

Lecture 14

Data Level Parallelism (2)

EEC 171 Parallel Architectures

John Owens

UC Davis

Credits

- © John Owens / UC Davis 2007–9.
- Thanks to many sources for slide material: Computer Organization and Design (Patterson & Hennessy) © 2005, Computer Architecture (Hennessy & Patterson) © 2007, Inside the Machine (Jon Stokes) © 2007, © Dan Connors / University of Colorado 2007, © Kathy Yelick / UCB 2007, © Wen-Mei Hwu/David Kirk, University of Illinois 2007, © David Patterson / UCB 2003–7, © John Lazzaro / UCB 2006, © Mary Jane Irwin / Penn State 2005, © John Kubiawicz / UCB 2002, © Krste Asinovic/Arvind / MIT 2002, © Morgan Kaufmann Publishers 1998.

Outline

- Vector machines (Cray 1)
- Vector complexities
- Massively parallel machines
(Thinking Machines CM-2)
- Parallel algorithms

Vector Processing

- Appendix F & slides by Krste Asanovic, MIT

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
- CDC 6600 (Cray, 1964) regarded as first supercomputer

Seymour Cray

- “Anyone can build a fast CPU. The trick is to build a fast system.”
- When asked what kind of CAD tools he used for the Cray-1, Cray said that he liked “#3 pencils with quadrille pads”. Cray recommended using the backs of the pages so that the lines were not so dominant.
- When he was told that Apple Computer had just bought a Cray to help design the next Apple Macintosh, Cray commented that he had just bought a Macintosh to design the next Cray.
- “Parity is for farmers.”

Supercomputer Applications

- Typical application areas
 - Military research (nuclear weapons, cryptography)
 - Scientific research
 - Weather forecasting
 - Oil exploration
 - Industrial design (car crash simulation)
- All involve huge computations on large data sets
- In 70s–80s, Supercomputer \equiv Vector Machine

Vector Supercomputers

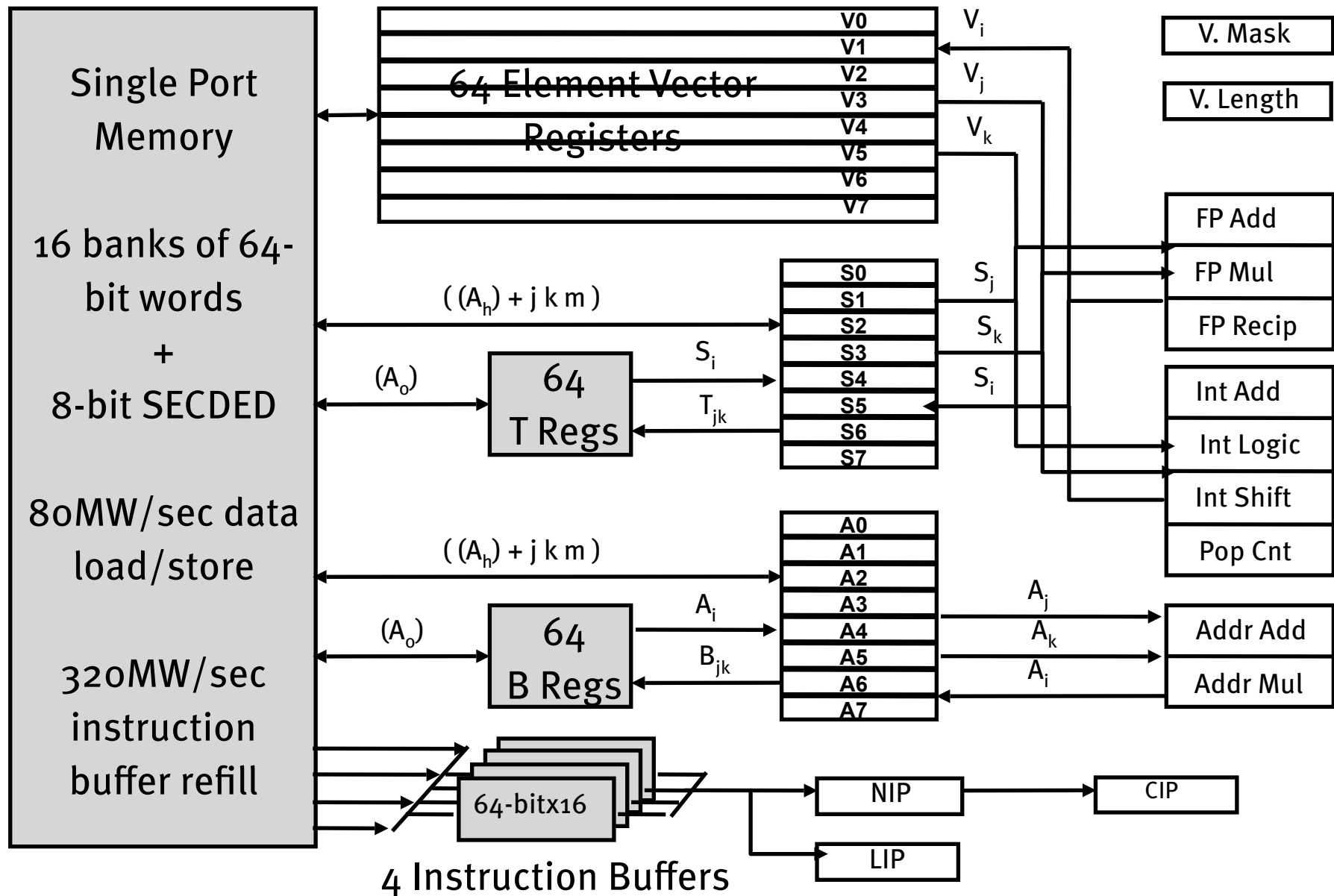
- Epitomized by Cray-1, 1976:
 - Scalar Unit + Vector Extensions
 - Load/Store Architecture
 - *Vector Registers*
 - *Vector Instructions*
 - Hardwired Control
 - Highly Pipelined Functional Units
 - Interleaved Memory System
 - No Data Caches
 - No Virtual Memory

Cray-1 (1976)

- 4 chip types (ECL):
 - 16x4 bit bipolar registers
 - 1024x1 bit SRAM
 - 4/5 input NAND gates
- 138 MFLOPS sustained,
250 MFLOPS peak



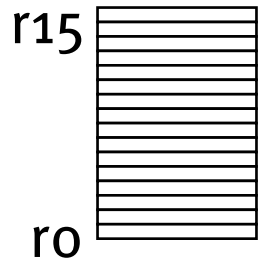
Cray-1 (1976)



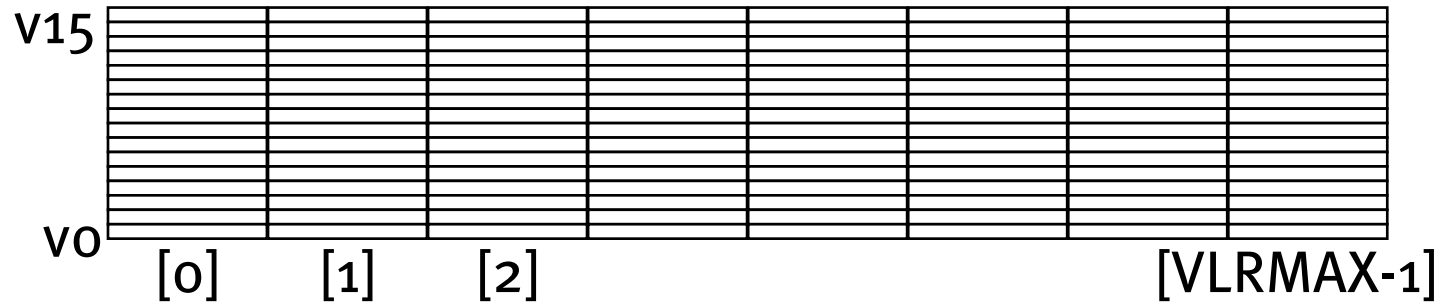
memory bank cycle 50 ns processor cycle 12.5 ns (80 MHz)

