

Vector Computers

Joel Emer

Computer Science & Artificial Intelligence Lab
M.I.T.

Loop Unrolled Code Schedule

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
  
```

Schedule →

	Int1	Int 2	M1	M2	FP+	FPx
loop:			ld f1			
			ld f2			
			ld f3			
	add r1		ld f4		fadd f5	
					fadd f6	
					fadd f7	
					fadd f8	
			sd f5			
			sd f6			
			sd f7			
	add r2	bne	sd f8			

Vector Supercomputers

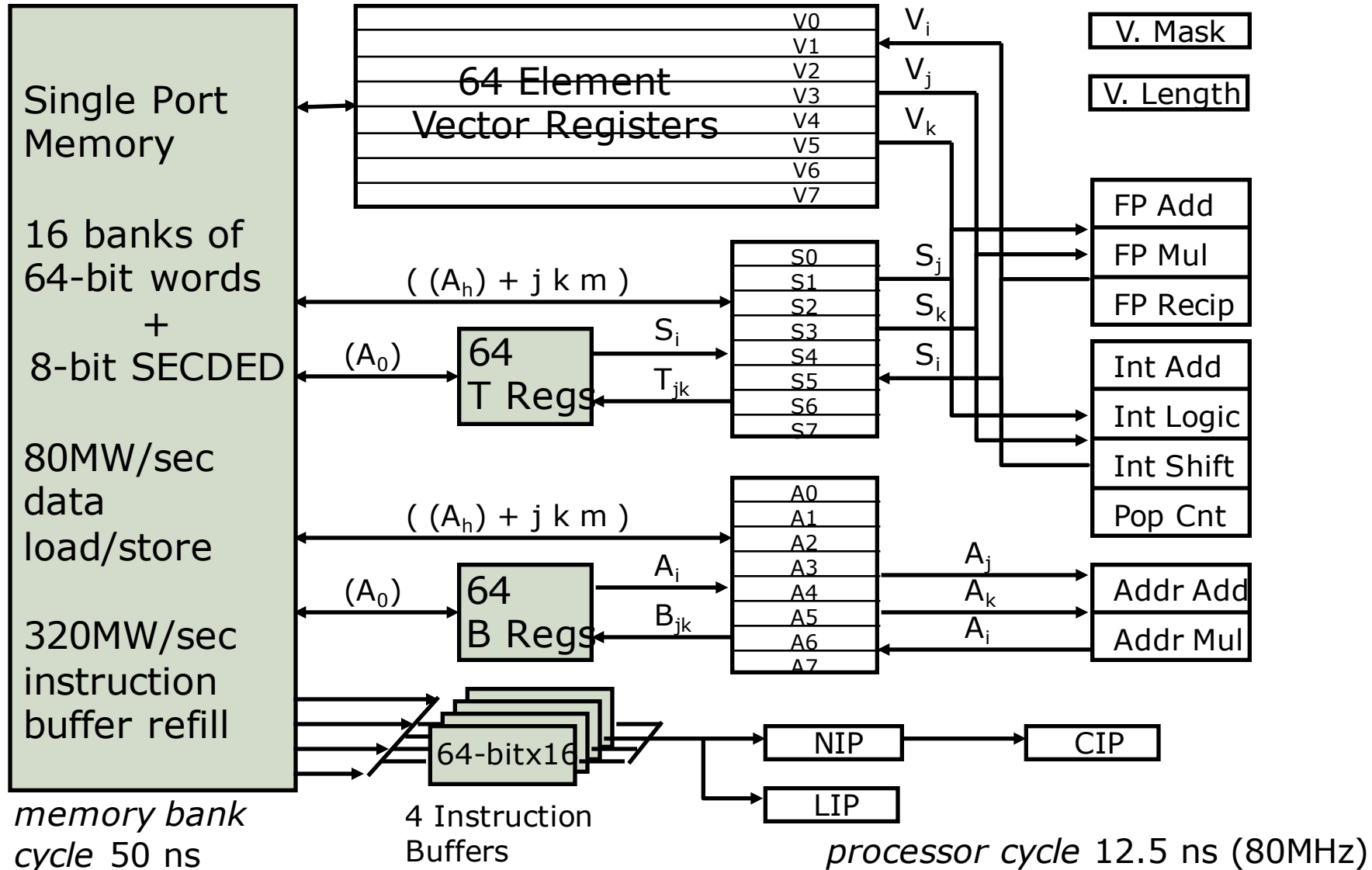
Epitomized by Cray-1, 1976:

- Scalar Unit
 - Load/Store Architecture
- Vector Extension
