CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

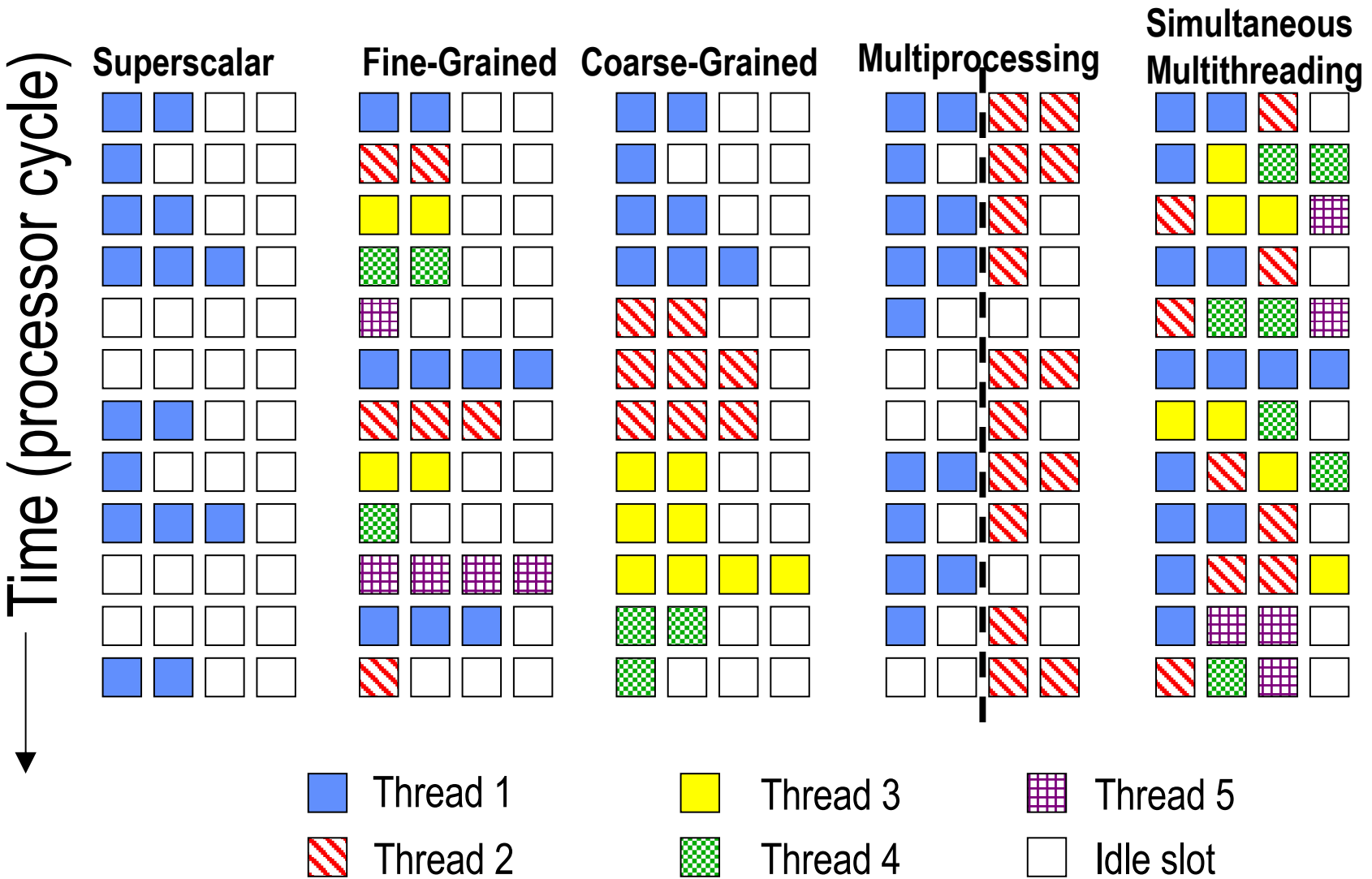## Lecture 15 – Vectors

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 Lecture 14: Multithreading

Time (processor cycle)

Superscalar — Fine-Grained — Coarse-Grained — Multiprocessing — Simultaneous Multithreading

Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Idle slot

2

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 $\equiv$ Vector Machine

Vector Supercomputers
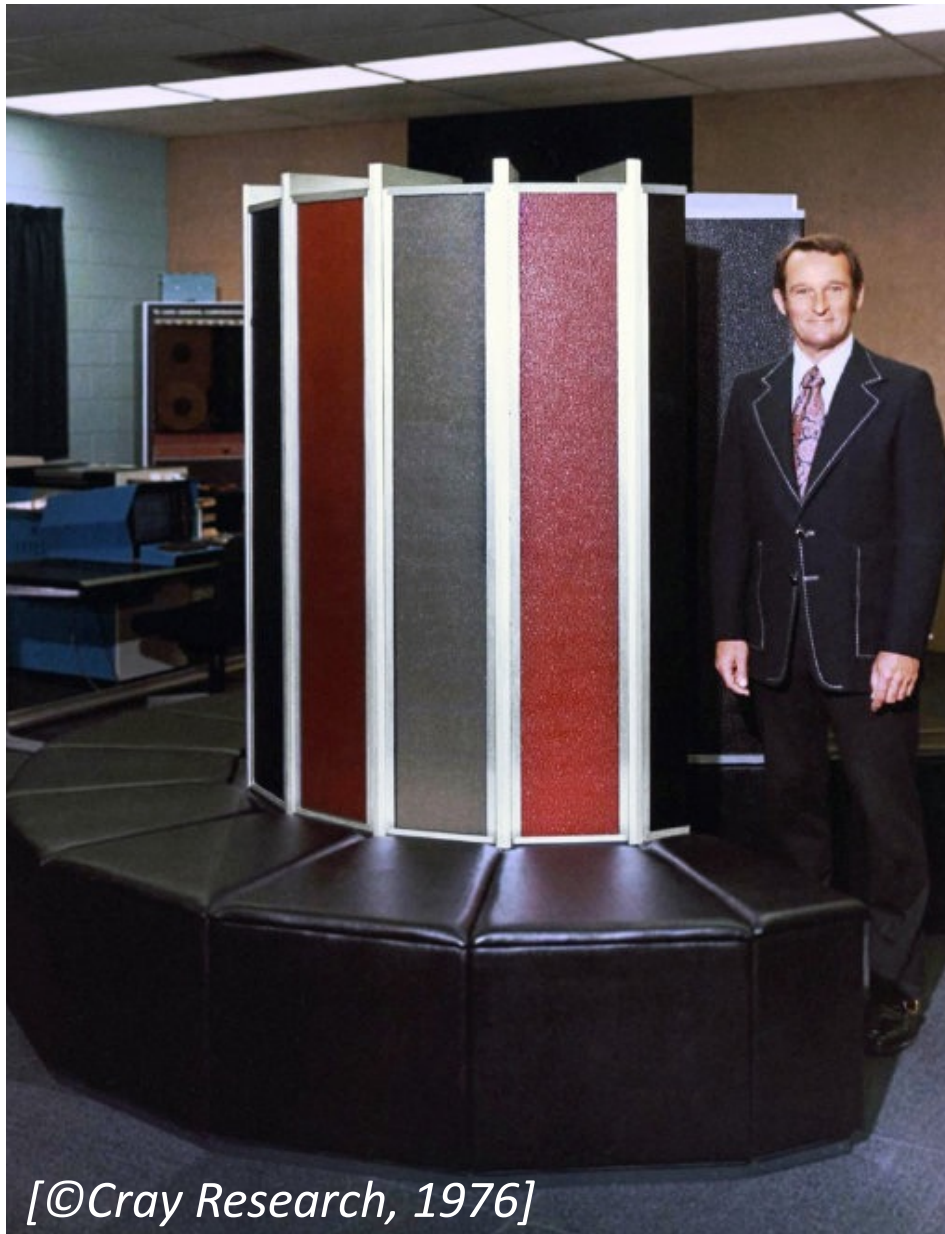
[©Cray Research, 1976]

