Vector Processors

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

Supercomputers

Definition of a supercomputer:

- Fastest machine in the 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

All involve huge computations on large data sets

In 70s-80s, Supercomputer $\equiv$ *Vector Machine*

Loop Unrolled Code Schedule

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

```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd.d f5, f0, f1
       fadd.d f6, f0, f2
       fadd.d f7, f0, f3
       fadd.d f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       fsd f8, 24(x2)
       add x2, 32
       bne x1, x3, loop
```

Schedule →

	Int1	Int 2	M1	M2	FP+	FPx
loop:			fld f1			
			fld f2			
			fld f3			
	add x1		fld f4		fadd.d f5	
					fadd.d f6	
					fadd.d f7	
					fadd.d f8	
			fsd f5			
			fsd f6			
			fsd f7			
	add x2	bne	fsd f8			

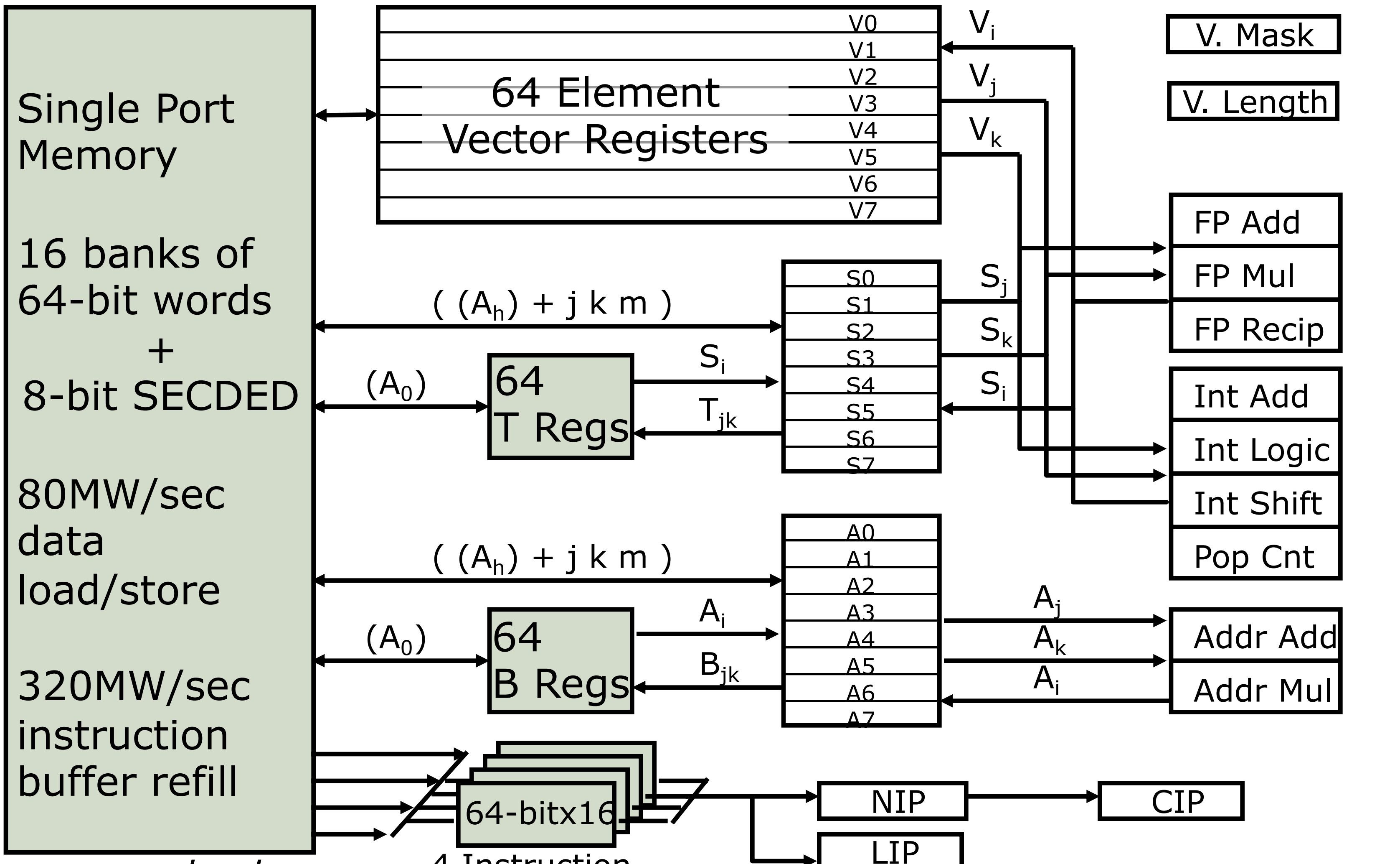
Vector Supercomputers

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)

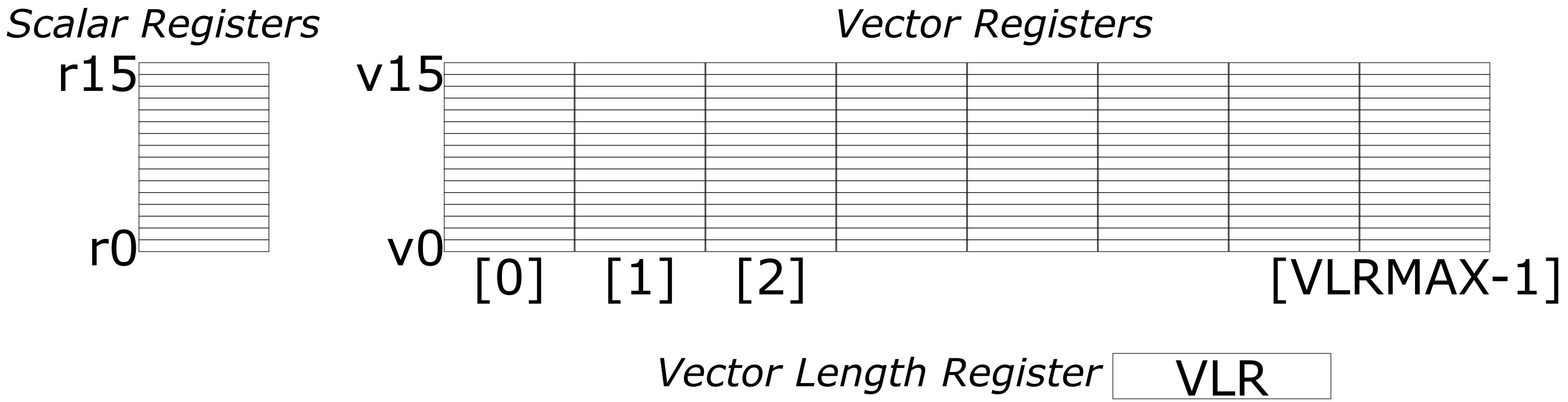


memory bank cycle 50 ns

4 Instruction Buffers

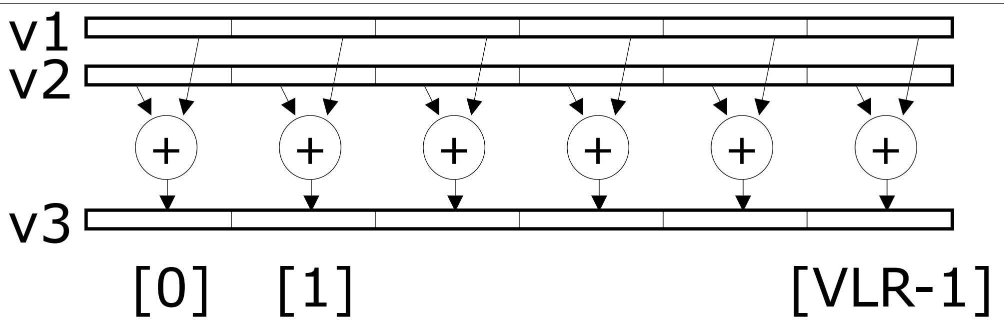
processor cycle 12.5ns (80MHz)

