

ARCHITECTURE OF

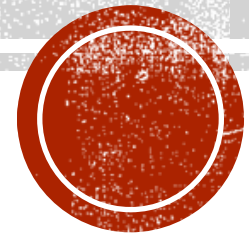
COMPUTER

SYSTEMS

LECTURE 15:

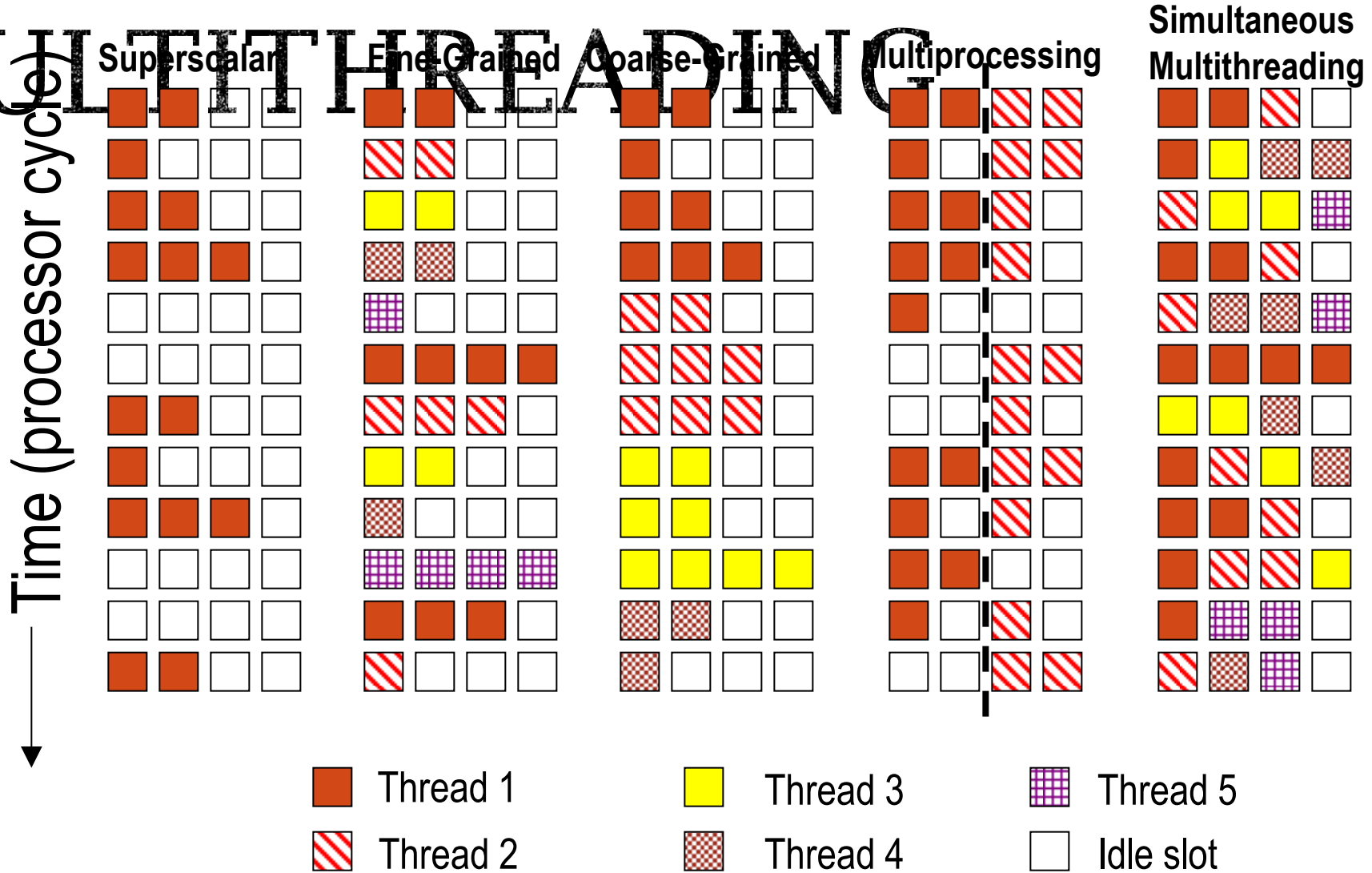
VECTOR

COMPUTERS



LAST TIME LECTURE 14:

MULTITHREADING



SUPERCOMPUTERS

- Definition of a supercomputer:
 - Fastest machine in world at given task
 - A device to turn a compute-bound problem into an I/O bound problem
 - Any machine costing \$30M+
 - Any machine designed by Seymour Cray
-
- CDC6600 (Cray, 1964) regarded as first supercomputer

CDC 6600 *SEYMOUR CRAY, 1963*



- A fast pipelined machine with 60-bit words
 - 128 Kword main memory capacity, 32 banks
- Ten functional units (parallel, unpipelined)
 - Floating Point: adder, 2 multipliers, divider
 - Integer: adder, 2 incrementers, ...
- Hardwired control (no microcoding)
- *Scoreboard* for dynamic scheduling of instructions
- Ten Peripheral Processors for Input/Output
 - a fast multi-threaded 12-bit integer ALU
- Very fast clock, 10 MHz (FP add in 4 clocks)
- >400,000 transistors, 750 sq. ft., 5 tons, 150 kW, novel freon-based technology for cooling
- Fastest machine in world for 5 years (until 7600)



IBM MEMO ON

CDC 6600

Thomas Watson Jr., IBM CEO, August 1963:

“Last week, Control Data ... announced the 6600 system. I understand that in the laboratory developing the system there are only 34 people including the janitor. Of these, 14 are engineers and 4 are programmers... Contrasting this modest effort with our vast development activities, I fail to understand why we have lost our industry leadership position by letting someone else offer the world's most powerful computer.”

To which Cray replied: *“It seems like Mr. Watson has answered his own question.”*



CDC 6600:

A LOAD/STORE

ARCHITECTURE

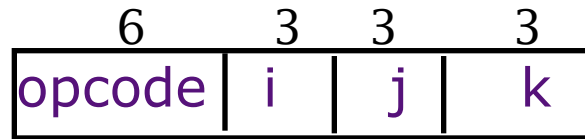
- Separate instructions to manipulate three types of reg.

8 60-bit data registers (X)

8 18-bit address registers (A)

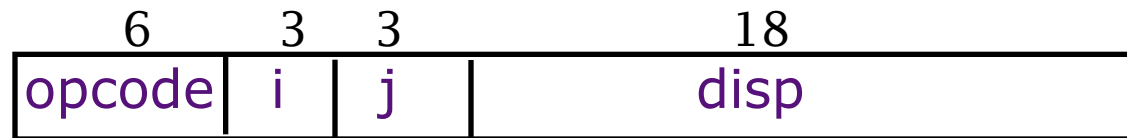
8 18-bit index registers (B)

- All arithmetic and logic instructions are reg-to-reg



$R_i \leftarrow (R_j) \text{ op } (R_k)$

- Only Load and Store instructions refer to memory!



