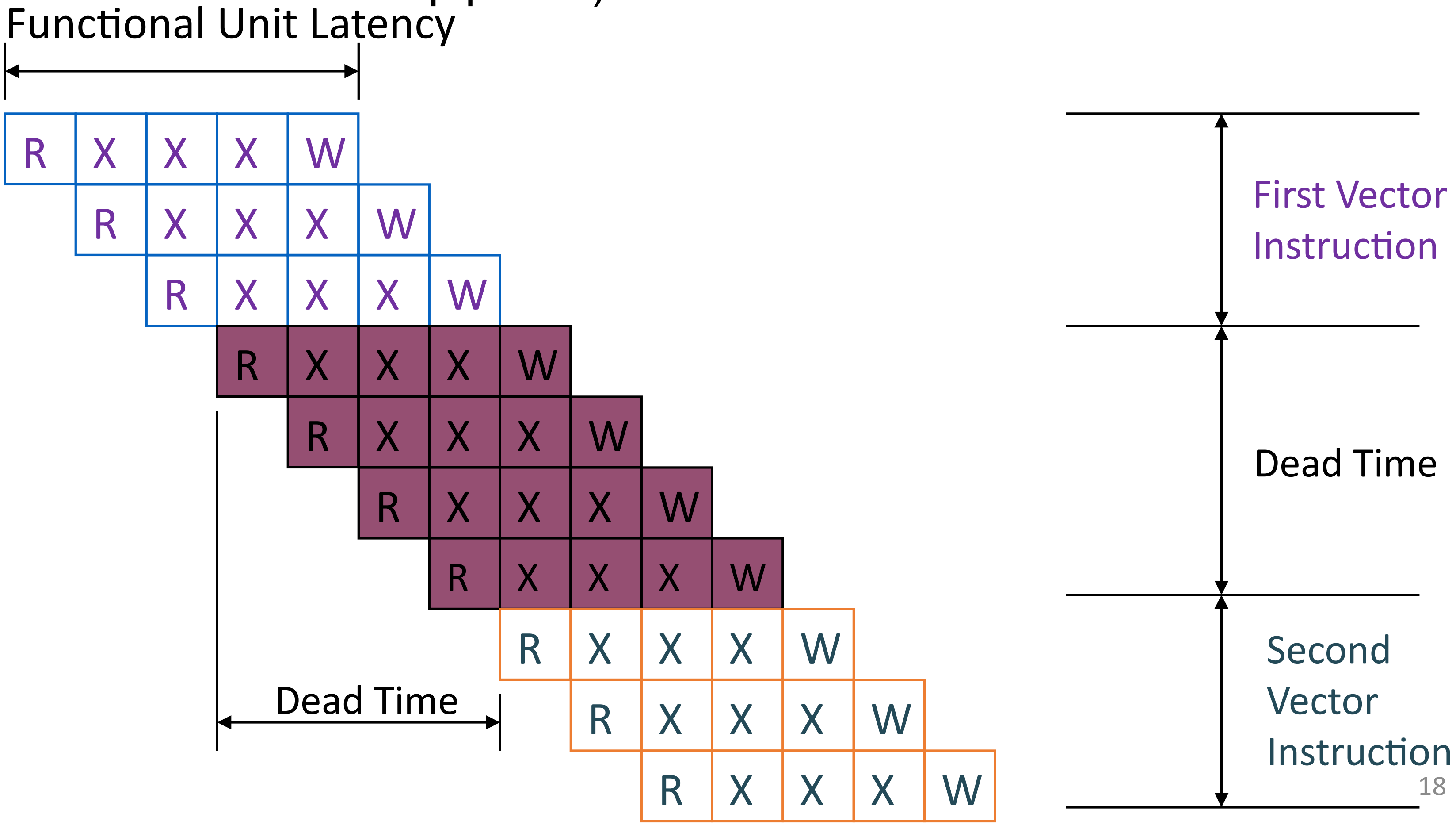


- With chaining, can start dependent instruction as soon as first result appears

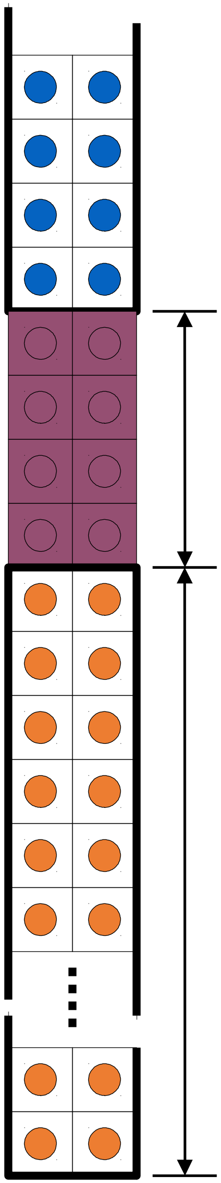


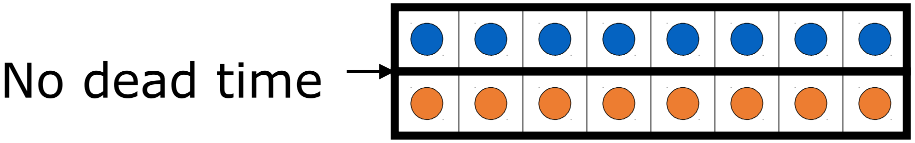
Vector Startup

- Two components of vector startup penalty
  - functional unit latency (time through pipeline)
  - dead time or recovery time (time before another vector instruction can start down pipeline)



Dead Time and Short Vectors



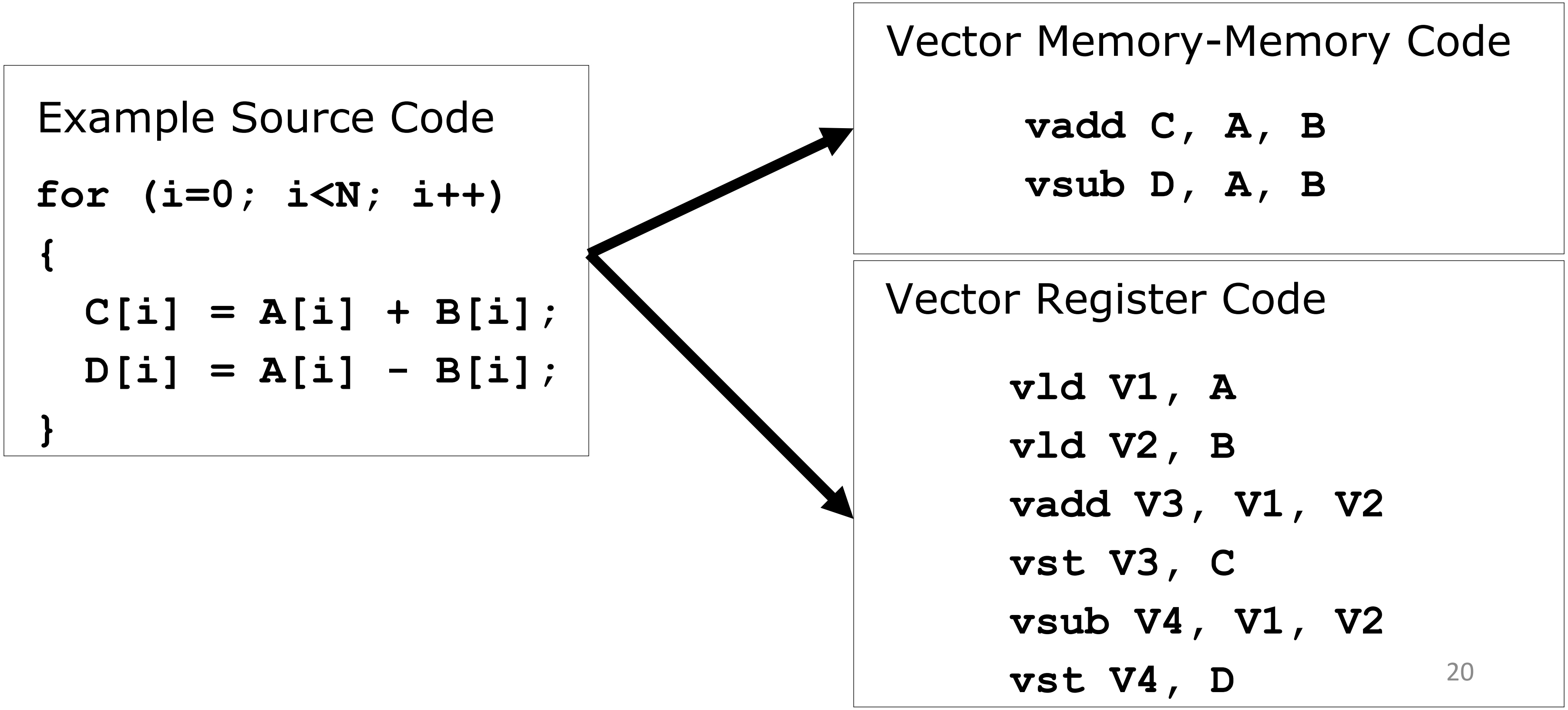


T0, Eight lanes
No dead time
100% efficiency with 8-element vectors

Cray C90, Two lanes
4-cycle dead time
Maximum efficiency 94% with 128-element vectors