Vector Programming Model

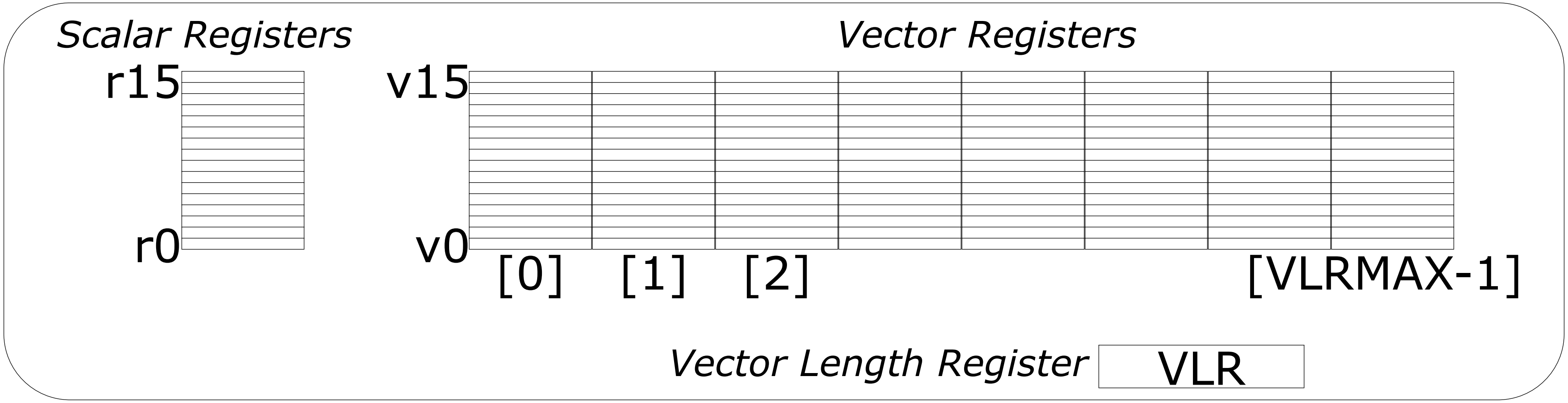


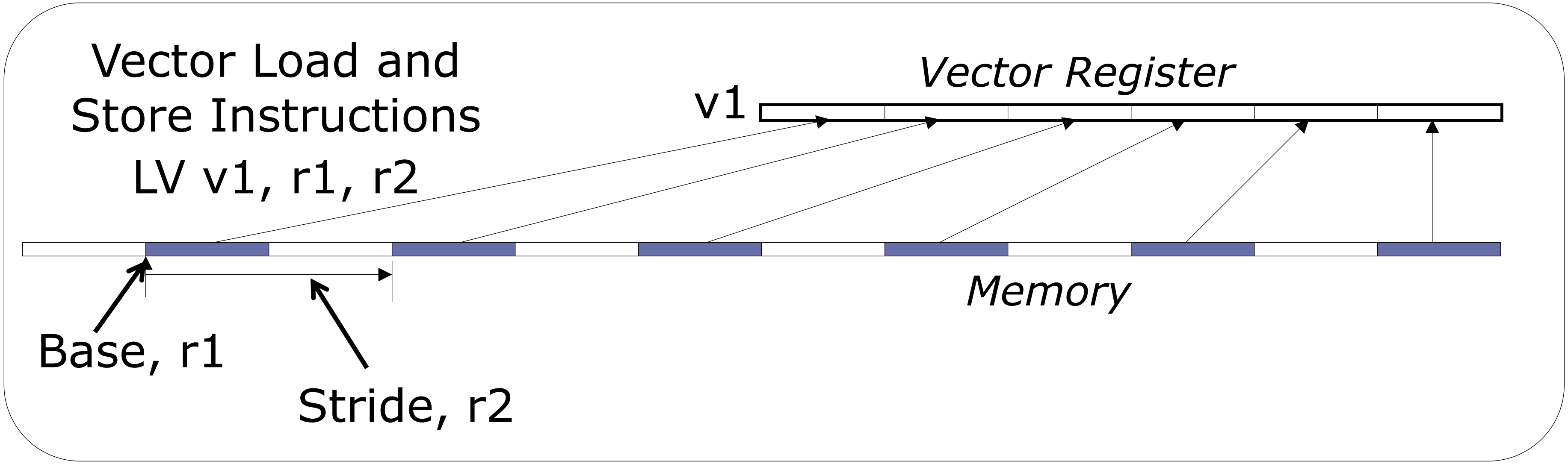
Vector Arithmetic Instructions

ADDV v3, v1, v2

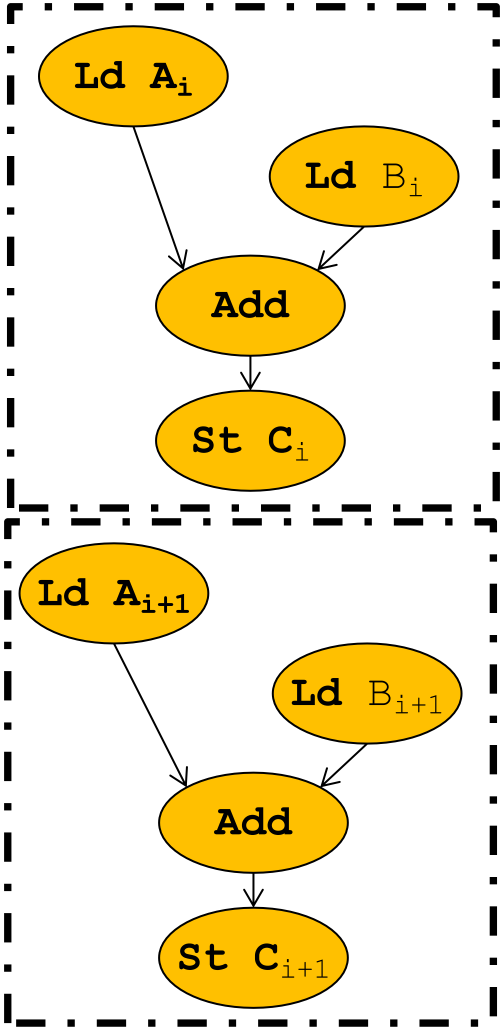


Vector Programming Model



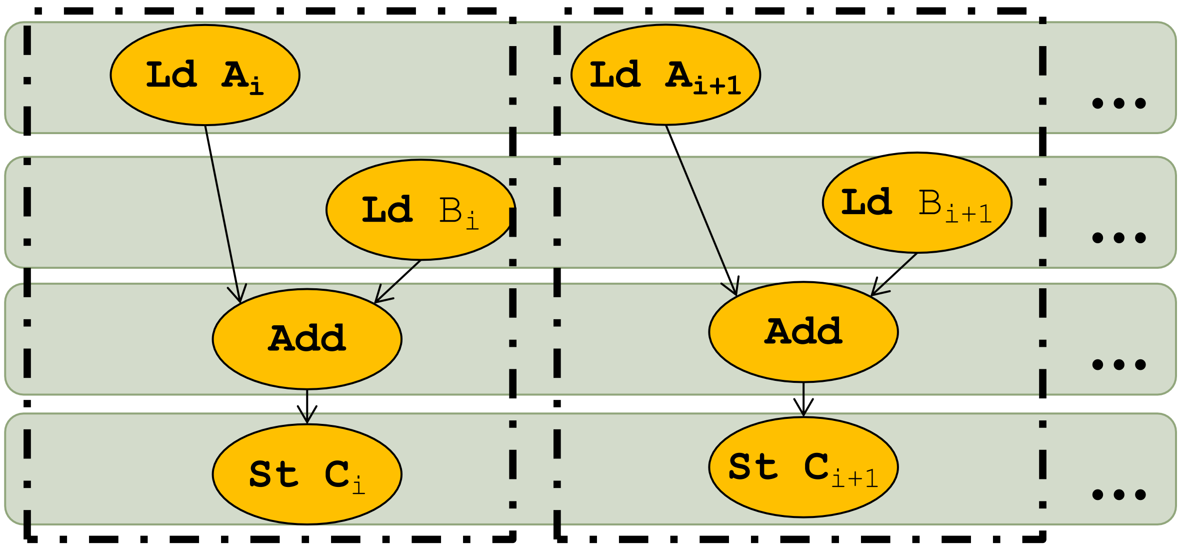


Compiler-based Vectorization




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

Compiler recognizes independent operations with loop dependence analysis



Scalar code

Vector code

Vector Code Example

`# C code`
`for (i=0; i<64; i++)`
`  C[i] = A[i] + B[i];`
	`# Scalar code`
	`  LI x14, 64`
	`loop:`
	`  FLD f0, 0(x11)`
	`  FLD f2, 0(x12)`
	`  FADD.D f4, f2, f0`
	`  FSD f4, 0(x13)`
	`  ADDI x11, 8`
	`  ADDI x12, 8`
	`  ADDI x13, 8`
	`  ADDI x14, -1`
	`  BNEZ x14, loop`
		`# Vector code`
		`LI vlr, 64`
		`LV v1, x11`
		`LV v2, x12`