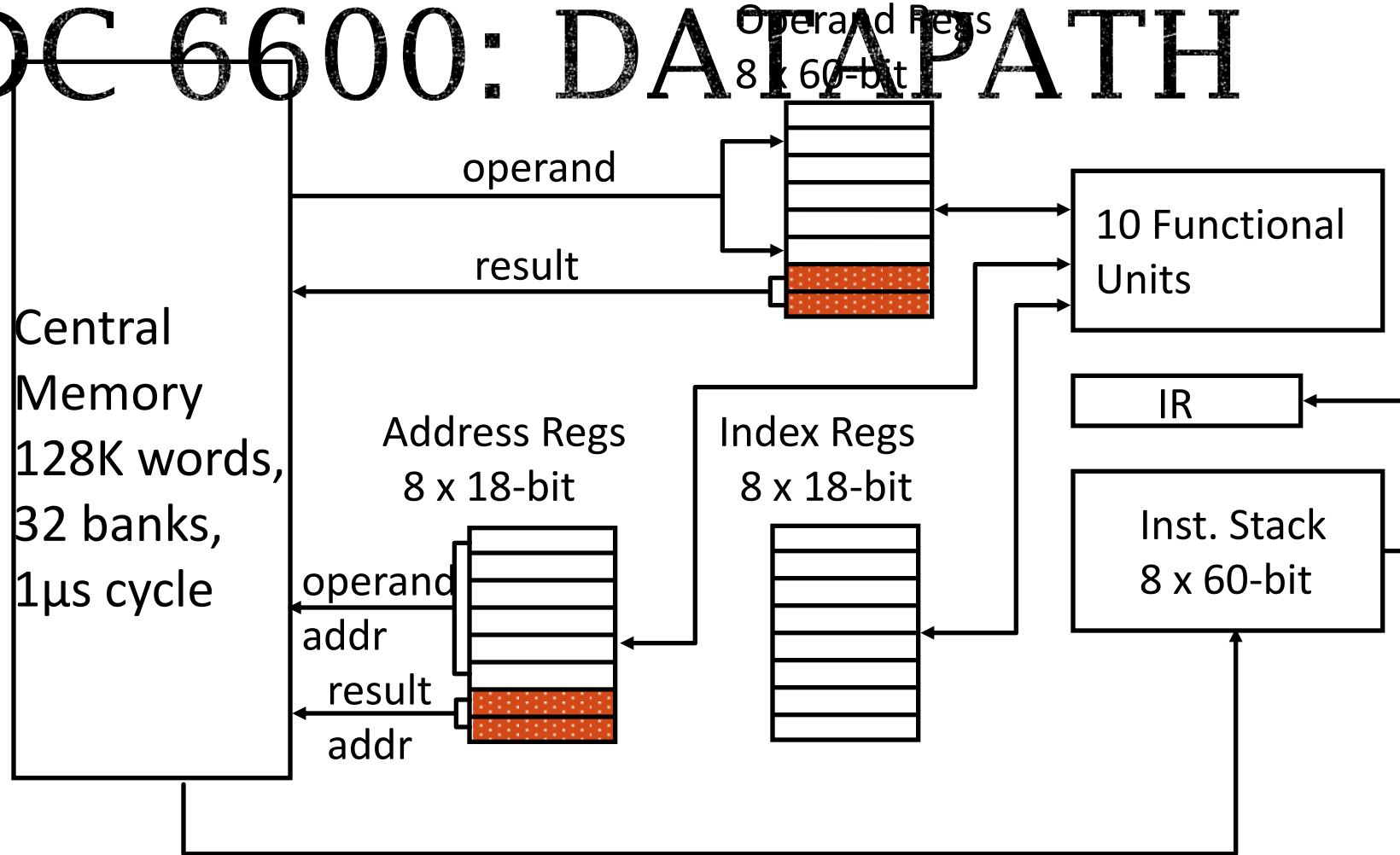
$R_i \leftarrow M[(R_j) + \text{disp}]$

Touching address registers 1 to 5 initiates a load

6 to 7 initiates a store

- *very useful for vector operations*

CDC 6600: DATAPATH



CDC6600 ISA DESIGNED TO SIMPLIFY HIGH- PERFORMANCE

- ## IMPLEMENTATION
- Use of three-address, register-register ALU instructions simplifies pipelined implementation
 - No implicit dependencies between inputs and outputs
 - Decoupling setting of address register (Ar) from retrieving value from data register (Xr) simplifies providing multiple outstanding memory accesses
 - Software can schedule load of address register before use of value
 - Can interleave independent instructions inbetween
 - CDC6600 has multiple parallel but unpipelined functional units
 - E.g., 2 separate multipliers
 - Follow-on machine CDC7600 used pipelined functional units
 - Foreshadows later RISC designs

CDC6600: VECTOR ADDITION

```
B0 [?] - n
loop: JZE B0, exit
      A0 [?] B0 + a0      load X0
      A1 [?] B0 + b0     load X1
      X6 [?] X0 + X1
      A6 [?] B0 + c0     store X6
      B0 [?] B0 + 1
      jump loop
```

A_i = address register

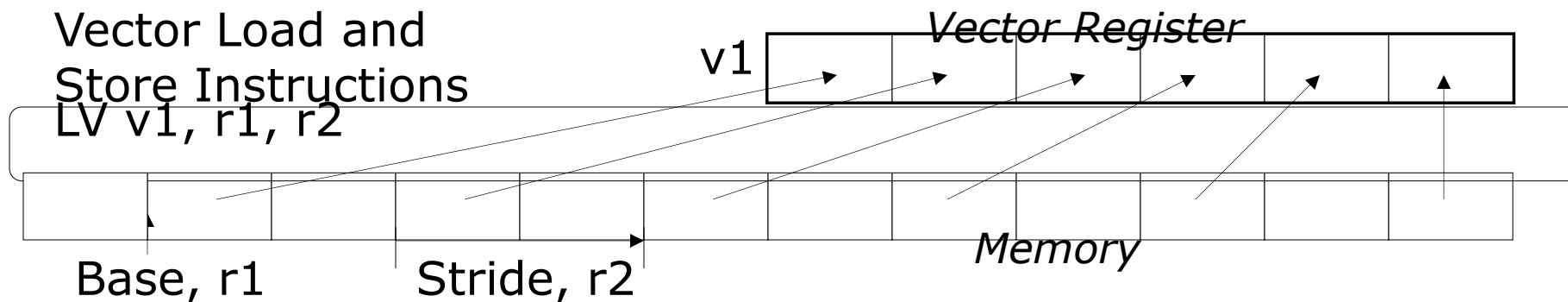
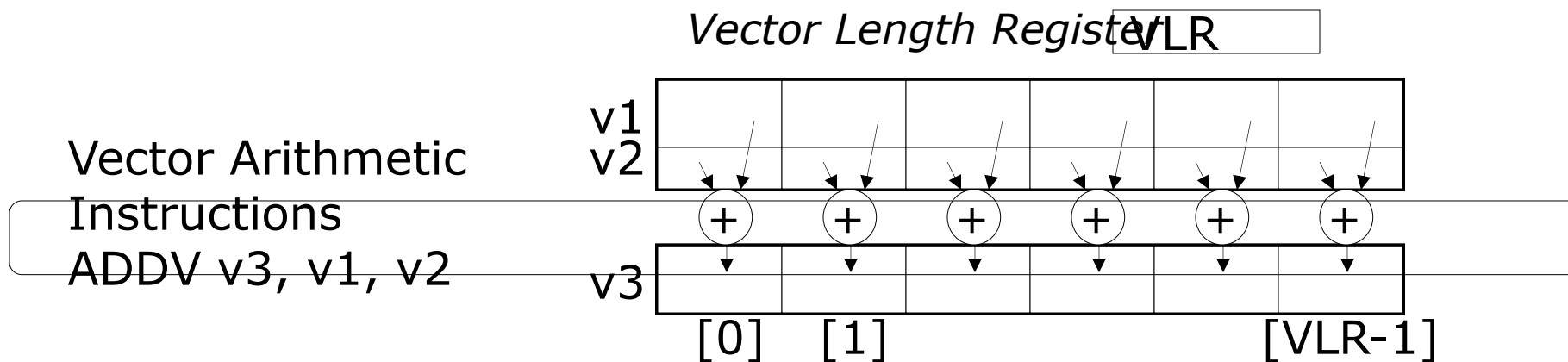
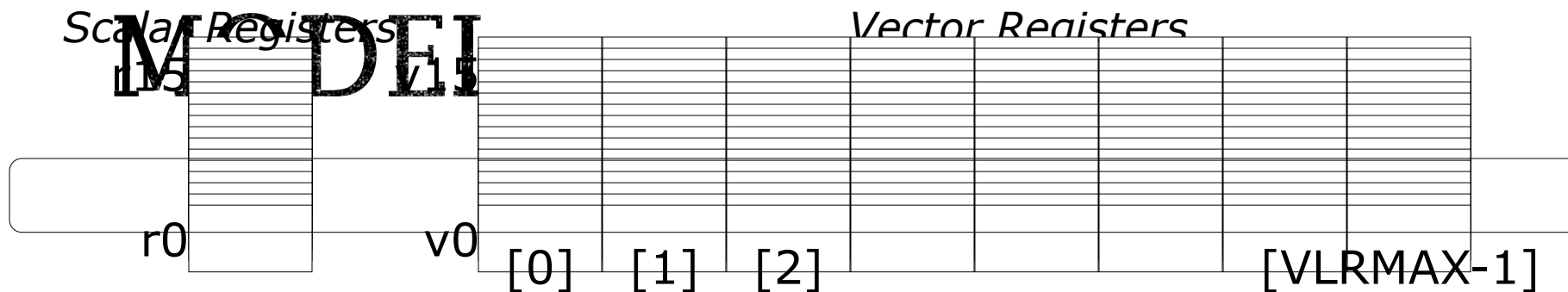
B_i = index register

X_i = data register

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 sets
- In 70s-80s, Supercomputer \equiv Vector Machine

PROGRAMMING



VECTOR CODE EXAMPLE

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

# Scalar Code
LI R4, 64
loop:
L.D F0, 0(R1)
L.D F2, 0(R2)
ADD.D F4, F2, F0
S.D F4, 0(R3)
DADDIU R1, 8
DADDIU R2, 8
DADDIU R3, 8
DSUBIU R4, 1
BNEZ R4, loop