Vector Memory-Memory versus Vector Register Machines

- Vector memory-memory instructions hold all vector operands in main memory
- The first vector machines, CDC Star-100 (‘73) and TI ASC (‘71), were memory-memory machines
- Cray-1 (’76) was first vector register machine



Example Source Code

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

Vector Memory-Memory Code

```
vadd C, A, B
vsub D, A, B
```

Vector Register Code

```
vld V1, A
vld V2, B
vadd V3, V1, V2
vst V3, C
vsub V4, V1, V2
vst V4, D
```

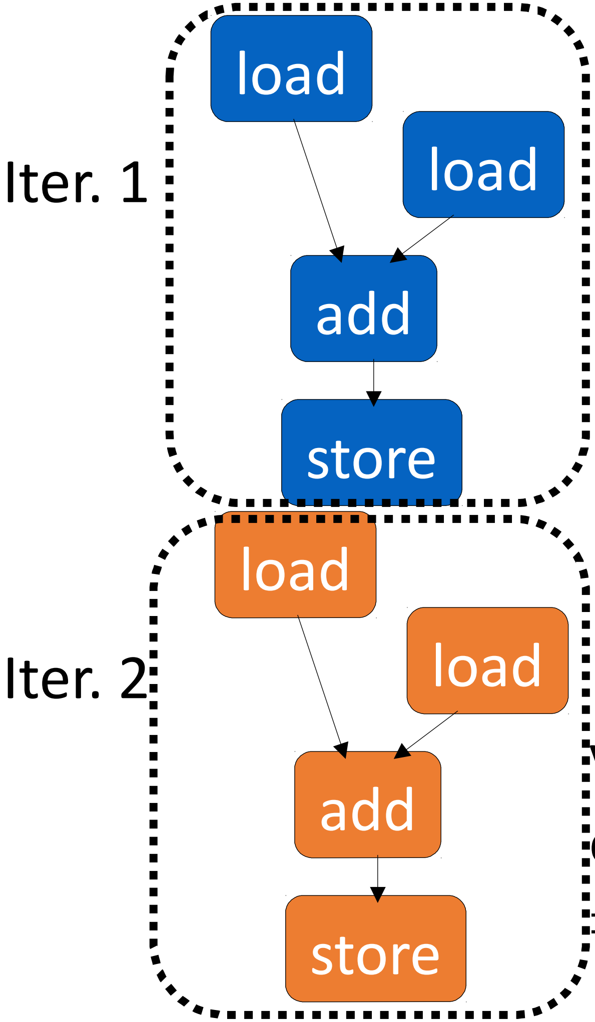
Vector Memory-Memory vs. Vector Register Machines

- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
  - All operands must be read in and out of memory
- VMMAs make if difficult to overlap execution of multiple vector operations, why?
  - Must check dependencies on memory addresses
- VMMAs incur greater startup latency
  - Scalar code was faster on CDC Star-100 for vectors < 100 elements
  - For Cray-1, vector/scalar breakeven point was around 2-4 elements
- Apart from CDC follow-ons (Cyber-205, ETA-10) all major vector machines since Cray-1 have had vector register architectures
  - (we ignore vector memory-memory from now on)

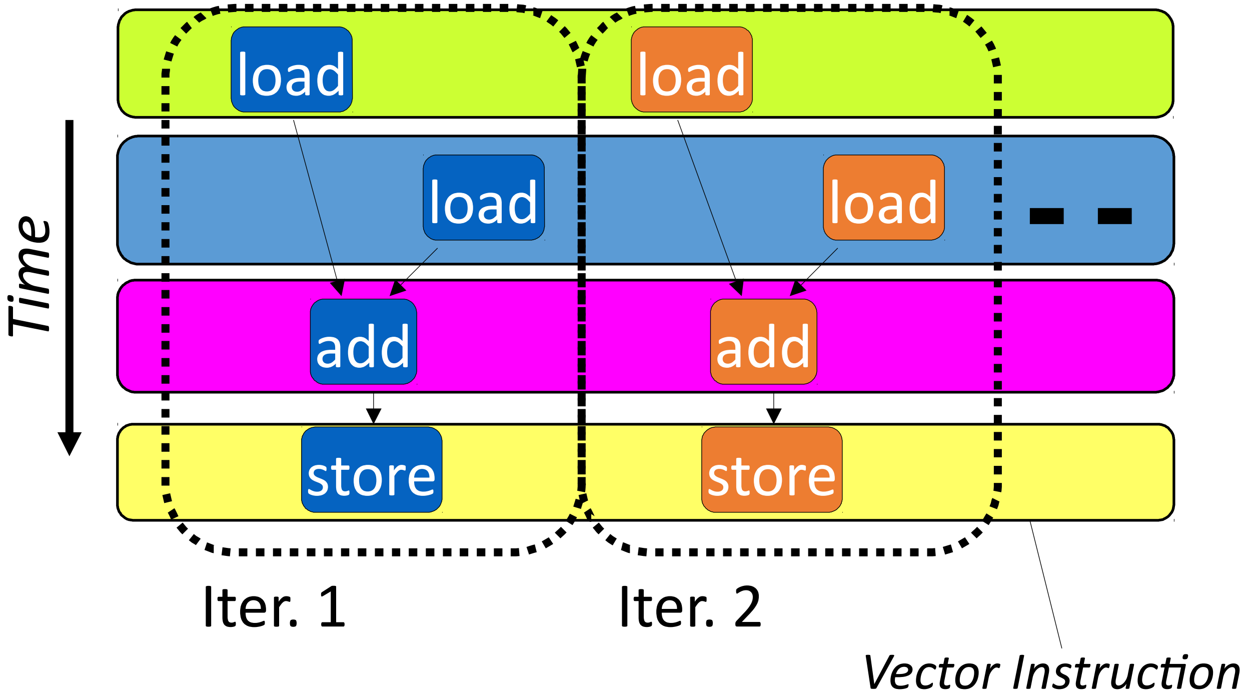
Automatic Code Vectorization

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

Scalar Sequential Code



Vectorized Code



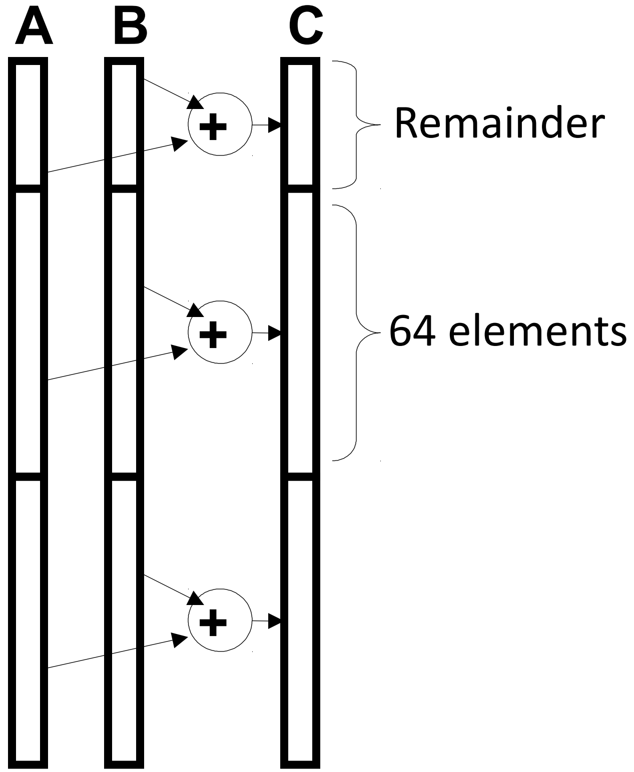
Vectorization is a massive compile-time reordering of operation sequencing

⇒ requires extensive loop-dependence analysis

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

Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:

```
for (i=0; i<N; i++)
    if (A[i]>0) then
        A[i] = B[i];
```

Solution: Add vector *mask* (or *flag*) registers

– vector version of predicate registers, 1 bit per element

…and *maskable* vector instructions

– vector operation becomes bubble ("NOP") at elements where mask bit is clear

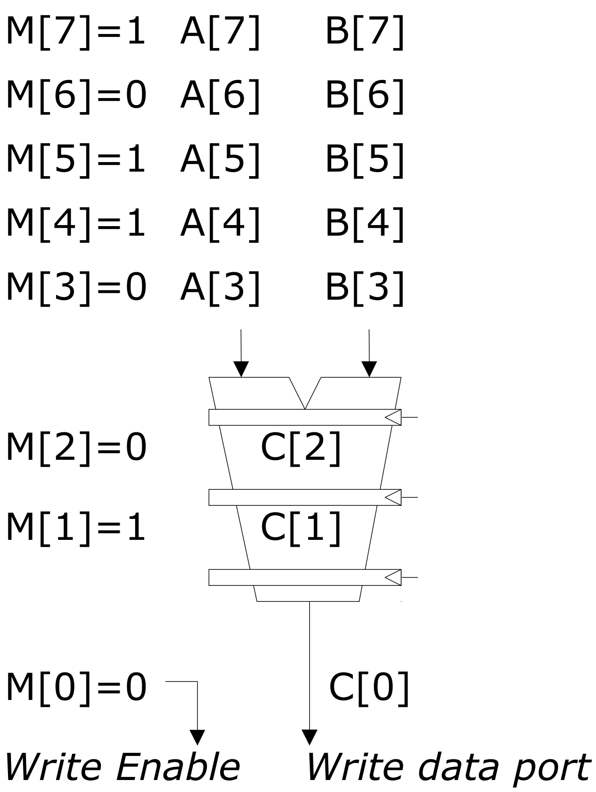
Code example:

```
cvm             # Turn on all elements
vld vA, xA      # Load entire A vector
vgt vA, f0      # Set bits in mask register where A>0
vld vA, xB      # Load B vector into A under mask
vst vA, xA      # Store A back to memory under mask
```