 - Vector Registers
 - Vector Instructions
- Implementation
 - Hardwired Control
 - Highly Pipelined Functional Units
 - No Data Caches
 - Interleaved Memory System
 - No Virtual Memory

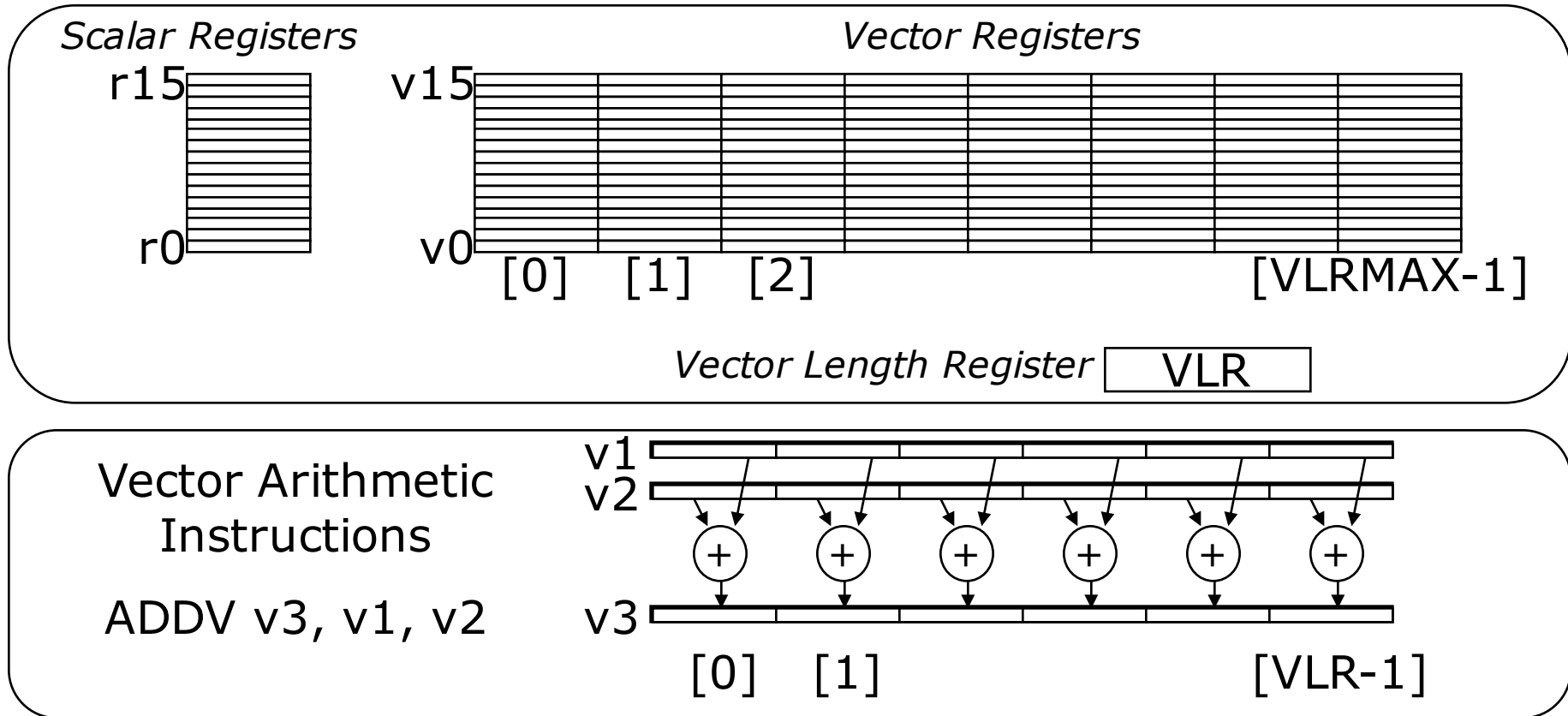
Cray-1 (1976)



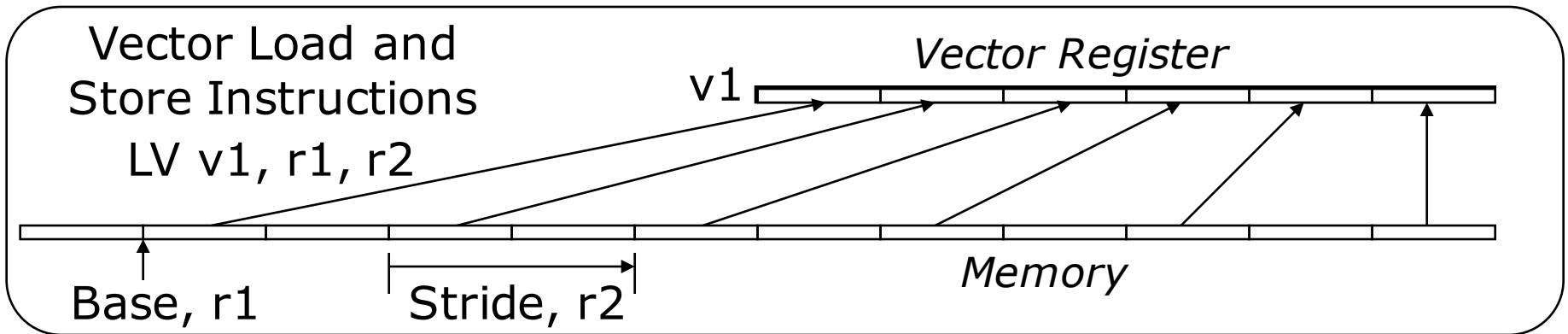
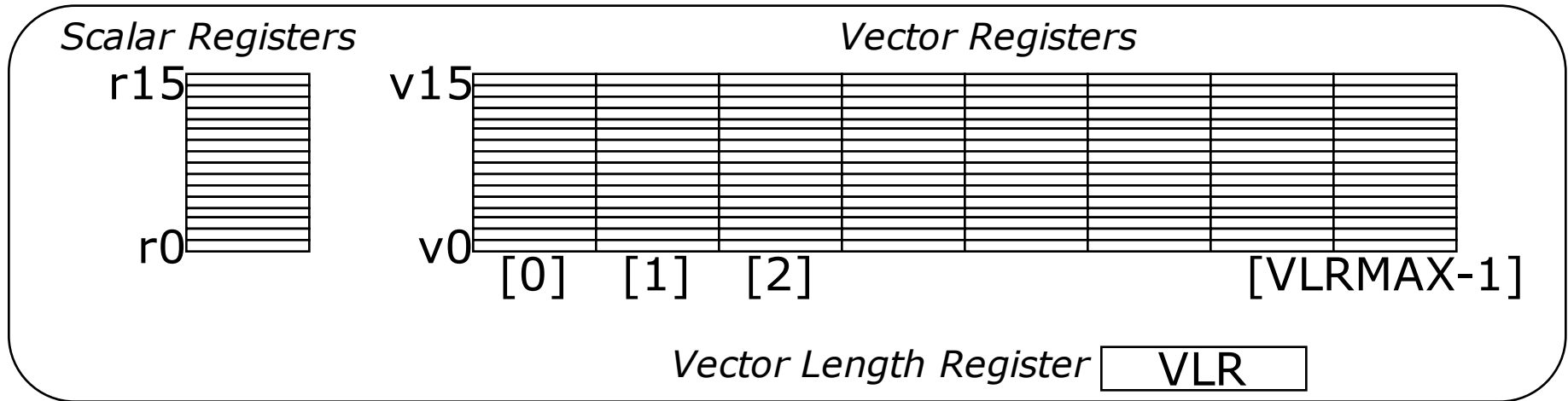
Cray-1 (1976)



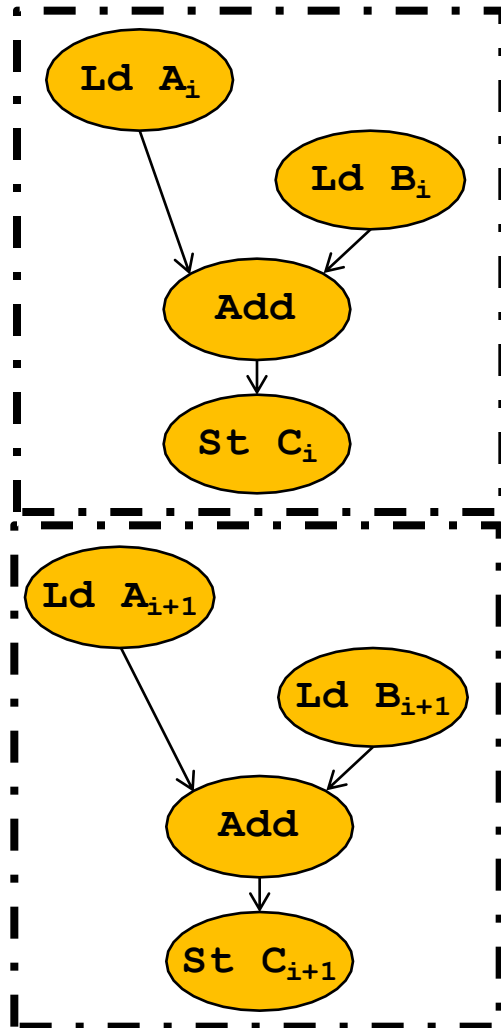
Vector Programming Model



Vector Programming Model



Compiler-based Vectorization

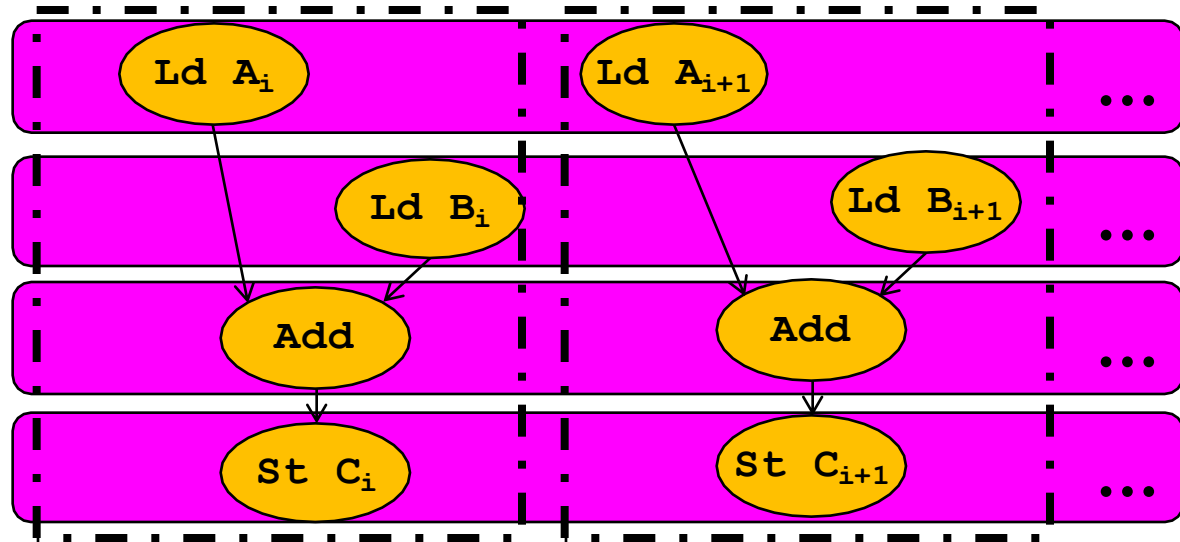


Scalar code