- **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

Scalar Registers

x31

x0

Vector Registers

v31

v0

[0] [1] [2]   [VLMAX-1]
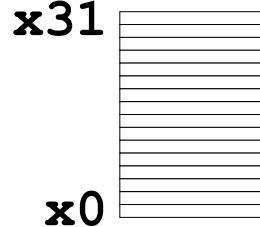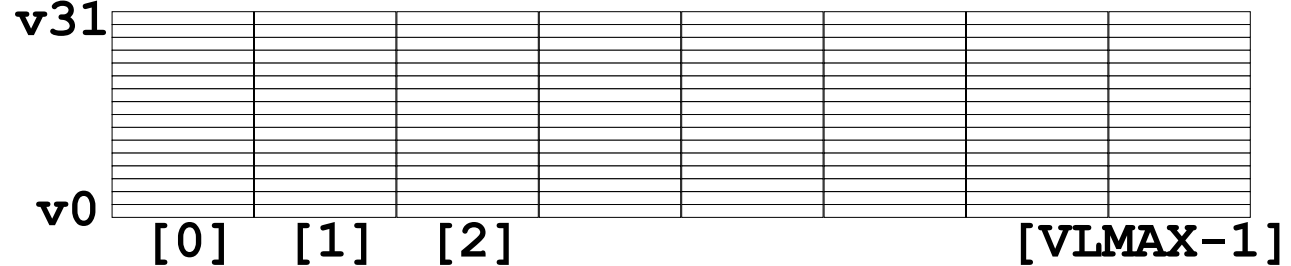
Vector Length Register | vl |

Vector Arithmetic
Instructions
vadd v3,v1,v2

v1
v2

(+) (+) (+) (+) (+) (+)

v3

[0]   [1]   [vl-1]
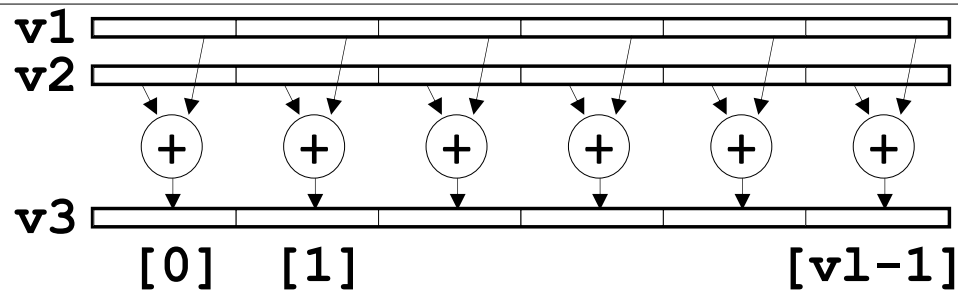
Vector Load and Store
Instructions
vls v1,(x1),x2

v1   *Vector Register*

Memory

Base, **x1**   Stride, **x2**
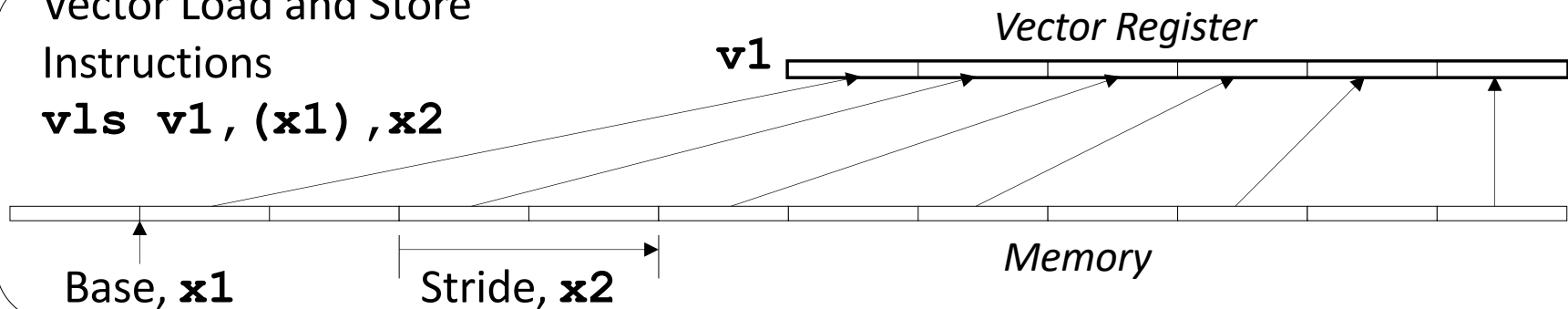
Vector Code Example

```
# 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, x1, 8
  addi x2, x2, 8
  addi x3, x3, 8
  subi x4, x4, 1
  bnez x4, loop
```
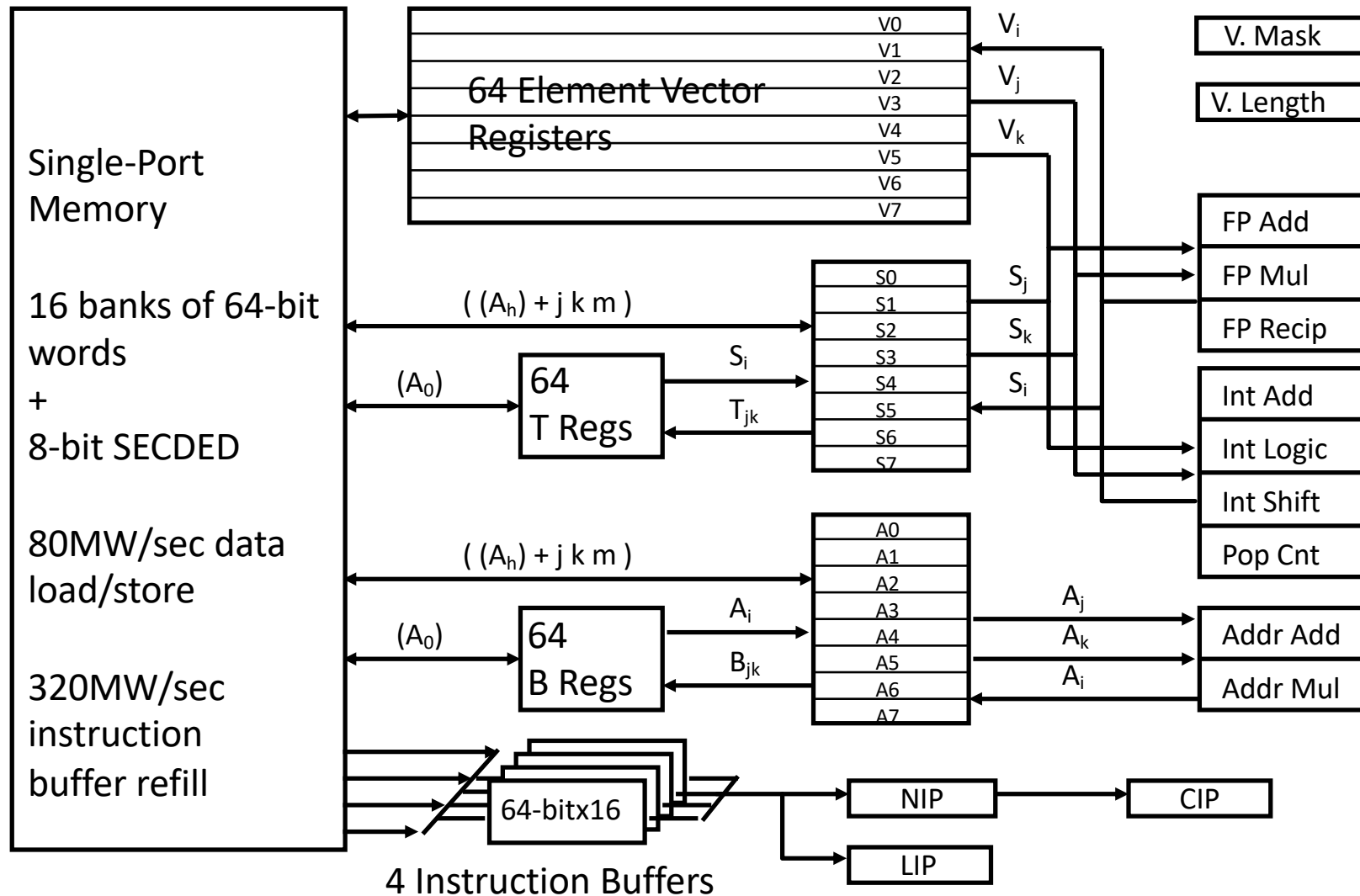
```
# Vector Code
  li x4, 64
  vsetvl x4
  vld v1, (x1)
  vld v2, (x2)
  vadd v3,v1,v2
  vst v3, (x3)
```