Masked Vector Instructions

Simple Implementation

- execute all N operations, turn off result writeback according to mask



M[7]=1 A[7] B[7]

M[6]=0 A[6] B[6]

M[5]=1 A[5] B[5]

M[4]=1 A[4] B[4]

M[3]=0 A[3] B[3]

M[2]=0 C[2]

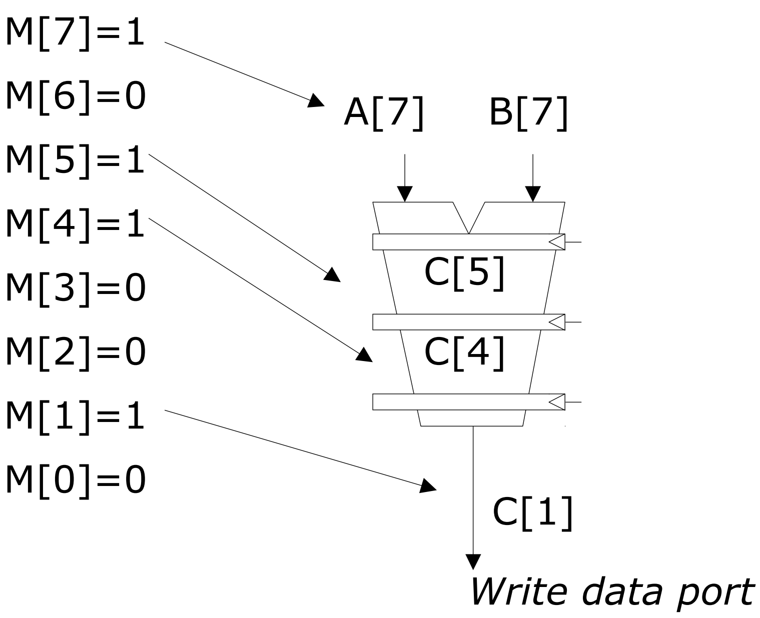
M[1]=1 C[1]

M[0]=0 C[0]

Write Enable *Write data port*

Density-Time Implementation

- scan mask vector and only execute elements with non-zero masks



M[7]=1

M[6]=0

M[5]=1

M[4]=1

M[3]=0

M[2]=0

M[1]=1

M[0]=0

A[7] B[7]

C[5]

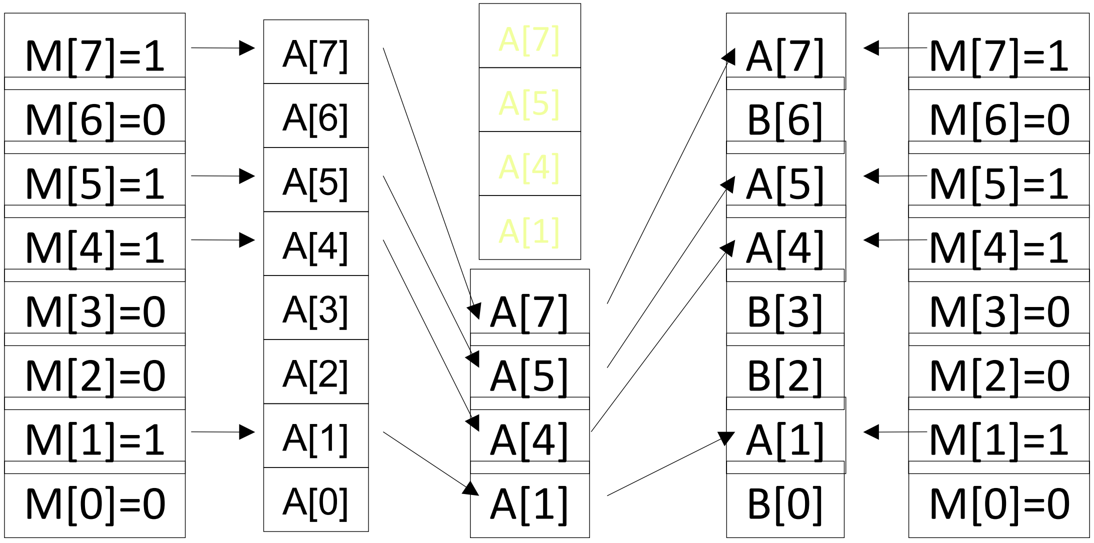
C[4]

C[1]

Write data port

Compress/Expand Operations

- Compress packs non-masked elements from one vector register contiguously at start of destination vector register
  - population count of mask vector gives packed vector length
- Expand performs inverse operation



Compress *Expand*

Used for density-time conditionals and also for general selection operations

Vector Reductions

Problem: Loop-carried dependence on reduction variables

```
sum = 0;
for (i=0; i<N; i++)
    sum += A[i];  # Loop-carried dependence on
  sum
```

Solution: Re-associate operations if possible, use binary tree to perform reduction

```
# Rearrange as:
sum[0:VL-1] = 0          # Vector of VL partial sums
for(i=0; i<N; i+=VL)     # Stripmine VL-sized chunks
    sum[0:VL-1] += A[i:i+VL-1]; # Vector sum
# Now have VL partial sums in one vector register
do {
    VL = VL/2;            # Halve vector length
    sum[0:VL-1] += sum[VL:2*VL-1] # Halve no. of
  partials
} while (VL>1)
```

Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

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

Indexed load instruction (*Gather*)

```
vld vD, xD          # Load indices in D vector
vldx vC, xC, vD     # Load indirect from xC base
vld vB, xB          # Load B vector
vadd vA,vB,vC       # Do add
vst vA, xA          # Store result
```

Vector Instruction Set Advantages

- Many vector machines have a very relaxed memory model, e.g.

```
vst v1, x1    # Store vector to x1
vld v2, x1    # Load vector from x1
```

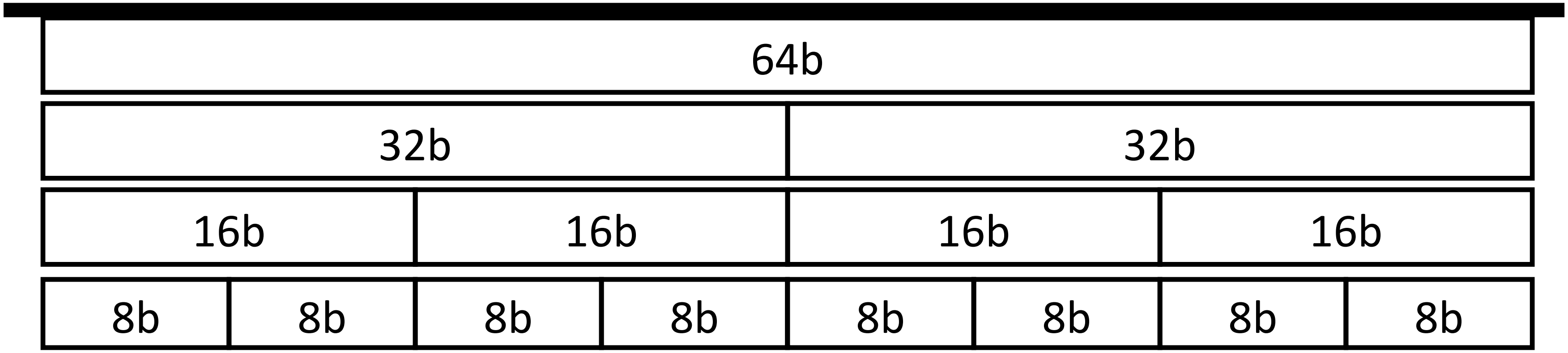
  - No guarantee that elements of v2 will have value of elements of v1 even when store and load execute by same processor!