```

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

Compiler recognizes independent operations with loop dependence analysis



Vector code

Vector Code Example

# C code	# Scalar Code	# Vector Code
<pre>for (i=0; i<64; i++) C[i] = A[i] + B[i];</pre>	<pre>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</pre>	<pre>LI VLR, 64 LV V1, R1 LV V2, R2 ADDV.D V3, V1, V2 SV V3, R3</pre>

Vector ISA Attributes

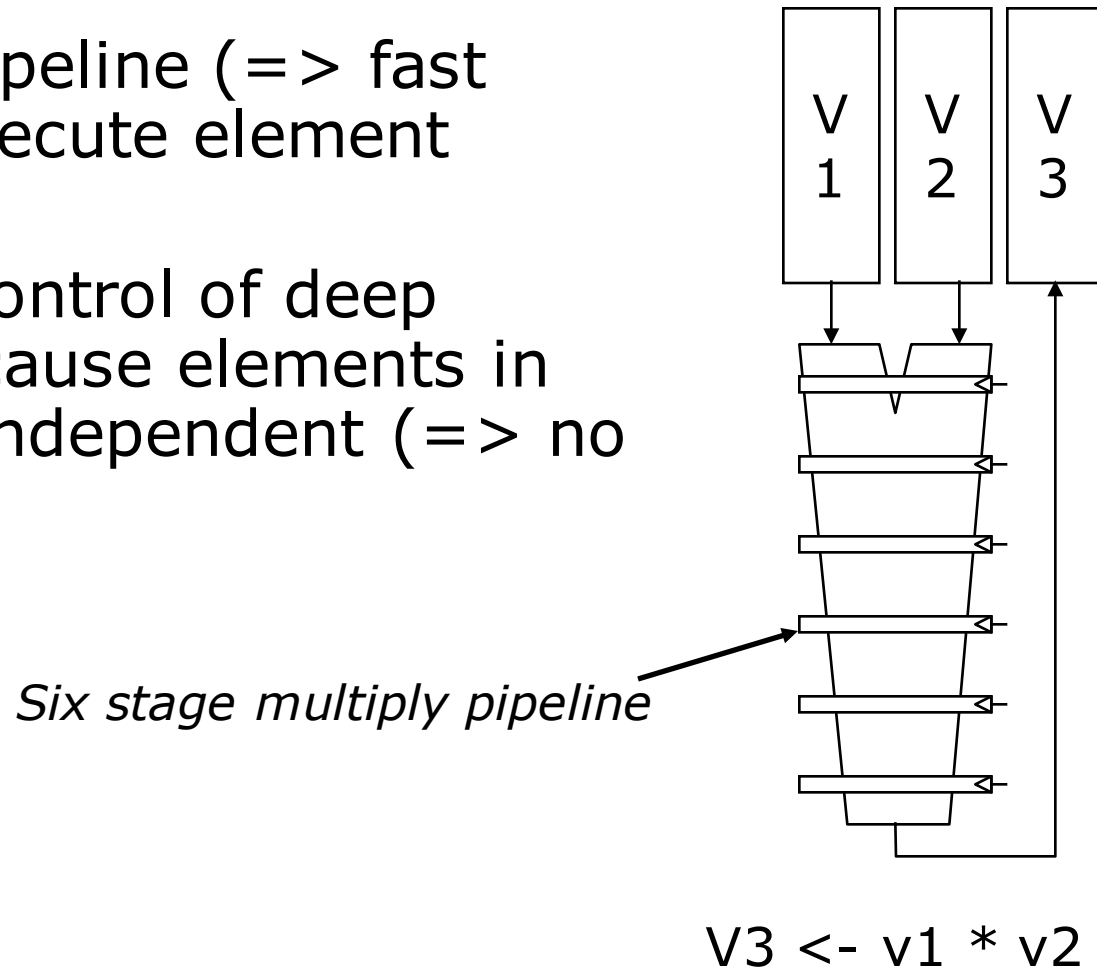
- Compact
 - one short instruction encodes N operations
 - many implicit bookkeeping/control operations
- Expressive, tells hardware that these N operations:
 - are independent
 - use the same functional unit
 - access disjoint registers
 - access registers in same pattern as previous instructions
 -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