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

FP Add
FP Mul
FP Recip

$((A_h) + j \, k \, m)$

(A_0)

64 T Regs

S_i
T_{jk}

S0
S1
S2
S3
S4
S5
S6
S7

S_j
S_k
S_i

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

NIP

LIP

CIP

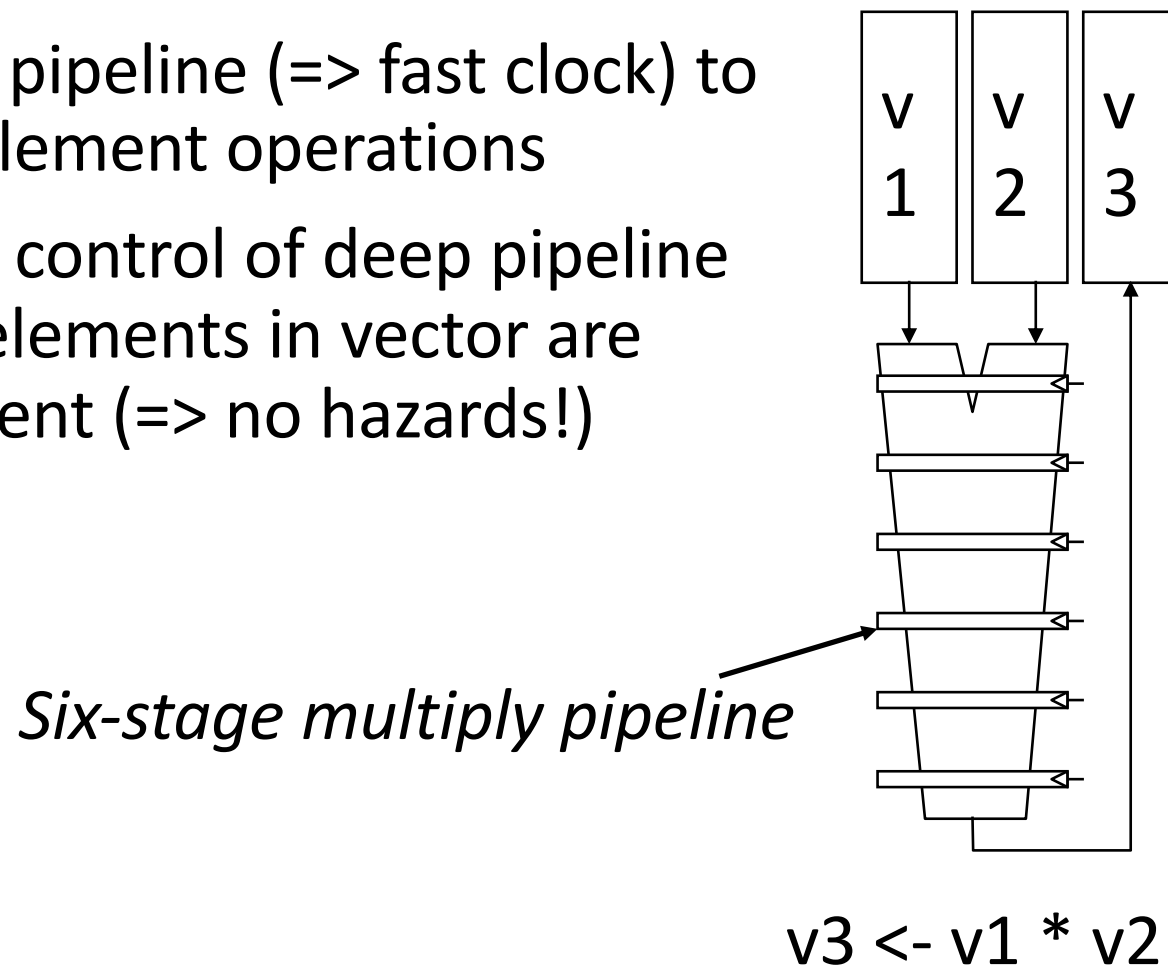**4 Instruction Buffers**

memory bank cycle 50 ns   *processor cycle* 12.5 ns (80MHz)
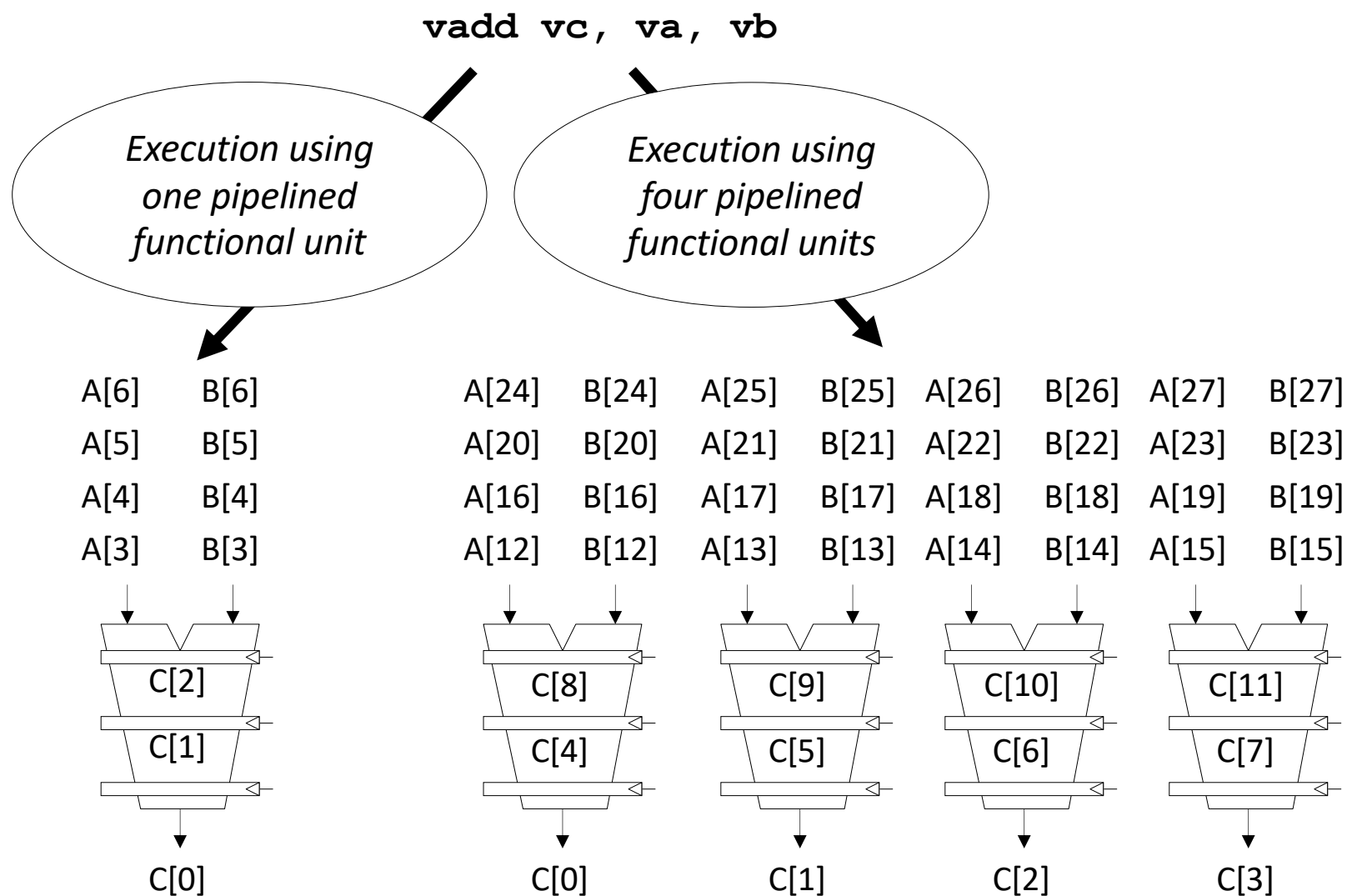
7

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!)

