Vector Computers

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

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

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 ≡ Vector Machine

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 →

loop:

Int1	Int 2	M1	M2	FP+	FPx
		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			

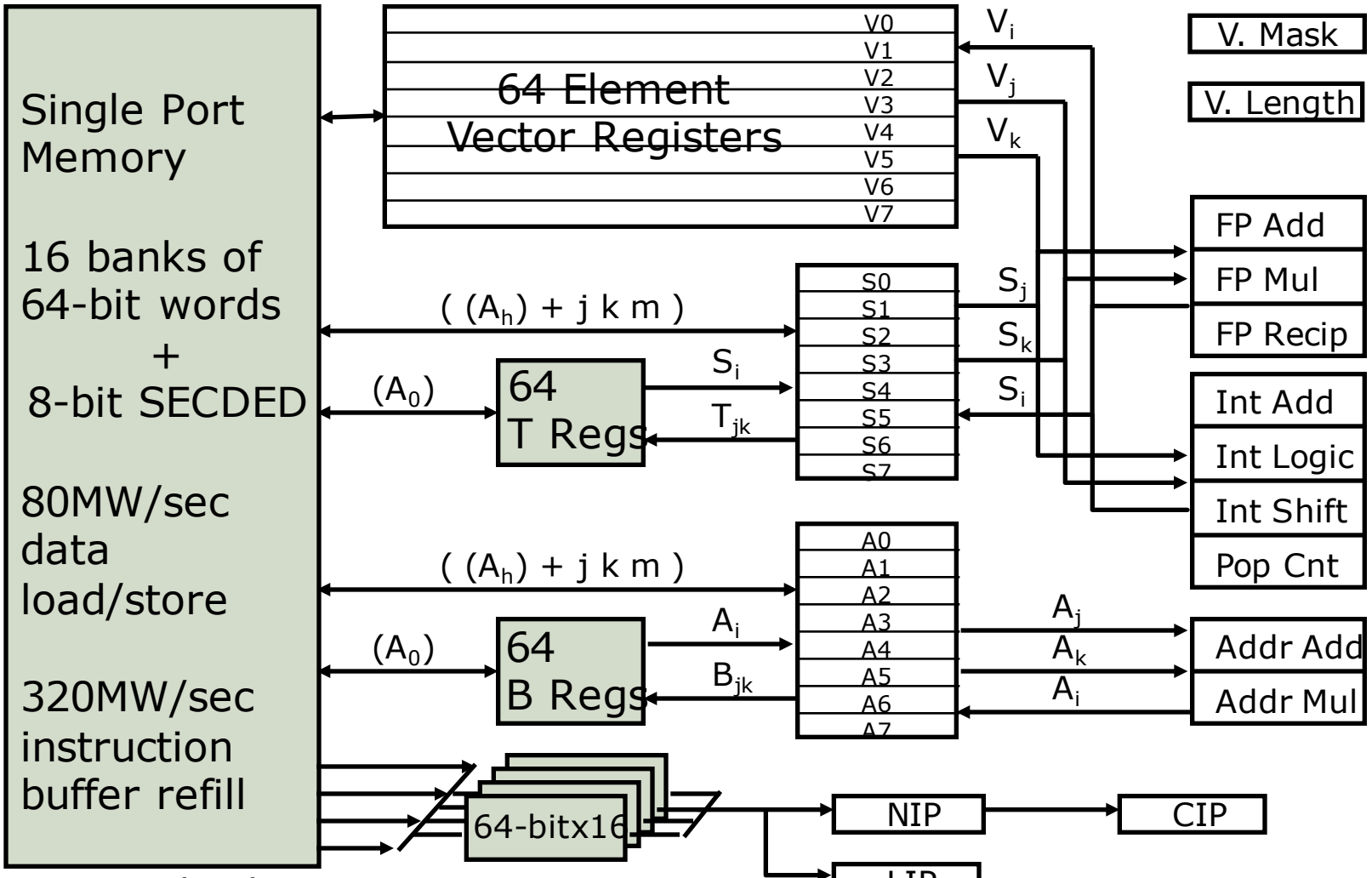
Vector Supercomputers

Epitomized by Cray-1, 1976:

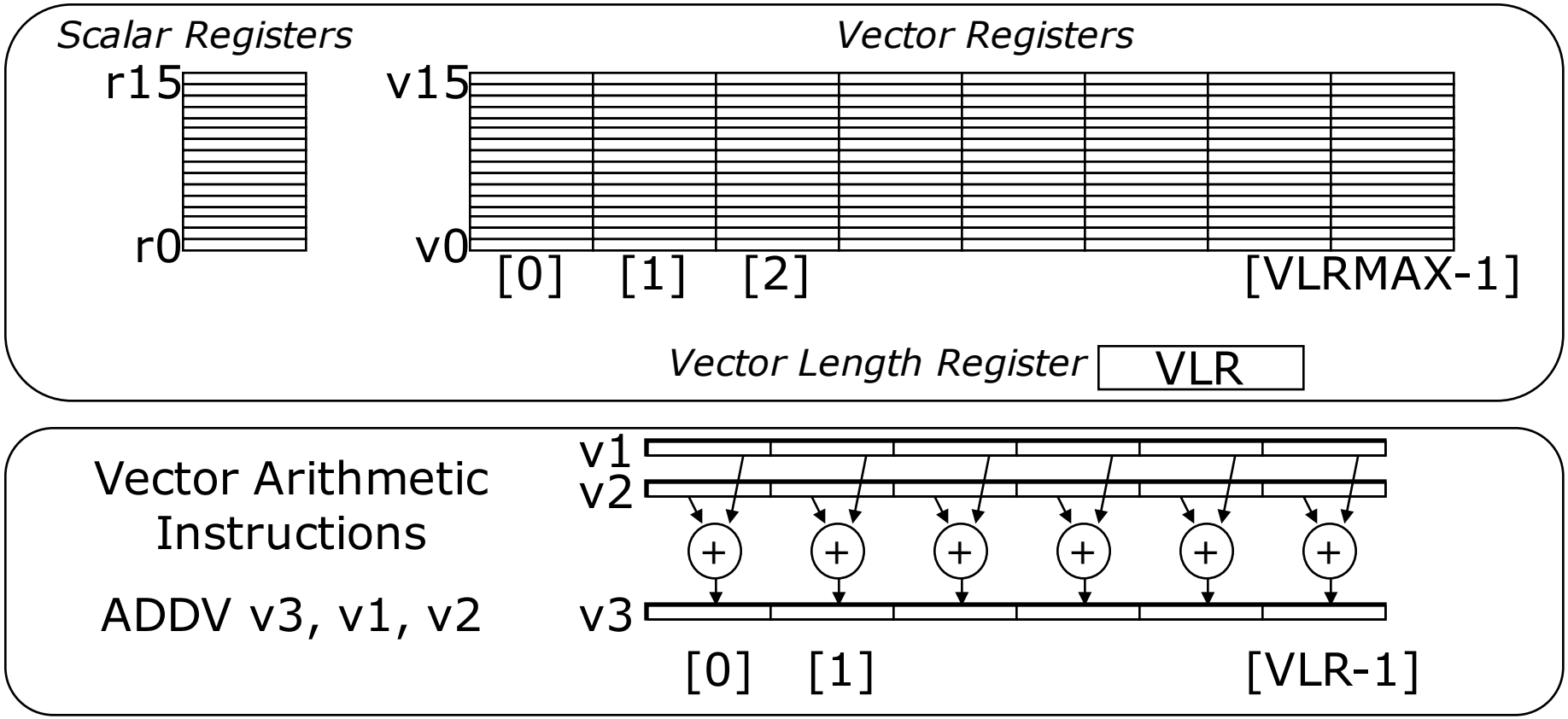
- Scalar Unit
  - Load/Store Architecture
- Vector Extension
  - Vector Registers
  - Vector Instructions
- Implementation
  - Hardwired Control
  - Highly Pipelined Functional Units
  - No Data Caches
  - Interleaved Memory System
  - No Virtual Memory