Vector Programming Model

Scalar Registers



Vector Registers

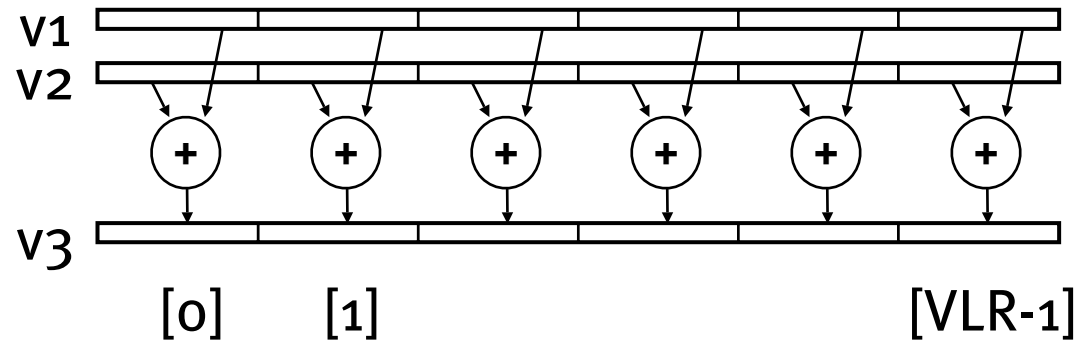


Vector Length Register



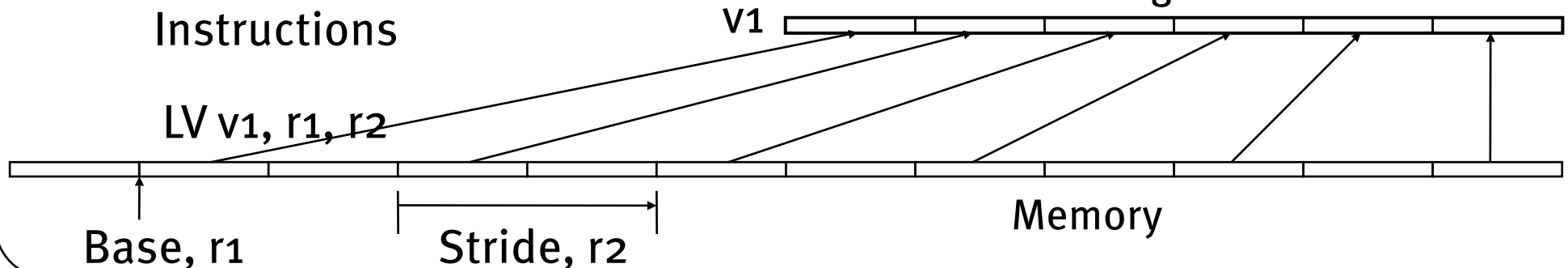
Vector Arithmetic
Instructions

ADDV v_3, v_1, v_2



Vector Load and Store
Instructions

LV v_1, r_1, r_2



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
```

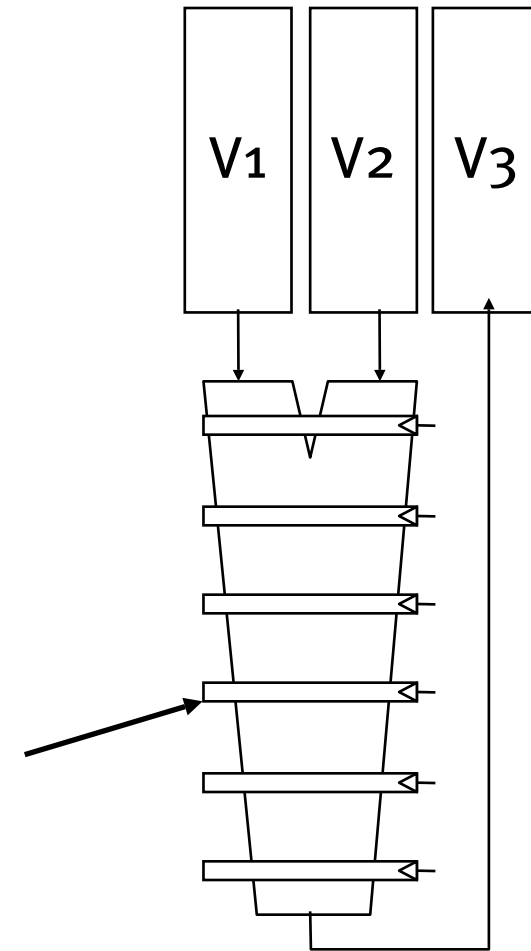
Vector Instruction Set Advantages

- 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 the same pattern as previous instructions
 - access a contiguous block of memory (unit-stride load/store), or
 - access memory in a known pattern (strided load/store)
- Scalable
 - can run same object code on more parallel pipelines or 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!)

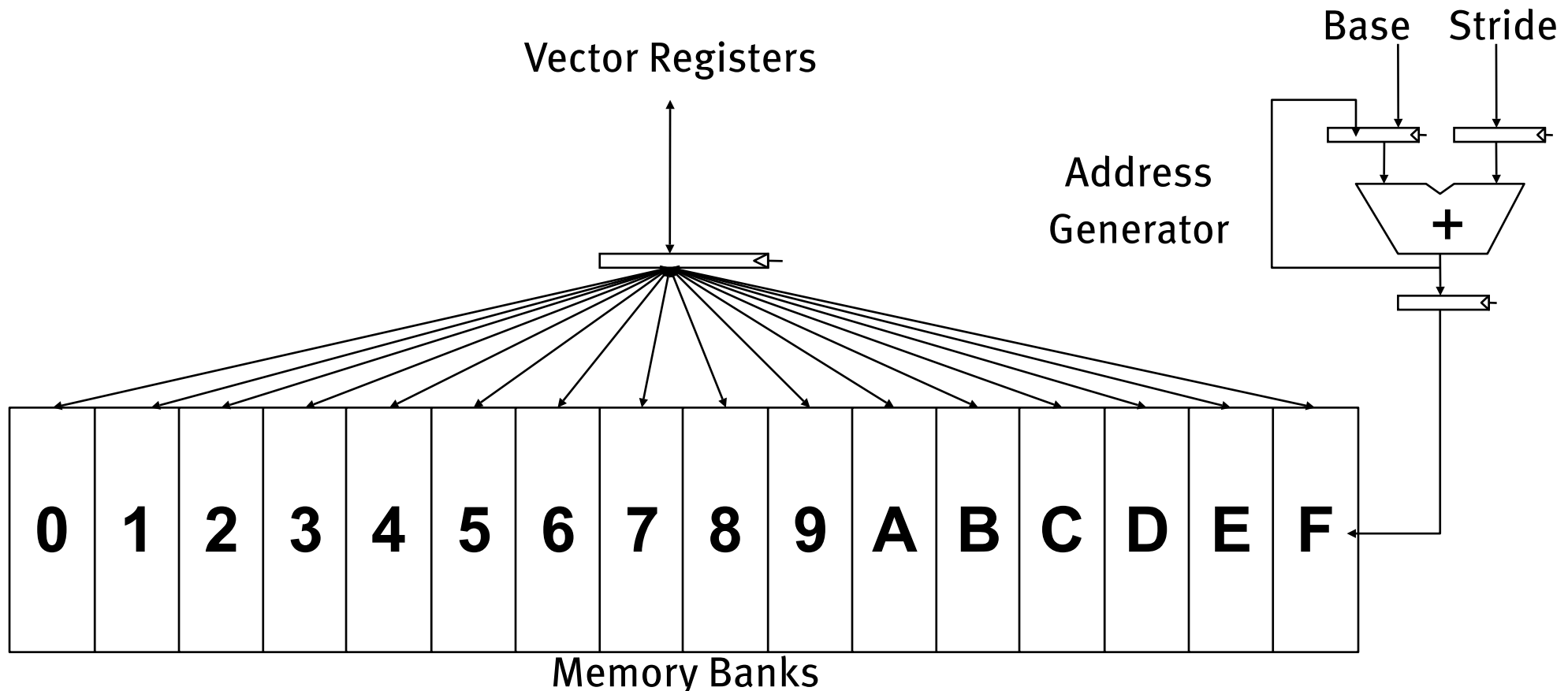
Six stage multiply pipeline



$$V_3 \leftarrow V_1 * V_2$$

Vector Memory System

- Cray-1: 16 banks, 64b wide per bank, 4 cycle bank busy time, 12 cycle latency
- Bank busy time: Cycles between accesses to same bank