Vector ISA Hardware Implications

- Large amount of work per instruction
 - > Less instruction fetch bandwidth requirements
 - > Allows simplified instruction fetch design
- Implicit bookkeeping operations
 - > Bookkeeping can run in parallel with main compute
- Disjoint vector element accesses
 - > Banked rather than multi-ported register files
- No data dependence within a vector
 - > Amenable to deeply pipelined/parallel designs
- Known regular memory access pattern
 - > Allows for banked memory for higher bandwidth

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!)

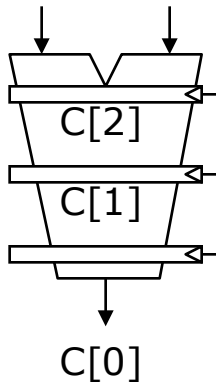


Vector Instruction Execution

ADDV C,A,B

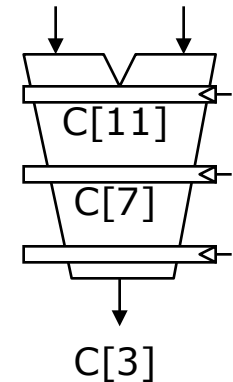
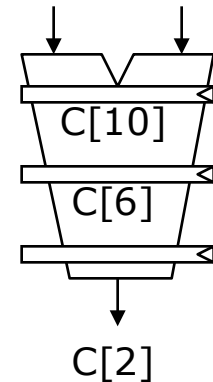
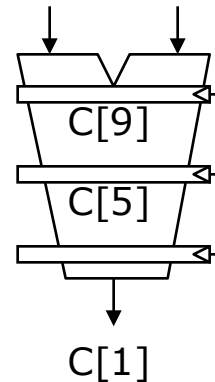
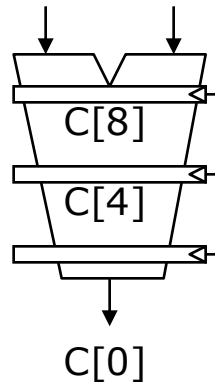
*Execution using
one pipelined
functional unit*

