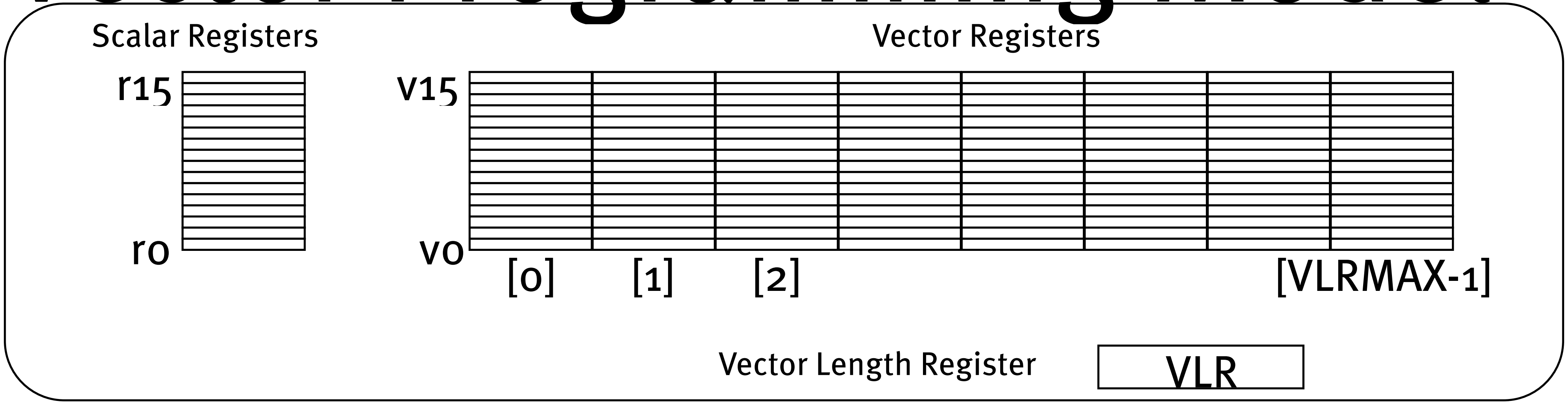
Lecture 16
Data Level Parallelism (2)