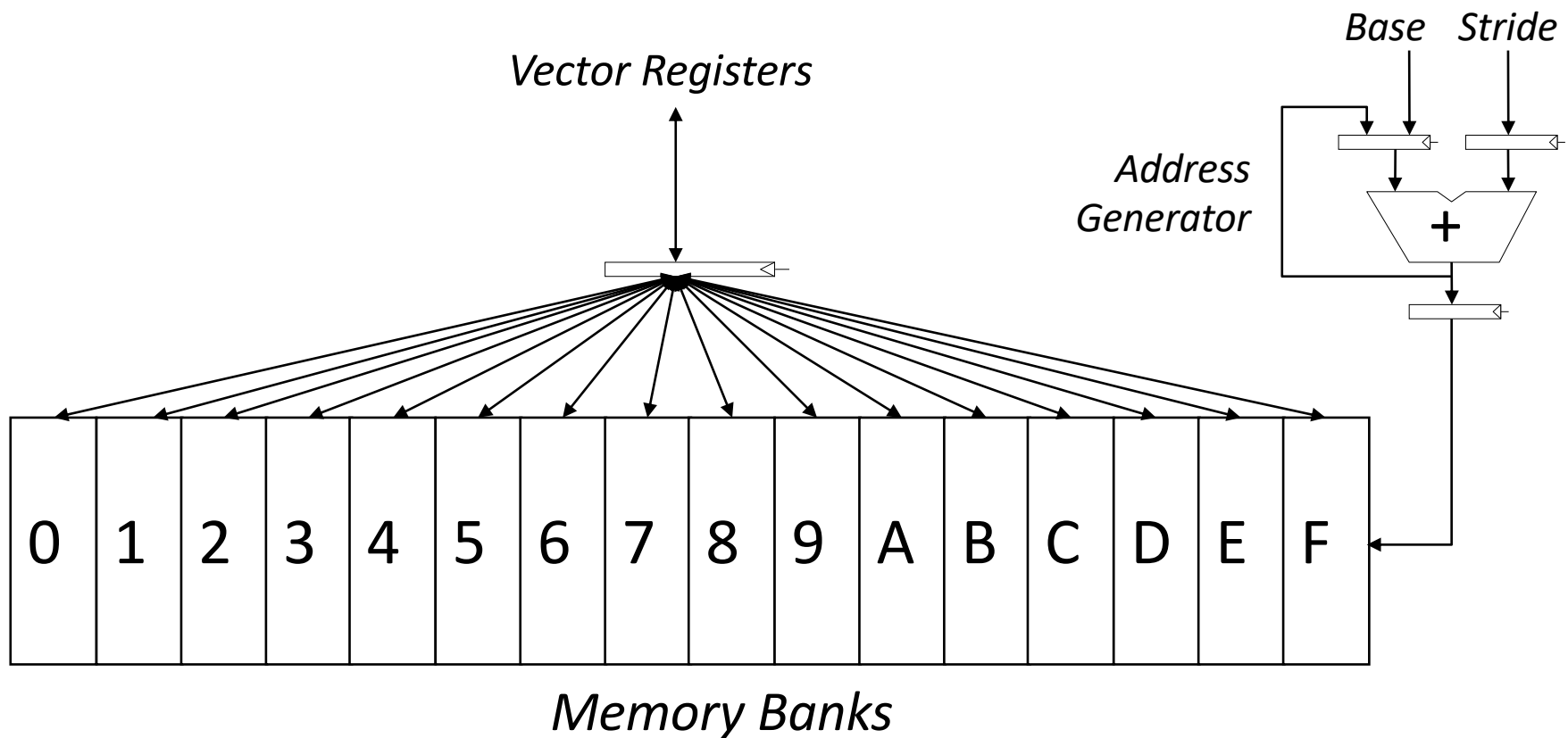*Six-stage multiply pipeline*

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]
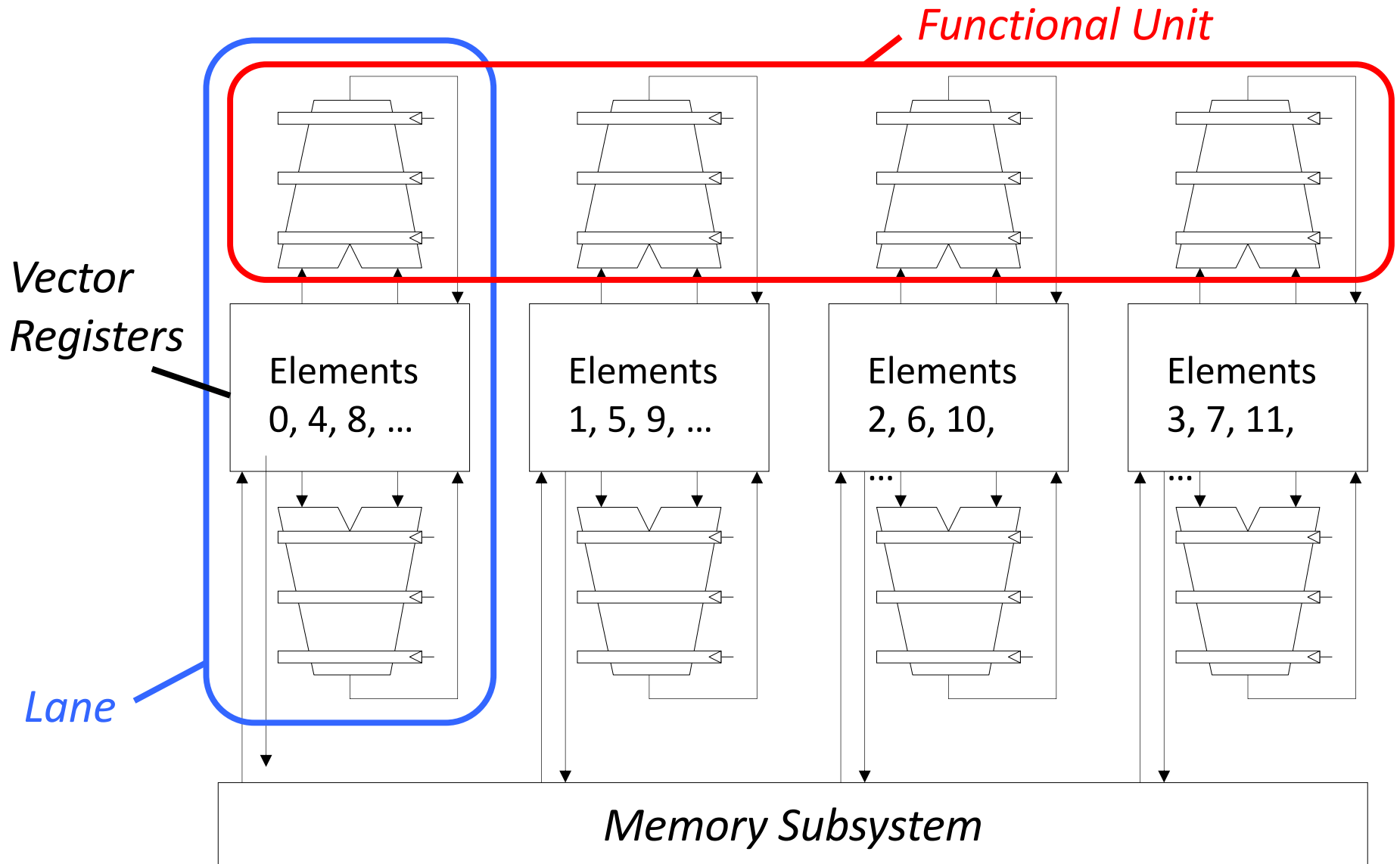
C[0]   C[1]   C[2]   C[3]