# Vector Code
LI VLR, 64
LV V1, R1
LV V2, R2
ADDV.D V3, V1, V2
SV V3, R3
```

	ADD.D F4, F2, F0	
	S.D F4, 0(R3)	

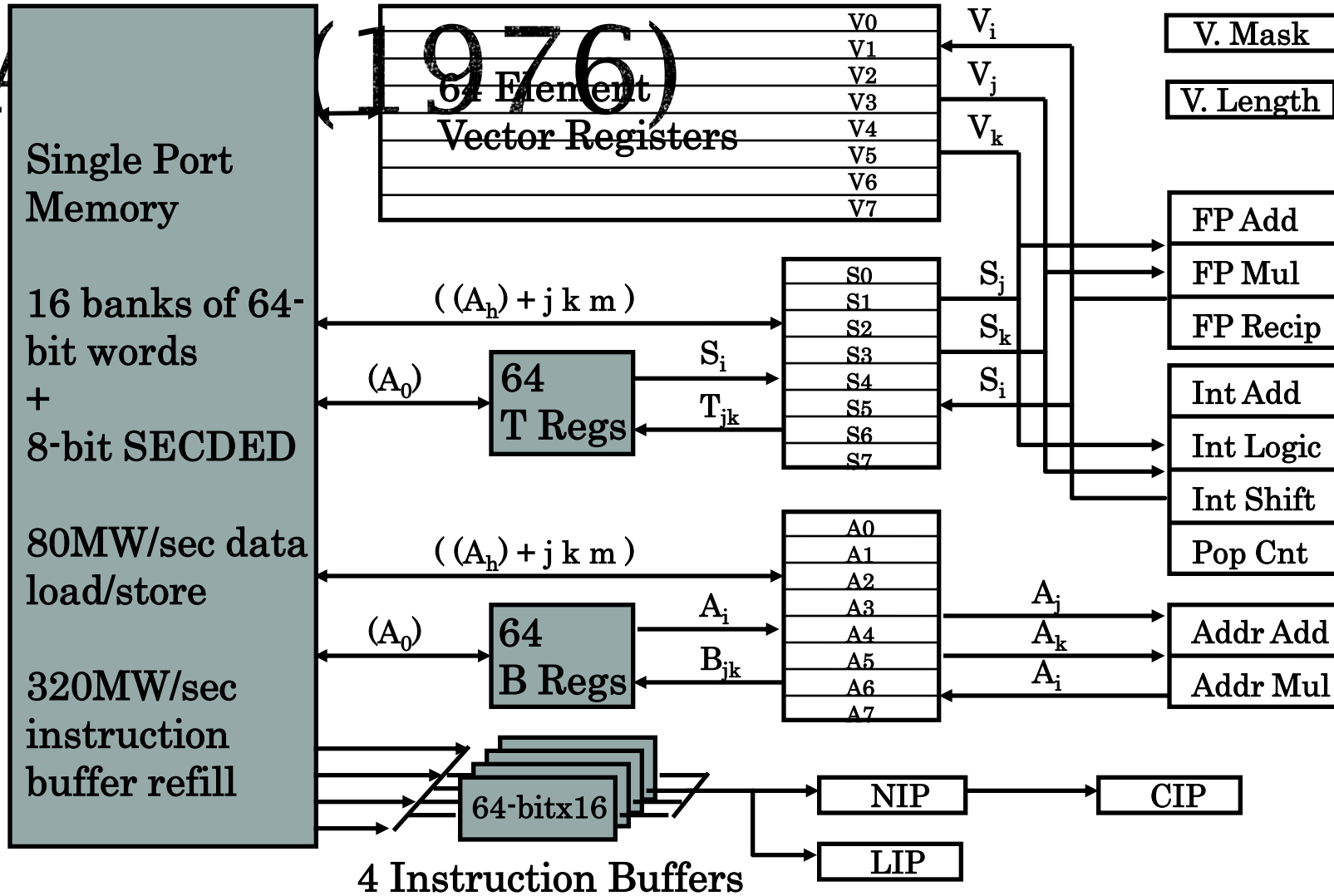
VECTORIZED SUPERCOMPUTERS

▪ 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



CRA



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

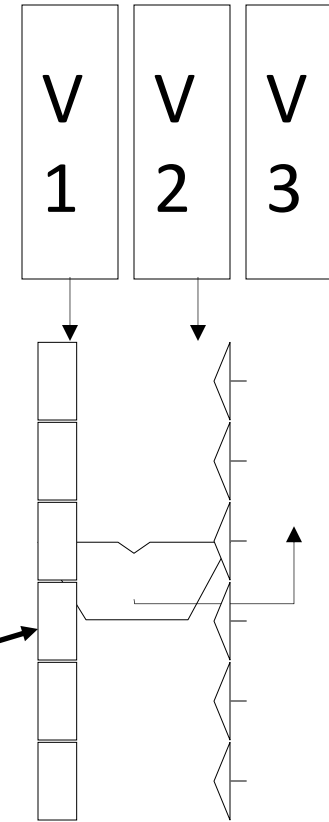
VECTOR INSTRUCTION SET

- ## 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 (\Rightarrow fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (\Rightarrow no hazards!)