EEC 171 Parallel Architectures

John Owens
UC Davis

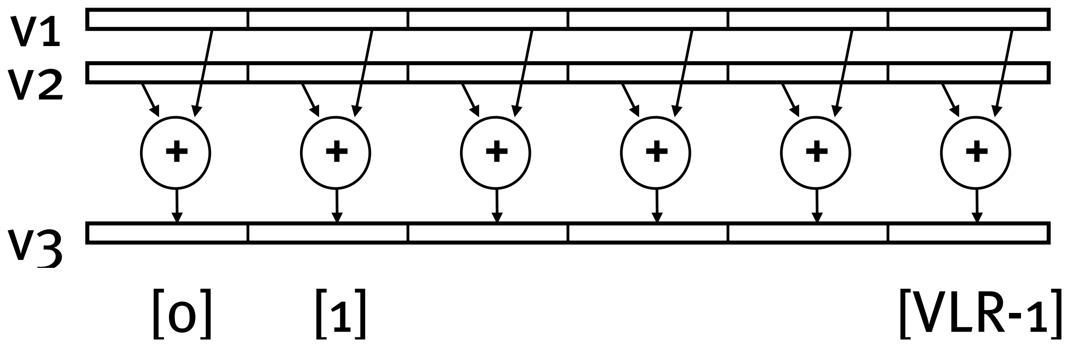
Credits

Vector Programming Model



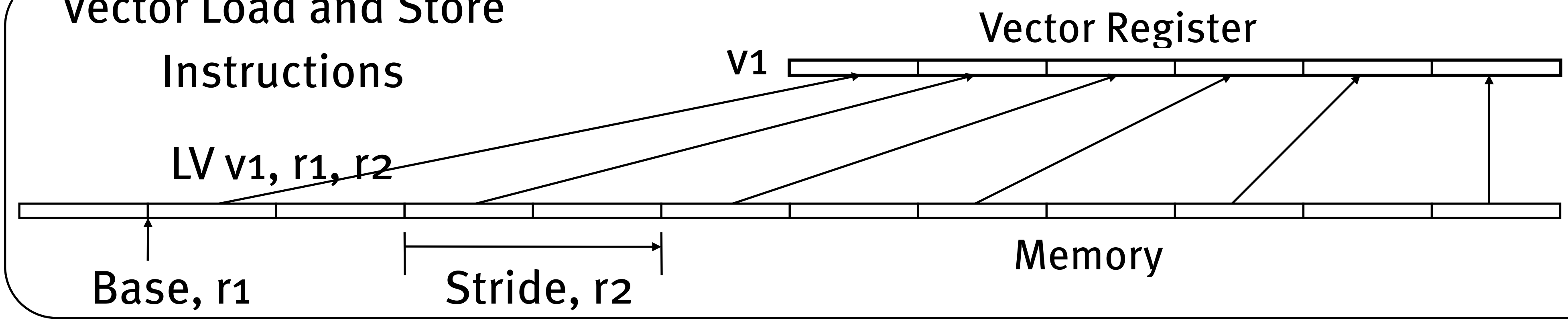
Vector Arithmetic Instructions

ADDV v3, v1, v2



Vector Load and Store Instructions

LV v1, r1, r2



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

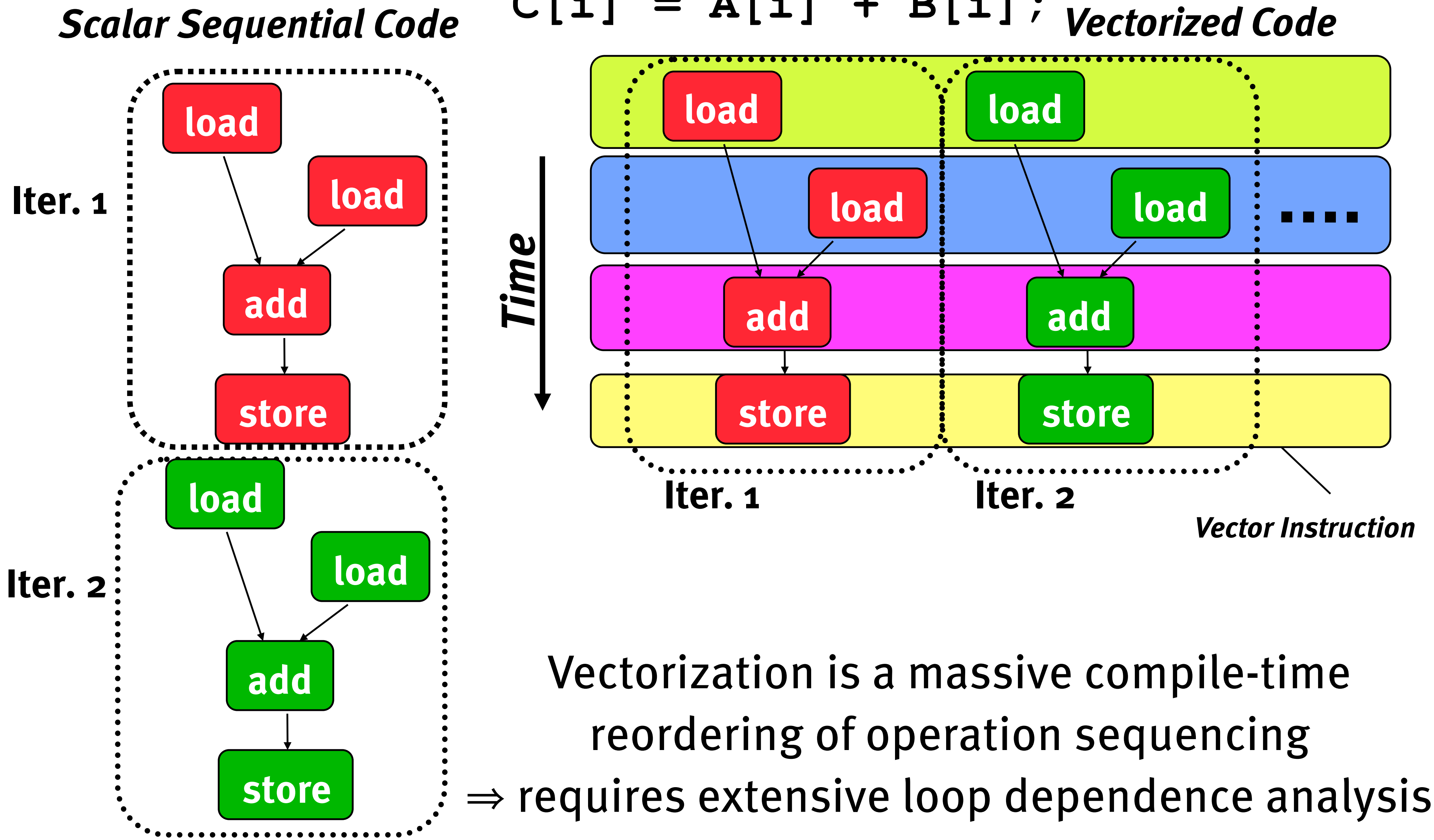
```
LV V1, A
LV V2, B
ADDV V3, V1, V2
SV V3, C
SUBV V4, V1, V2
SV V4, D
```