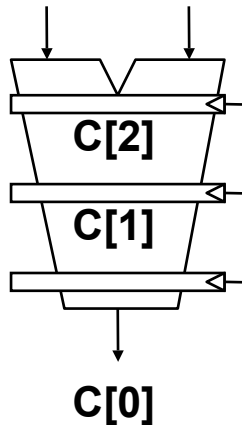
Vector Instruction Execution

ADDV C, A, B

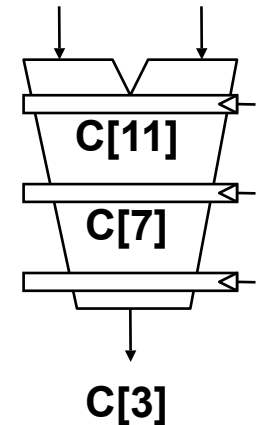
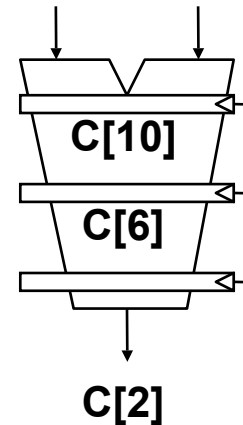
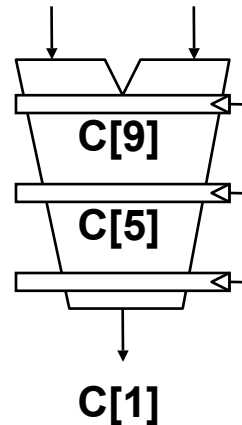
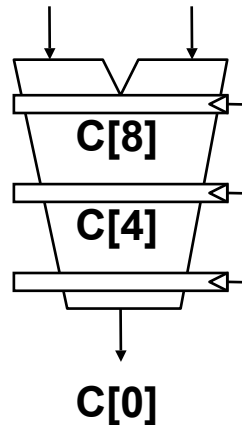
Execution using one pipelined functional unit

Execution using four pipelined functional units

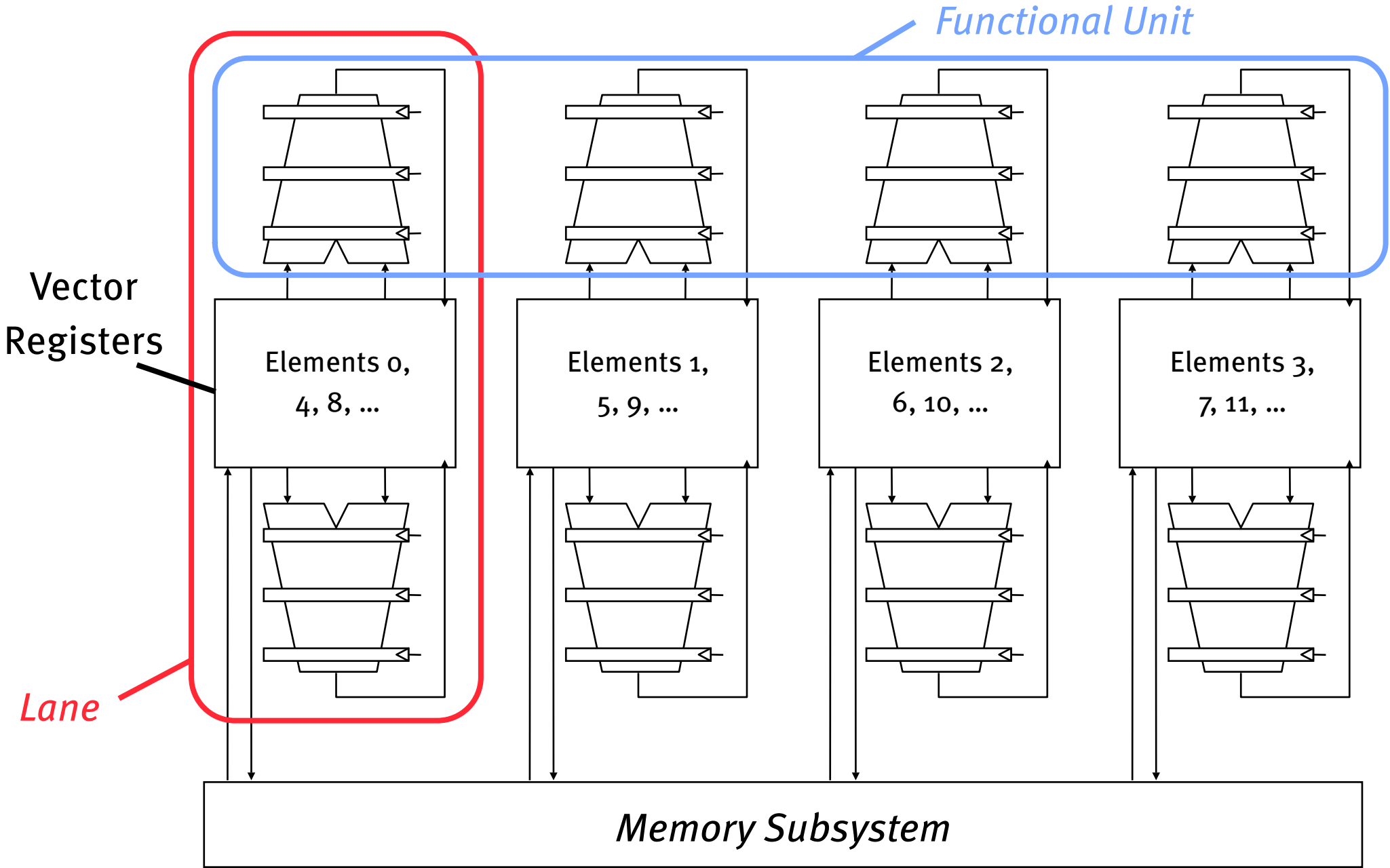
A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]



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]

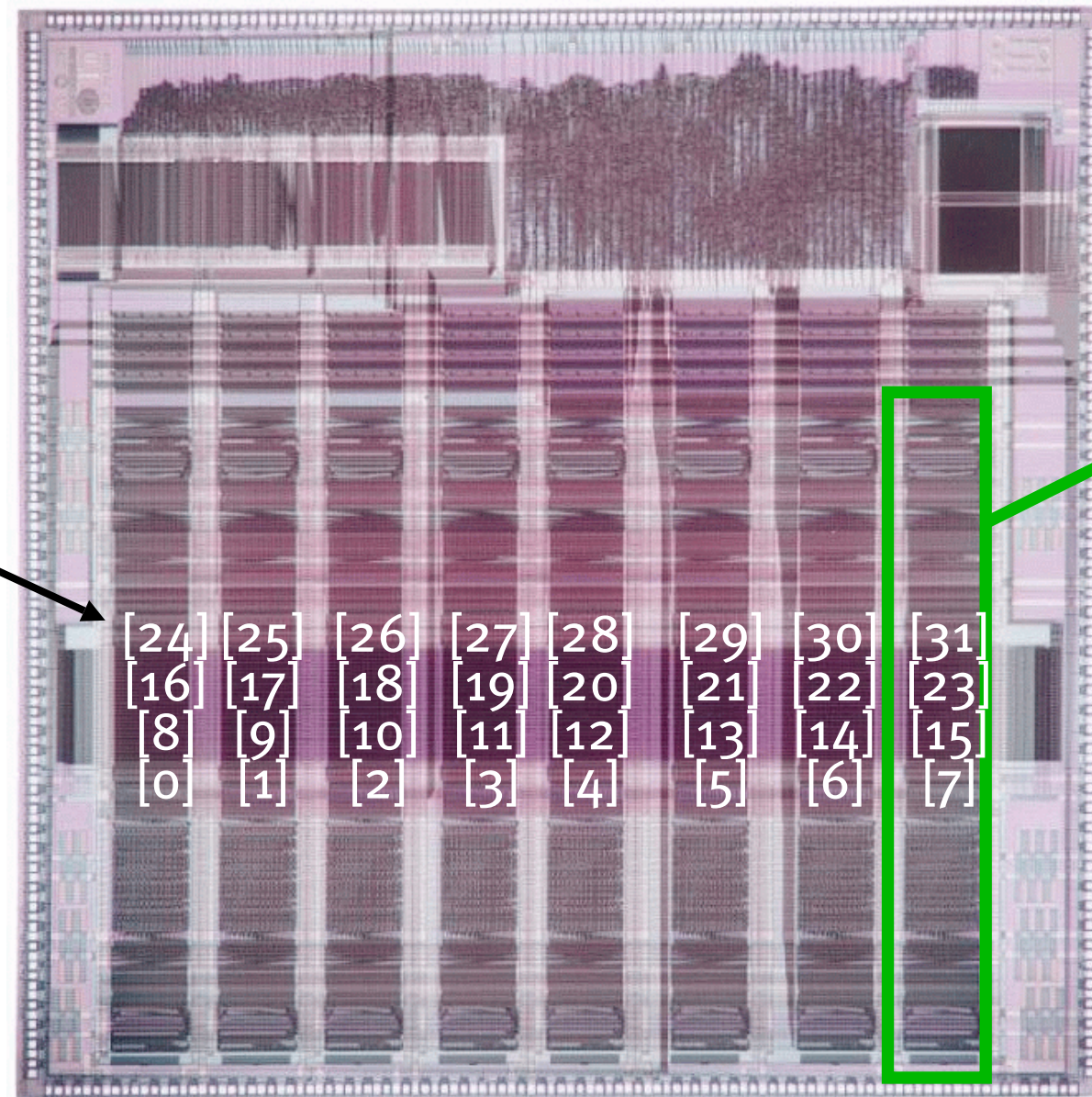


Vector Unit Structure



To Vector Microprocessor (1995)

*Vector register
elements striped
over lanes*



Lane

Vector Memory-Memory vs. 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?
- VMMA make it difficult to overlap execution of multiple vector operations, why?
- 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)

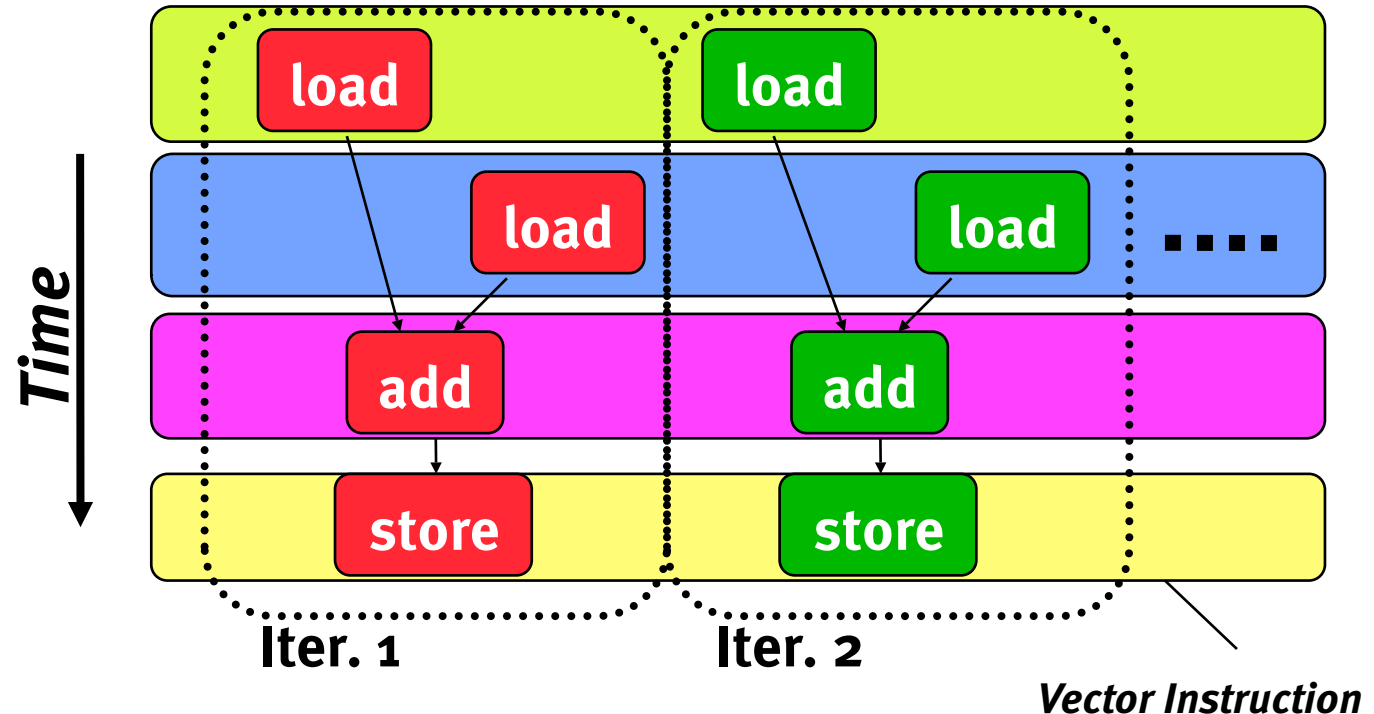
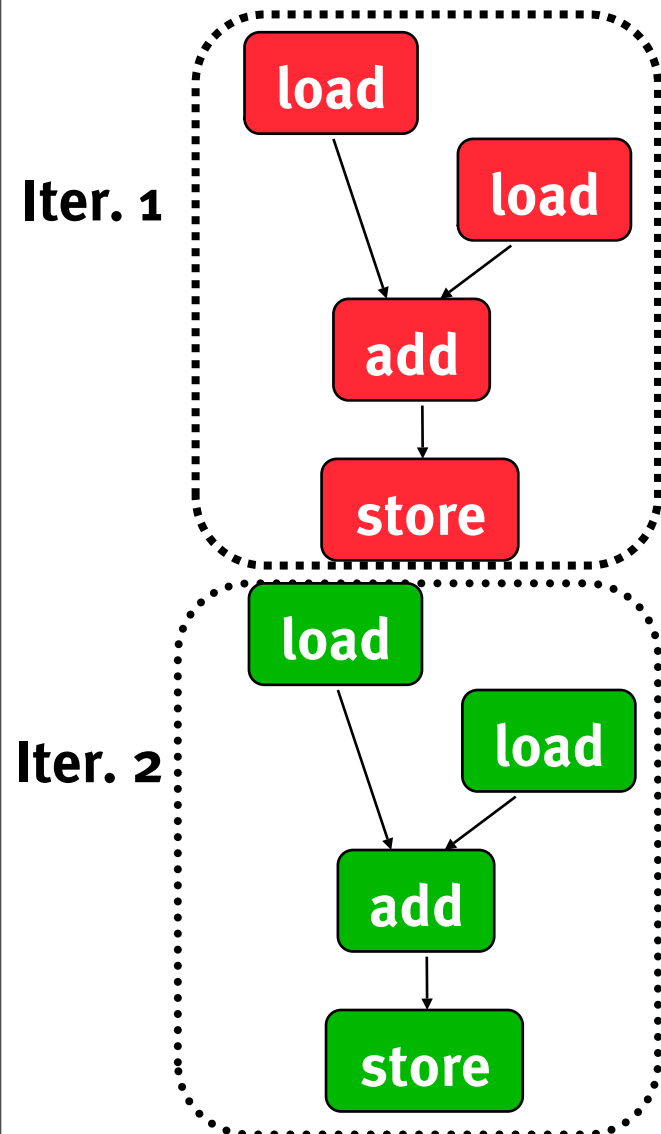
Automatic Code Vectorization