		`ADDV.D v3, v1, v2`
		`SV v3, x13`

How many instructions?	1+9*64 instrs	5 instrs

Vector ISA Attributes

- Compact
  - One short instruction encodes N operations (plus bookkeeping)

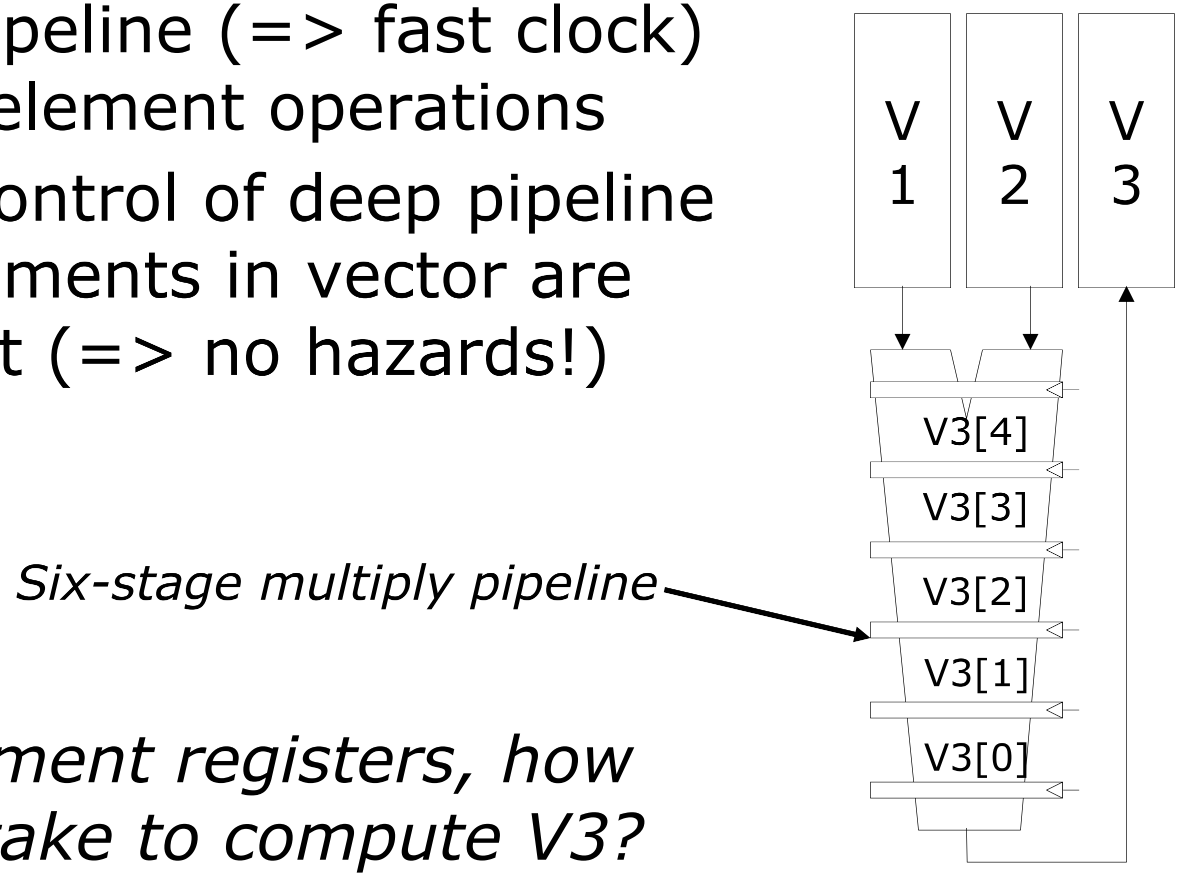
- Expressive, tells hardware that these N operations:
  - Are independent
  - Use the same functional unit
  - Access disjoint elements in vector registers
  - Access registers in same pattern as previous instructions
  - Access a contiguous block of memory (unit-stride load/store)
  - Access memory 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
- No data dependence within a vector
  - → Amenable to deeply pipelined/parallel designs
- Disjoint vector element accesses
  - → Banked rather than multi-ported register files
- 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!)