Vector Memory-Memory vs. Vector Register Machines

- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
- VMMAs make it difficult to overlap execution of multiple vector operations, why?
- VMMAs 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];
```



Vectorization is a massive compile-time reordering of operation sequencing

⇒ requires extensive loop dependence analysis

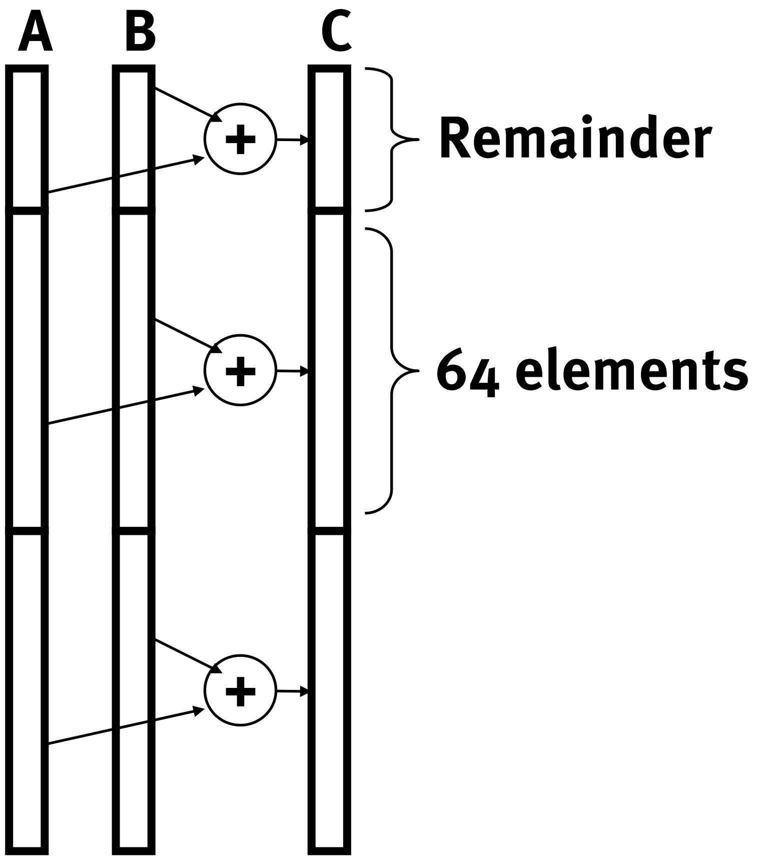
Guy Steele, Dr Dobbs Journal 24 Nov 2005

- "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”

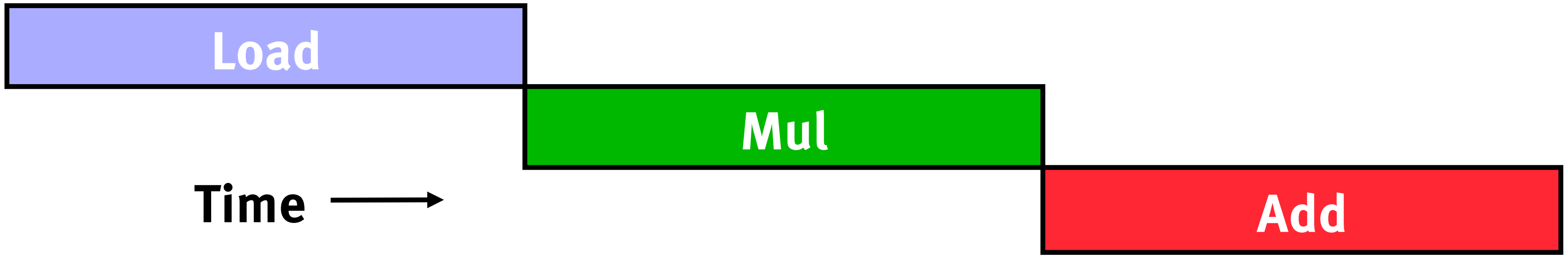
```
for (i=0; i<N; i++)
    C[i] = A[i]+B[i];
```