```
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

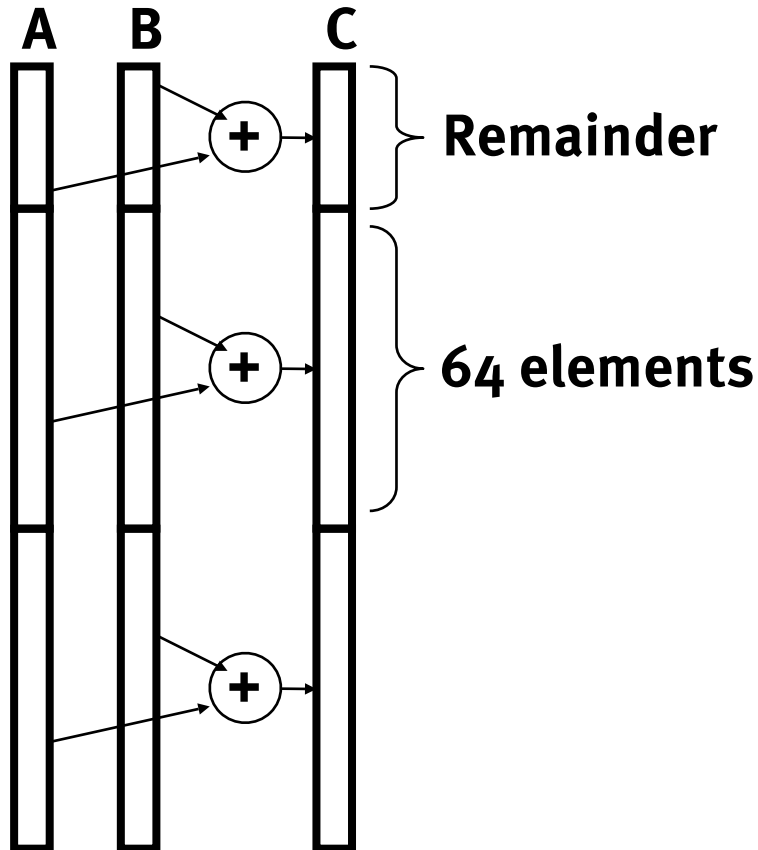
Guy Steele, Dr Dobbs Journal 24 Nov 2005

- “What might a language look like in which parallelism is the default? How about data-parallel languages, in which you operate, at least conceptually, on all the elements of an array at the same time? These go back to APL in the 1960s, and there was a revival of interest in the 1980s when data-parallel computer architectures were in vogue. But they were not entirely satisfactory. I'm talking about a more general sort of language in which there are control structures, but designed for parallelism, rather than the sequential mindset of conventional structured programming. What if do loops and for loops were normally parallel, and you had to use a special declaration or keyword to indicate sequential execution? That might change your mindset a little bit.”