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

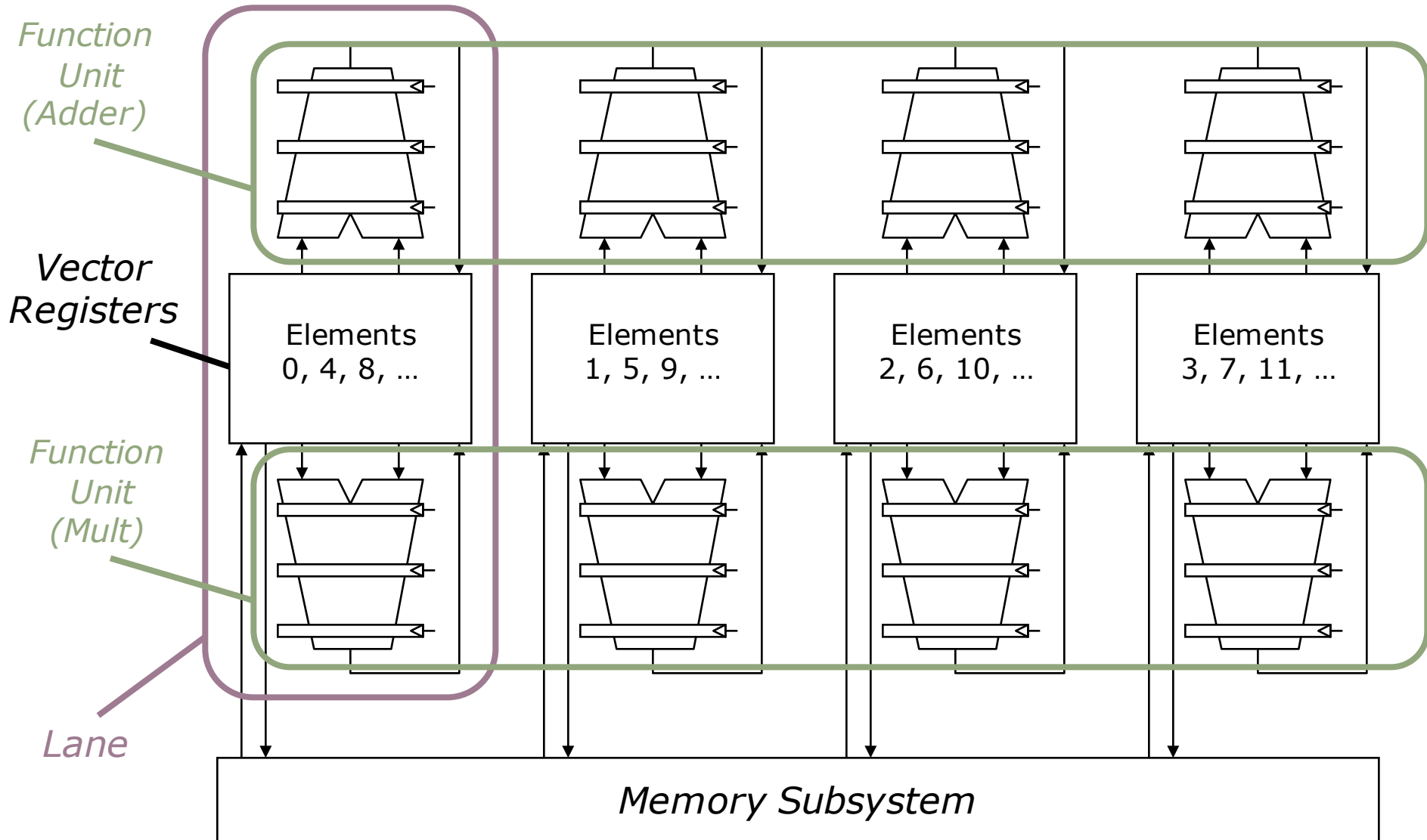


*Execution using
four pipelined
functional units*

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]