Cray-1 (1976)

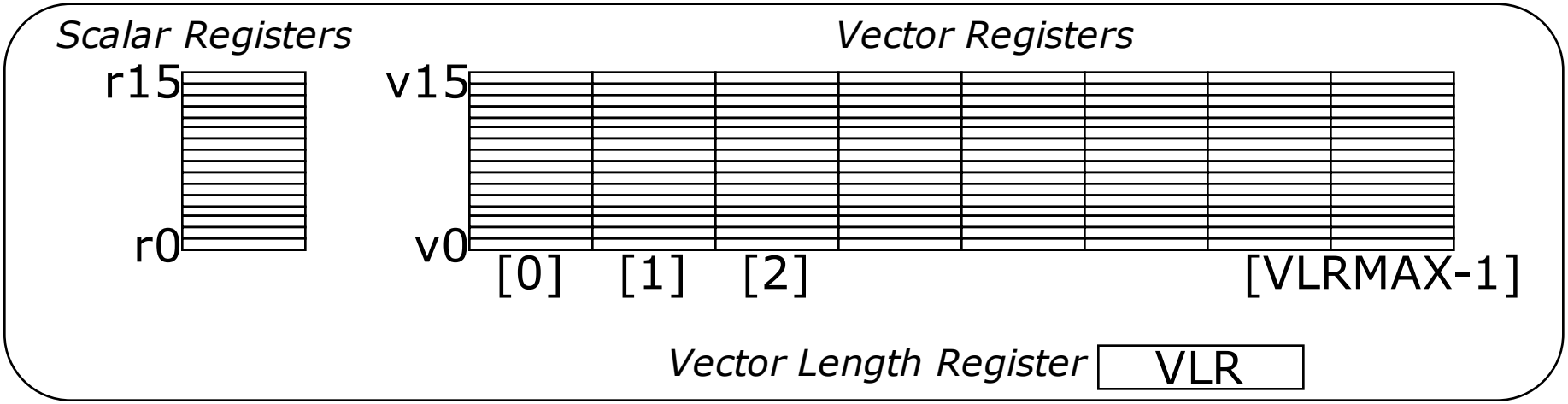
Cray-1 (1976)

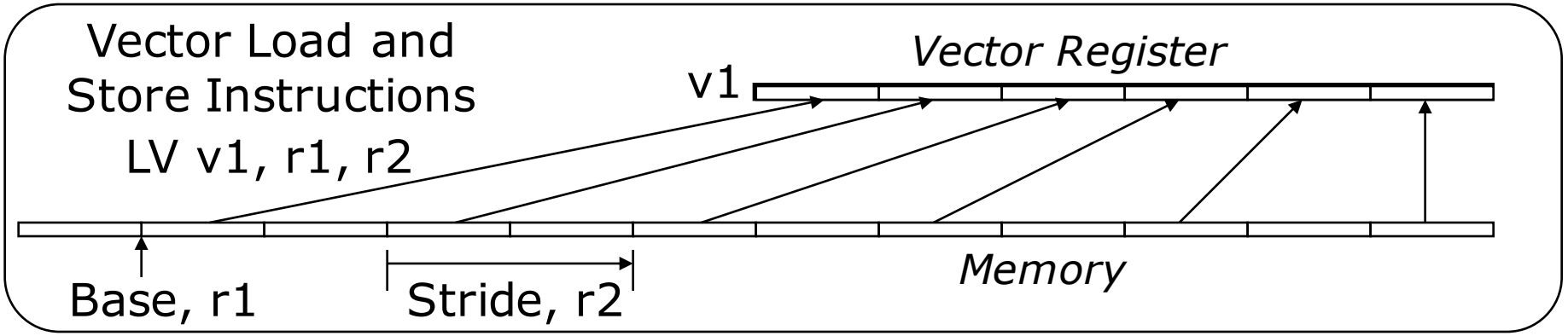

Single Port Memory
16 banks of 64-bit words + 8-bit SECDED
80MW/sec data load/store
320MW/sec instruction buffer refill
64 Element Vector Registers
V0
V1
V2
V3
V4
V5
V6
V7
V. Mask
V. Length
FP Add
FP Mul
FP Recip
Int Add
Int Logic
Int Shift
Pop Cnt
Addr Add
Addr Mul
((Ah) + j k m)
(A0)
64 T Regs
64 B Regs
S0
S1
S2
S3
S4
S5
S6
S7
A0
A1
A2
A3
A4
A5
A6
A7
64-bitx16
NIP
CIP
LIP
memory bank cycle 50 ns
4 Instruction Buffers
processor cycle 12.5 ns (80MHz)