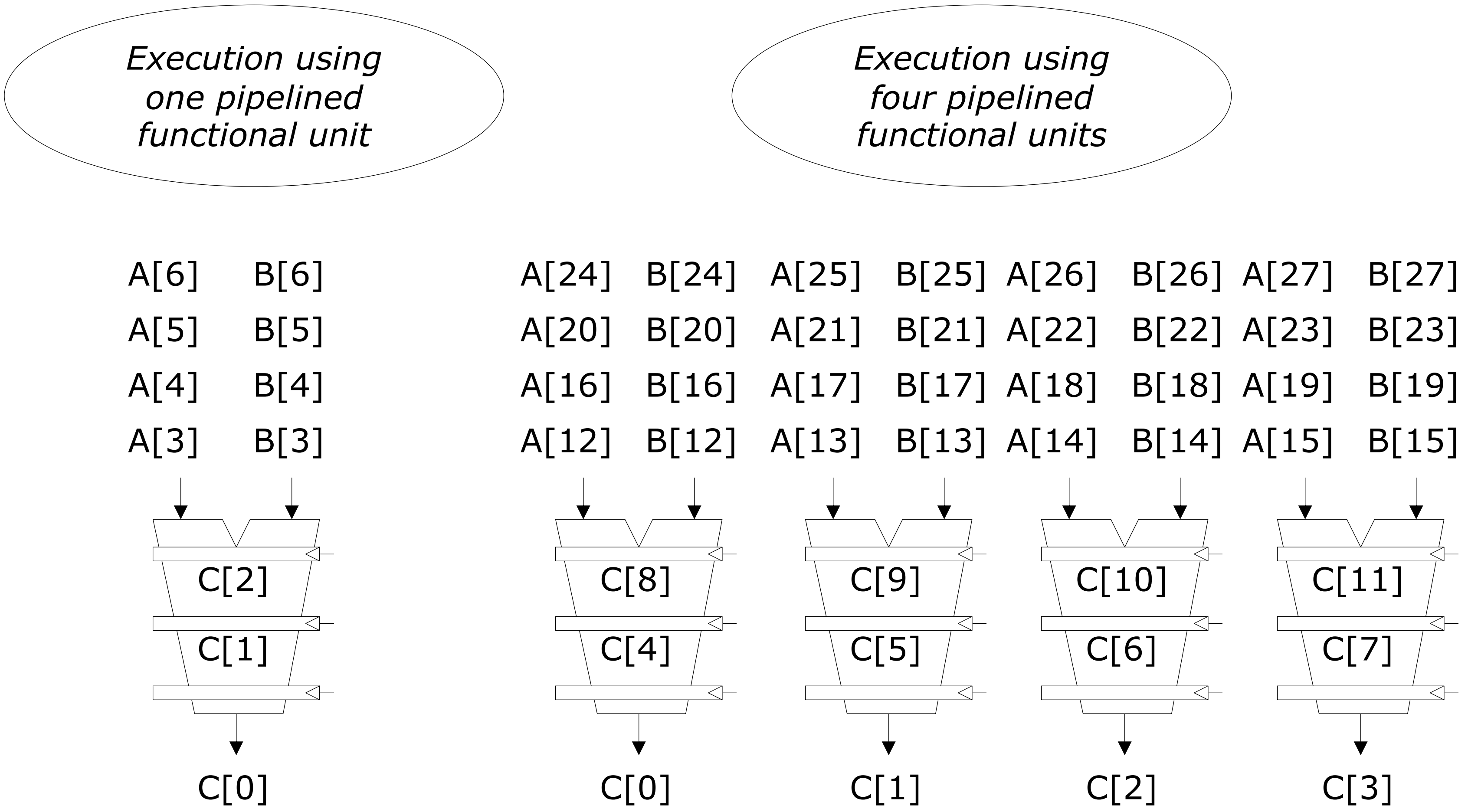
Given 64-element registers, how long does it take to compute V3?