```
 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

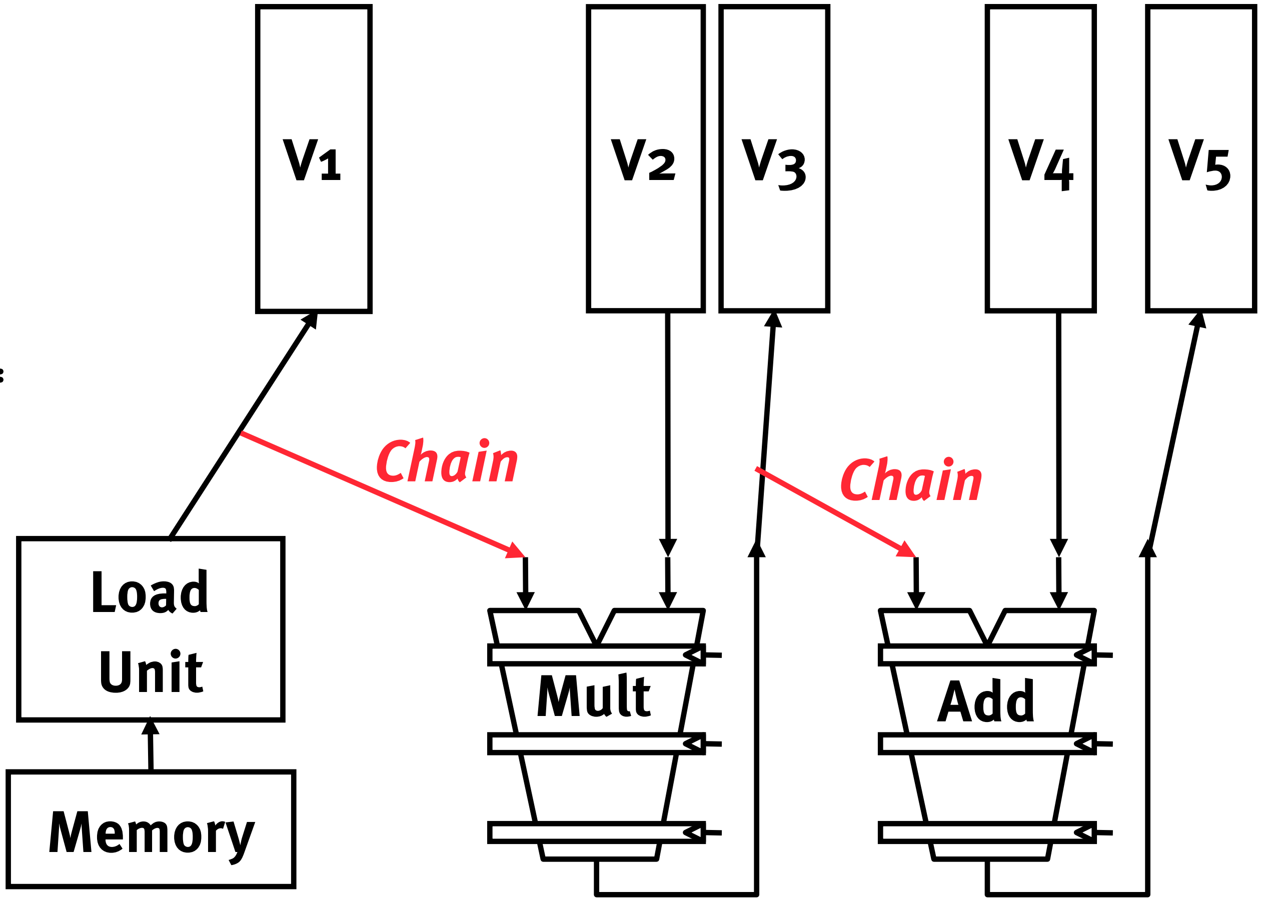
- Must wait for last element of result to be written before starting dependent instruction