Vector Programming Model

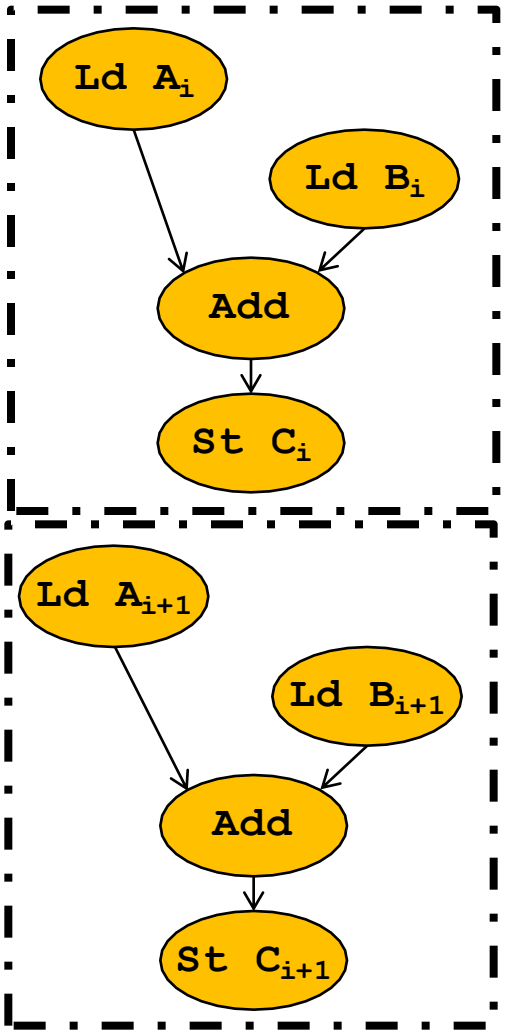


Vector Programming Model





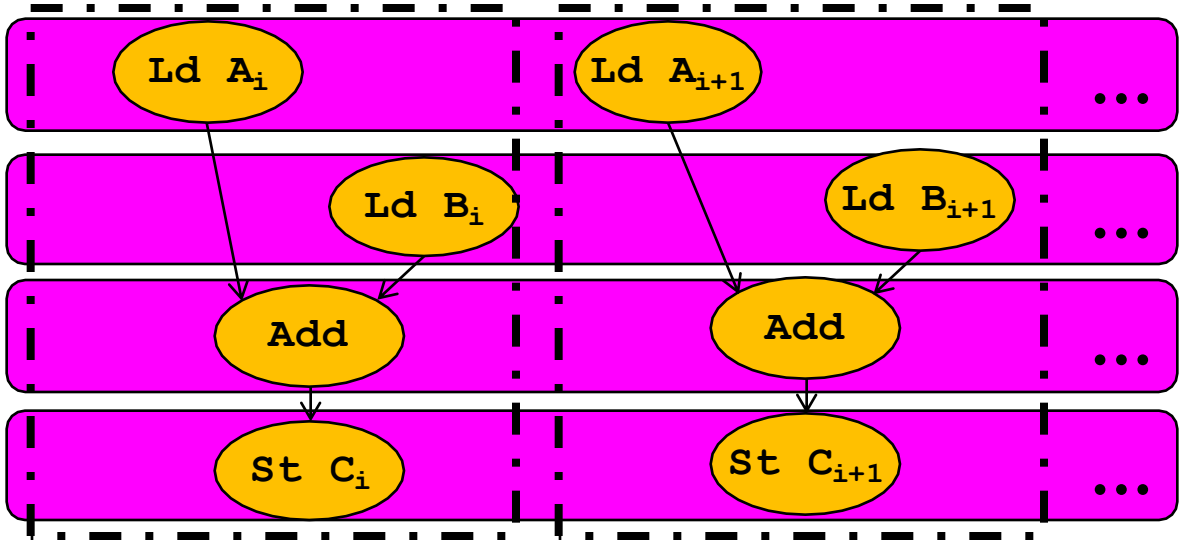
Compiler-based Vectorization



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
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
```

Vector ISA Attributes

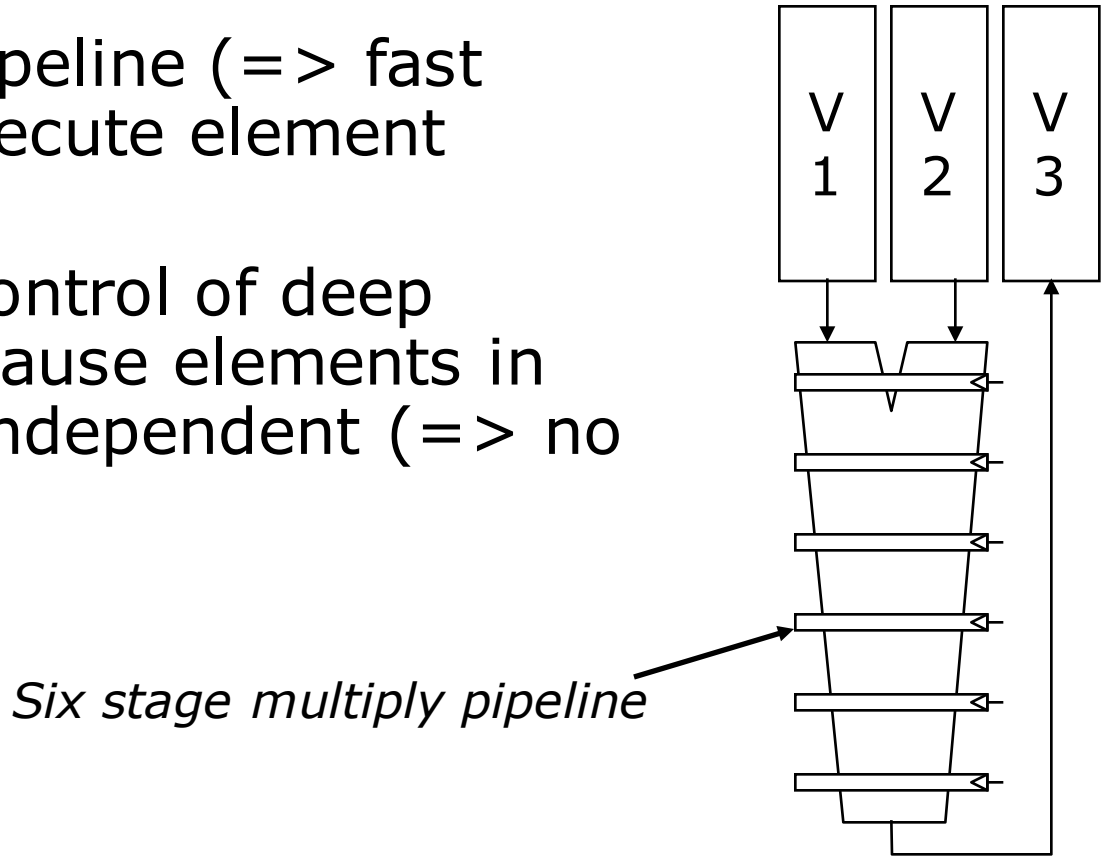
- Compact
  - one short instruction encodes N operations
  - many implicit bookkeeping/control operations