6+63 cycles

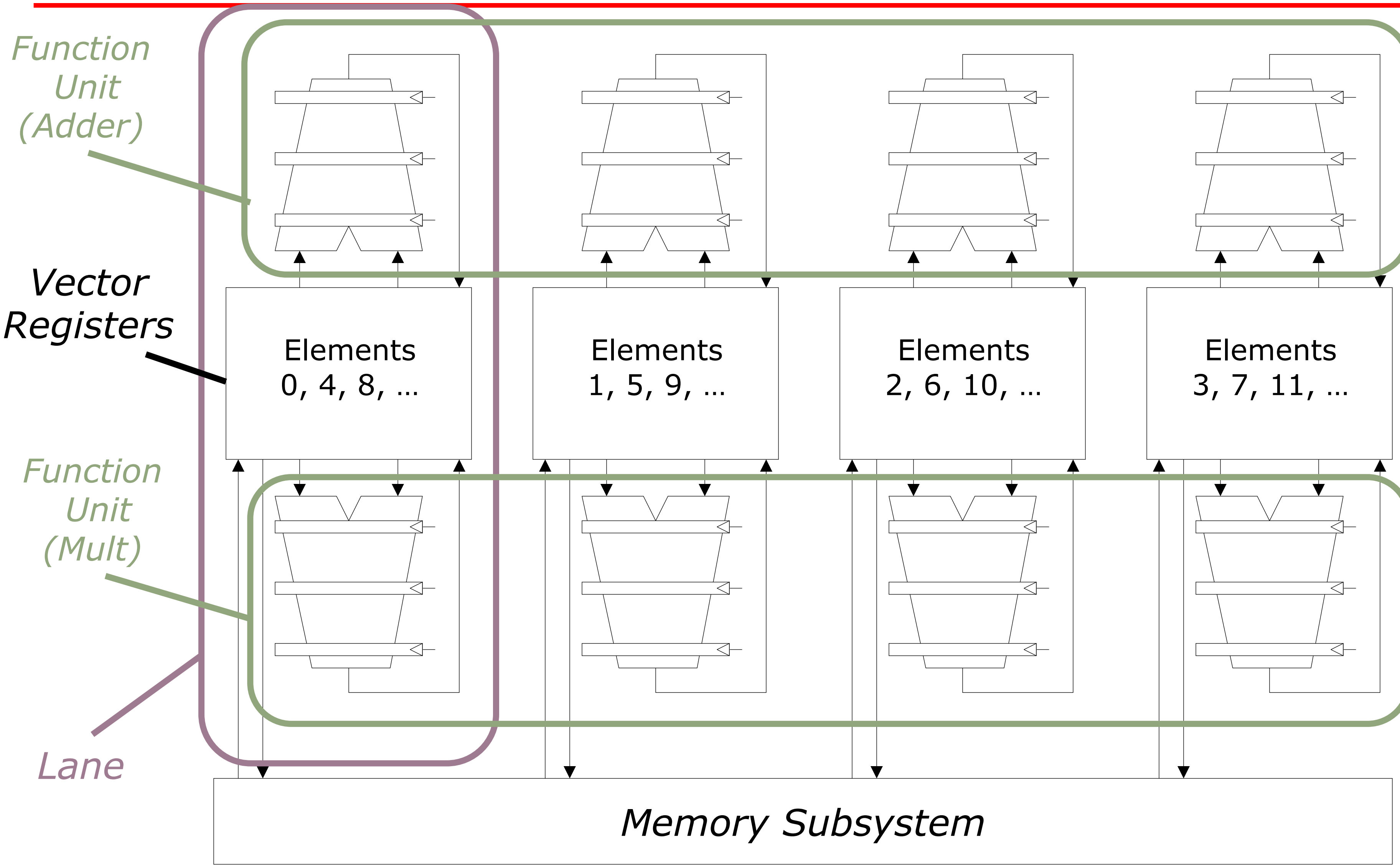
V3 ← V1 * V2

Vector Instruction Execution

ADDV C,A,B



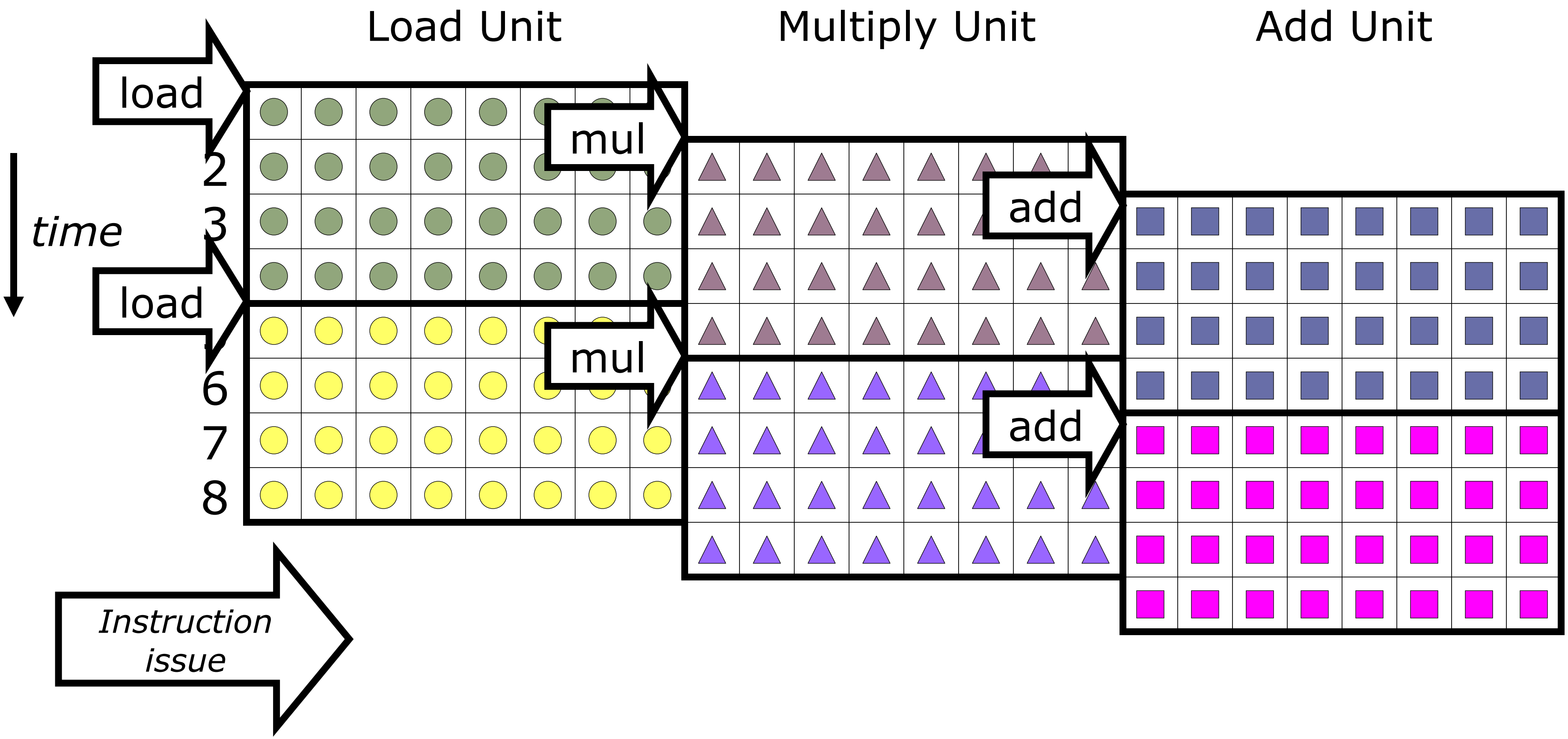
Vector Unit Structure



Vector Instruction Parallelism

Can overlap execution of multiple vector instructions

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

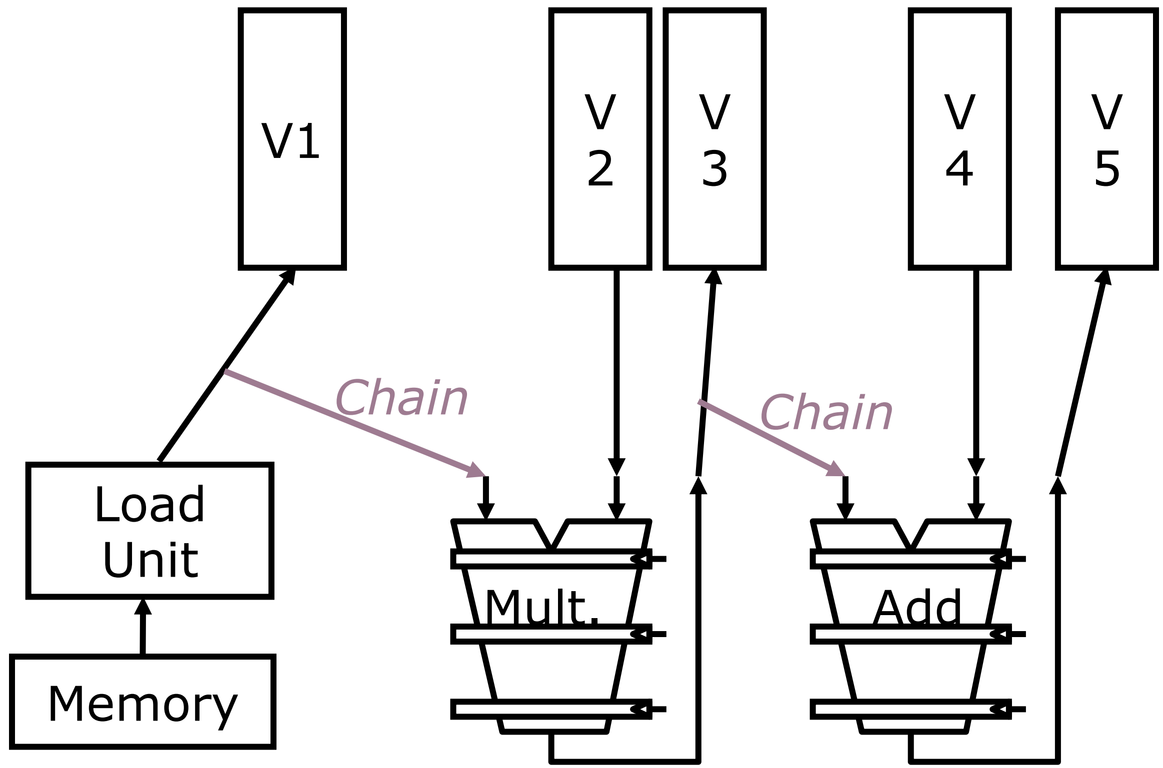


Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Chaining

Problem: Long latency for RAW register dependencies