Interleaved Vector Memory System

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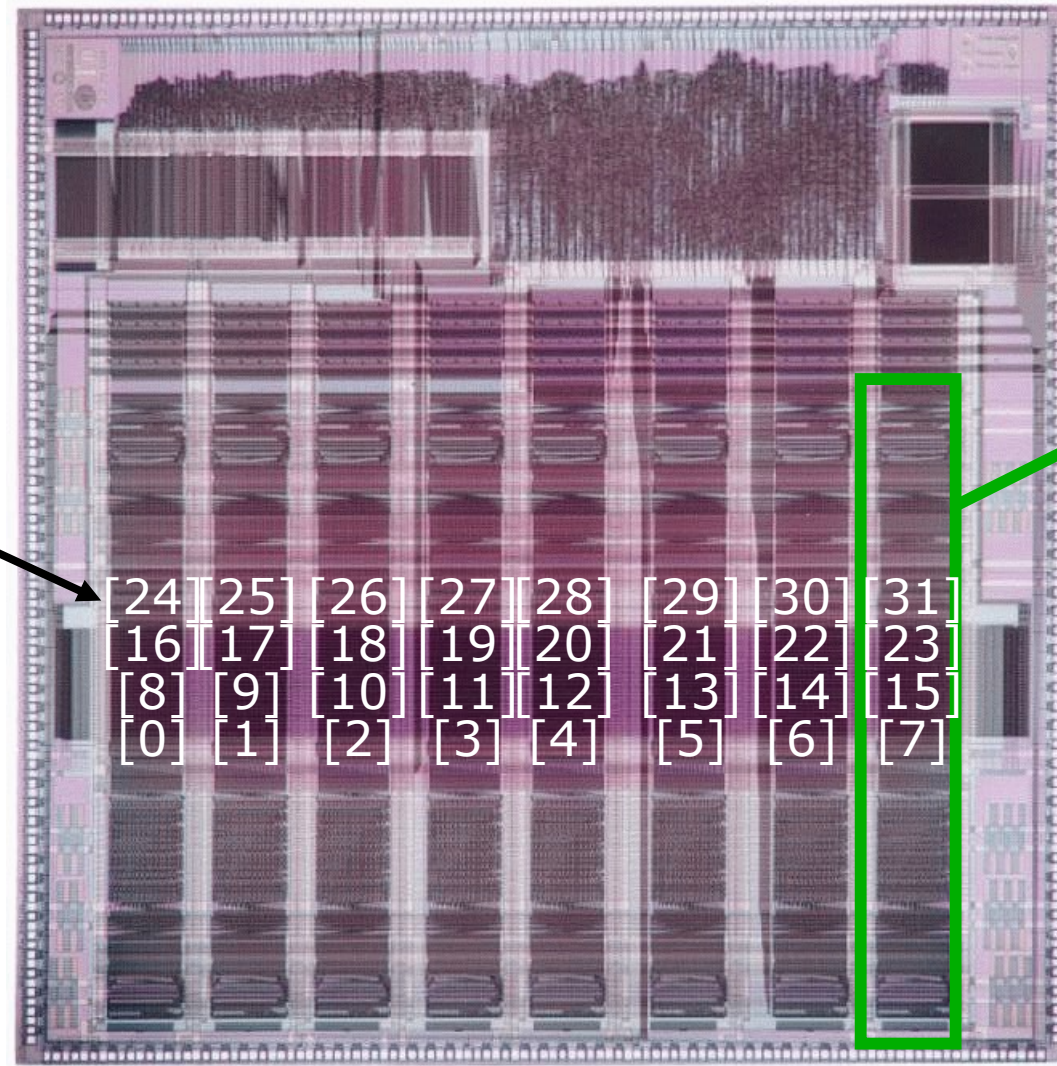- Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

Base   Stride

Vector Registers

Address
Generator

0 1 2 3 4 5 6 7 8 9 A B C D E F

Memory Banks

Vector Unit Structure

Functional Unit

Vector Registers

Elements 0, 4, 8, …

Elements 1, 5, 9, …

Elements 2, 6, 10, …

Elements 3, 7, 11, …

Lane

Memory Subsystem

T0 Vector Microprocessor (UCB/ICSI, 1995)

Vector register elements striped over lanes

Lane

[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]

Vector Instruction Parallelism

- Can overlap execution of multiple vector instructions
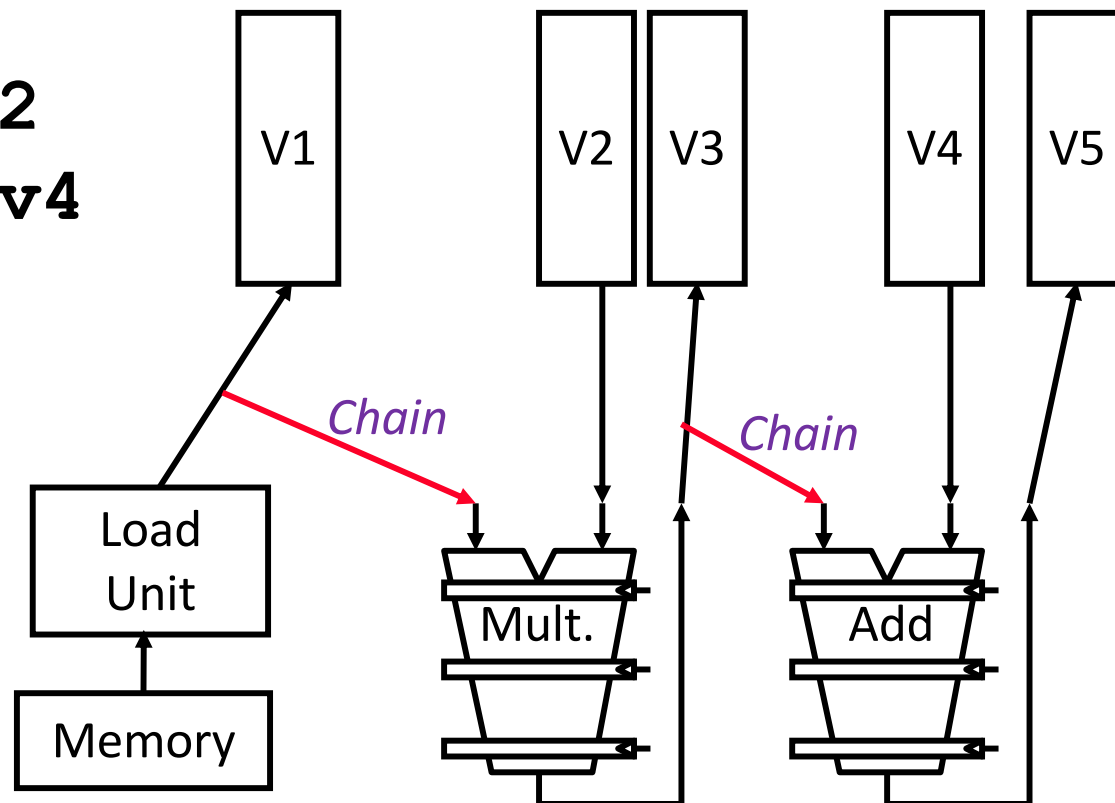 - example machine has 32 elements per vector register and 8 lanes
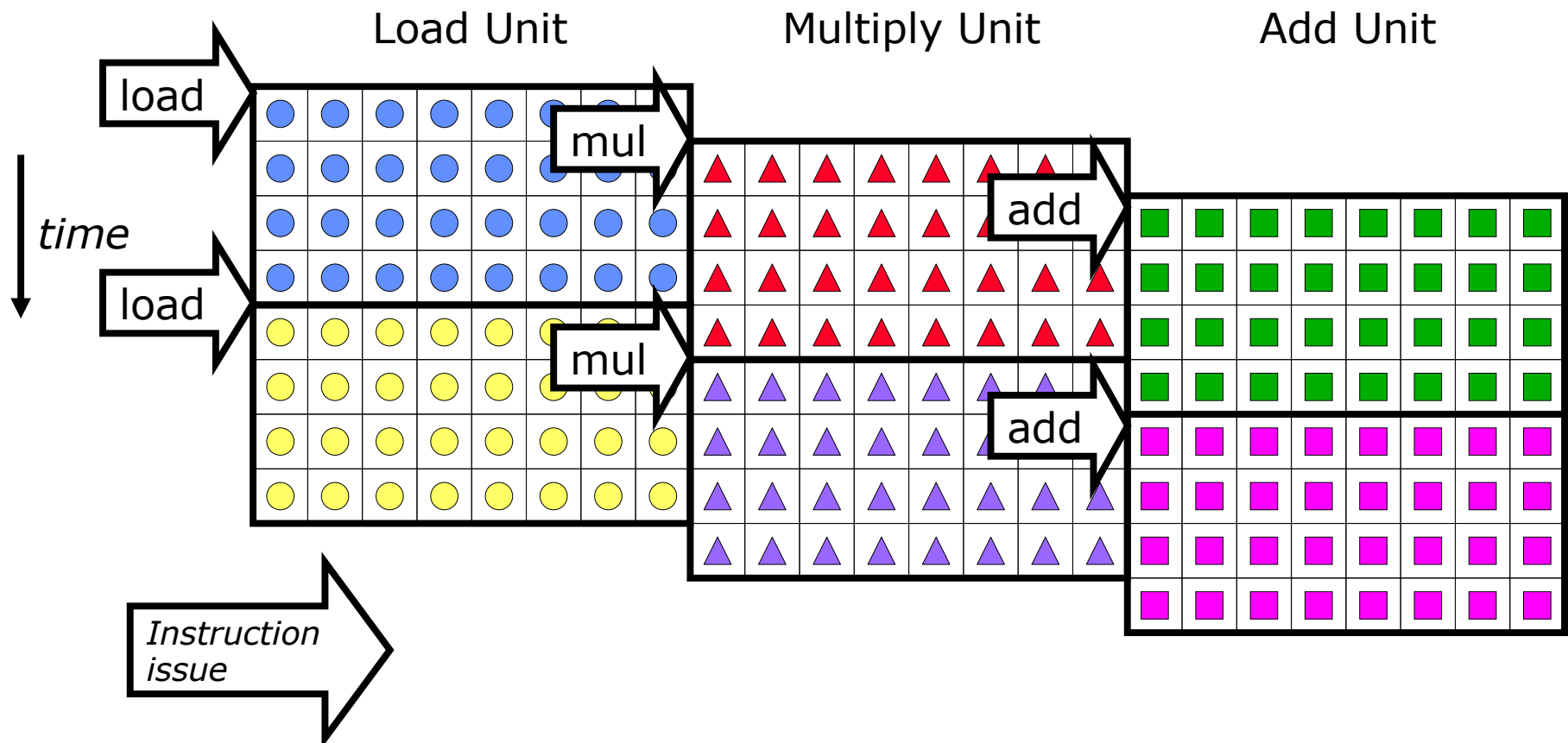
Complete 24 operations/cycle while issuing 1 short instruction/cycle