Vector Chaining

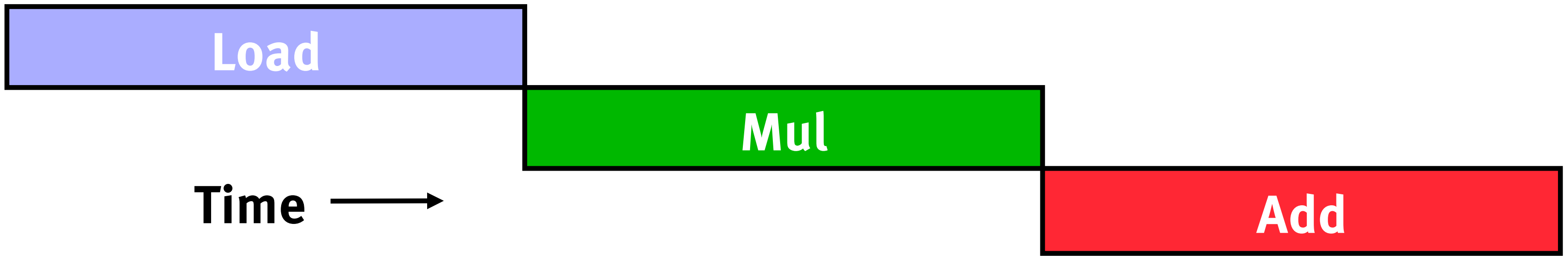
- Vector version of register bypassing
  - introduced with Cray-1