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

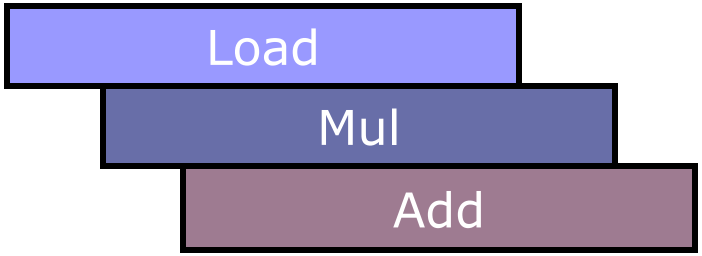


- Vector version of register bypassing
  - introduced with Cray-1

Vector Chaining Advantage

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

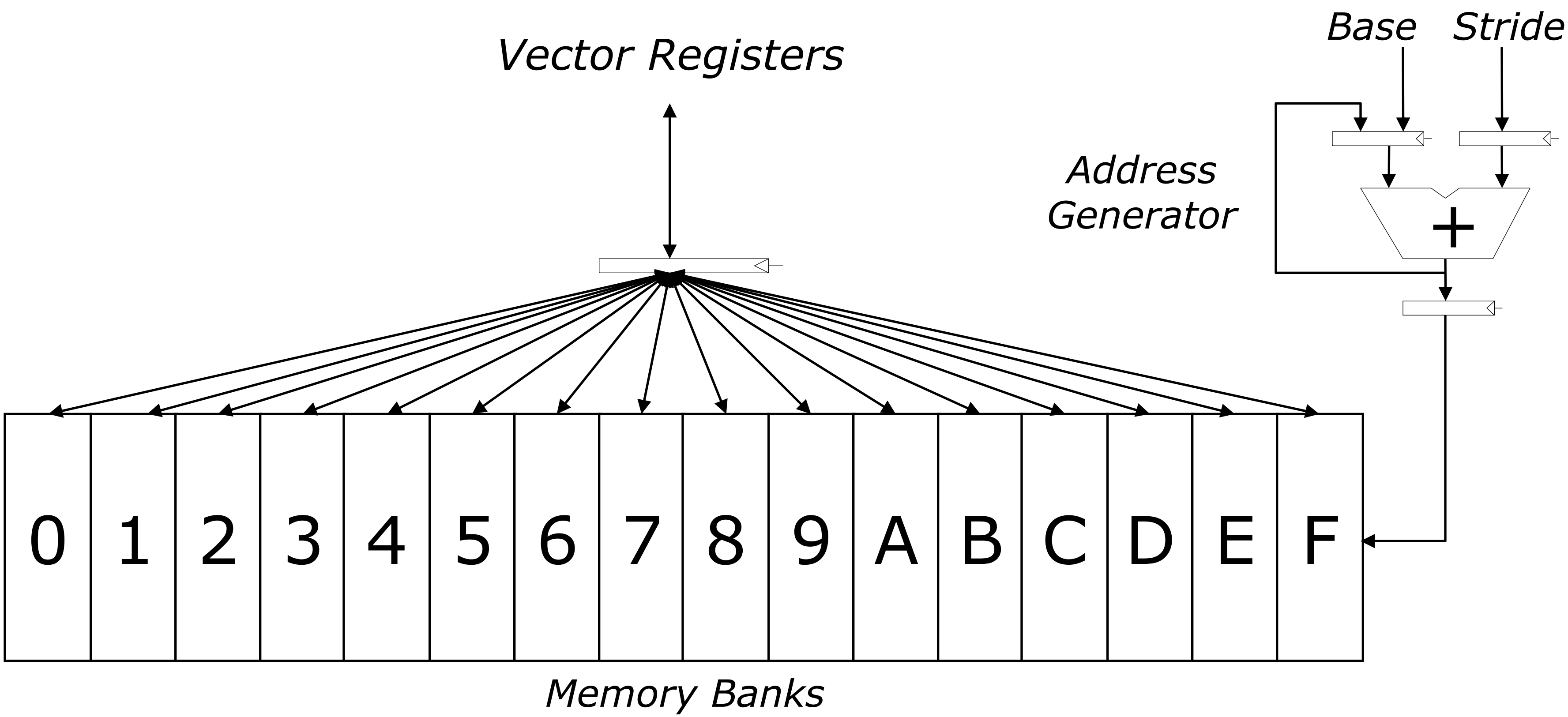
- With chaining, can start dependent instruction as soon as first result appears



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)