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
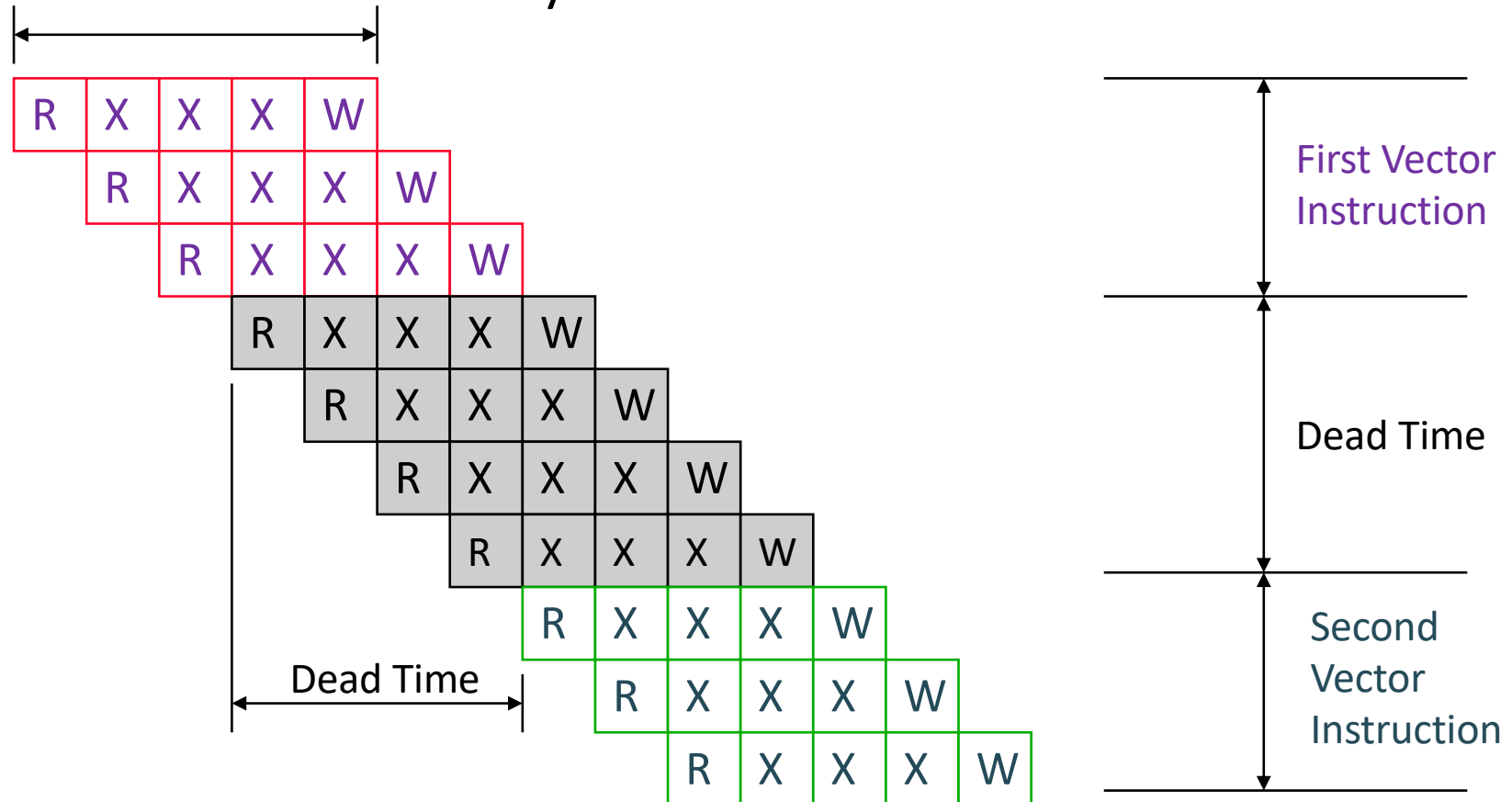

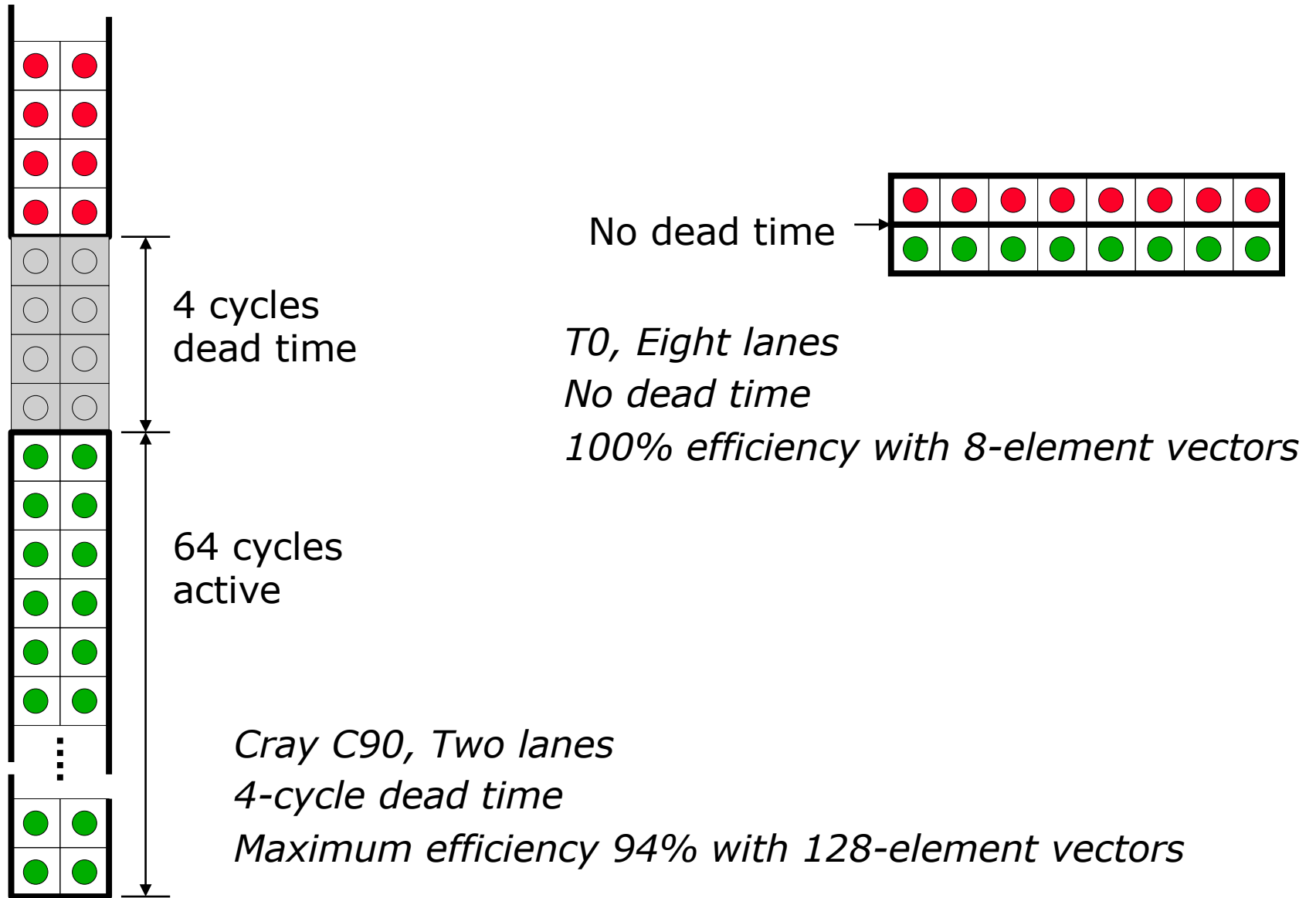- With chaining, can start dependent instruction as soon as first result appears