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

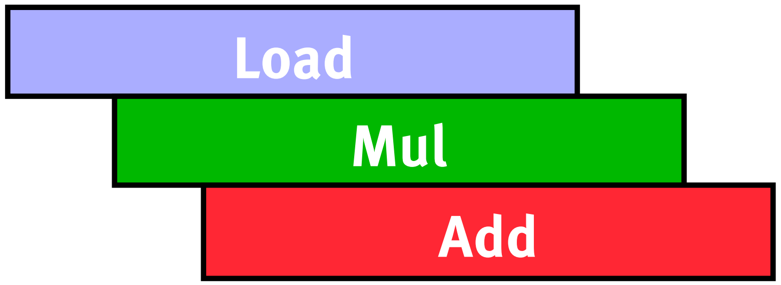


Vector Chaining Advantage

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

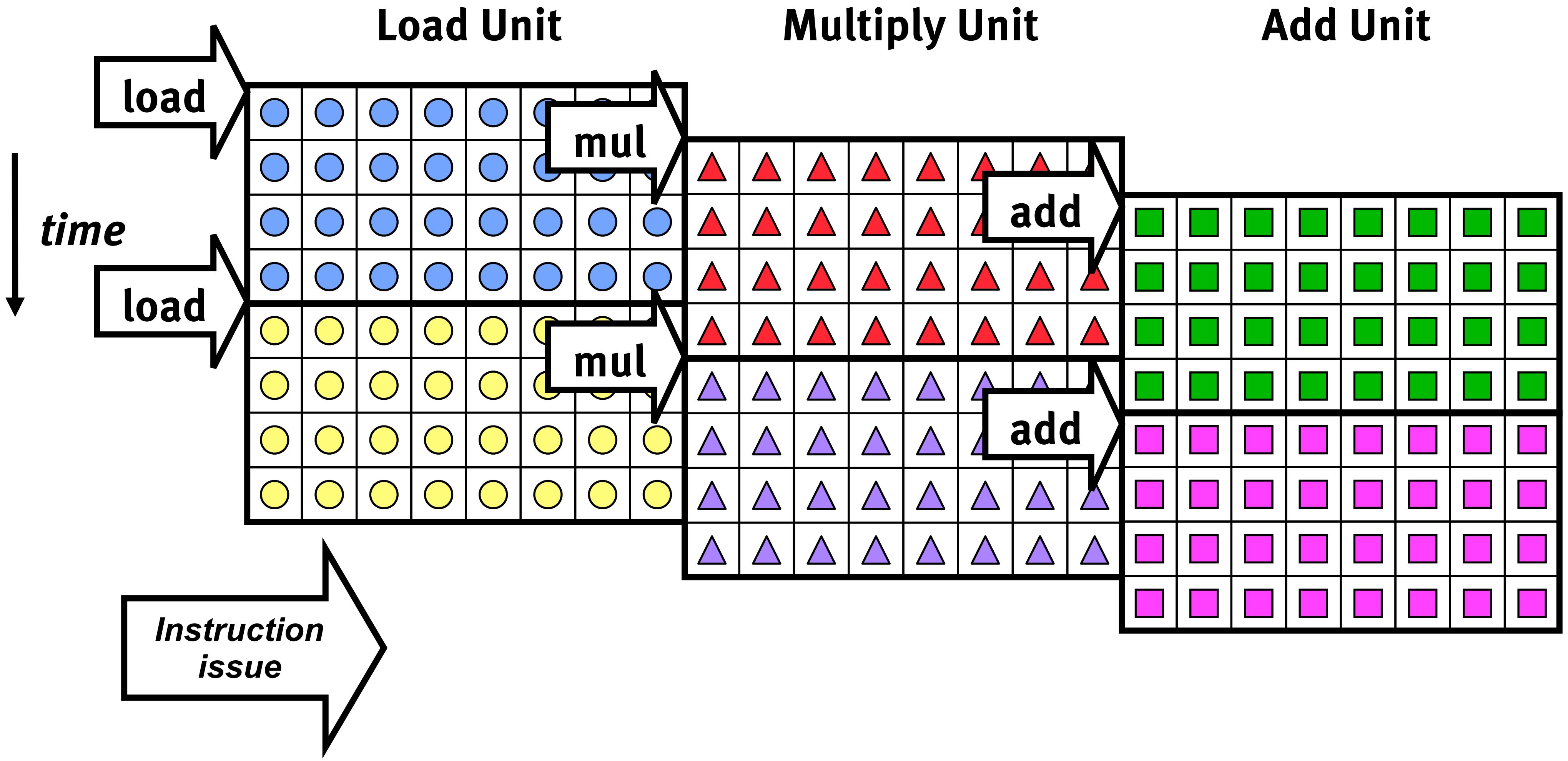