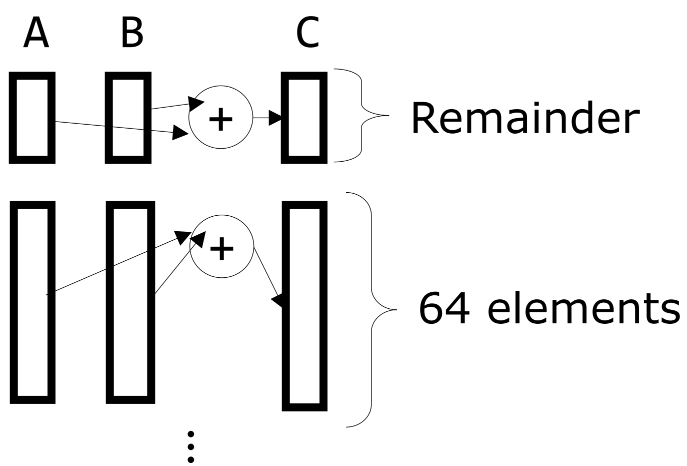


Vector Stripmining

Problem: Vector registers have finite length

Solution: Break loops into pieces that fit in registers, *"Strip mining"*

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




```
 ANDI x11, xN, 63     # N mod 64
 MTC1 vlr, x11        # Do remainder
loop:
 LV v1, xA
 SLL x12, x11, 3      # Multiply by 8
 ADD xA, xA, x12      # Bump A pointer
 LV v2, xB
 ADD xB, xB, x12      # Bump B pointer
 ADDV.D v3, v1, v2
 SV v3, xC
 ADD xC, xC, x12      # Bump C pointer
 SUB xN, xN, x11      # Subtract elements
 LI x11, 64
 MTC1 vlr, x11        # 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 NOP at elements where mask bit is clear

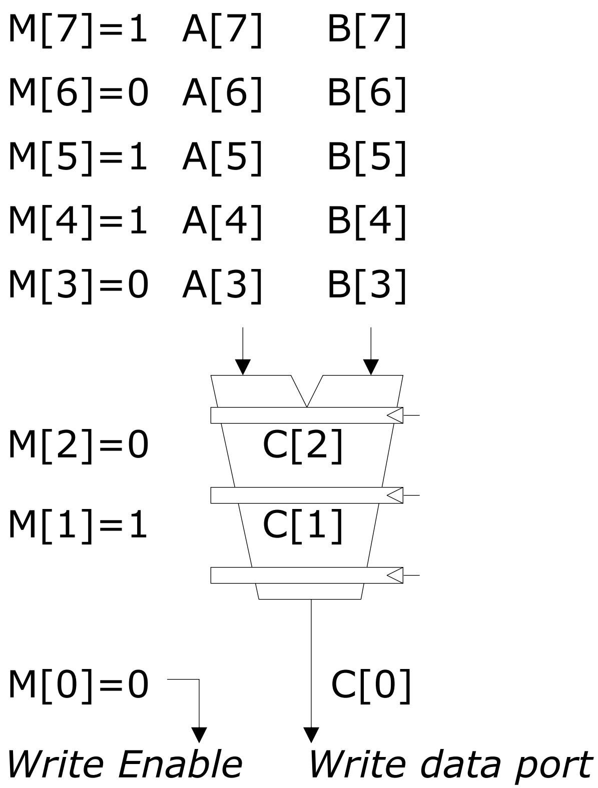
Code example:

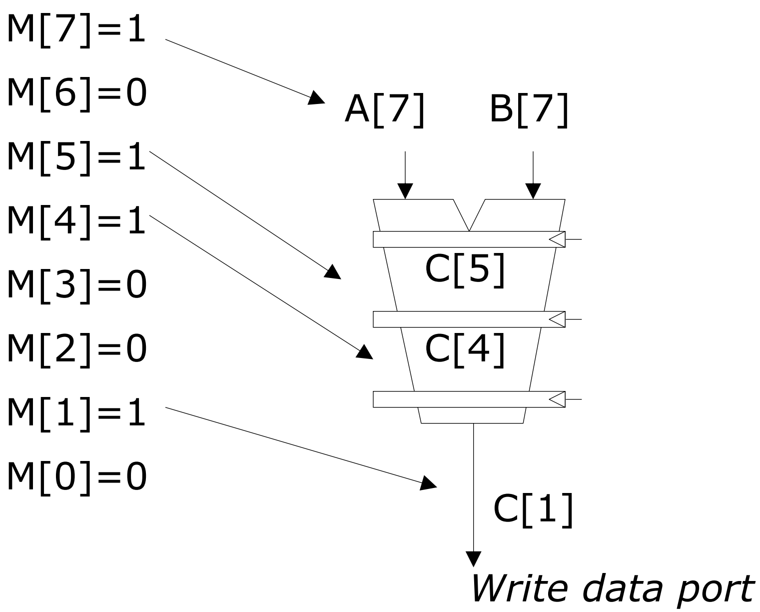
```
CVM               # Turn on all elements
LV vA, xA         # Load entire A vector
SGTVS.D vA, f0    # Set bits in mask register where A>0
LV vA, xB         # Load B vector into A under mask
SV vA, xA         # Store A back to memory under mask
```

Masked Vector Instructions

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

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





$$C[i]=A[i]+B[i]$$

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, xD            # Load indices in D vector
LVI vC, xC, vD       # Load indirect from xC base
LV vB, xB            # Load B vector
ADDV.D vA, vB, vC    # Do add
SV vA, xA            # Store result
```

Is this a correct translation? Yes if A doesn't overlap with the other vectors

Vector Scatter/Gather

Scatter example:

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

Is the following a correct translation?

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

Incorrect if B may contain repeated 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)

NEC Vector Machines

Single node SX systems

System	Introduction	Max. CPUs	Peak CPU double precision GFLOPS	Peak system GFLOPS	Max. main memory	System memory B/W (GB/s)	Memory B/W per CPU (GB/s)
SX-7[11]	2002[11]	32	8.83	282	256 GB	1129	35.3
SX-8[17][11]	2004[17][11]	8	16[11]	128	128 GB	512	64
SX-8i[citation needed]	2005				32 GB		
SX-8R[citation needed]	2006	8	35.2	281.6	256 GB	563.2	70.4
SX-9[11]	2007	16	102.4[11]	1638	1 TB	4096	256
SX-ACE[citation needed]	2013	1	256	256	1 TB	256	256
SX-Aurora TSUBASA[citation needed]	2017	8	2450	19600	8×48GB	8×1200	1200
SX-Aurora TSUBASA Type 20[citation needed]	2021	8	3070	24560	8×48GB	8×1530	1530

Source: Wikipedia

Multimedia/SIMD 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 incarnations initially reused existing registers
  - e.g., 64-bit registers split into 2x32bits or 4x16bits or 8x8bits
- Trend towards larger vector support in microprocessors
  - e.g. x86:
    - MMX (64 bits)
    - SSEx (128 bits)
    - AVX (256 bits)
    - AVX-512 (512 bits/masks)