Six stage multiply pipeline



$$V3 \leftarrow v1 * v2$$

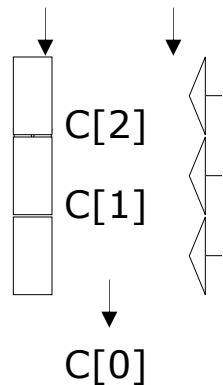
VECTOR INSTRUCTION EXECUTION

ADDV C,A,B

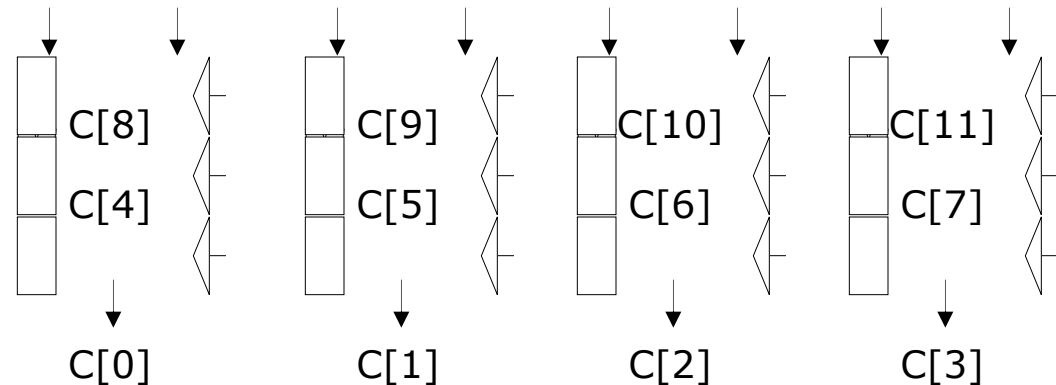
Execution using
one pipelined
functional unit

Execution using
four pipelined
functional units

A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]



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]

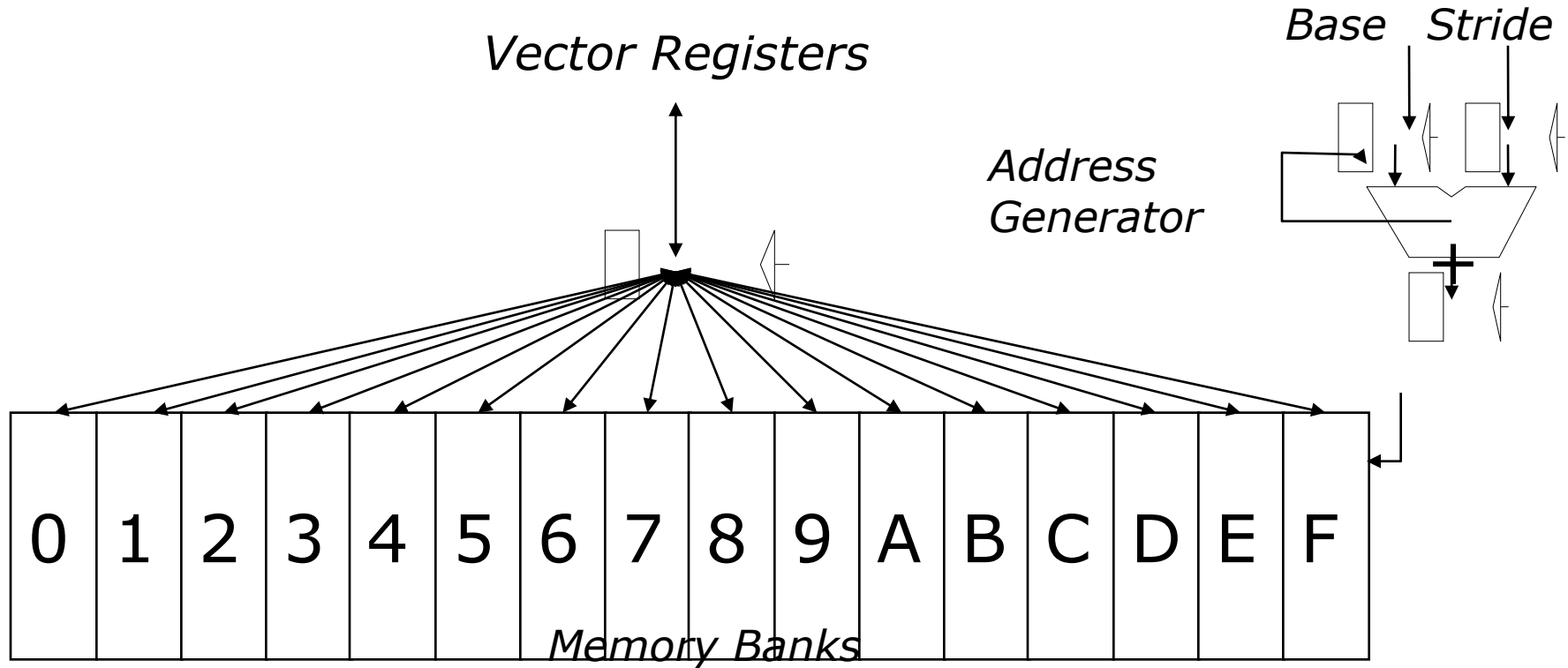


INTERLEAVED VECTOR

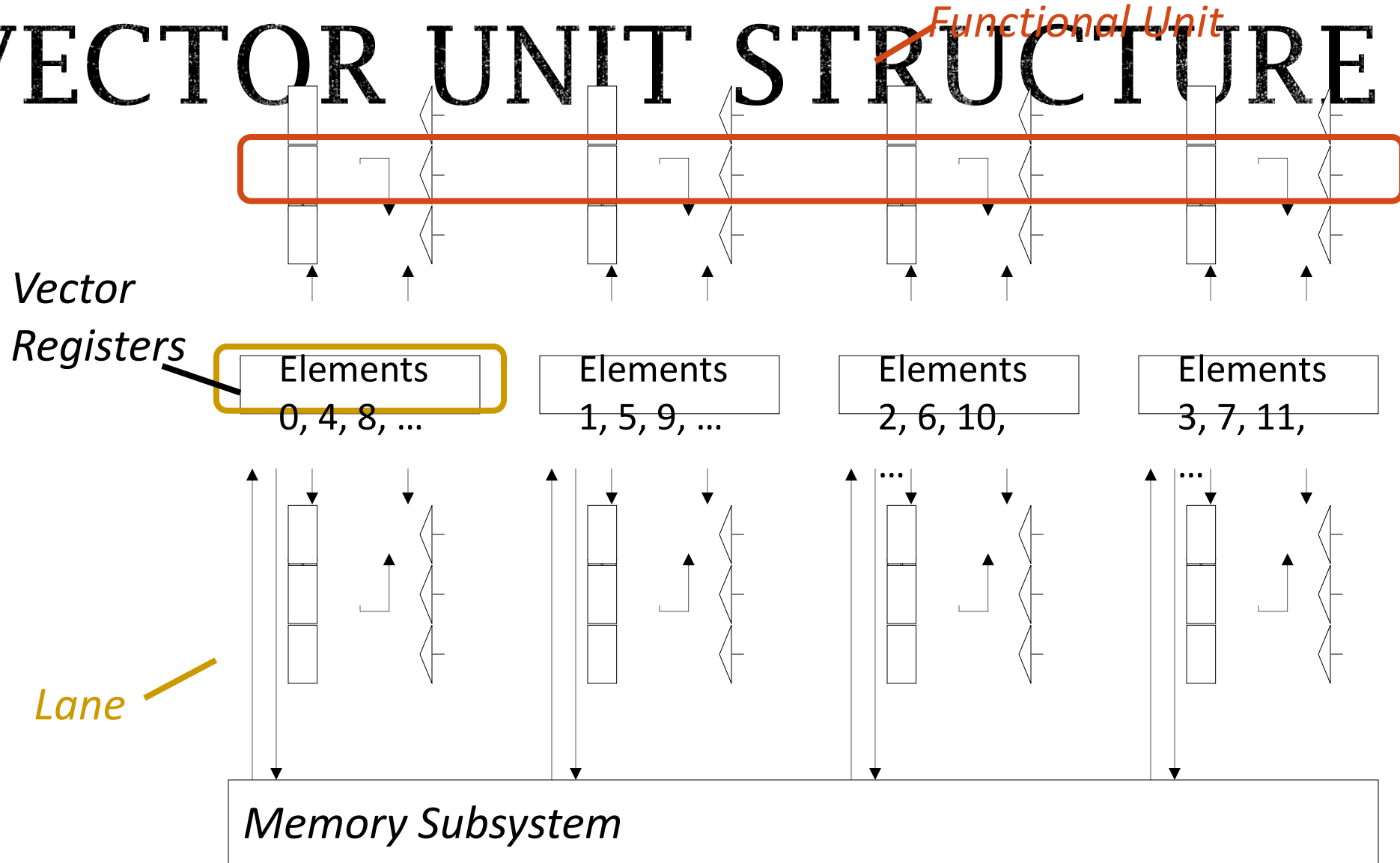
Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