- Expressive, tells hardware that these N operations:
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in same pattern as previous instructions
  - 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 (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)



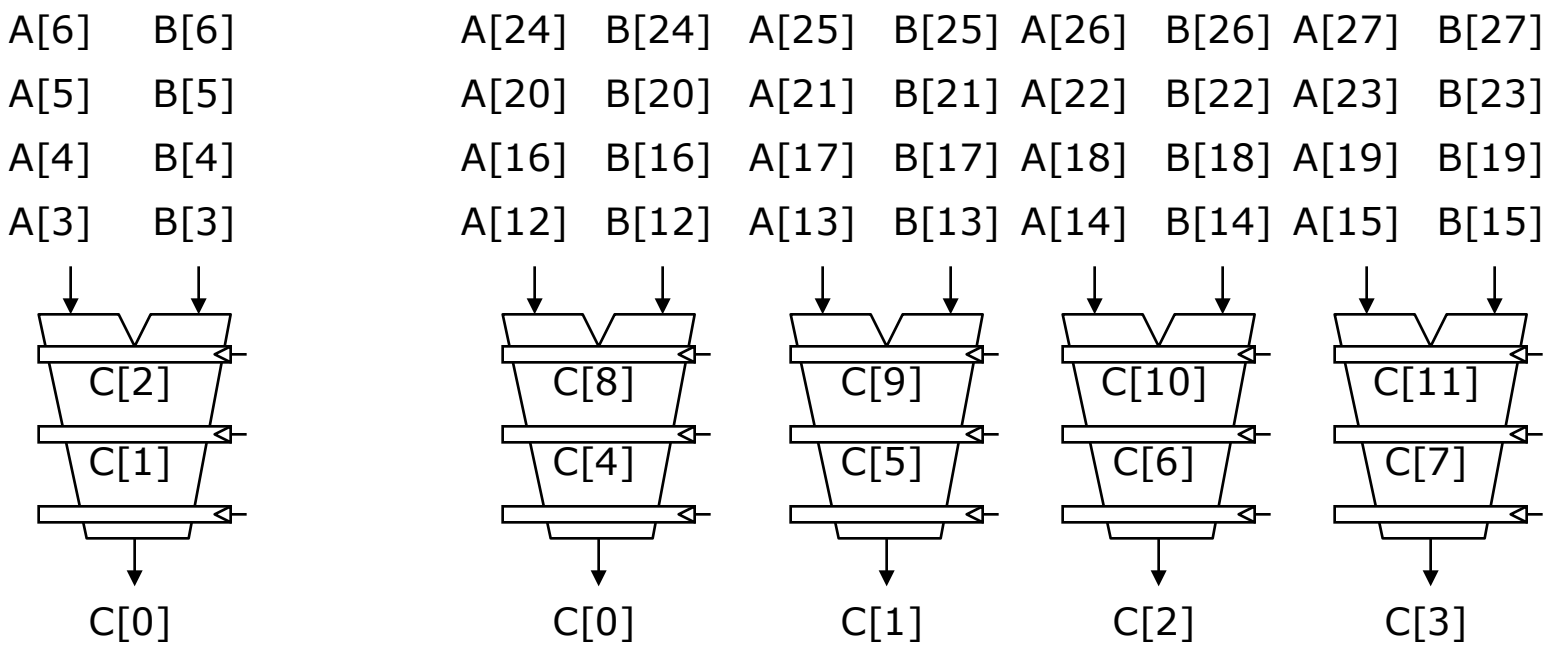
V3 <- v1 * v2

Vector Instruction Execution

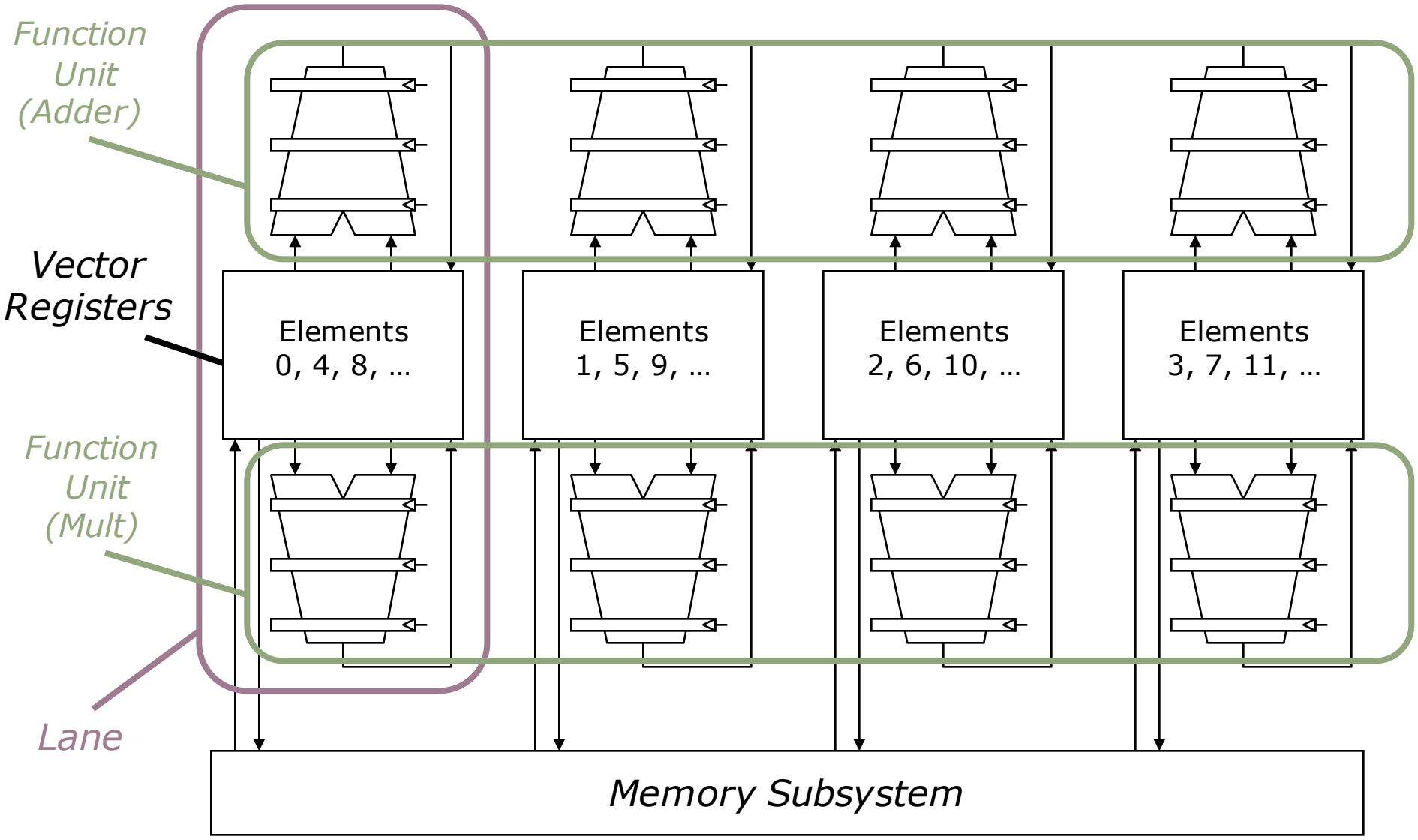
ADDV C,A,B

Execution using one pipelined functional unit