Vector Stripmining

- Problem: Vector registers have finite length
- Solution: Break loops into pieces that fit into vector 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?
```

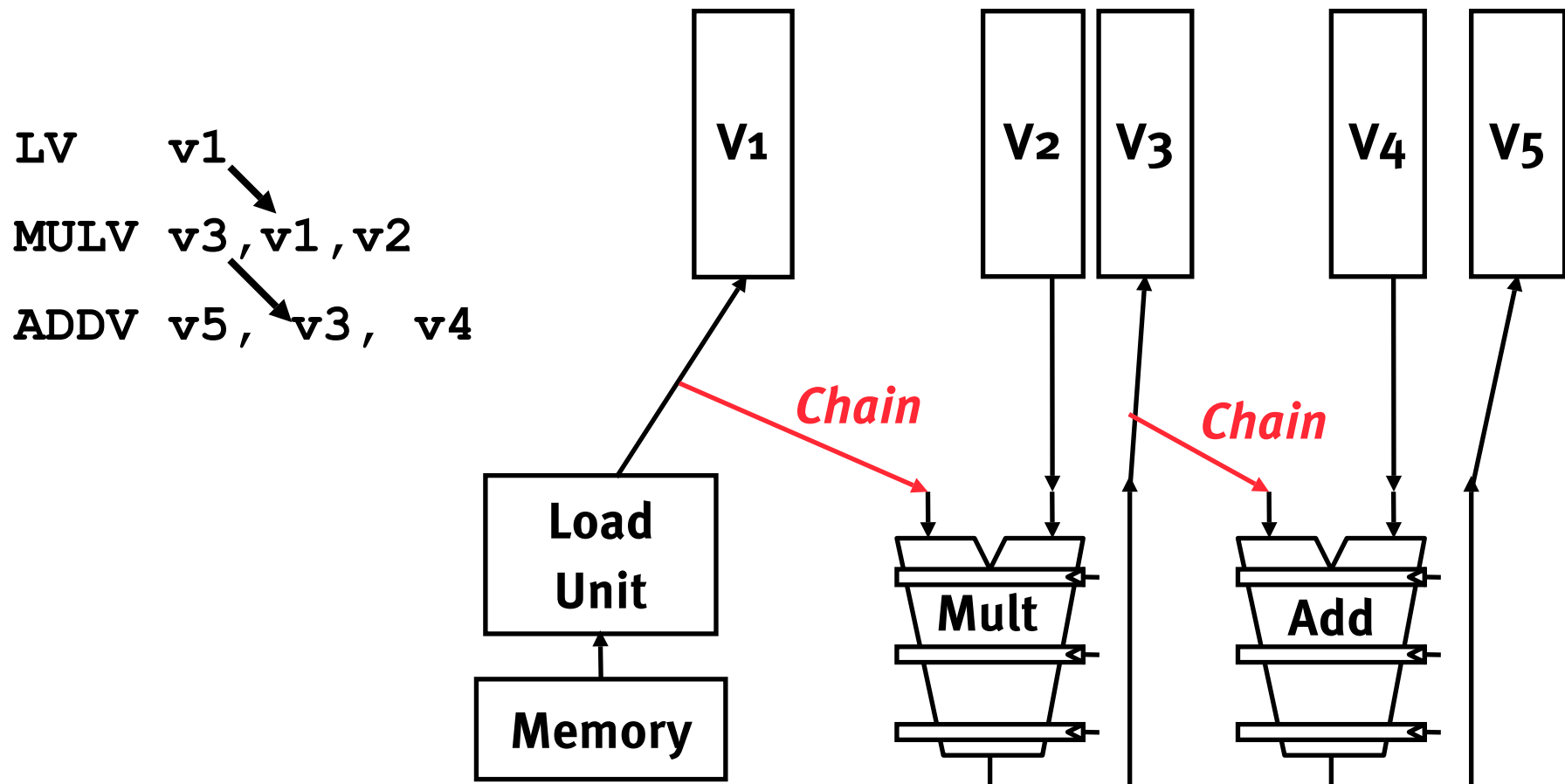
Vector Inefficiency

- Must wait for last element of result to be written before starting dependent instruction



Vector Chaining

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

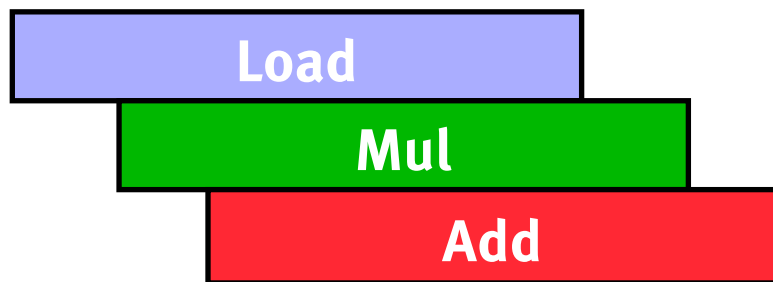


Vector Chaining Advantage

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

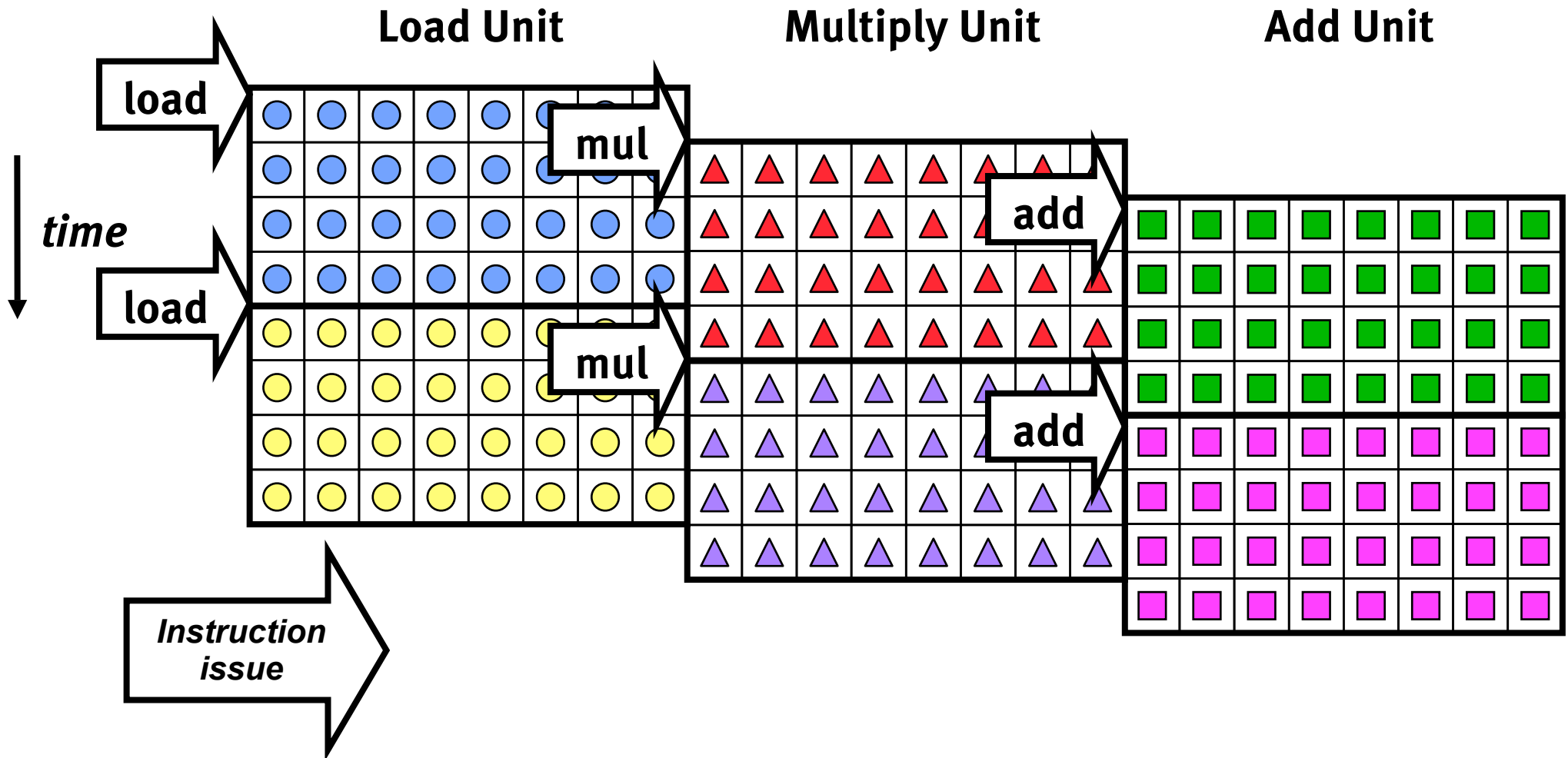


- With chaining, can start dependent instruction as soon as first result appears



Vector Instruction Parallelism

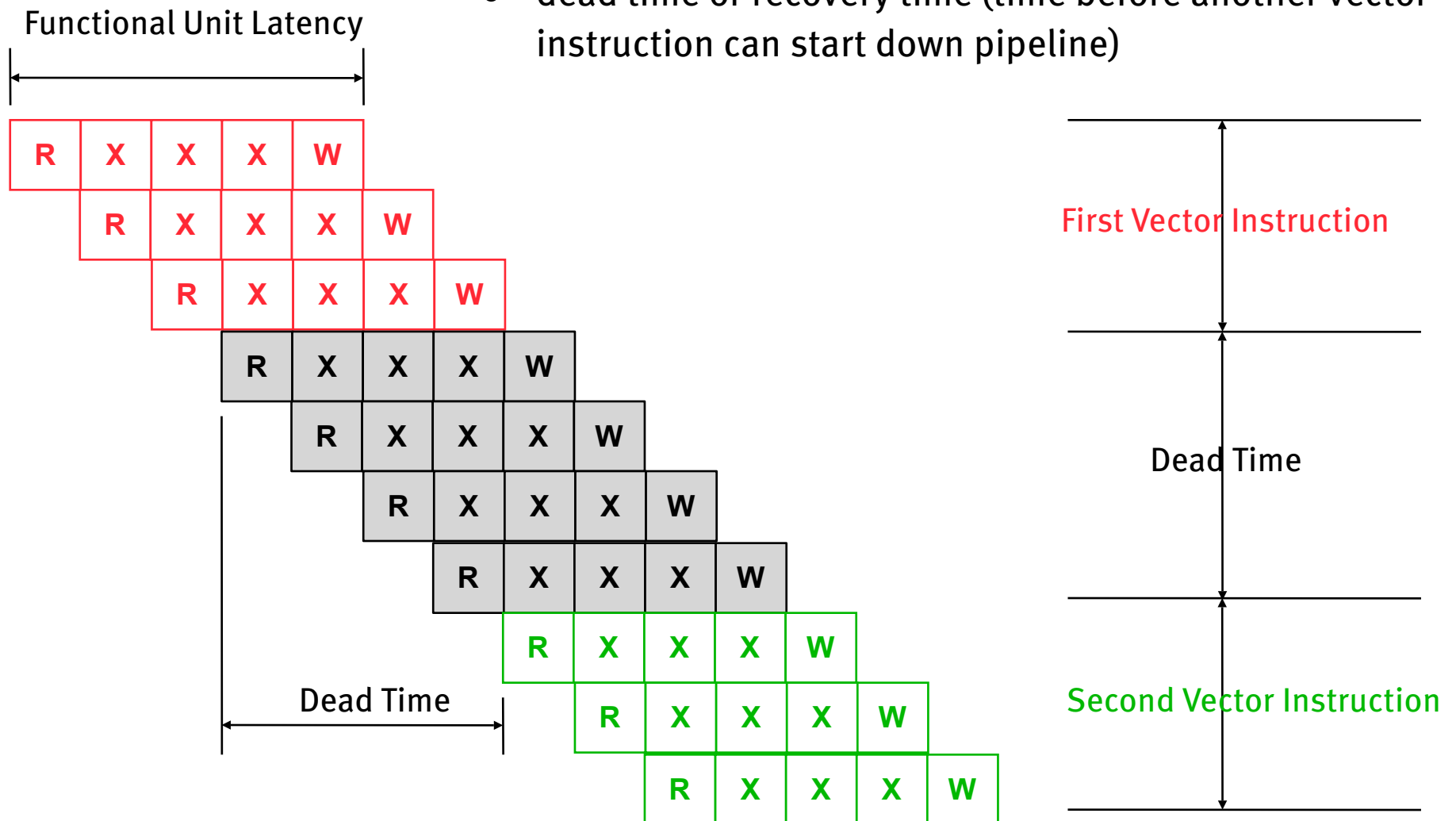
- Can overlap execution of multiple vector instructions
 - example machine has 32 elements per vector register and 8 lanes



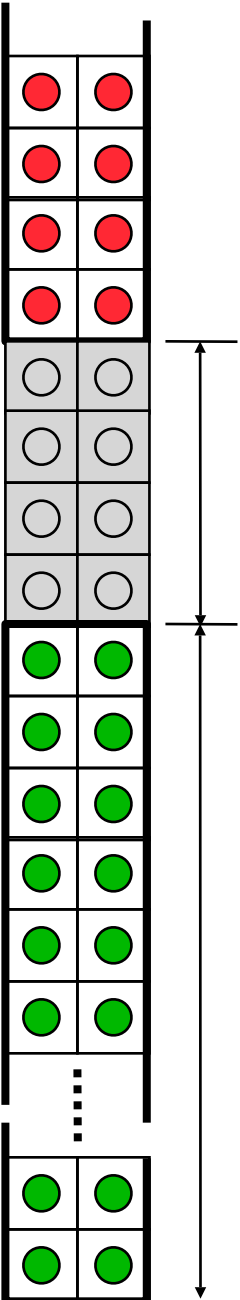
Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector 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



4 cycles dead time

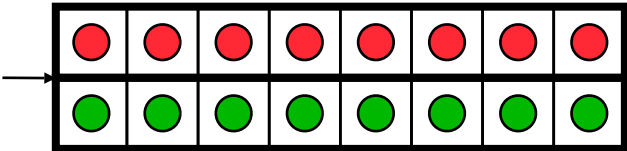
64 cycles active

Cray C90, Two lanes

4 cycle dead time

*Maximum efficiency 94% with
128 element vectors*

No dead time



To, Eight lanes

No dead time

*100% efficiency with 8 element
vectors*

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 is indexed write

- Scatter example:

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

- Gather then scatter ...