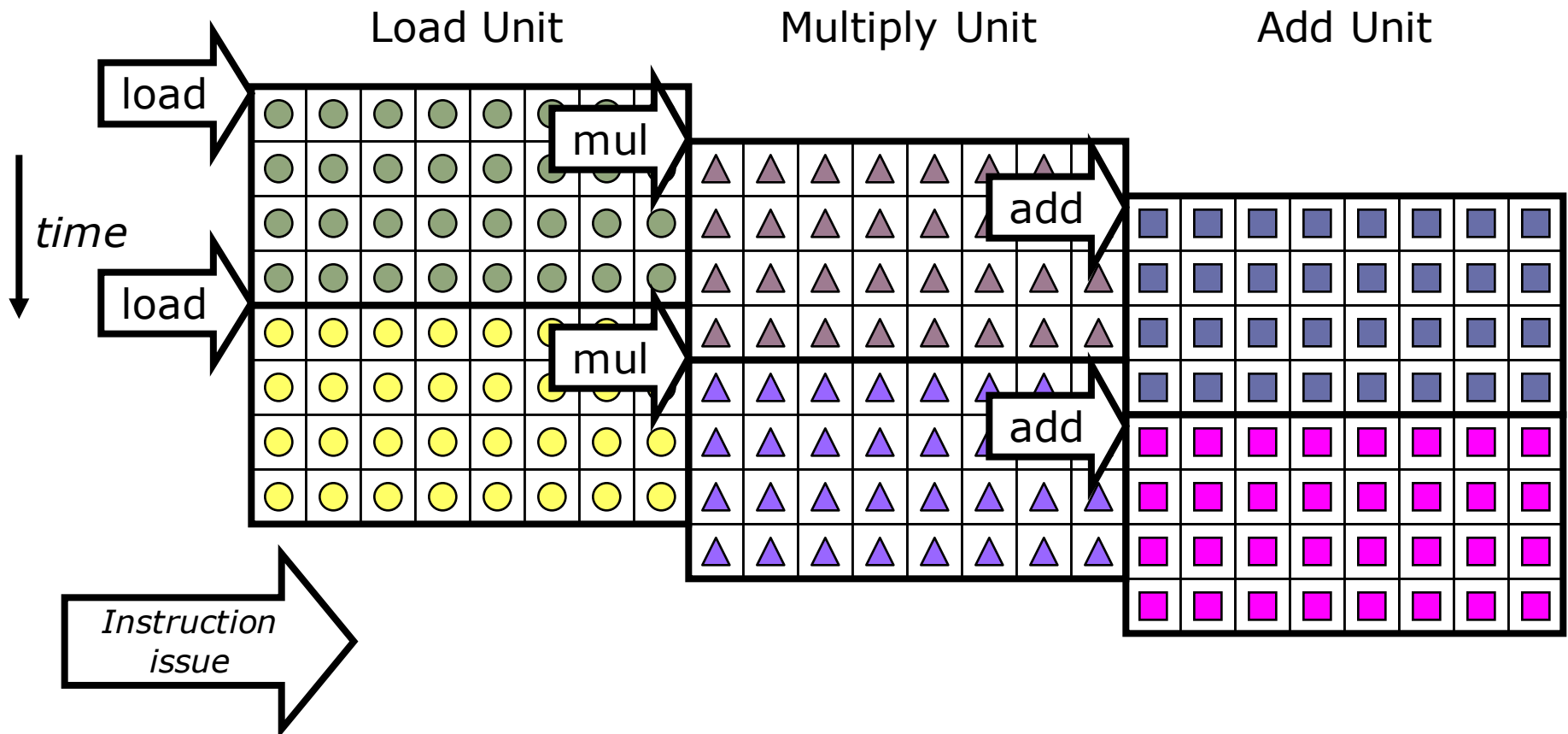
Vector Unit Structure



Vector Instruction Parallelism

Can overlap execution of multiple vector instructions

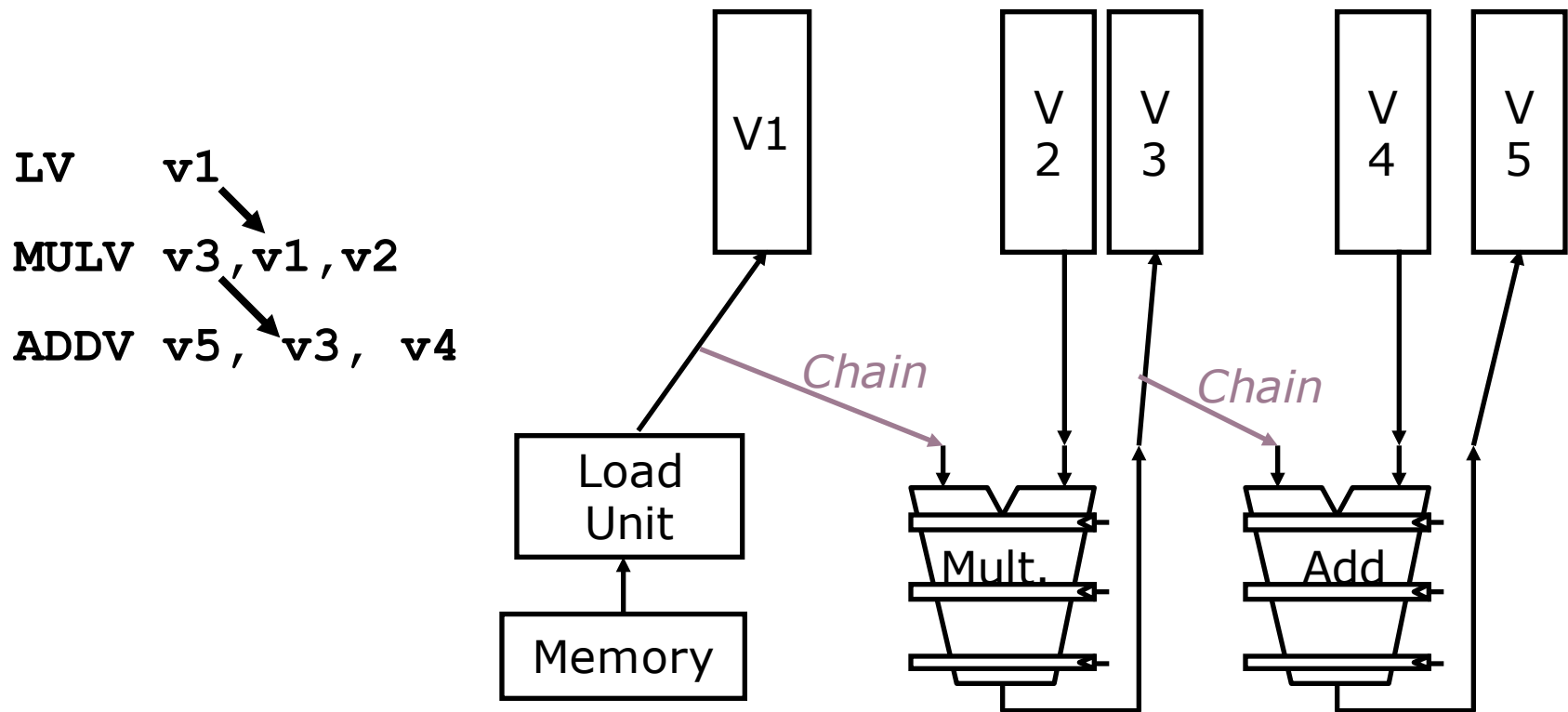
- example machine has 32 elements per vector register and 8 lanes



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

Vector Chaining

Problem: Long latency for RAW register dependencies



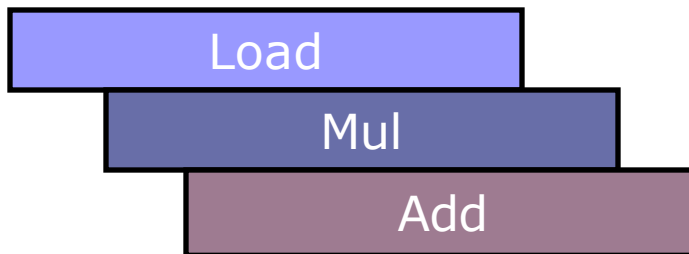
- Vector version of register bypassing
 - introduced with Cray-1

Vector Chaining Advantage

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



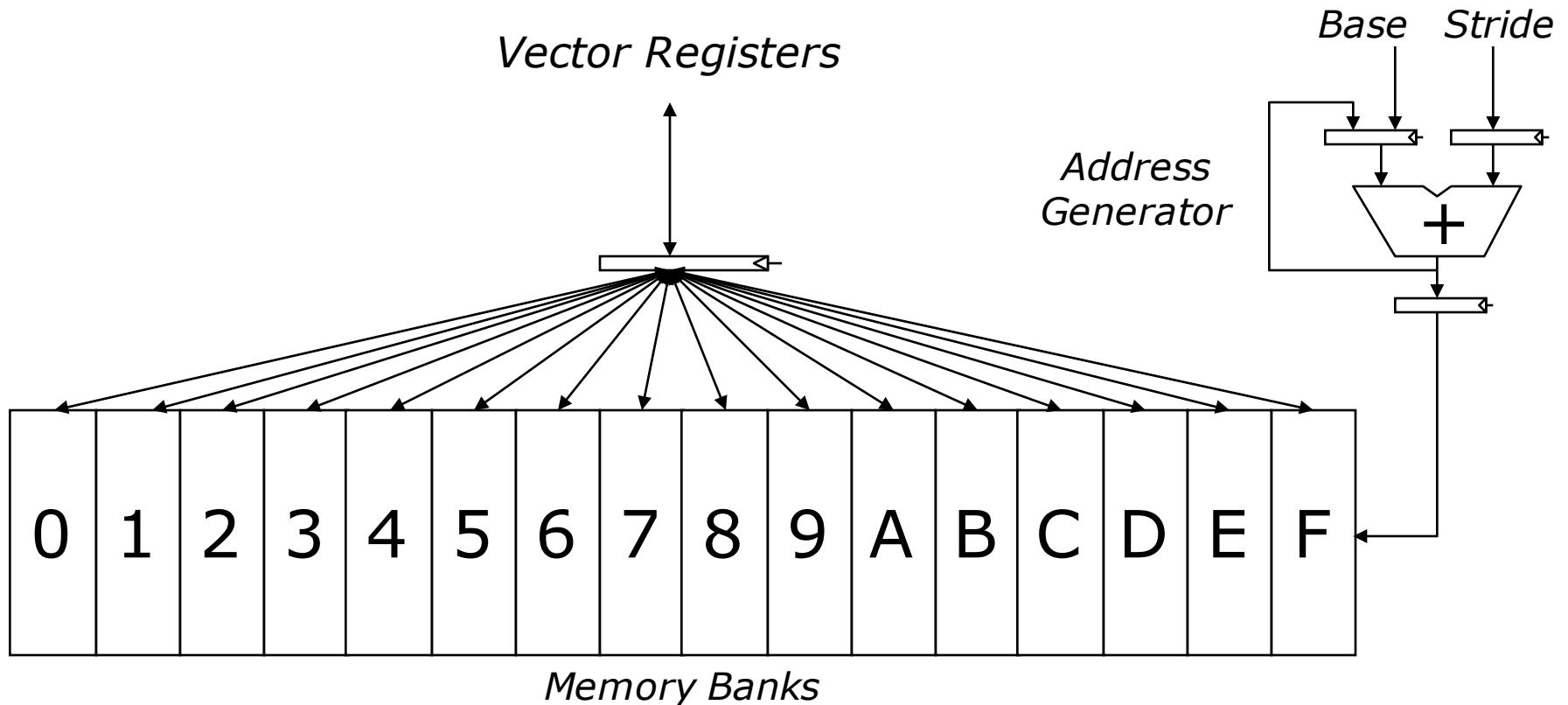
- With chaining, can start dependent instruction as soon as first result appears