- Requires explicit memory barrier or fence

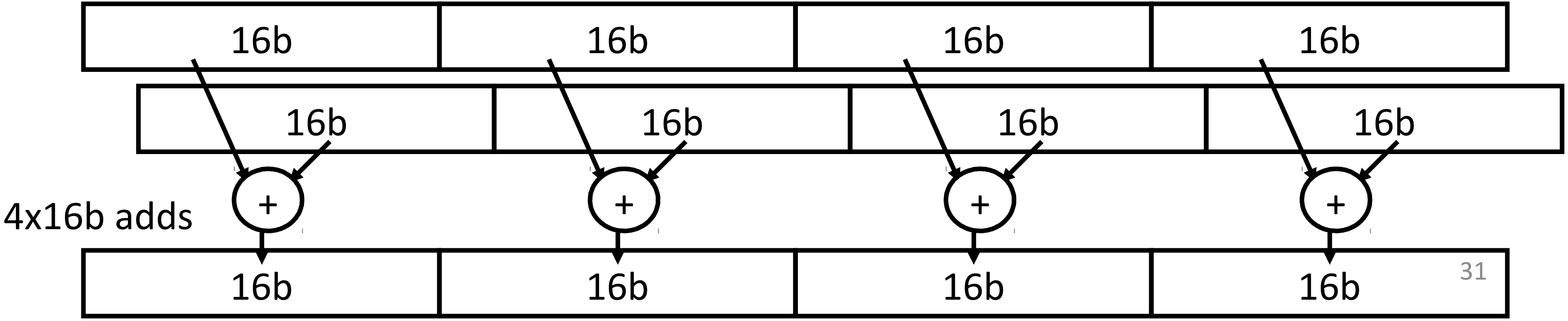
```
vst v1, x1    # Store vector to x1
fence         # Enforce ordering s->l
vld v2, x1    # Load vector from x1
```

- Vector machines support highly parallel memory systems (multiple lanes and multiple load and store units) with long latency (100+ clock cycles)
  - hardware coherence checks would be prohibitively expensive
  - vectorizing compiler can eliminate most dependencies

Packed SIMD Extensions



- Very short vectors added to existing ISAs for microprocessors
- Use existing 64-bit registers split into 2x32b or 4x16b or 8x8b
  - Lincoln Labs TX-2 from 1957 had 36b datapath split into 2x18b or 4x9b
  - Newer designs have wider registers
    - 128b for PowerPC Altivec, Intel SSE2/3/4
    - 256b/512b for Intel AVX
- Single instruction operates on all elements within register



Packed SIMD versus Vectors

- Limited instruction set:
  - no vector length control
  - no strided load/store or scatter/gather
  - unit-stride loads must be aligned to 64/128-bit boundary
- Limited vector register length:
  - requires superscalar dispatch to keep multiply/add/load units busy
  - loop unrolling to hide latencies increases register pressure
- Trend towards fuller vector support in microprocessors
  - Better support for misaligned memory accesses
  - Support of double-precision (64-bit floating-point)
  - New Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b), gather added, scatter to follow
  - ARM Scalable Vector Extensions (SVE)

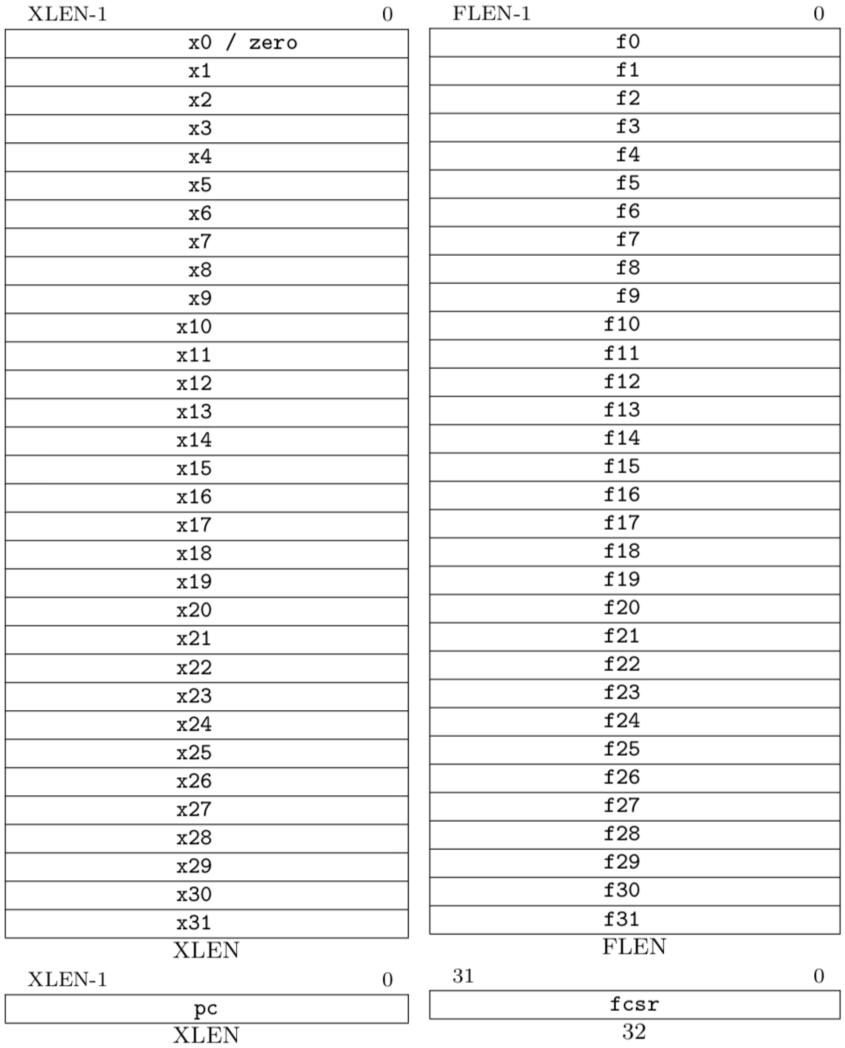
New RISC-V “V” Vector Extension

- 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, v0.7
  - v0.7 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

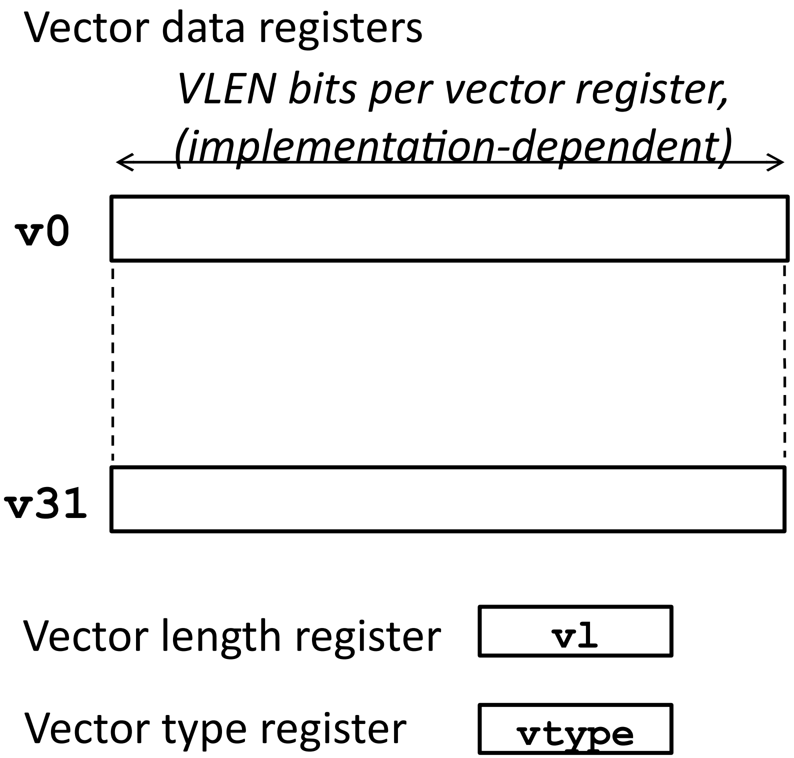
RISC-V Scalar State

- 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)



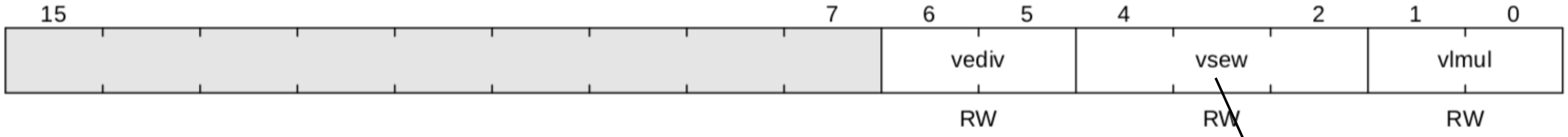
Vector Extension Additional State

- 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`**



Vector Type Register

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)