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)

Functional Unit Latency

First Vector Instruction

Dead Time

Second Vector Instruction

Dead Time

17

Dead Time and Short Vectors

4 cycles
dead time

64 cycles
active

No dead time →

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

18

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)
```

19

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)

20

CS152 Administrivia

- PS 4 out today, due Monday April 5

- Lab 3 due Monday April 5

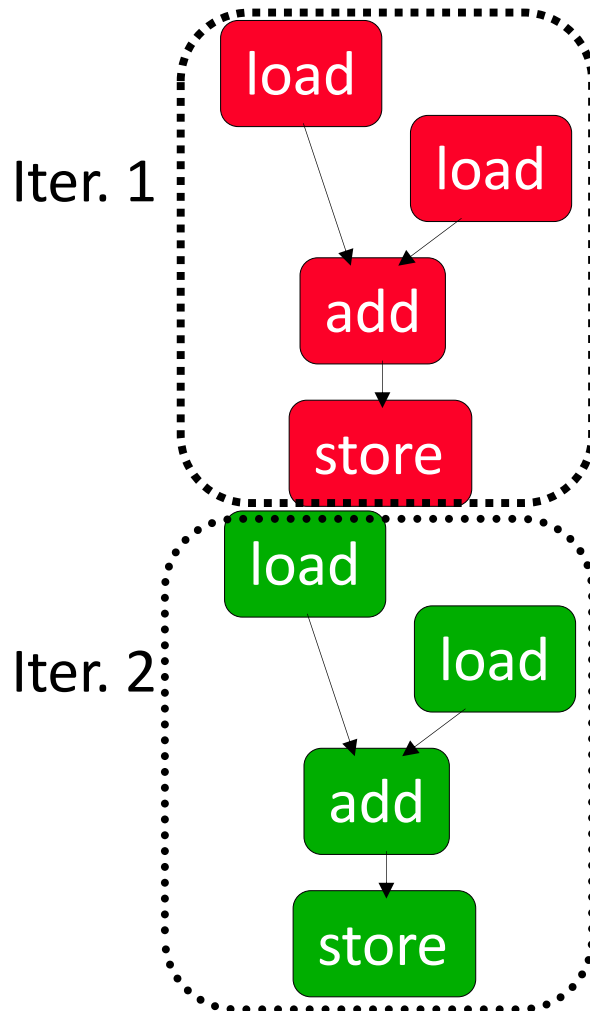- No lectures or sections next week!
    - Spring Break (March 22-26)

CS252 Administrivia

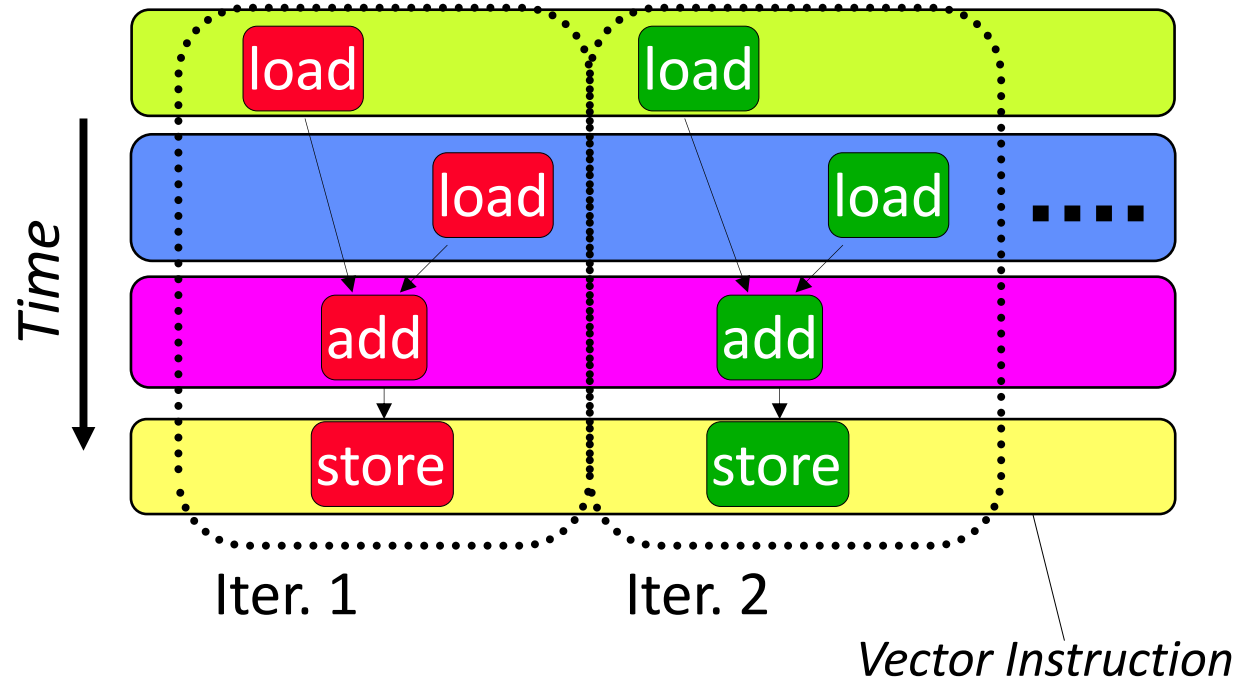- Readings on OoO architectures this Thursday

Automatic Code Vectorization

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

Scalar Sequential Code

Vectorized Code

Time

Iter. 1

Iter. 2

Iter. 1 Iter. 2

Vector Instruction

Vectorization is a massive compile-time reordering of operation sequencing
$\Rightarrow$ 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];
```


A B C

Remainder
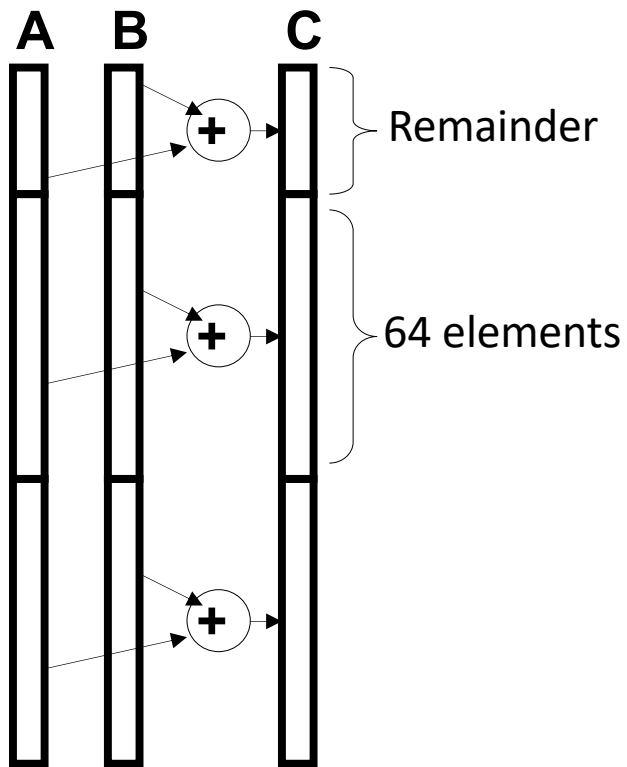
64 elements

```
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?
```