MEMORY SYSTEM

Bank busy time: Time before bank ready to accept next request

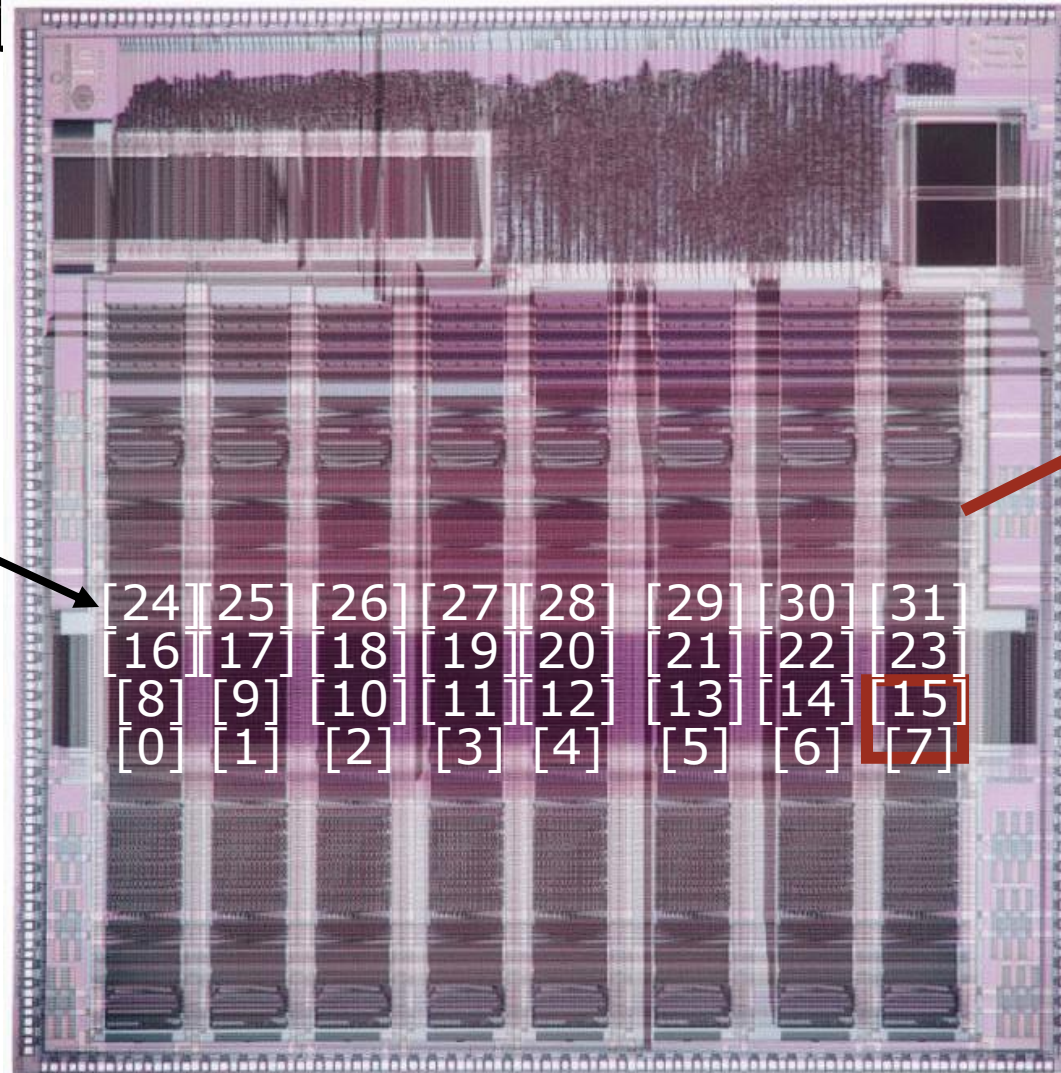


VECTOR UNIT STRUCTURE



MICROPROCESSOR (UCB/ICST 1005)

*Vector register
elements striped
over lanes*

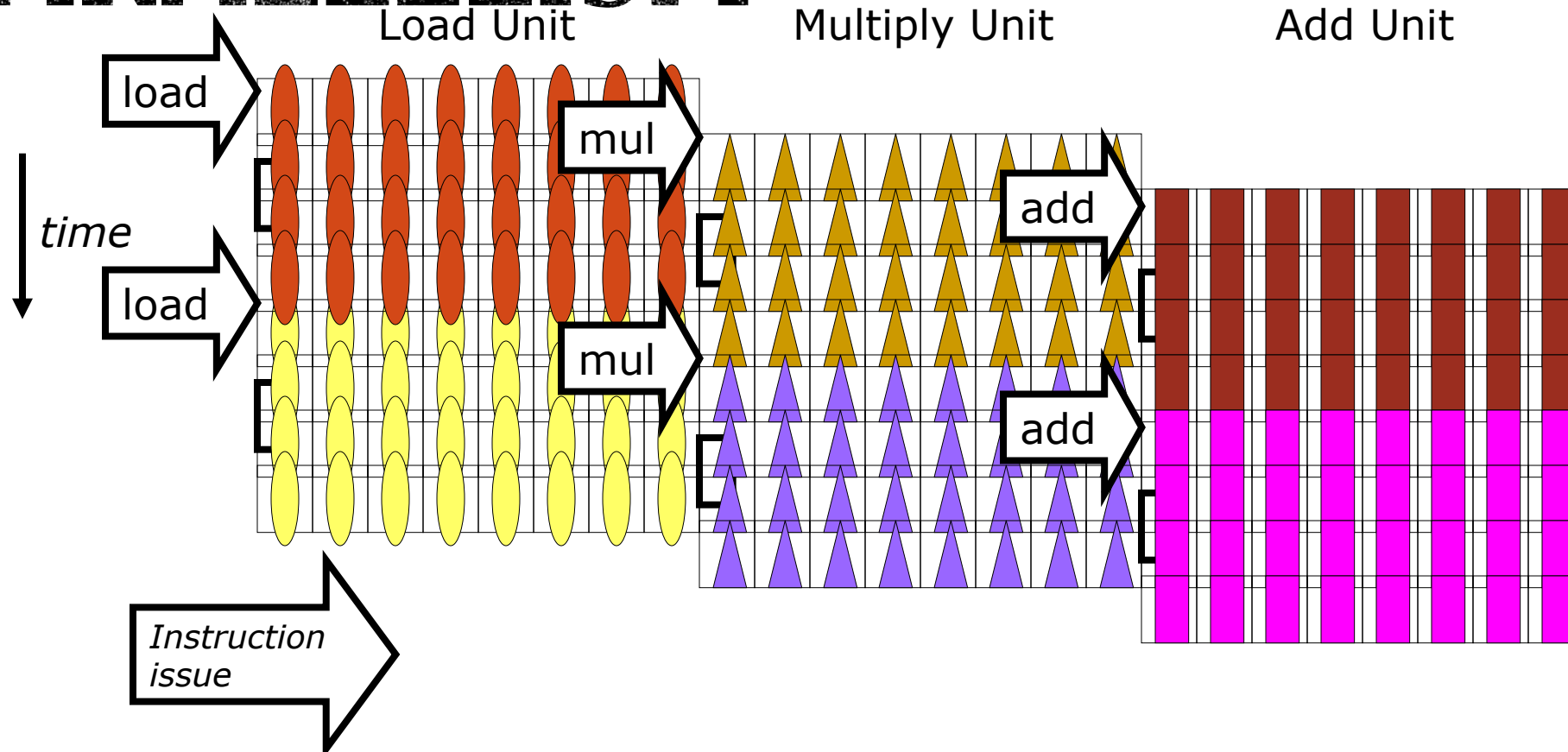


VECTOR INSTRUCTION

- Can overlap execution of multiple vector instructions

PARALLELISM

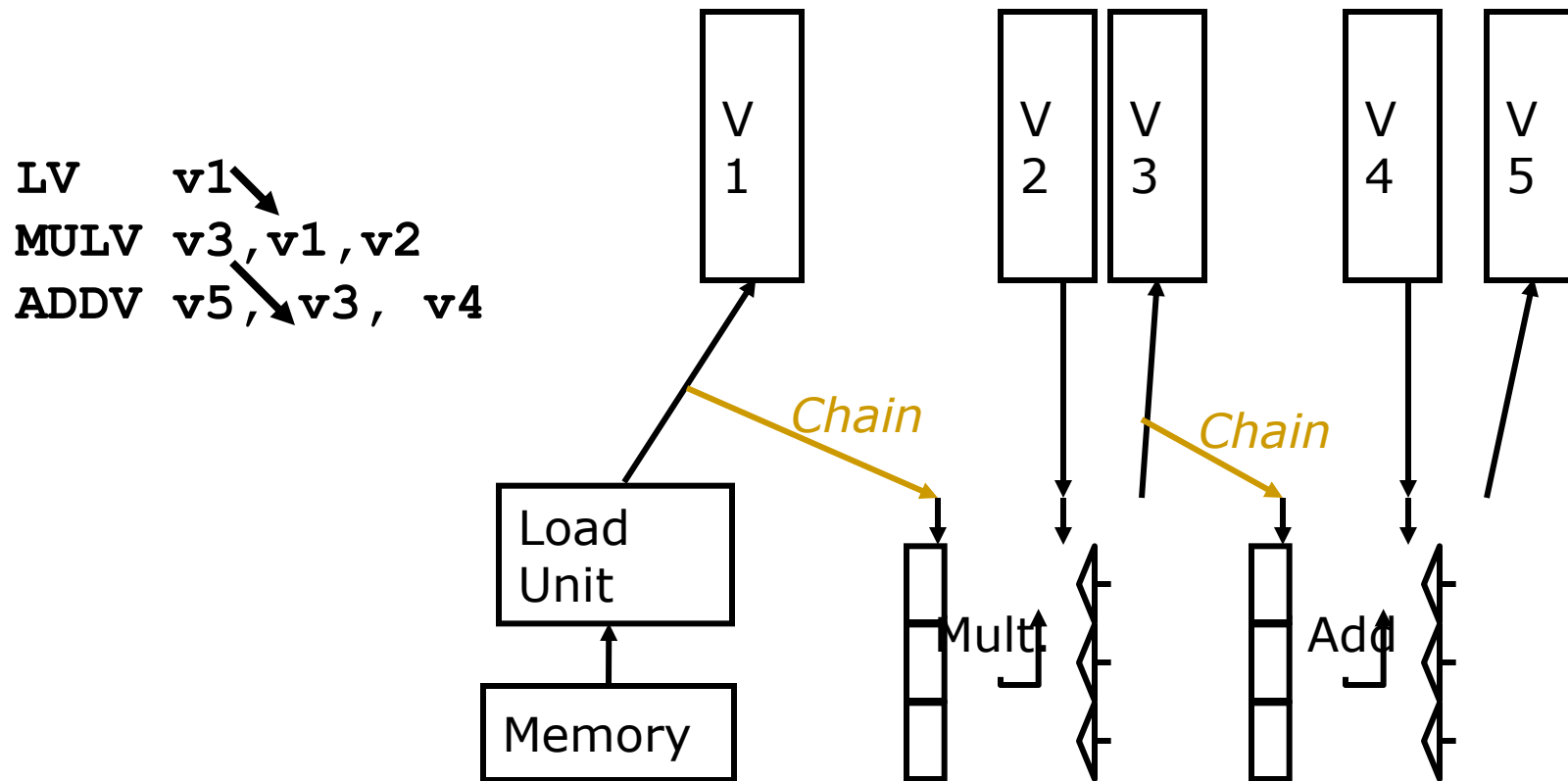
- example machine has 32 elements per vector register and 8 lanes



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