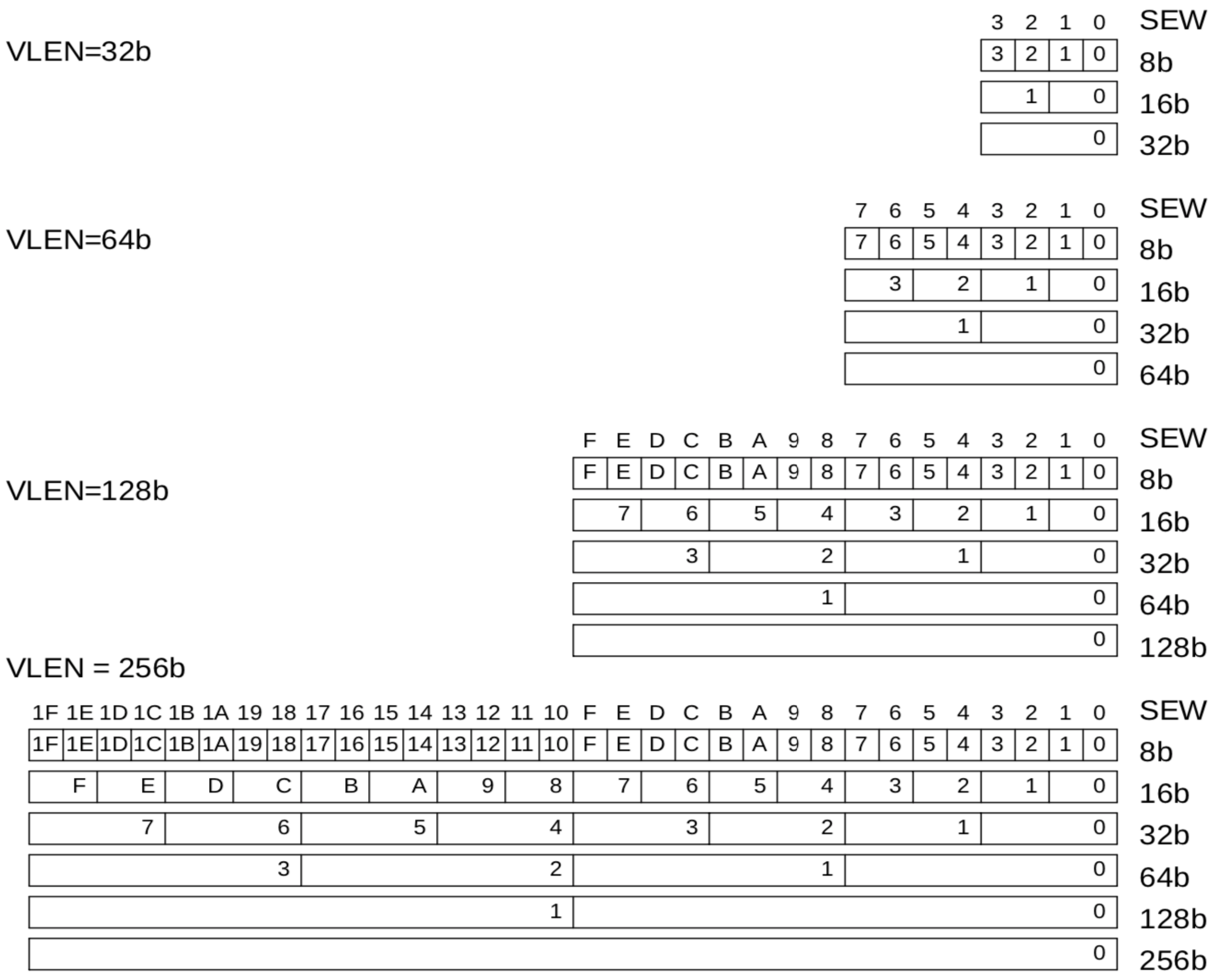
vsew[2:0]			SEW
0	0	0	8
0	0	1	16
0	1	0	32
0	1	1	64
1	0	0	128
1	0	1	256
1	1	0	512
1	1	1	1024

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

`vlmul[1:0]` encodes vector register length multiplier (LMUL = $2^{\mathtt{vlmul}}$ = 1-8)

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

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

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/vsetvl`

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

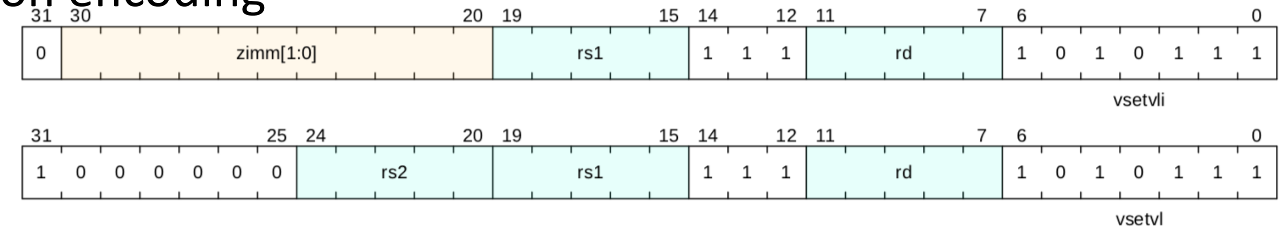
```
vsetvli rd, rs1, e8 # Set SEW=8, vl=min(VLEN/SEW,rs1),
rd=vl
```

Resulting machine vector length setting

Requested application vector length

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

Instruction encoding



31	30 … 20	19 … 15	14 … 12	11 … 7	6 … 0
0	zimm[1:0]	rs1	1 1 1	rd	1 0 1 0 1 1 1

vsetvli

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
1 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	1 0 1 0 1 1 1

vsetvl

Usually use immediate form, **`vsetvli`,** to set **`vtype`** parameters. The register version **`vsetvl`** is usually used for context save/restore

`vsetvl{i}` operation

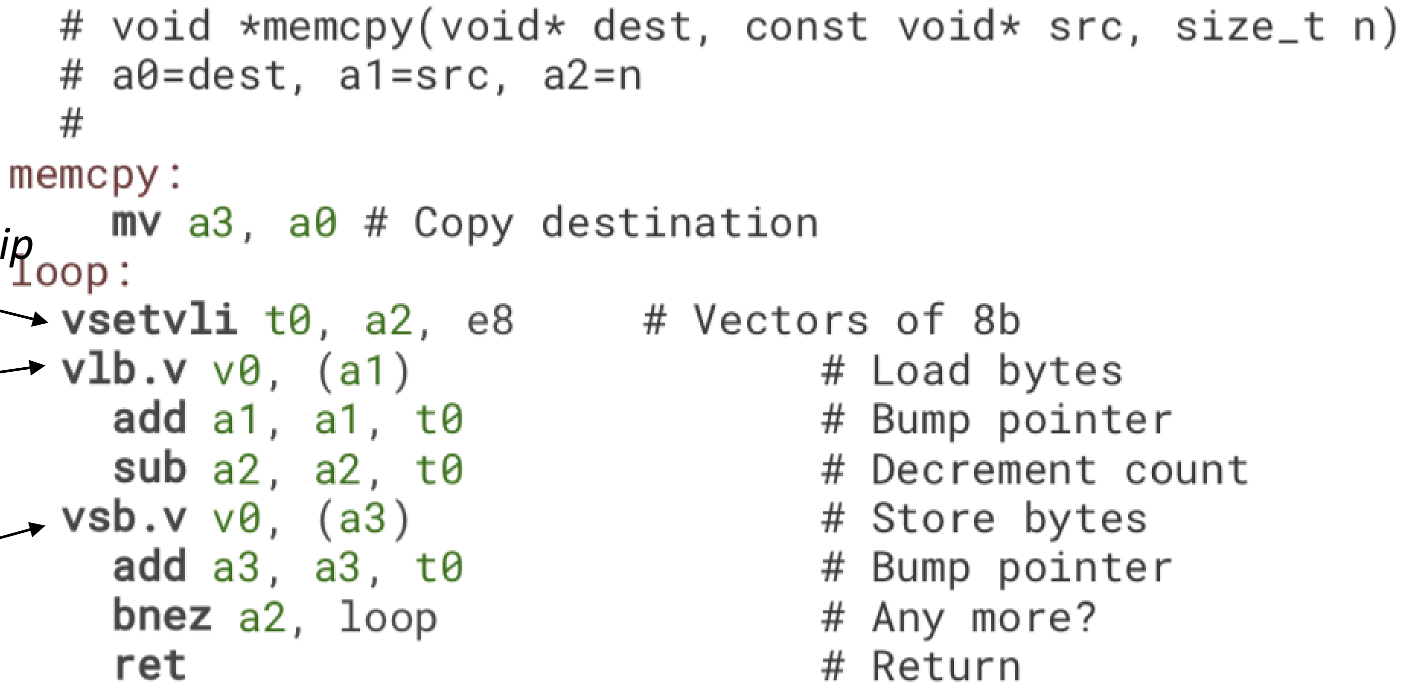
- The first scalar register argument, rs1, is the requested application vector length (AVL)
- The type argument (either immediate or second register) indicates how the vector registers should be configured
  - Configuration includes size of each element
- The vector length is set to the minimum of requested 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