- With chaining, can start dependent instruction as soon as first result appears



Vector Instruction Parallelism

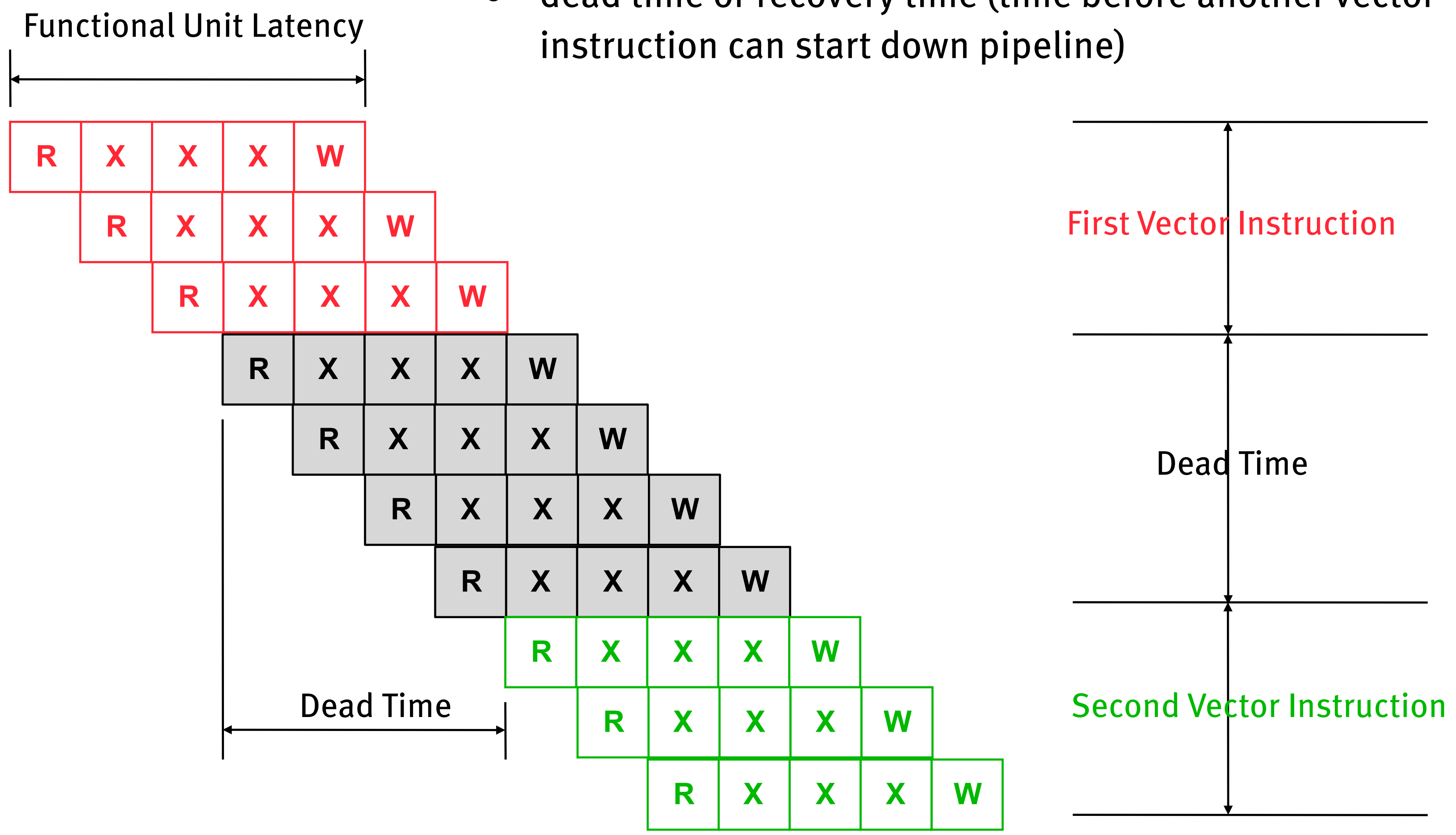
- Can overlap execution of multiple vector instructions
  - example machine has 32 elements per vector register and 8 lanes