Execution using four pipelined functional units



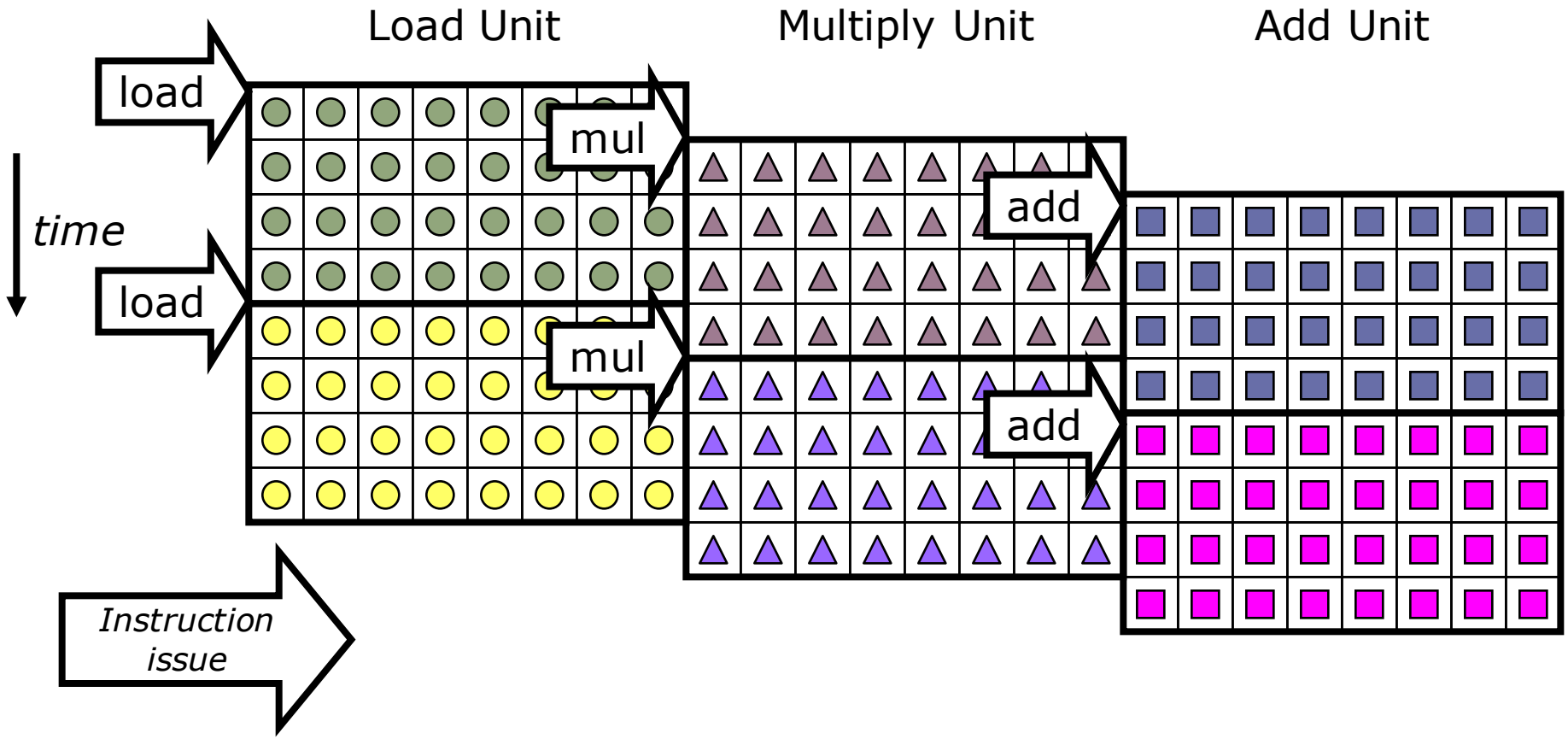
Vector Unit Structure



Vector Instruction Parallelism

Can overlap execution of multiple vector instructions

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

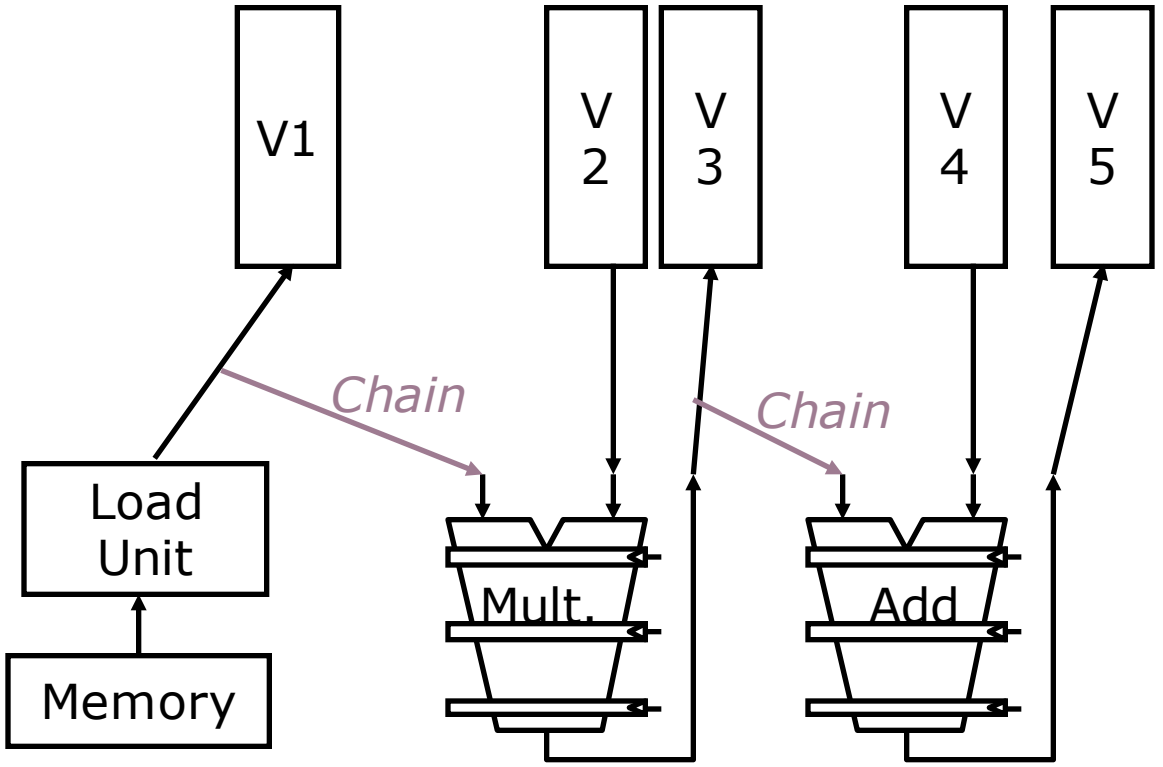


Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Chaining

Problem: Long latency for RAW register dependencies