Set configuration, calculate vector strip length

Unit-stride vector load bytes

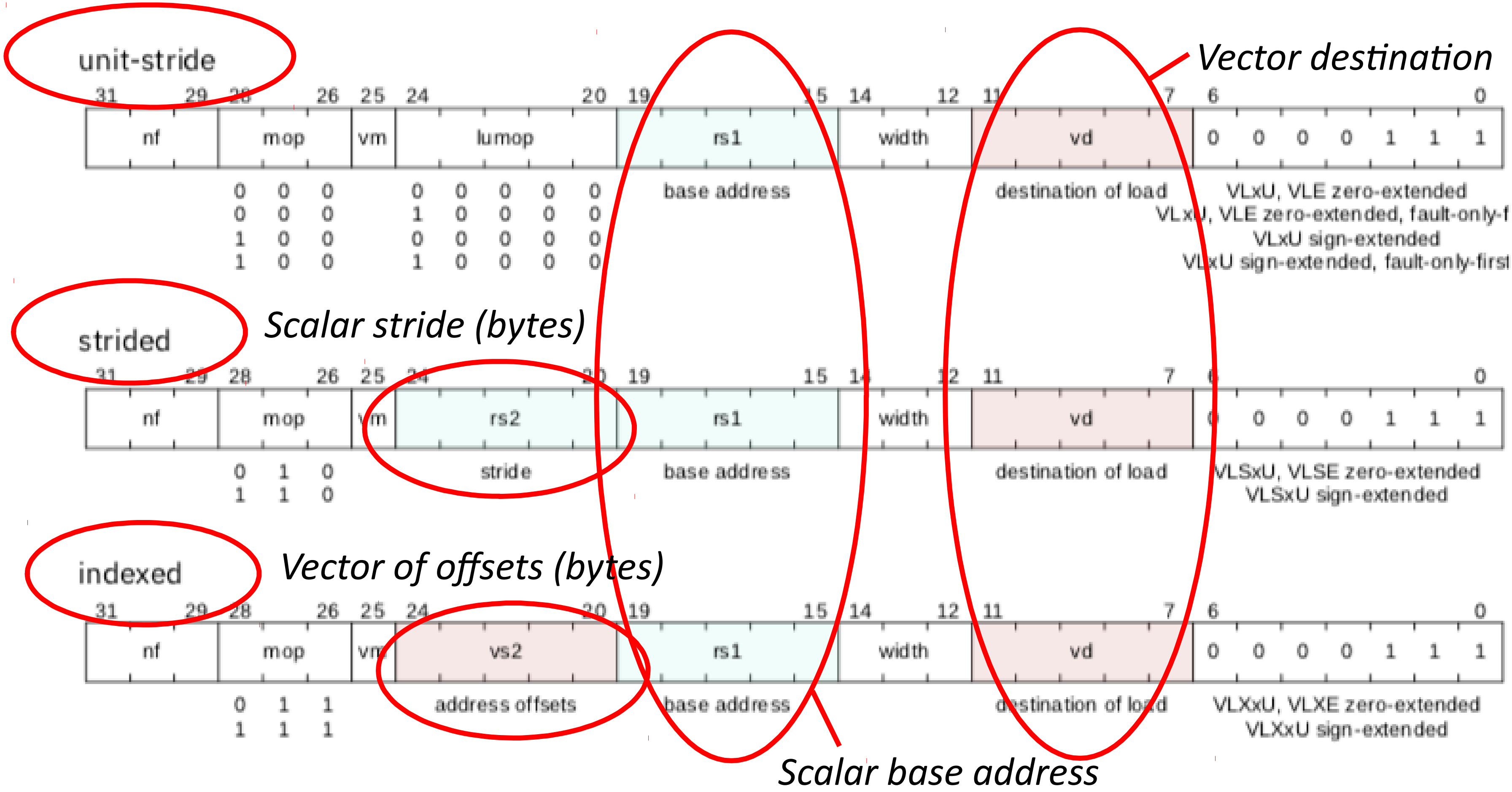
Unit-stride vector store bytes



```
 # 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
 vlb.v v0, (a1)                  # Load bytes
  add a1, a1, t0                 # Bump pointer
  sub a2, a2, t0                 # Decrement count
 vsb.v v0, (a3)                  # Store bytes
  add a3, a3, t0                 # Bump pointer
  bnez a2, loop                  # Any more?
  ret                            # Return
```

Binary machine code can run on machines with any VLEN!

Vector Load Instructions



Vector destination

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

Scalar stride (bytes)

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

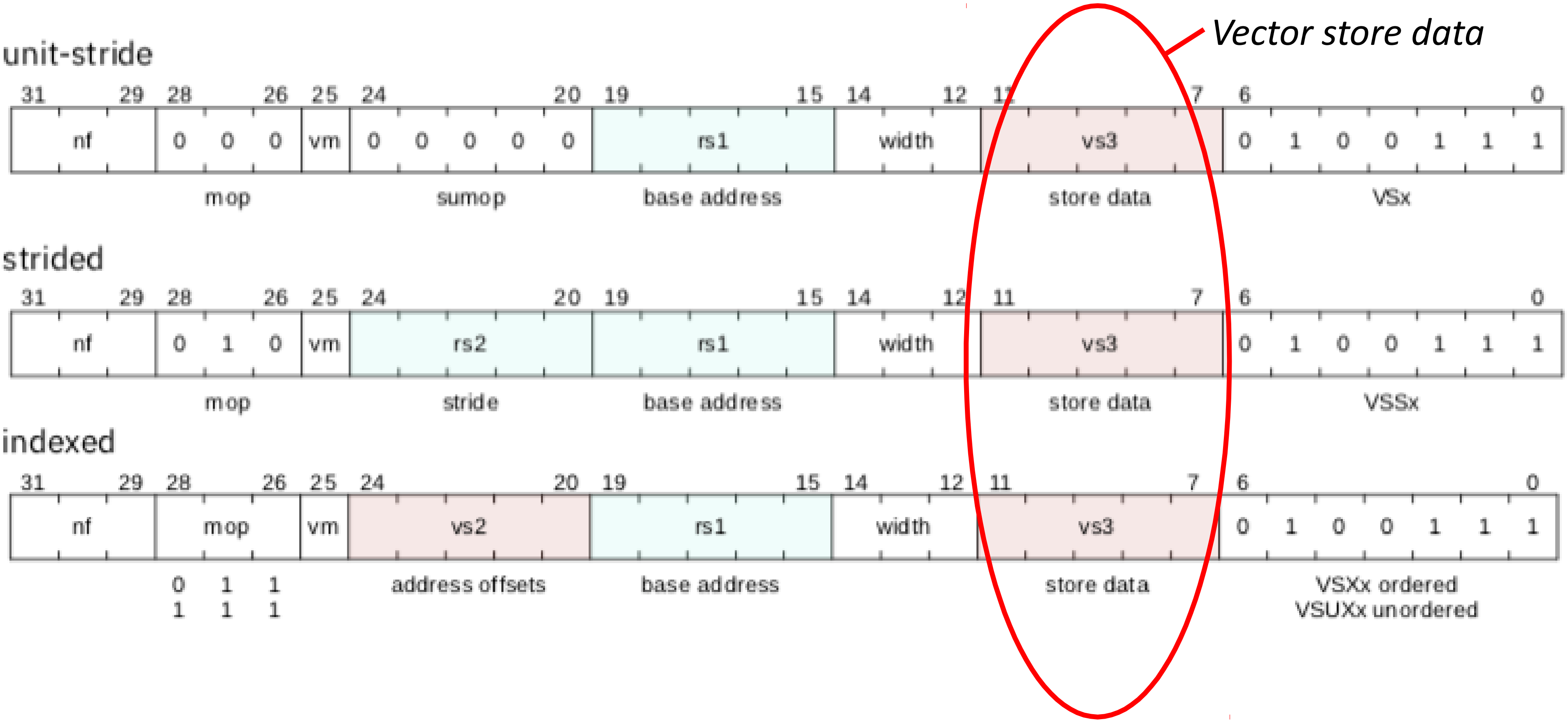
indexed

Vector of offsets (bytes)