VECTOR CHAINING

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

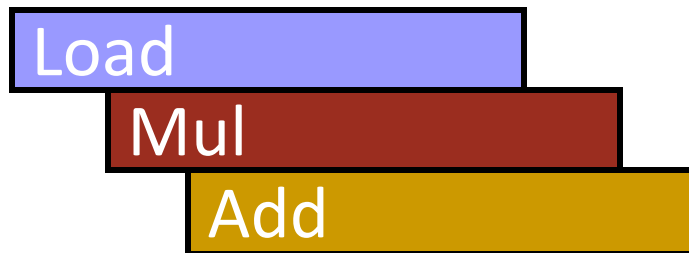


VECTOR CHAINING

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

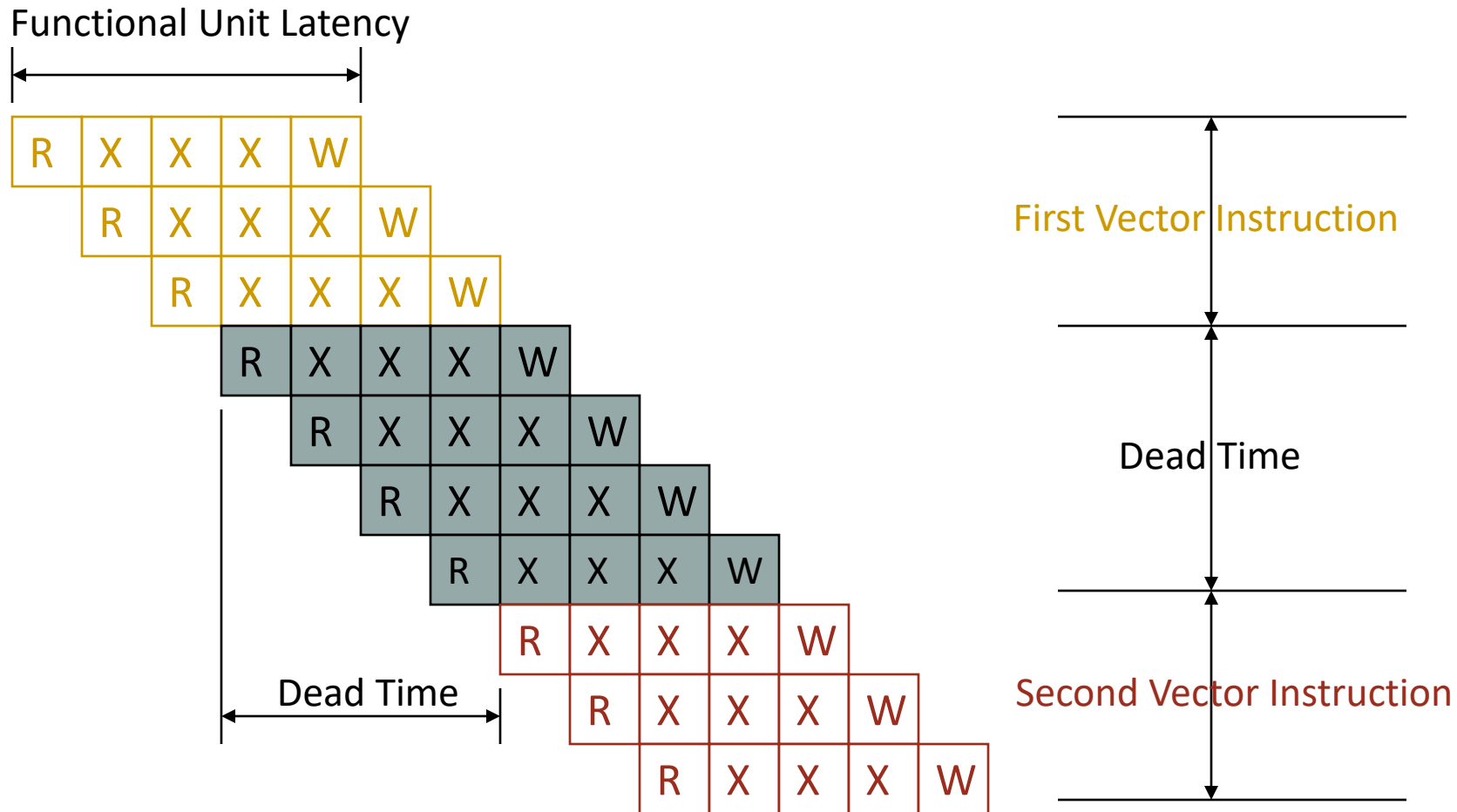


- With chaining, can start dependent instruction as soon as first result appears

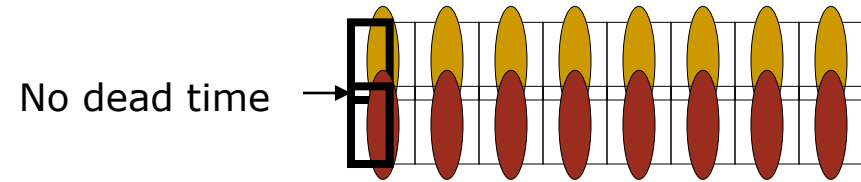
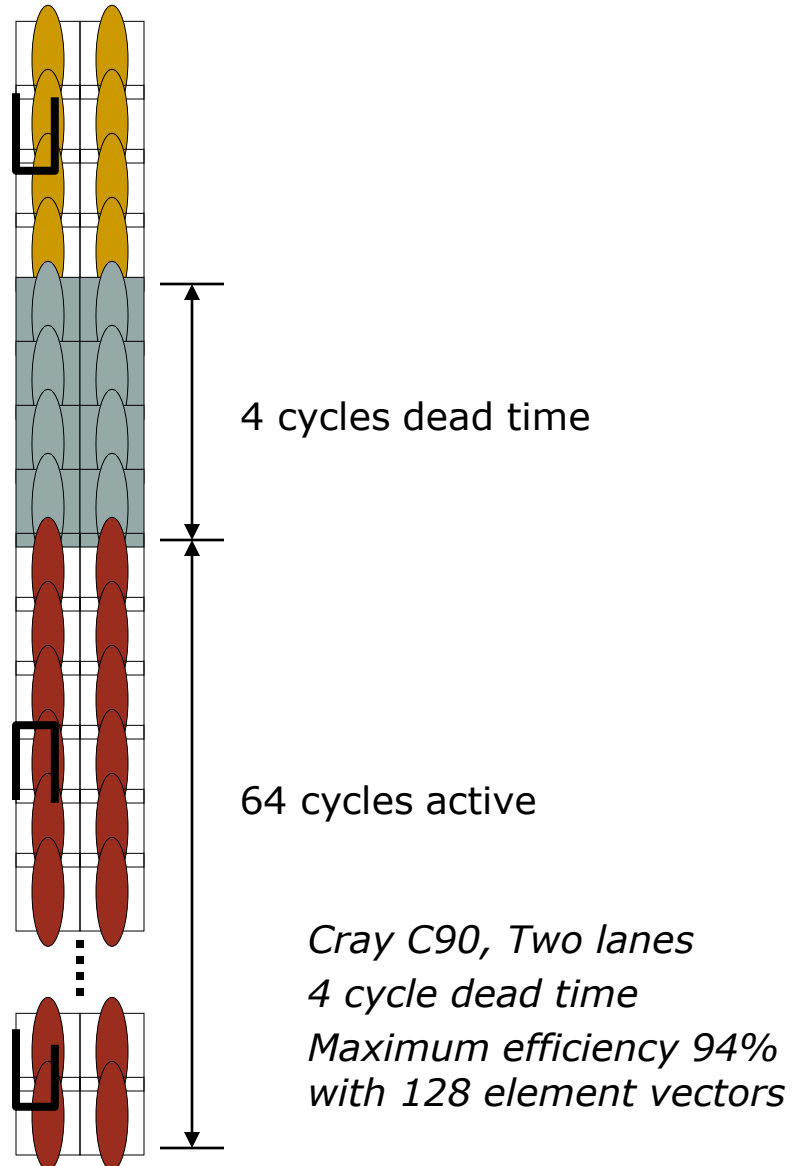


VECTOR STARTUP

- 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

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