Vector Memory System

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

- *Bank busy time*: Cycles between accesses to same bank
- Allows 16 parallel accesses (if data in different banks)

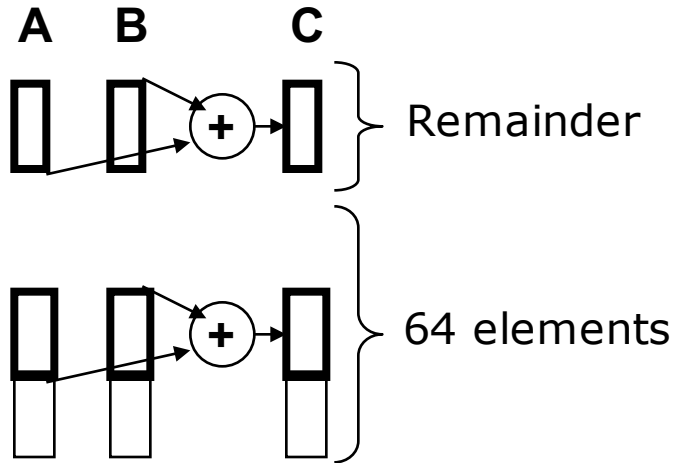


Vector Stripmining

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 R1, N, 63      # N mod 64
MTC1 VLR, R1       # Do remainder
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?
```

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 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 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]

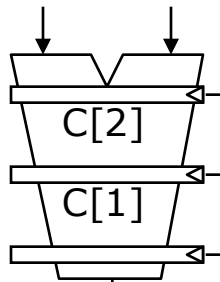
M[4]=1 A[4] B[4]

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

M[2]=0

M[1]=1

M[0]=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

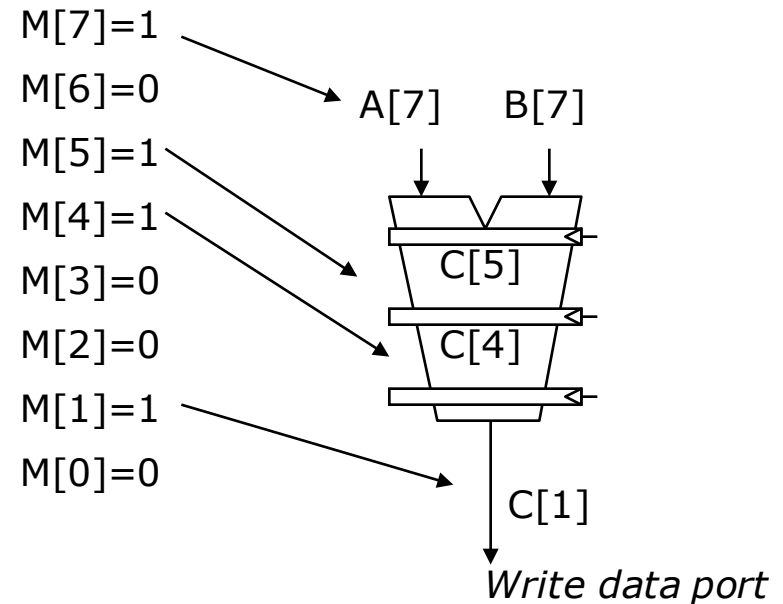
A[7] B[7]

C[5]

C[4]

C[1]

Write data port



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

Scatter example:

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

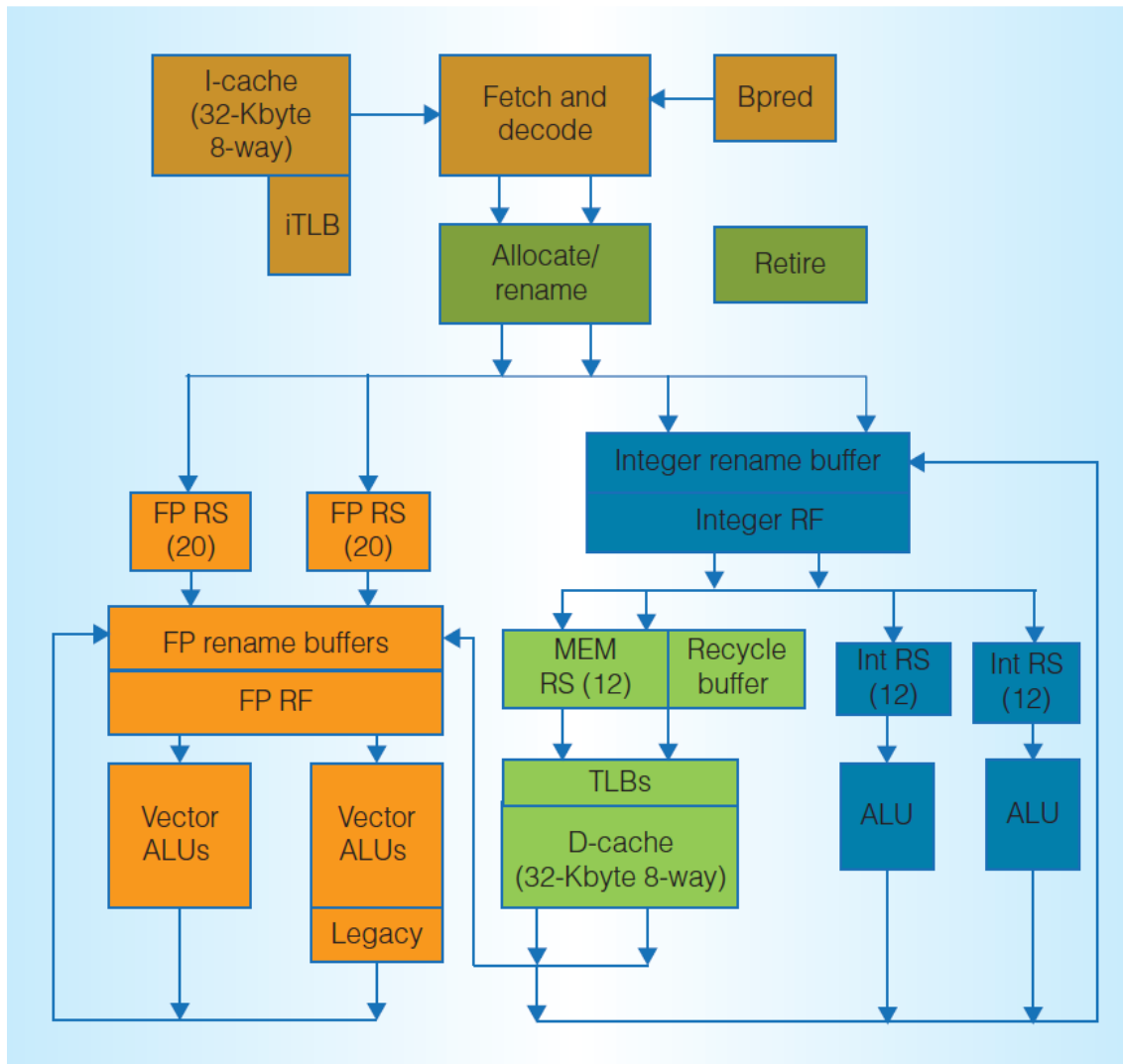
Is following a correct translation?