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

Scalar base address

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

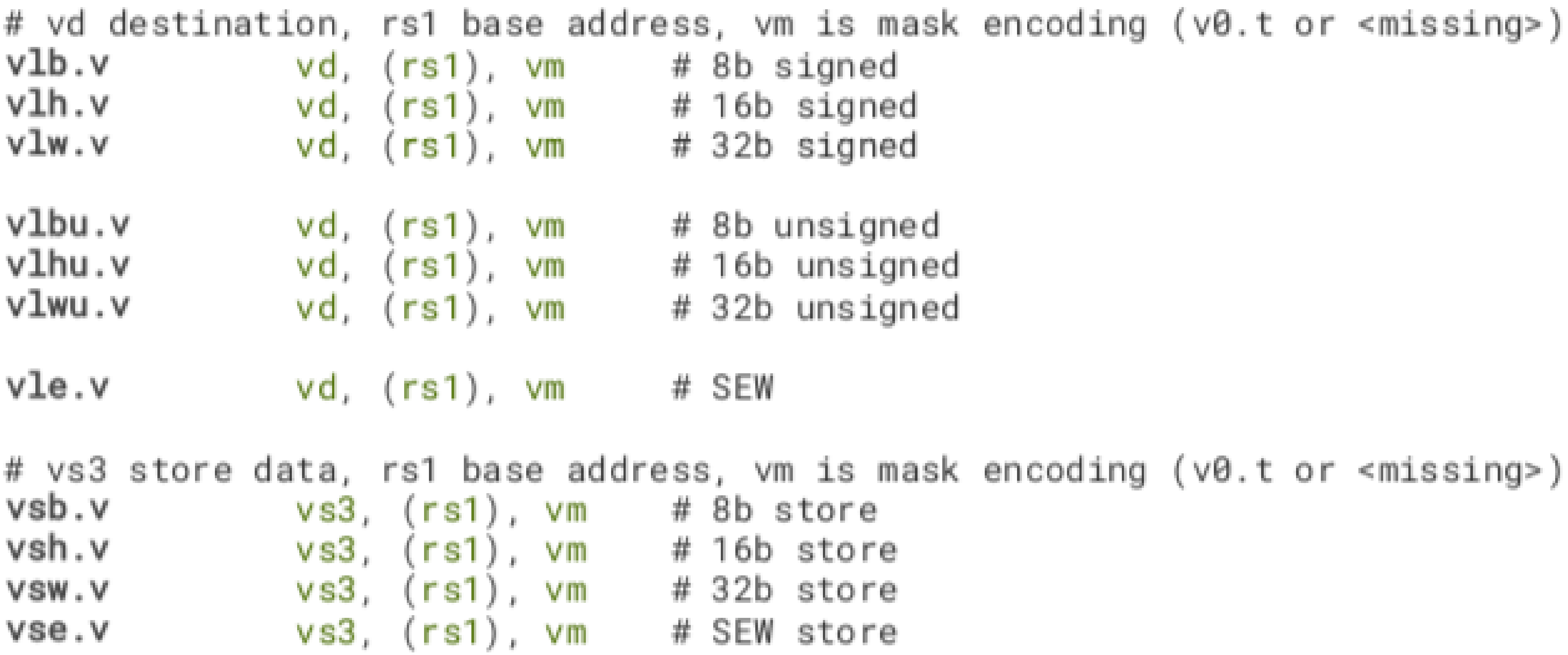
strided

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

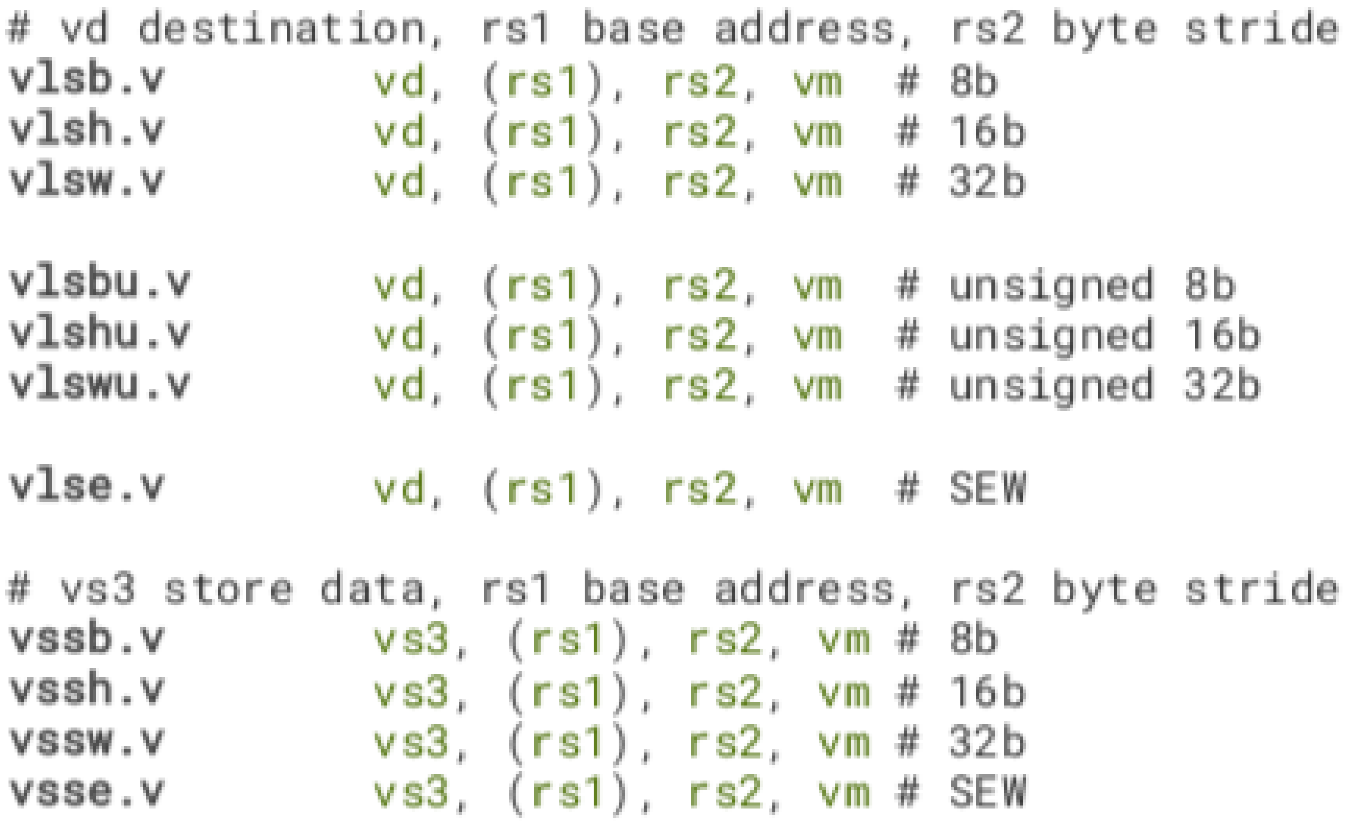
31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
nf	mop	vm	vs2	rs1	width	vs3	0 1 0 0 1 1 1
	0 1 1 1 1 1		address offsets	base address		store data	VSXx ordered VSUXx unordered

Vector Unit-Stride Loads/Stores



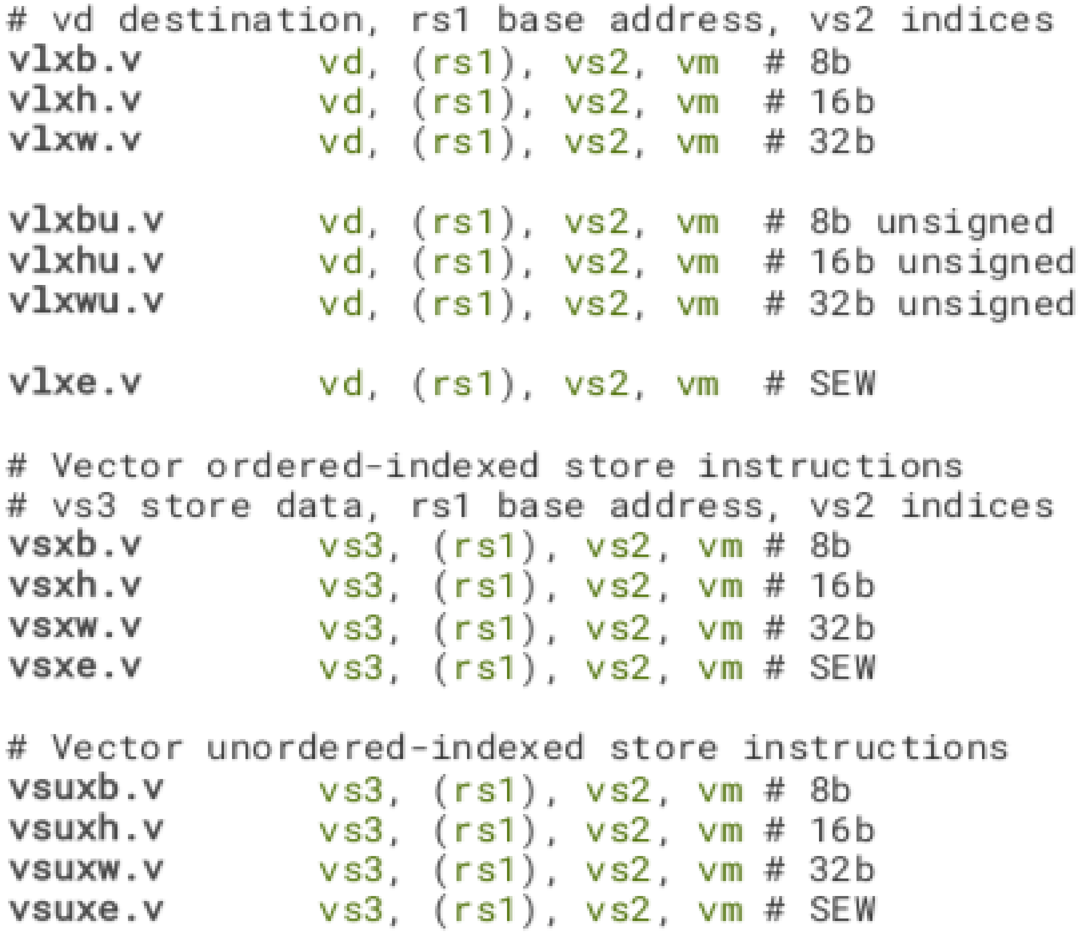
```
# vd destination, rs1 base address, vm is mask encoding (v0.t or <missing>)
vlb.v         vd, (rs1), vm     # 8b signed
vlh.v         vd, (rs1), vm     # 16b signed
vlw.v         vd, (rs1), vm     # 32b signed

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
```