24

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
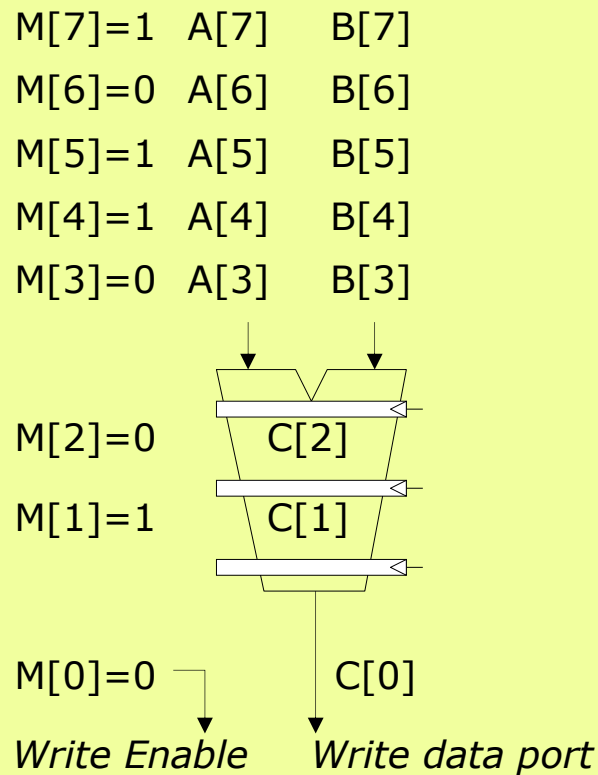
Code example:

```
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   A[7]   B[7]

M[5]=1

M[4]=1

M[3]=0   C[5]

M[2]=0   C[4]

M[1]=1

M[0]=0   C[1]

Write data port
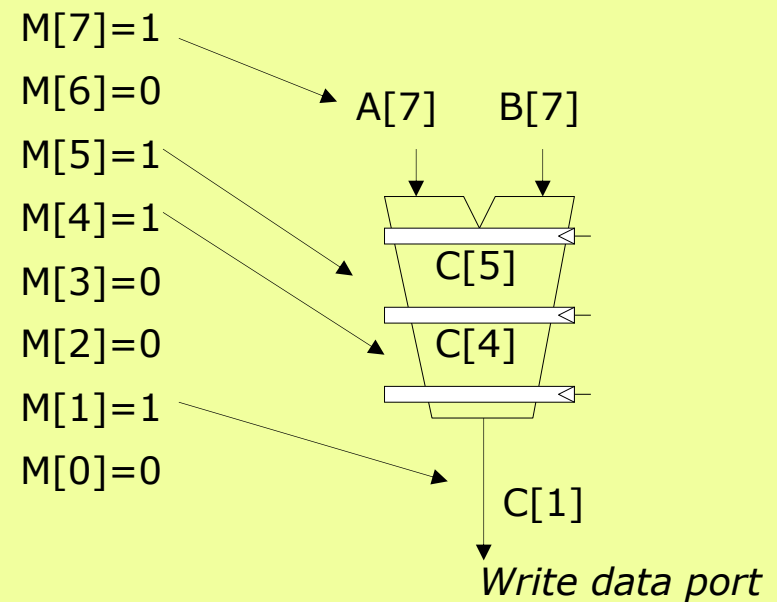
Compress/Expand Operations

- 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

M[7]=1 →	A[7]		A[7] ←	M[7]=1
M[6]=0	A[6]		B[6]	M[6]=0
M[5]=1 →	A[5]		A[5] ←	M[5]=1
M[4]=1 →	A[4]		A[4] ←	M[4]=1
M[3]=0	A[3]	A[7]	B[3]	M[3]=0
M[2]=0	A[2]	A[5]	B[2]	M[2]=0
M[1]=1 →	A[1]	A[4]	A[1] ←	M[1]=1
M[0]=0	A[0]	A[1]	B[0]	M[0]=0

Compress   *Expand*
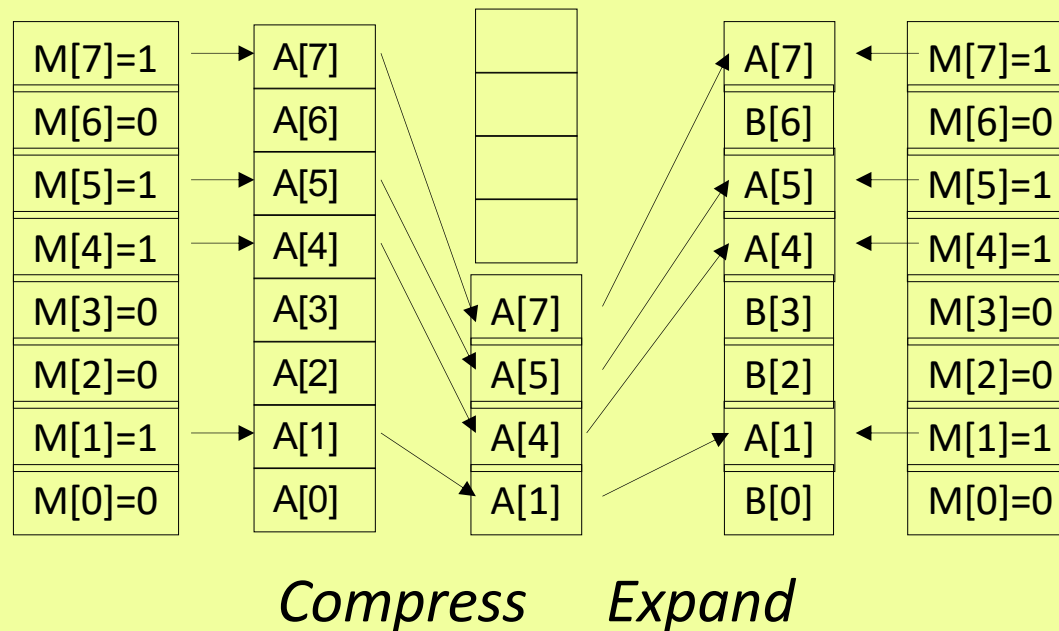
Used for density-time conditionals and also for general selection operations

CS252

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. 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
vlx vC, (xC), vD    # Load indexed from xC base
vld vB, (xB)        # Load B vector
vadd vA,vB,vC       # Do add
vst vA, (xA)        # Store result
```

Histogram with Scatter/Gather

Histogram example:

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

Is following a correct translation?

```
vld vB, (xB)       # Load indices in B vector
vlx vA, (xA), vB   # Gather initial A values
vadd vA, vA, 1     # Increment
vsx vA, (xA), vB   # Scatter incremented values
```

Vector Memory Models

- Some 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!

- So require 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

CS252

Packed SIMD Extensions

64b							
32b				32b			
16b		16b		16b		16b	
8b	8b	8b	8b	8b	8b	8b	8b

- 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

16b	16b	16b	16b

4x16b adds

16b	16b	16b	16b

16b	16b	16b	16b
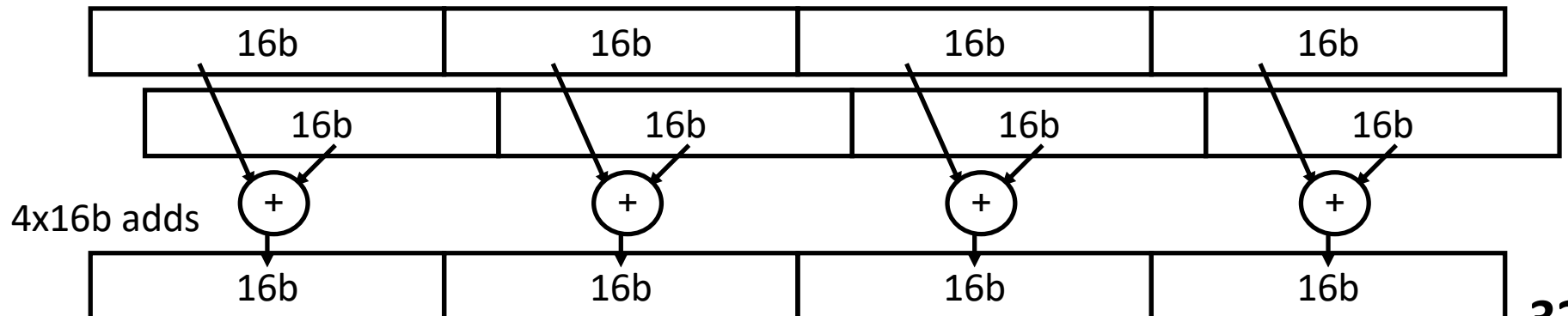
Packed SIMD versus Vectors

- 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)

Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)