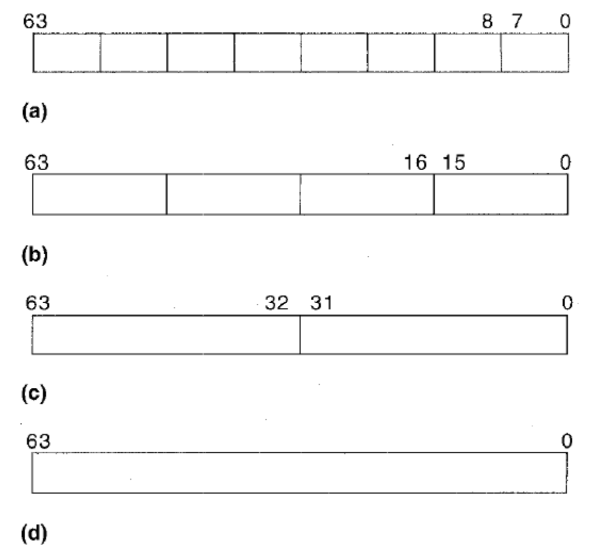


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

Intel SIMD Evolution

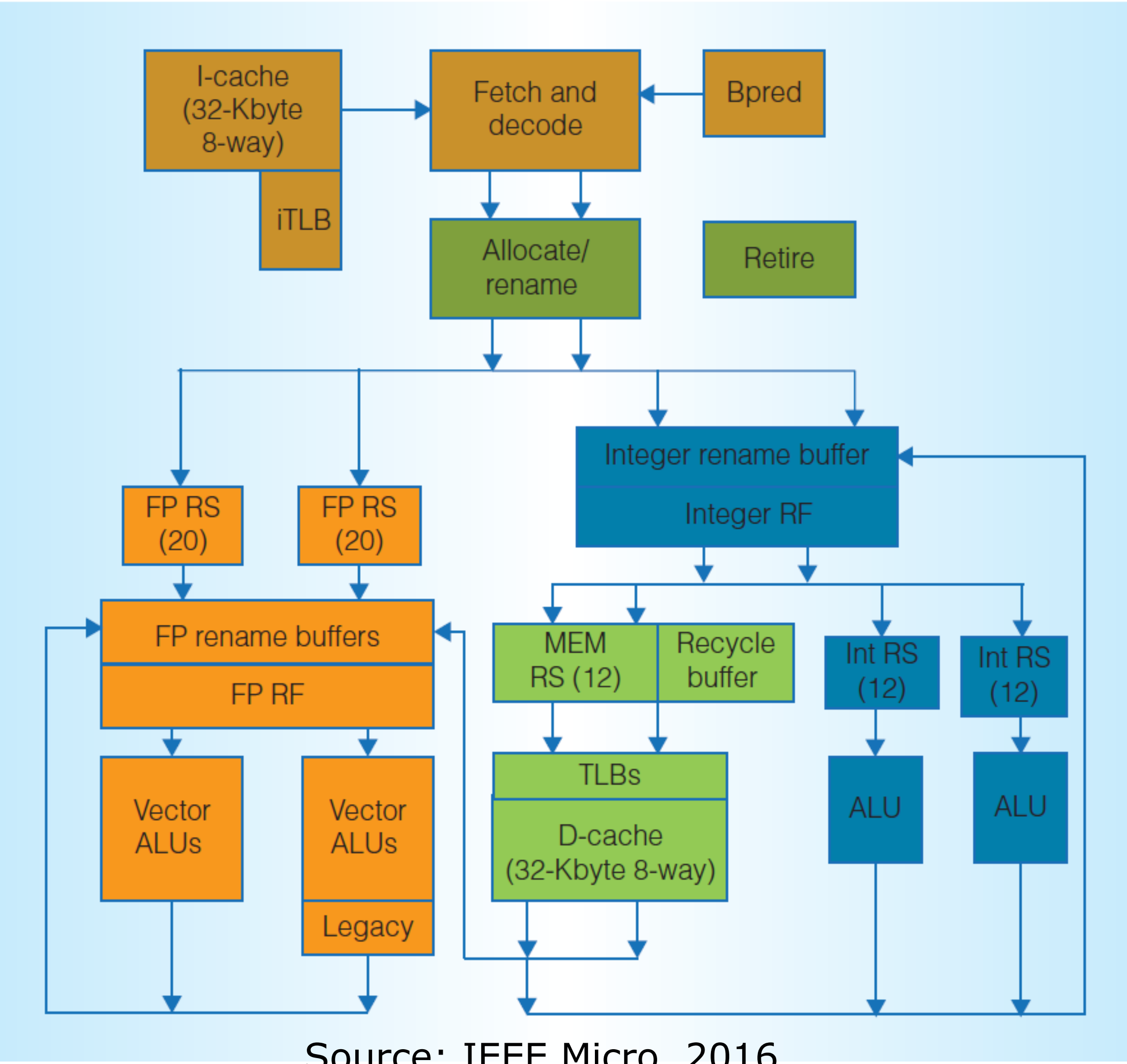
Implementations:

- Intel MMX (1996) – 64bits
  - Eight 8-bit integer ops, or
  - Four 16-bit integer ops
  - Two 32-bit integer ops
- Streaming SIMD Extensions (SSE) (1999) – 128bits
  - 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) – 256bits
  - Four 64-bit integer/fp ops (and smaller fp types)
- AVX-512 (2017) – 512bits
  - New instructions: scatter/gather, mask registers

Multimedia Extensions vs Vectors

- Limited instruction set
  - No vector length control
  - Usually no masks
  - Up until recently, 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

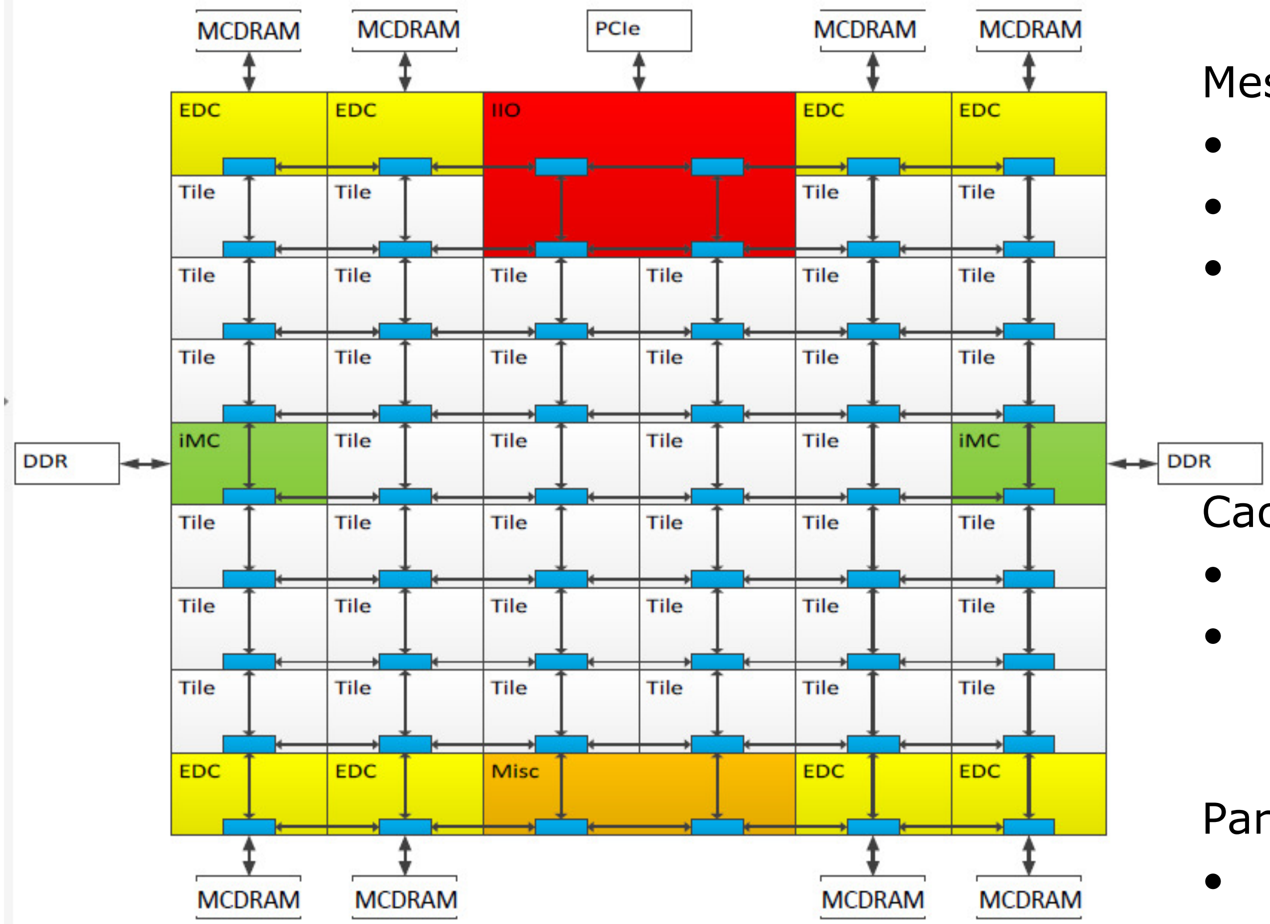
Knights Landing (KNL) CPU



Source: IEEE Micro, 2016

- 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 buffer, RS: dynamically partitioned
- Several thread choosers

Knights Landing (KNL) Mesh



Source: IEEE Micro, 2016

Mesh of Rings
- Rows/columns (half) ring
- YX routing
- Message arbitration on:
  - Injection
  - Turns

Cache Coherent Interconnect
- MESIF protocol
- Distributed directory
  - to filter snoops

Partitioning modes
- All-to-all
- Quadrant
- Sub-NUMA

Thank you!

Next Lecture: GPUs