```
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
```

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 (vector mask is implicit in this instruction set):

```
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]

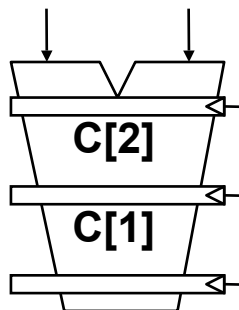
M[4]=1 A[4] B[4]

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

M[2]=0

M[1]=1

M[0]=0



C[0]

Write data port

Write Enable

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

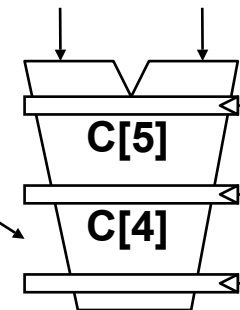
M[3]=0

M[2]=0

M[1]=1

M[0]=0

A[7] B[7]

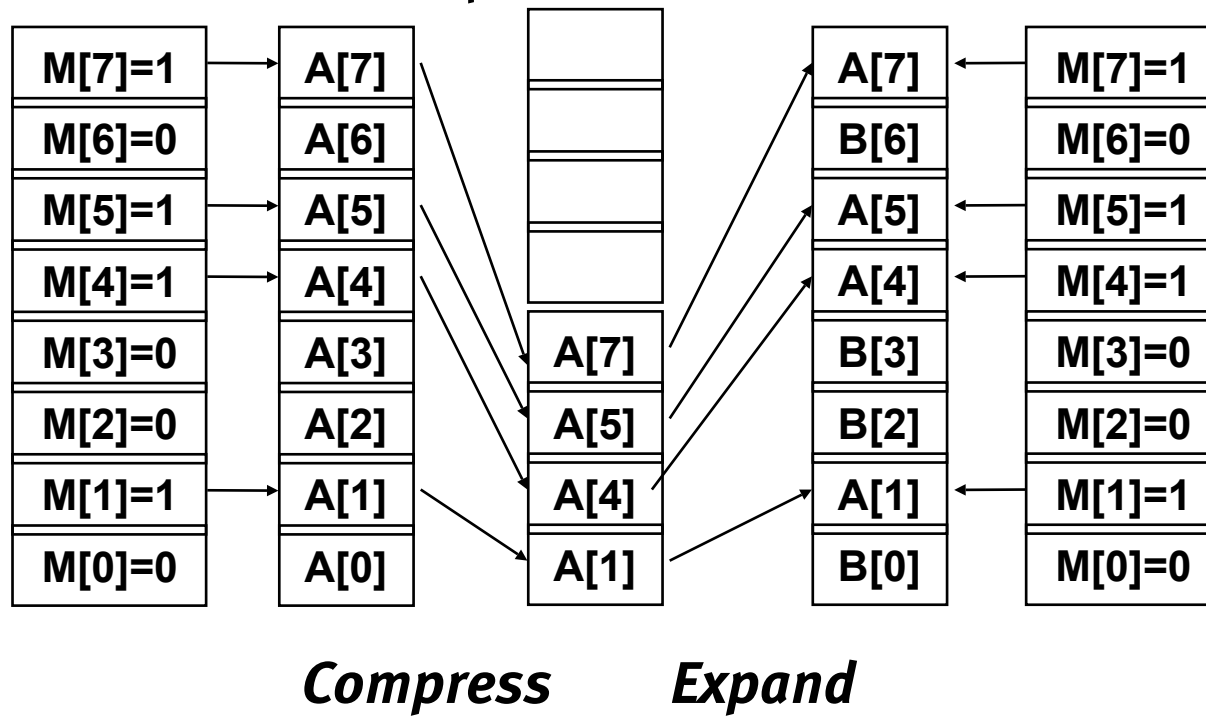


C[1]