```
LV   v1
MULV v3,v1,v2
ADDV v5, v3, v4
```

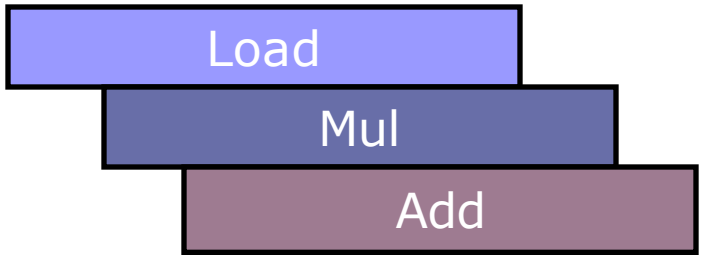


- Vector version of register bypassing
  - introduced with Cray-1

Vector Chaining Advantage

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

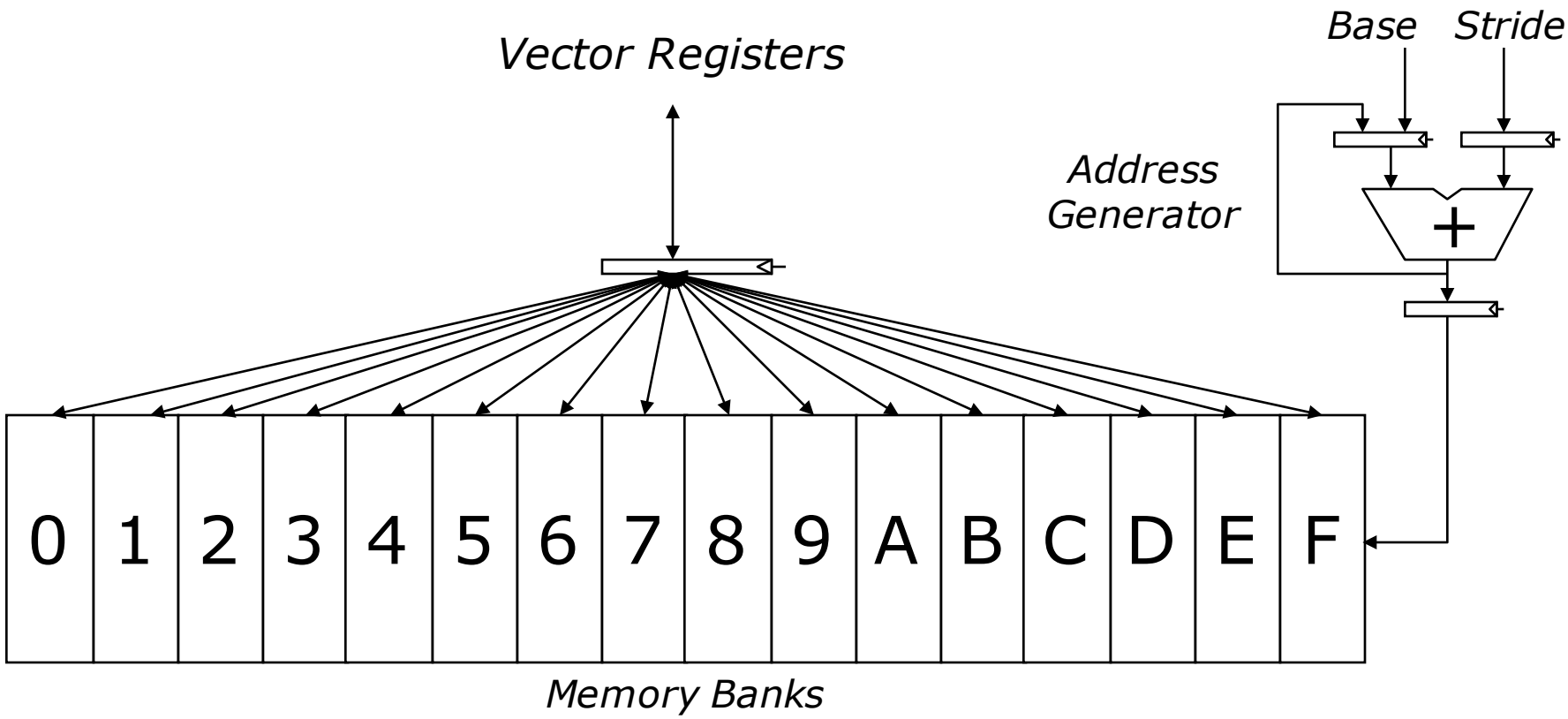
- With chaining, can start dependent instruction as soon as first result appears



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)

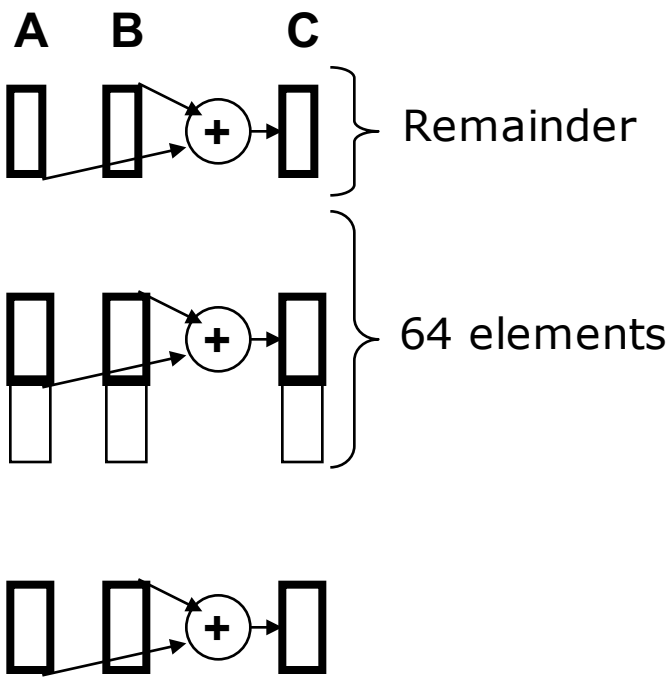


Vector Stripmining

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
 DSLL 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

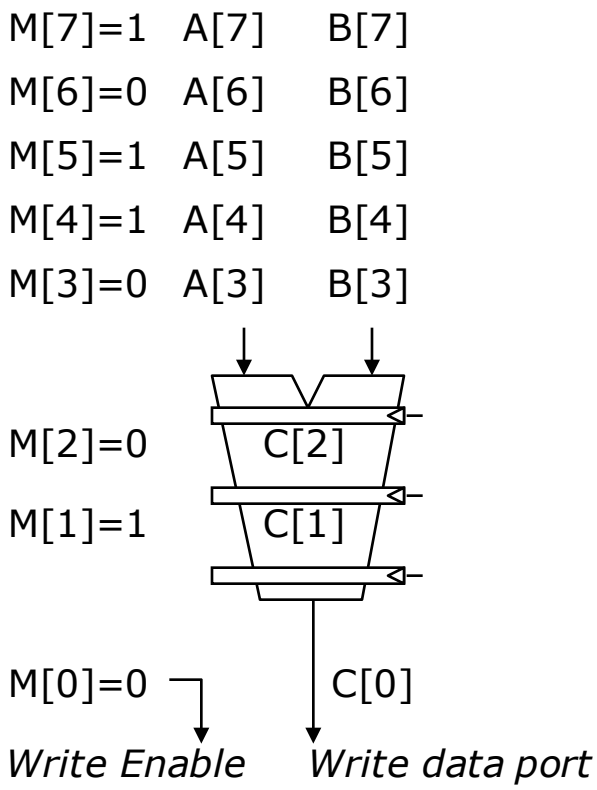
Code example:

```
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

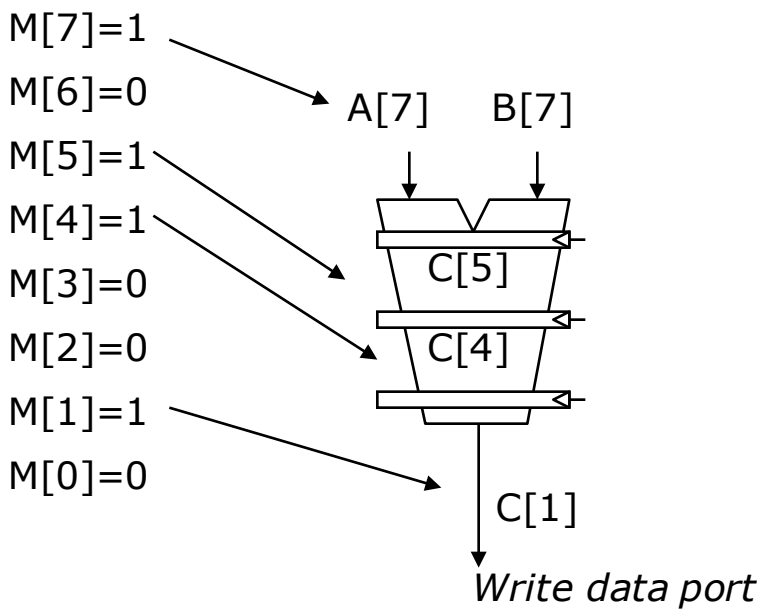
Simple Implementation

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



Density-Time Implementation

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



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
```

A Later Generation Vector Super:
NEC SX-6 (2003)

- CMOS Technology
  - 500 MHz CPU, fits on single chip
  - SDRAM main memory (up to 64GB)
- Scalar unit
  - 4-way superscalar
  - with out-of-order and speculative execution
  - 64KB I-cache and 64KB data cache
- Vector unit
  - 8 foreground VRegs + 64 background VRegs (256x64-bit elements/VReg)
  - 1 multiply unit, 1 divide unit, 1 add/shift unit, 1 logical unit, 1 mask unit
  - 8 lanes (8 GFLOPS peak, 16 FLOPS/cycle)
  - 1 load & store unit (32x8 byte accesses/cycle)
  - 32 GB/s memory bandwidth per processor
- SMP structure
  - 8 CPUs connected to memory through crossbar
  - 256 GB/s shared memory bandwidth (4096 interleaved banks)

Multimedia Extensions

- Short vectors added to existing general-purpose ISAs
- Idea first used on Lincoln Labs TX-2 computer in 1957, with 36b datapath split into 2x18b or 4x9b.
- Recent incarnation initially, 64-bit registers split into:
  - 2x32bits or 4x16bits or 8x8bits
- Trend towards larger vector support in microprocessors
  - e.g. x86: MMX → SSEx (128 bits) → AVX (256 bits) → AVX-512 (512 bits/masks)

Intel SIMD Evolution

Implementations

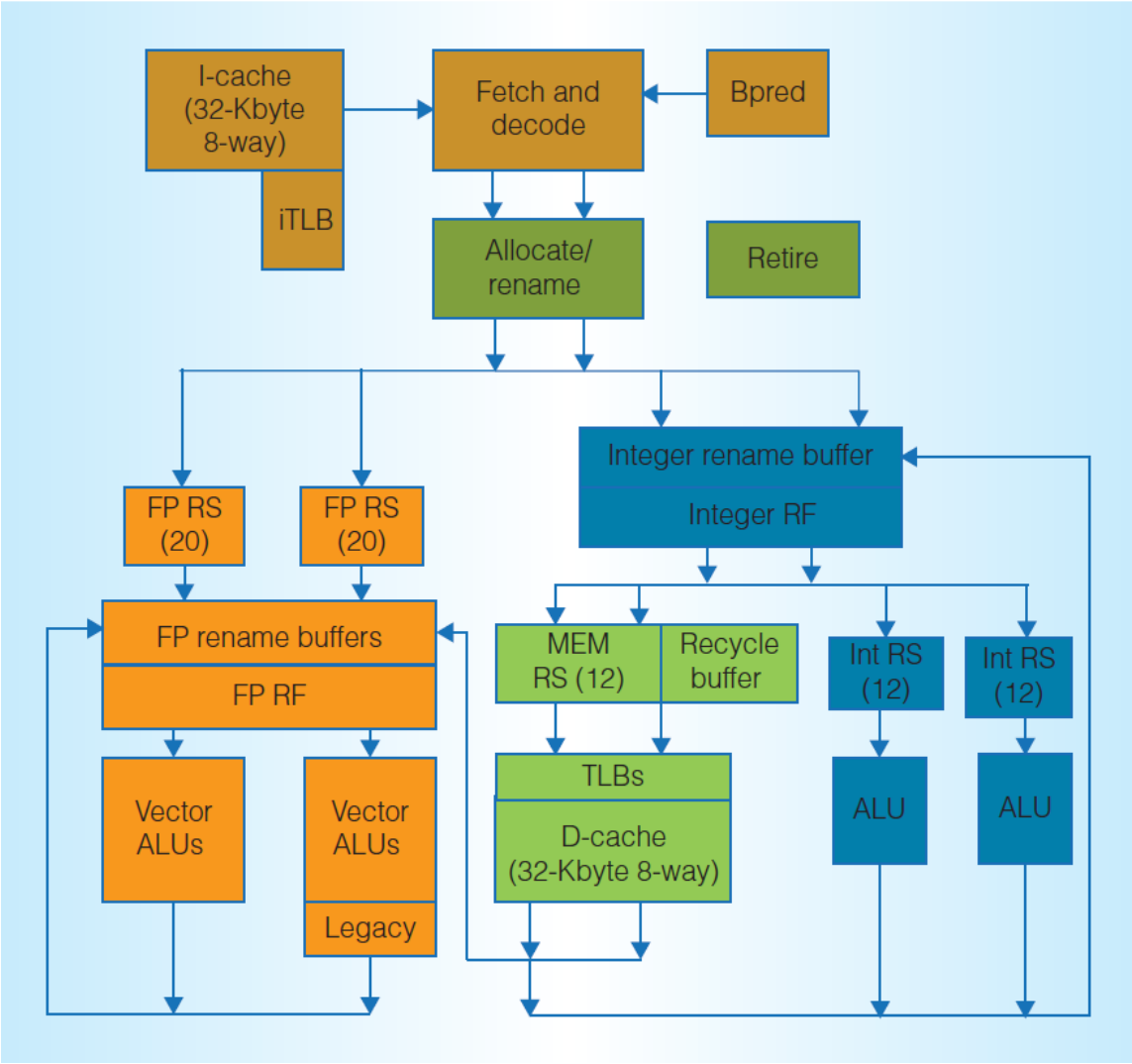