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

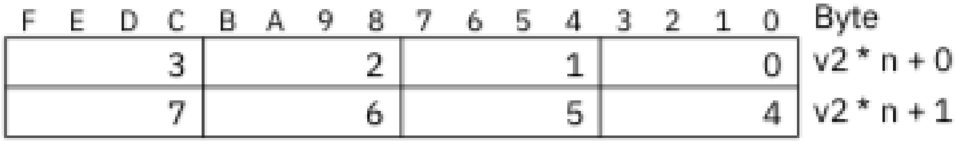


```
# vd destination, rs1 base address, vs2 indices
vlxb.v      vd, (rs1), vs2, vm  # 8b
vlxh.v      vd, (rs1), vs2, vm  # 16b
vlxw.v      vd, (rs1), vs2, vm  # 32b

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

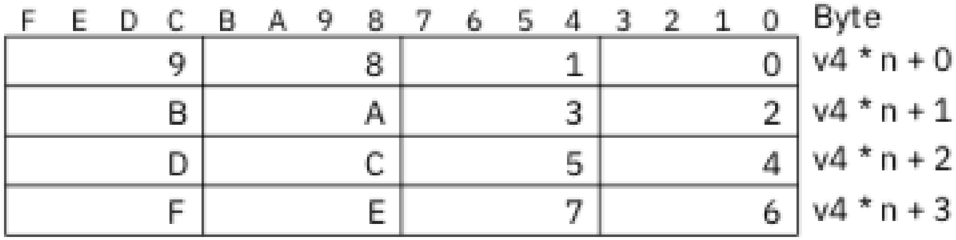
- Gives fewer but longer vector registers
- Set by **`vlmul[1:0]`** field in **`vtype`** during **`setvli`**

LMUL=2



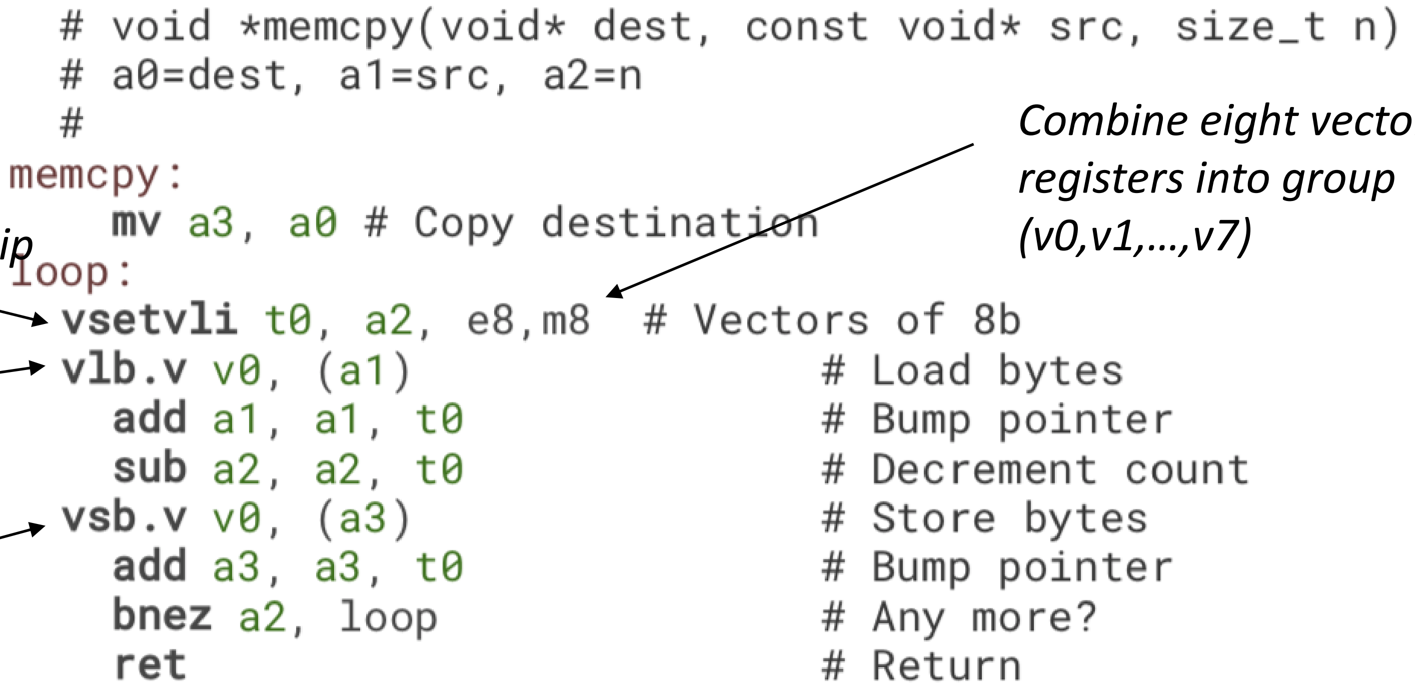
F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
			3				2				1				0	v2 * n + 0
			7				6				5				4	v2 * n + 1

LMUL=4



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

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
 vlb.v v0, (a1)                # Load bytes
  add a1, a1, t0               # Bump pointer
  sub a2, a2, t0               # Decrement count
 vsb.v v0, (a3)                # Store bytes
  add a3, a3, t0               # Bump pointer
  bnez a2, loop                # Any more?
  ret                          # Return
```

Set configuration, calculate vector strip length

Combine eight vector registers into group (v0,v1,…,v7)

Unit-stride vector load bytes

Unit-stride vector store bytes

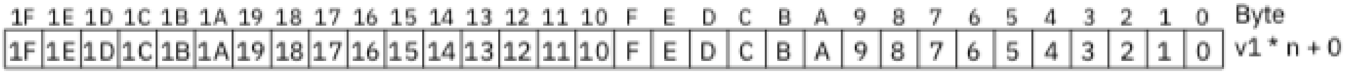
Binary machine code can run on machines with any VLEN!

Mixed-Width Loops

- Have different element widths in one loop, even in one instruction
- Want same number of elements in each vector register, even if different bits/element
- Solution: Keep SEW/LMUL constant

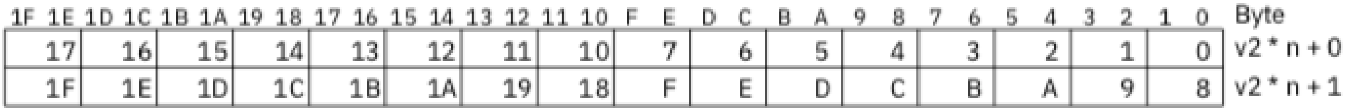
VLEN=256b, SLEN=128b

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



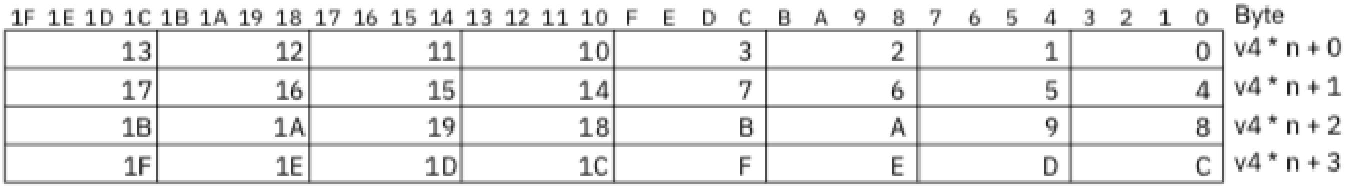
1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	v1 * n + 0

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



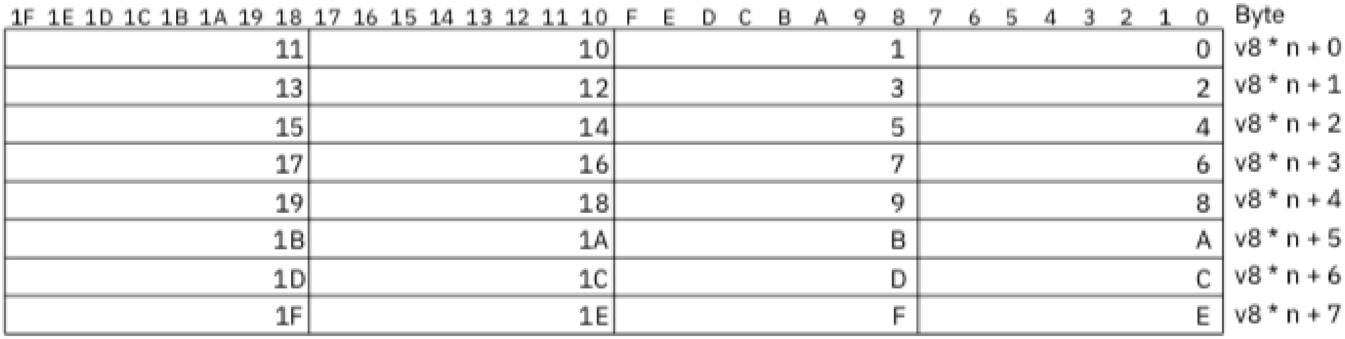
1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
	17		16		15		14		13		12		11		10		7		6		5		4		3		2		1		0	v2 * n + 0
	1F		1E		1D		1C		1B		1A		19		18		F		E		D		C		B		A		9		8	v2 * n + 1

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