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



Used for density-time conditionals and also for general selection operations

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)
```

A Modern Vector Super: NEC SX-6 (2003)

- CMOS Technology
 - 500 MHz CPU, fits on single chip
 - SDRAM main memory (up to 64 GB)
- Scalar unit
 - 4-way superscalar with out-of-order and speculative execution
 - 64 KB I-cache and 64 KB data cache



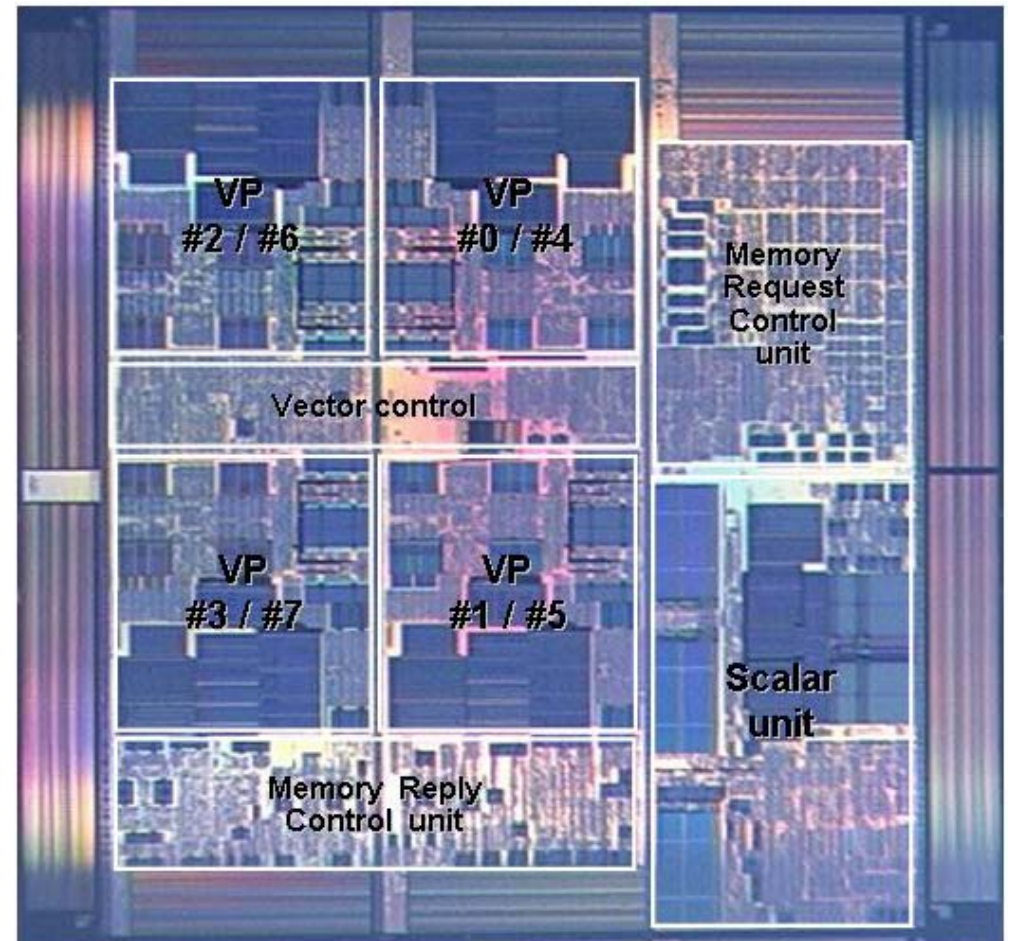
A Modern Vector Super: NEC SX-6 (2003)

- 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 per lane
 - 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)



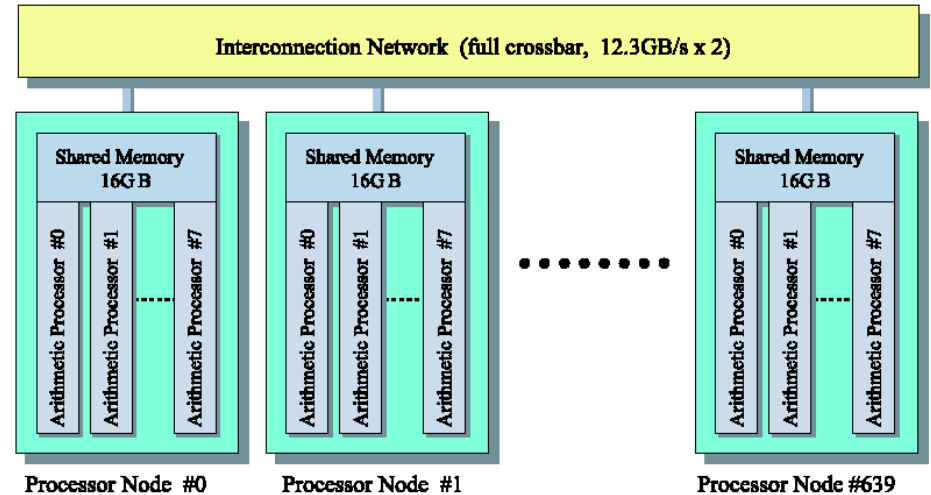
SX-6 Die Photo

- 0.15 μm CMOS
- 60M transistors
- 432 mm^2
- 500 MHz scalar, 1 GHz vector



NEC Earth Simulator

- 5120 CPUs, 41 TFLOPS peak, 35 sustained
- Each node: 8 CPUs, 32 memory modules
- 16 GB local memory
- 32 GB/s to local memory per CPU
- Interconnect: full 640x640 crossbar



TIME

2002 Best Inventions

Rank	Manufacturer Computer/ Procs	GFLOPS
1	NEC Earth-Simulator/ 5120	35860.00
2	Hewlett-Packard ASCI Q - AlphaServer SC ES45/1.25 GHz/ 4096	7727.00
3	Hewlett-Packard ASCI Q - AlphaServer SC ES45/1.25 GHz/ 4096	7727.00
4	IBM ASCI White, SP Power3 375 MHz/ 8192	7226.00
5	Linux NetworX MCR Linux Cluster Xeon 2.4 GHz - Quadrics/ 2304	5694.00
6	Hewlett-Packard AlphaServer SC ES45/1 GHz/ 3016	4463.00
7	Hewlett-Packard AlphaServer SC ES45/1 GHz/ 2560	3980.00
8	HPTi Aspen Systems, Dual Xeon 2.2 GHz - Myrinet2000/ 1536	3337.00
9	IBM pSeries 690 Turbo 1.3GHz/ 1280	3241.00
10	IBM pSeries 690 Turbo 1.3GHz/ 1216	3164.00

What we've learned

- SIMD instructions
 - Fixed width (usually 4), fit into standard scalar instruction set
 - Examples: MMX, SSE, AltiVec
- Vector instructions
 - Operate on arbitrary length vectors
 - HW techniques: vector registers, lanes, chaining, masks

What's Next

- Massively parallel machines
- Big idea: Write one program, run it on lots of processors
 - First we're going to look at hardware
 - Thinking Machines CM-2
 - Then we're going to look at algorithms

Name That Film!



Thinking Machines

- Goals: AI, symbolic processing, eventually scientific computing
- “In 1990, seven years after its founding, Thinking Machines was the market leader in parallel supercomputers, with sales of about \$65 million. Not only was the company profitable; it also, in the words of one IBM computer scientist, had cornered the market ‘on sex appeal in high-performance computing’.” (Inc Magazine, 15 September 1995)
- Richard Feynman, when told by Danny Hillis that he was planning to build a computer with a million processors: “That is positively the dopiest idea I ever heard.”
- Founded 1982, profitable 1989, bankrupt in 1994



1-Slide Programming Model

- Specify a discrete domain for a program (“grid”)
 - Example: Image processing, 512x128 image
- Assign a processor to each element in the grid
 - Example: 1 processor per element, so 64k processors
- Write a program for one processor
- All processors run that program

Questions To Think About

- Should the program look like a serial program that runs on one processor, or should it look like a parallel program?
- How do different elements of the program talk to each other?
- How do they synchronize, if necessary?
- What happens when some of the processors want to branch one way and some want to branch another way?
- What happens when processor store ops conflict?

CM-2 Overview

- “The Connection Machine processors are used whenever an operation can be performed simultaneously on many data objects. Data objects remain in the Connection Machine memory during execution of the program and are operated upon in parallel. This model differs from the serial model, where data objects in a computer's memory are processed one at a time, by reading each one in turn, operating on it, and then storing the result back in memory before processing the next object.”