```
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
```

Multimedia Extensions

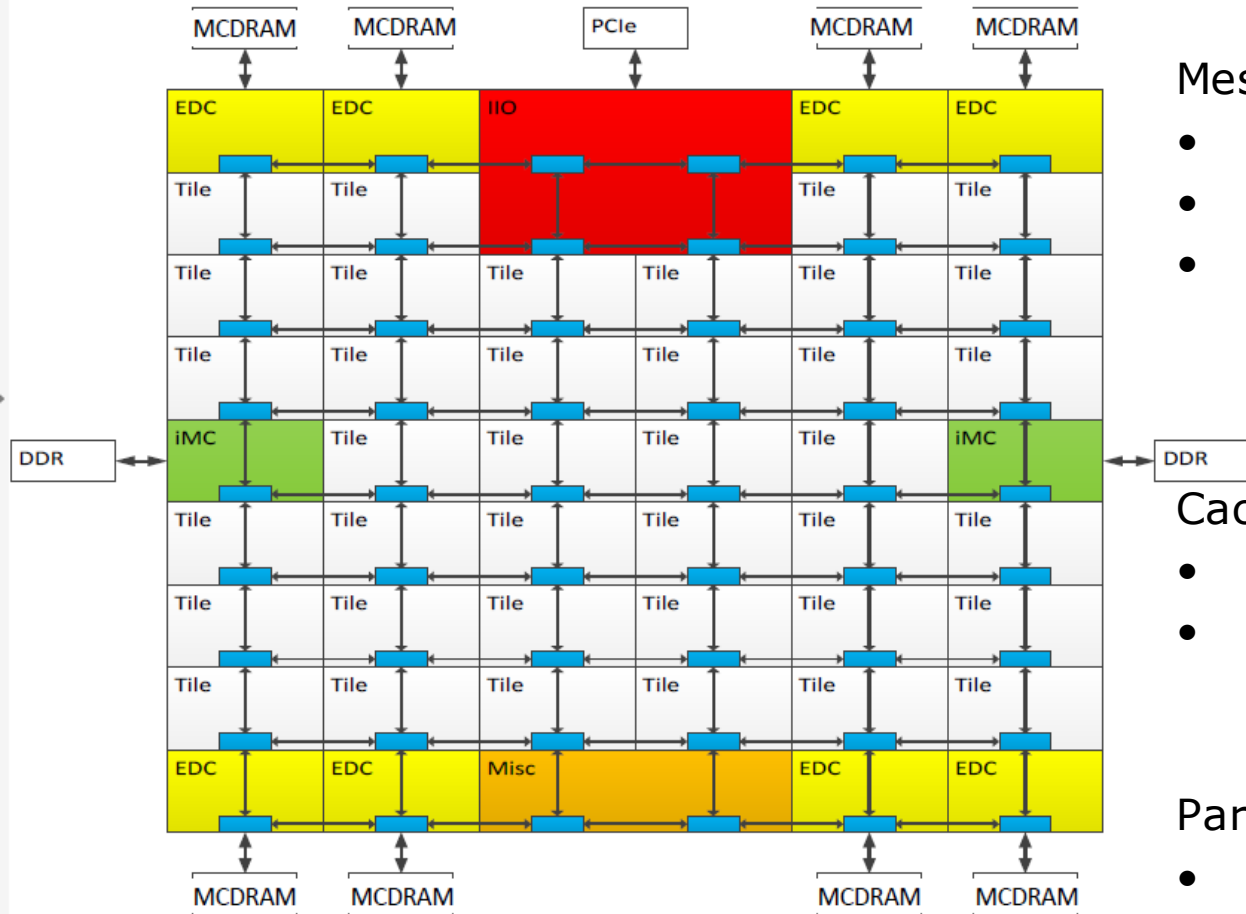
- Short vectors added to existing general-purpose ISAs
- Initially, 64-bit registers split into 2x32b or 4x16b or 8x8b
- Limited instruction set:
 - No vector length control
 - No vector masks
 - No strided load/store or scatter/gather
 - Loads must be aligned to 64-bit boundary
- Limited vector register 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
 - e.g. x86: MMX → SSE (128 bits) → AVX (256 bits) → AVX-512 (512 bits/masks)

Knight's Landing (KNL) CPU



- 2-wide decode/retire
- 6-wide execute
- 72-entry ROB
- 64B cache ports
- 2 load/1 store ports
- Fast unaligned access
- Fast scatter/gather
- OoO int/fp RS
- In-order mem RS
- 4 thread SMT
- Many shared resources
 - ROB, rename, RS...
 - Caches, TLB
- Several thread choosers

Knight's Landing (KNL) Mesh



Source: IEEE Micro, 2016

Mesh of Rings

- Rows/columns (half) ring
- YX routing
- Messages arbitration on:
 - Injection
 - Turns

Cache Coherent Interconnect

- MESIF protocol
- Distributed directory
 - to filter snoops

Partitioning modes

- All-to-all
- Quadrant
- Sub-NUMA

Next: GPUs
Thank you !