```
ADDV C, A, B  
SUBV D, A, B
```

Vector Register Code

```
LV V1, A  
LV V2, B  
ADDV V3, V1, V2  
SV V3, C  
SUBV V4, V1, V2  
SV V4, D
```

VECTOR MEMORY-MEMORY VS. VECTOR REGISTER MACHINES

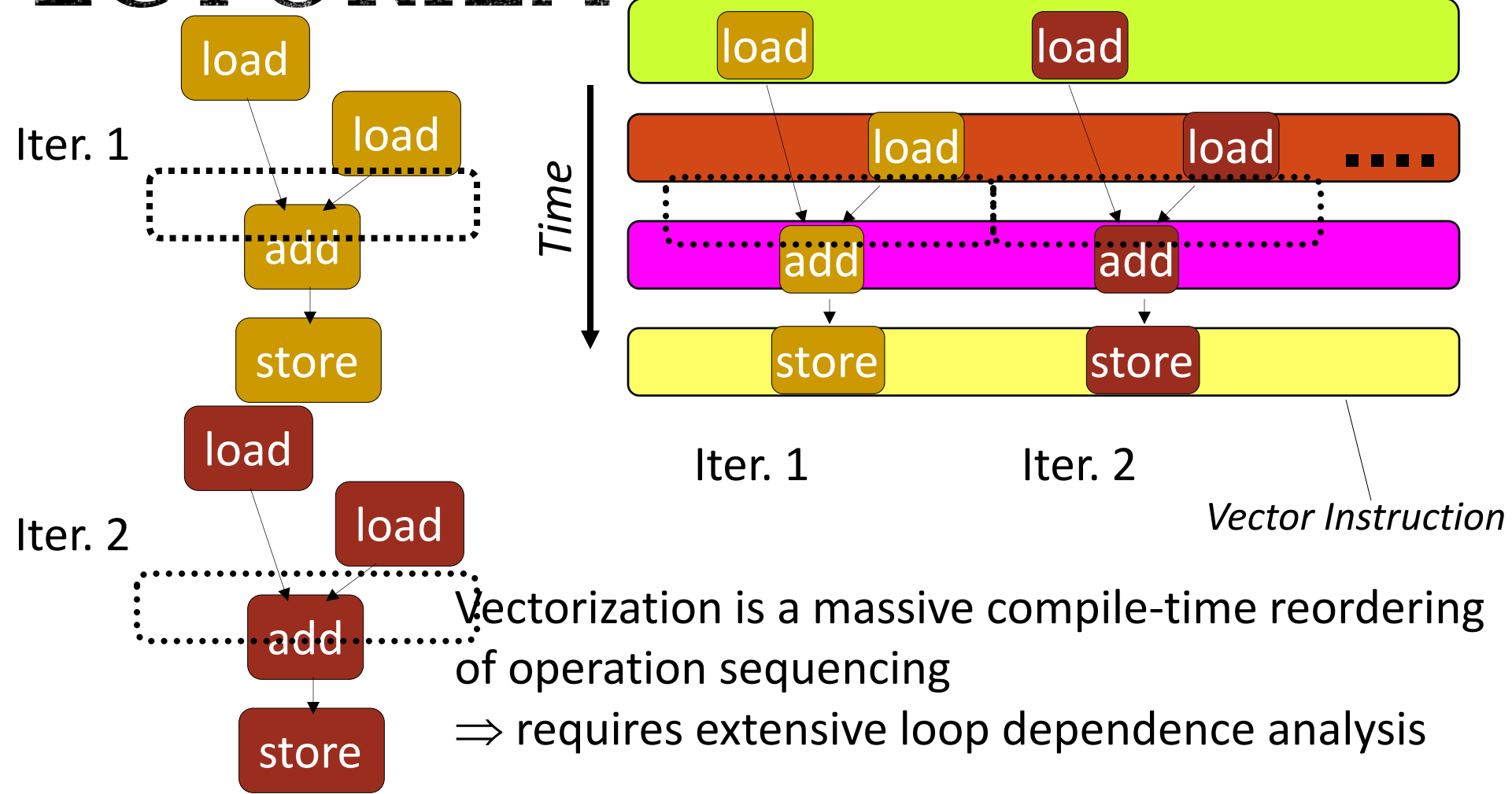
- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
 - All operands must be read in and out of memory
- VMMA make it difficult to overlap execution of multiple vector operations, why?
 - Must check dependencies on memory addresses
- VMMA incur greater startup latency
 - Scalar code was faster on CDC Star-100 for vectors < 100 elements
 - For Cray-1, vector/scalar breakeven point was around 2 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

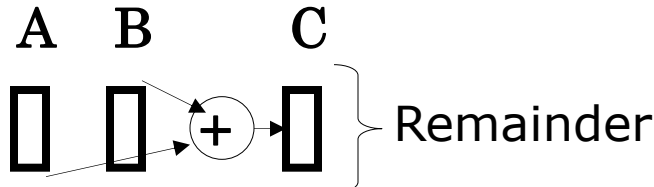
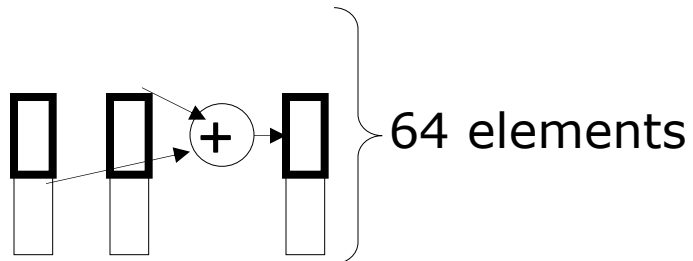
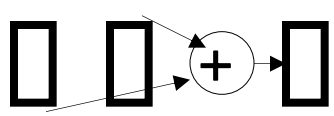


Vectorization is a massive compile-time reordering of operation sequencing
 ⇒ requires extensive loop dependence analysis

VECTOR STRIPMINING

Problem: Vector registers have finite length

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

	<pre> ANDI R1, N, 63 # N mod 64 MTC1 VLR, R1 # Do remainder </pre>
<pre> for (i=0; i<N; i++) C[i] = A[i]+B[i]; </pre>	<pre> loop: LV V1, RA DSSL R2, R1, 3 # Multiply by 8 DADDU RA, RA, R2 # Bump pointer LV V2, RB DADDU RB, RB, R2 ADDV.D V3, V1, V2 SV V3, RC DADDU RC, RC, R2 DSUBU N, N, R1 # Subtract elements LI R1, 64 MTC1 VLR, R1 # Reset full length BGTZ N, loop # Any more to do? </pre>
	
	
	

VECTOR CONDITIONAL

Problem. Want to vectorize loops with conditional code:

```
EXECUTION 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
LV vA, rA          # Load entire A vector
SGTVS.D vA, F0    # Set bits in mask register where A>0
LV vA, rB          # Load B vector into A under mask
SV vA, rA          # Store A back to memory under mask
```

MASKED VECTOR

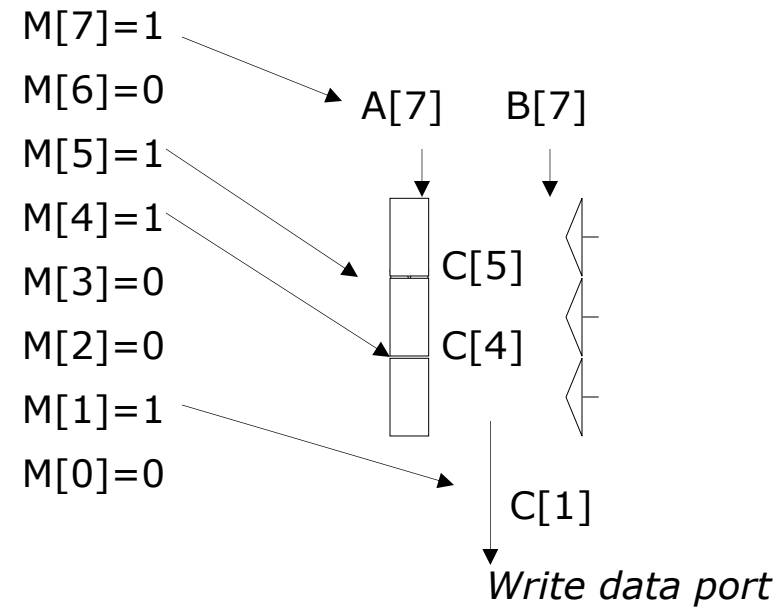
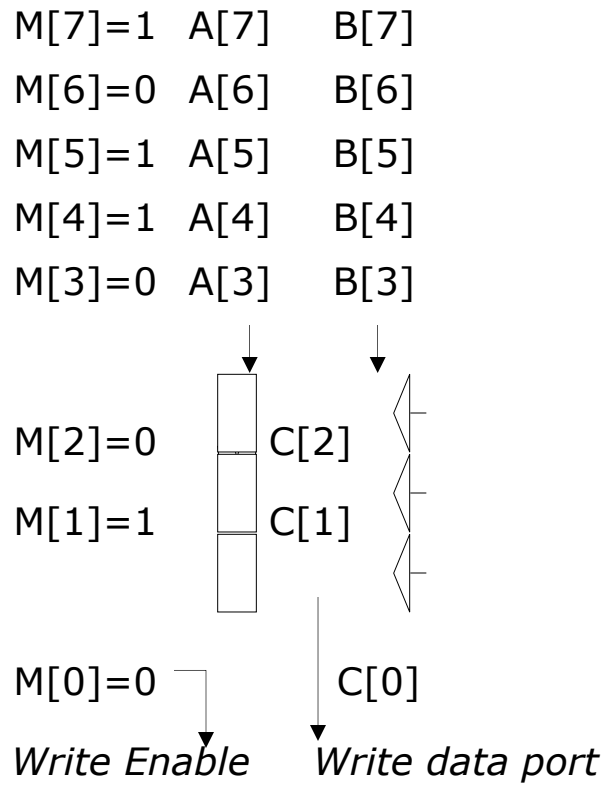
Simple Implementation

Density-Time Implementation

INSTRUCTIONS

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

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



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*)