CM-2 Overview

- 16k–64k processors
 - Up to 128 kB of memory per processor
 - Processors communicate with each other and with peripherals, all in parallel
- Front-end computer handles serial computation, interface with CM-2 back-end

Virtual Processors

- Natural way to program in parallel is to assign one processor per parallel element
 - Example: Image processing 512x128 rectangle, 64k elements
 - Think in these terms when you program!
- If you have 64k processors, great.
- If you don't, create 64k virtual processors and assign them to the physical processors
 - In a 16k processor CM-2, that's 4 virtual processors per physical processor
 - Data is striped across physical processors
 - Benefit: Allows same program to run on different-sized machines

Communication Patterns

- Global operations
 - `scalar = sum(array)`
- Matrix (row-column structure)
- Finite-differences (neighbor communication)
- Spatial to frequency domain (butterfly)
- Irregular communication

CM-2 and Communication

- Applications are generally structured:
 - First step: gather data from other elements
 - Second step: do local computation (no communication necessary)
- CM-2 has:
 - Ability to communicate with nearest neighbors using special-purpose hardware (NEWS)
 - General-purpose network to communicate with any other processors

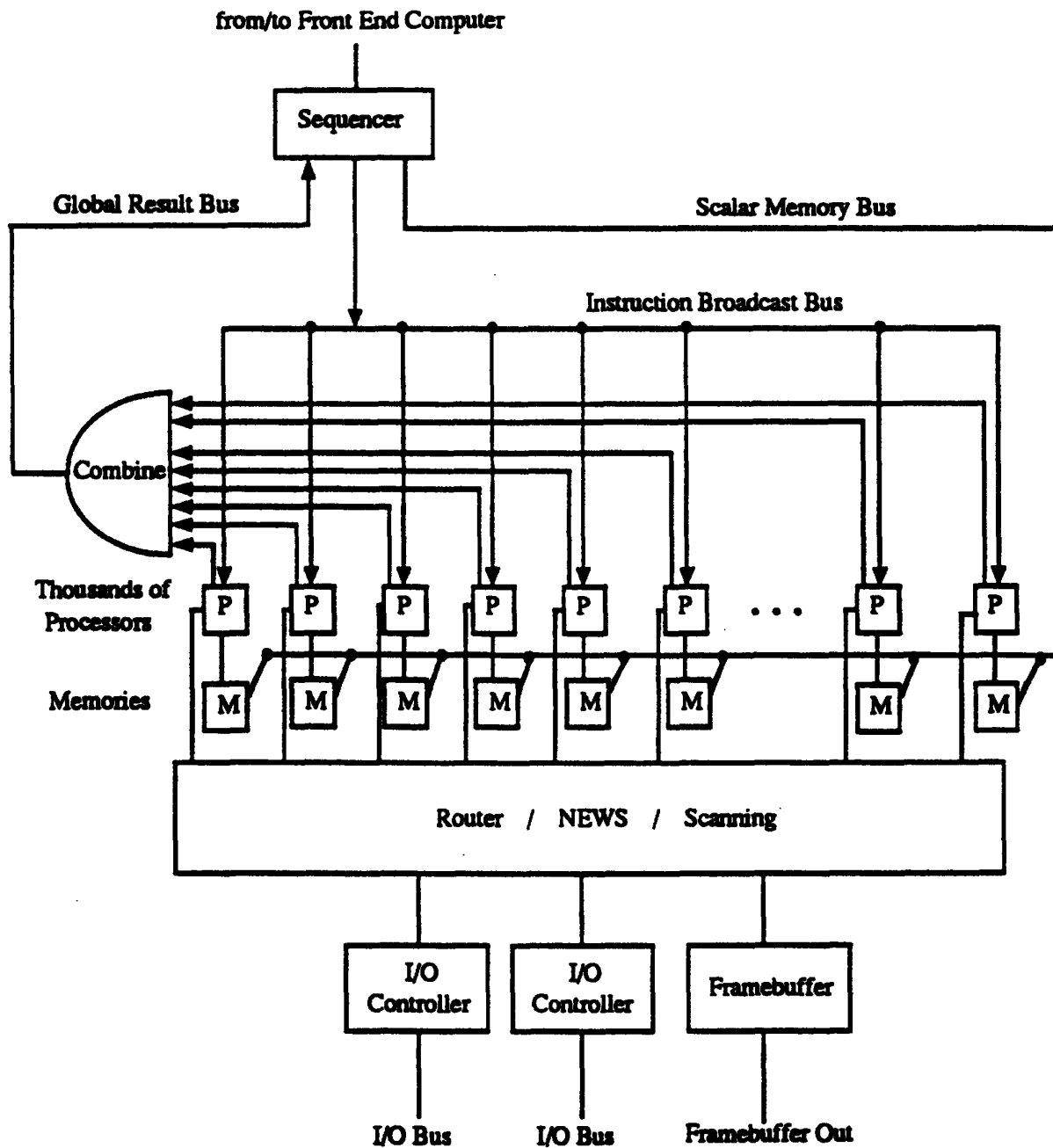
Communication Primitives

- send-with-overwrite
- send-with-logand
- send-with-logior
- send-with-logxor
- send-with-s-add
- send-with-s-multiply
- send-with-u-add
- send-with-u-multiply
- send-with-f-add
- send-with-f-multiply
- send-with-c-add
- send-with-c-multiply
- send-with-s-max
- send-with-s-min
- send-with-u-max
- send-with-u-min
- send-with-f-max
- send-with-f-min

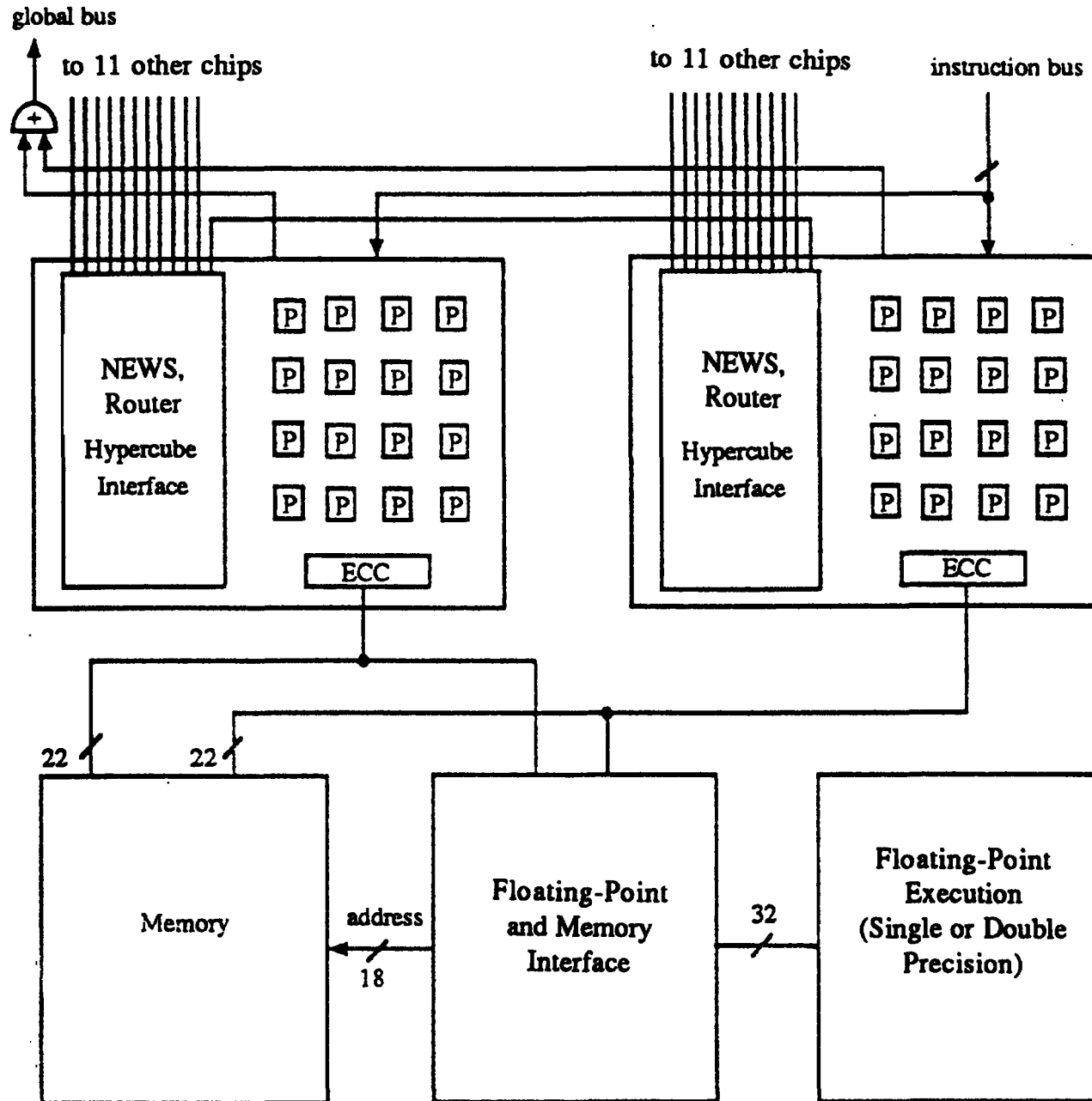
Computation + Communication Primitives

- Scan
 - Sum (or other op) of all preceding elements in a row
- Reduce
 - Sum (or other op) of all elements in a row
- Global
 - Sum (or other op) of ALL elements
- Spread
 - Sum (or other op) of particular element is distributed to all in row
- Multispread
 - Spread across multiple dimensions

CM-2 Hardware Overview



CM-2 Data Processing Node



CM-2 ALU

- 3-input, 2-output logic element
- ALU cycle:
 - Read 2 data bits from memory
 - Read 1 data bit from flag
 - Compute two results:
 - 1 written to data memory
 - 1 written to flag
 - Conditional on “context” flag
- Can compute any 2 boolean functions (1 byte each)

CM-2 k-bit add

- Clear flag “c” (carry bit)
- Iterate k times:
 - Read one bit of each operand (2 bits)
 - Read carry bit
 - Compute sum, store to memory
 - Compute carry-out, store to flag
- Last cycle stores carry-out separately (to check for overflow)

CM-2 Router

- Any processor can send a message to any other processor through the router
- (or) The router allows any processor to access any memory location in the machine, in parallel between processors
- Each CM-2 processor chip (16 processors) contains one router node
- Network is a 12-cube
 - Router node i is connected to router node j if $|i-j| = 2^k$

CM-2 Specialized Transfer

- Virtual processors on the same physical processor don't have to use the network at all
- 16 physical processors per chip—communication doesn't have to leave the chip
- Regular communication patterns (like nearest neighbor) avoid router overhead / calculation of destination address
 - Use “NEWS” network

On to the CM-5 ...

- CM-2 was designed for AI apps
- Not many AI labs could afford a \$5M machine
- Instead it was used for (and DARPA was interested in) scientific computing
- Successor, the CM-5, had MIMD organization and commodity microprocessors (Sun SPARC) with special-purpose floating-point and I/O hardware
 - Also cool blinky lights