- Intel MMX (1996)
  - Eight 8-bit integer ops, or
  - Four 16-bit integer ops
  - Two 32-bit integer ops
- Streaming SIMD Extensions (SSE) (1999)
  - Four 32-bit integer ops (and smaller integer types)
  - Four 32-bit integer/fp ops, or
  - Two 64-bit integer/fp ops
- Advanced Vector Extensions (2010)
  - Four 64-bit integer/fp ops (and smaller fp types)

Operands must be consecutive and aligned memory locations

Multimedia Extensions vs Vectors

- Limited instruction set:
  - No vector length control
  - No masks
  - No strided load/store or scatter/gather
  - Unit-stride loads must be aligned to 64/128-bits
- Limited vector register length:
  - requires superscalar dispatch to keep units busy
  - loop unrolling to hide latencies increases register pressure
- Trend towards fuller vector support
  - Better support for misaligned memory accesses
  - Support of double-precision (64-bit floating-point)
  - Support for masked operations

Knight's Landing (KNL) CPU



- 2-wide decode/retire
- 6-wide execute

- 72-entry ROB

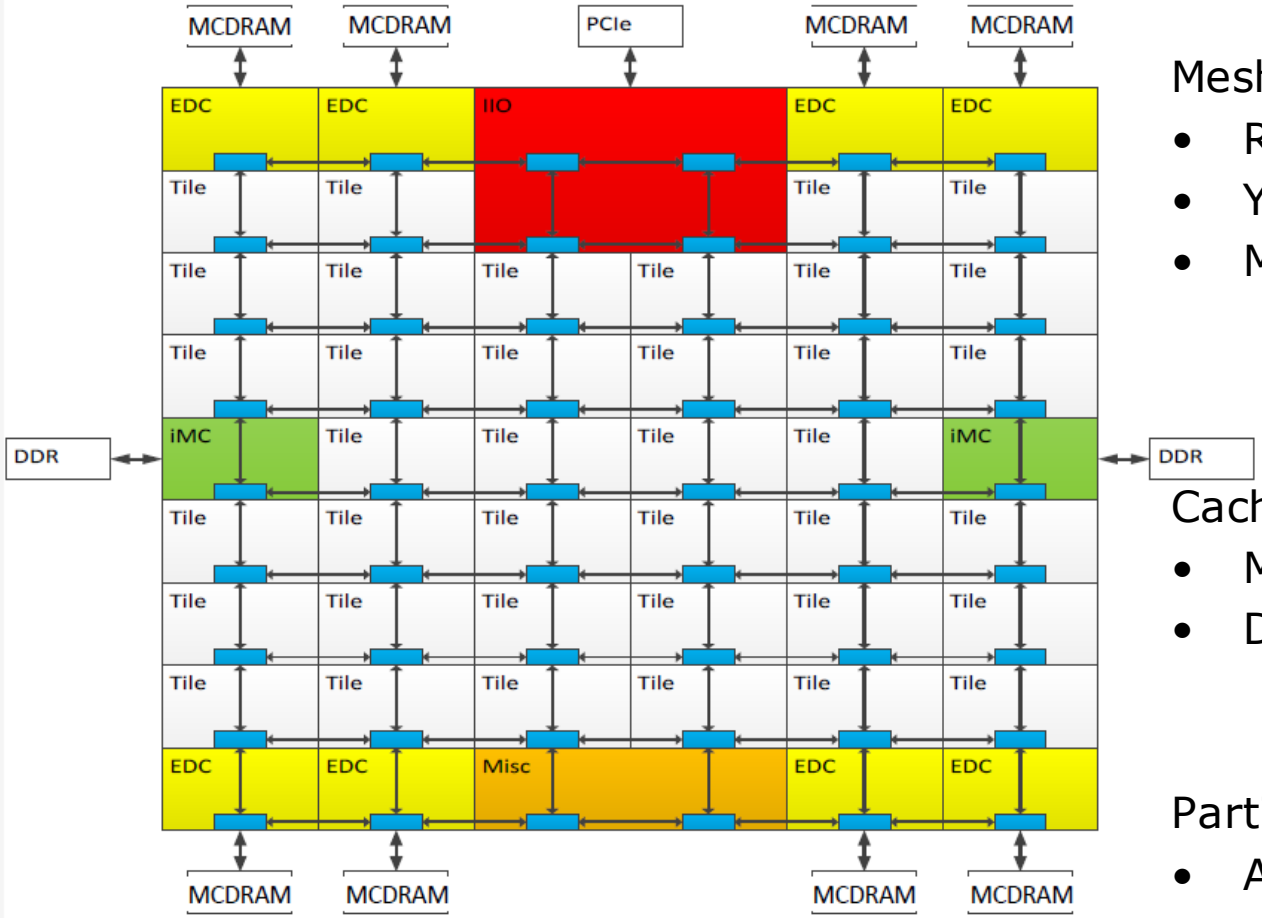
- 64B cache ports
- 2 load/1 store ports
- Fast unaligned access
- Fast scatter/gather

- OoO int/fp RS
- In-order mem RS

- 4 thread SMT
- Many shared resources
  - ROB, rename, RS...
  - Caches, TLB
- Several thread choosers

Source: IEEE Micro, 2016

Knight's Landing (KNL) Mesh



Source: IEEE Micro, 2016

Mesh of Rings
- Rows/columns (half) ring
- YX routing
- Messages arbitation 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 !