```
LV vD, rD          # Load indices in D vector  
LVI vC, rC, vD     # Load indirect from rC base  
LV vB, rB          # Load B vector  
ADDV.D vA, vB, vC  # Do add  
SV vA, rA          # Store result
```

VECTOR SCATTER/GATHER

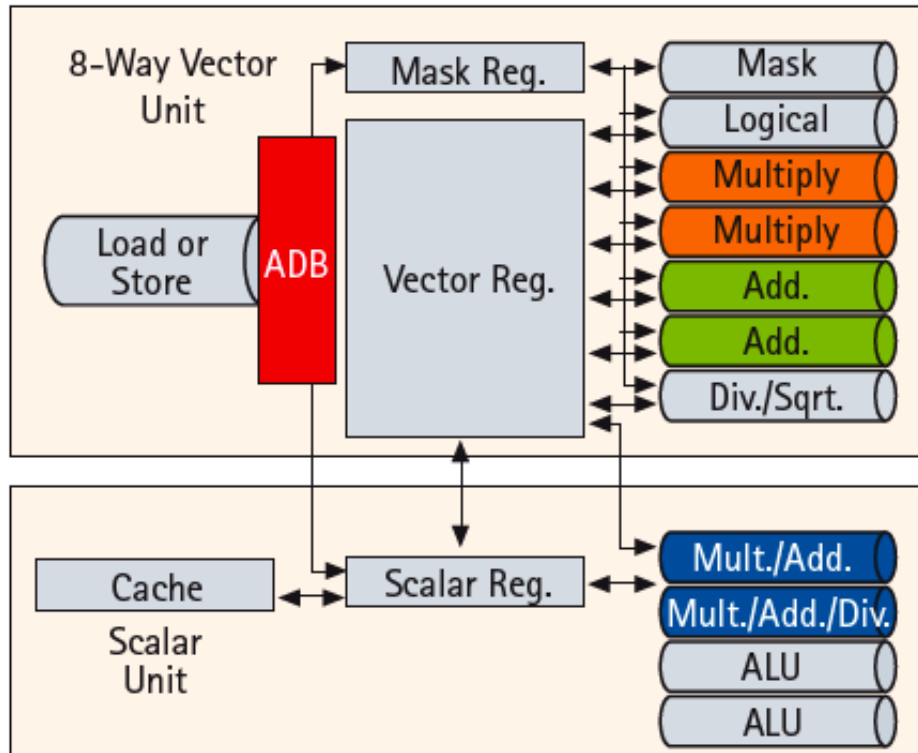
Histogram example:

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

Is following a correct translation?

```
LV vB, rB          # Load indices in B vector  
LVI vA, rA, vB     # Gather initial A values  
ADDV vA, vA, 1     # Increment  
SVI vA, rA, vB     # Scatter incremented values
```

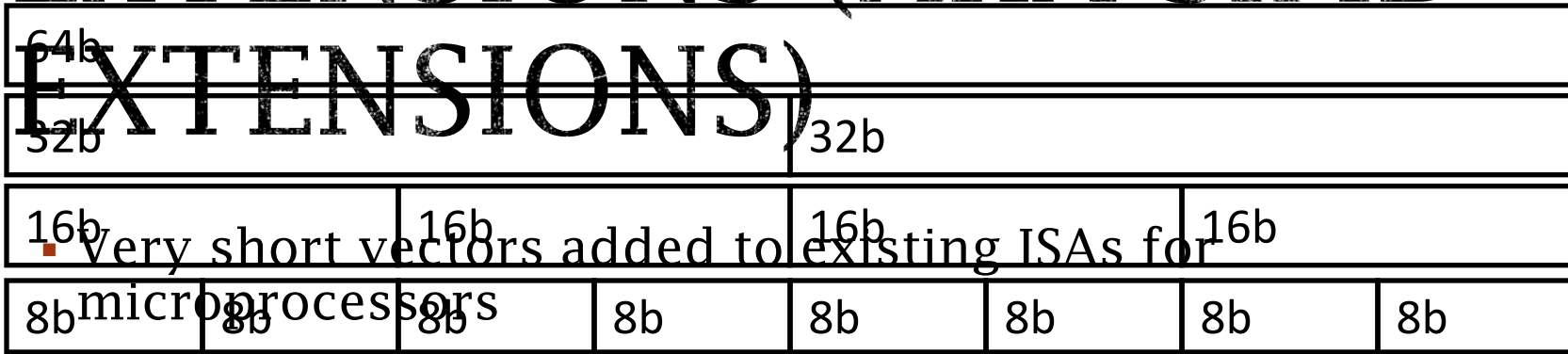
A MODERN VECTOR SUPER: NEC SX-9 (2008)



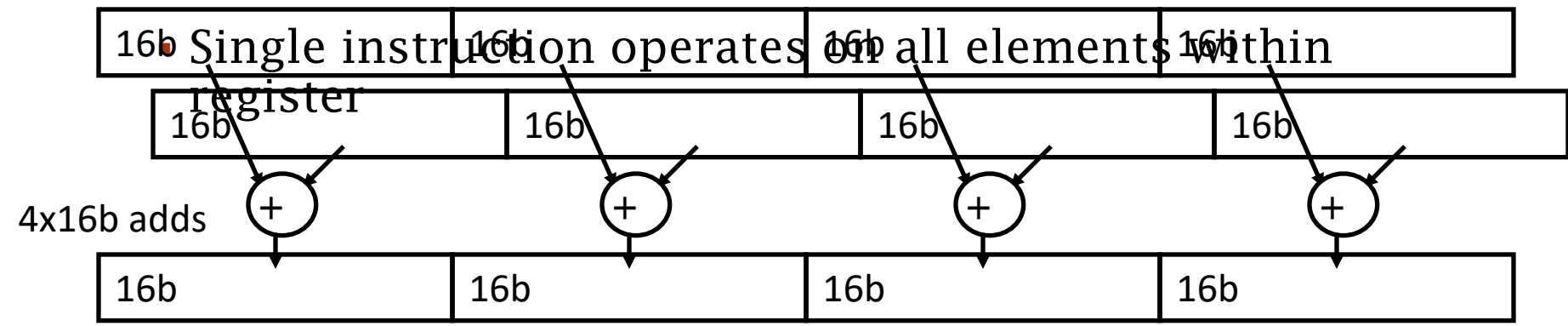
- 65nm CMOS technology
- Vector unit (3.2 GHz)
 - 8 foreground VRegs + 64 background VRegs (256x64-bit elements/VReg)
 - 64-bit functional units: 2 multiply, 2 add, 1 divide/sqrt, 1 logical, 1 mask unit
 - 8 lanes (32+ FLOPS/cycle, 100+ GFLOPS peak per CPU)
 - 1 load or store unit (8 x 8-byte accesses/cycle)
- Scalar unit (1.6 GHz)
 - 4-way superscalar with out-of-order and speculative execution
 - 64KB I-cache and 64KB data cache

- Memory system provides 256GB/s DRAM bandwidth per CPU
- Up to 16 CPUs and up to 1TB DRAM form shared-memory *node*
 - total of 4TB/s bandwidth to shared DRAM memory
- Up to 512 nodes connected via 128GB/s network links (message passing between nodes)

EXTENSIONS (AKA SIMD)



- 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 for Intel AVX



MULTIMEDIA EXTENSIONS

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)

ACKNOWLEDGEMENTS

- These slides contain material developed and copyright by:
 - Arvind (MIT)
 - Krste Asanovic (MIT/UCB)
 - Joel Emer (Intel/MIT)
 - James Hoe (CMU)
 - John Kubiatowicz (UCB)
 - David Patterson (UCB)
- MIT material derived from course 6.823
- UCB material derived from course CS252