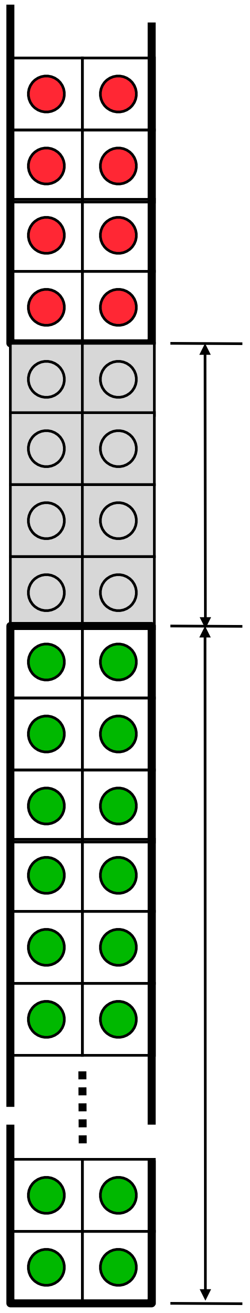
Complete 24 operations/cycle while issuing 1 short instruction/cycle

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)



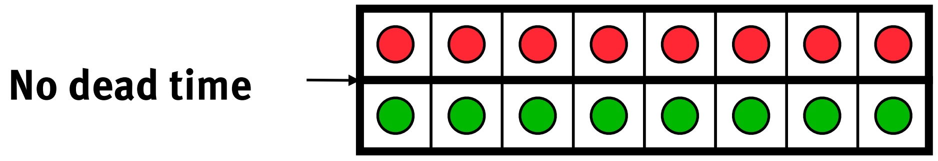
Dead Time and Short Vectors



Cray C90, Two lanes

4 cycle dead time

Maximum efficiency 94% with 128 element vectors



T0, 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):

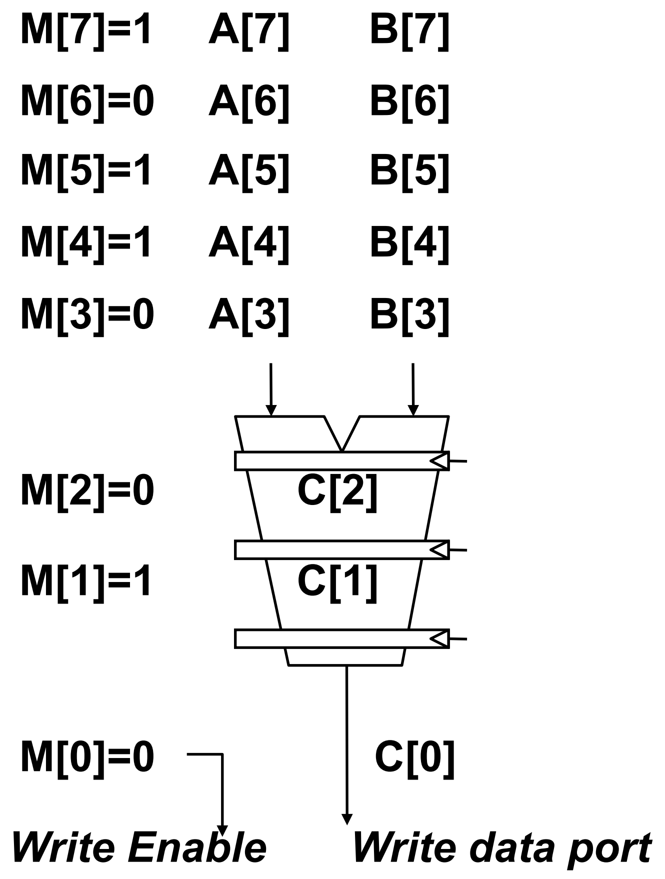
```
CVM              # Turn on all elements
LV vA, rA        # Load entire A vector
SGTVS.D vA, F0   # Set bits in mask register where A>0
LV vA, rB        # Load B vector into A under mask
SV vA, rA        # Store A back to memory under mask
```

This is what NVIDIA hides with its “SIMT” model of execution.

Masked Vector Instructions

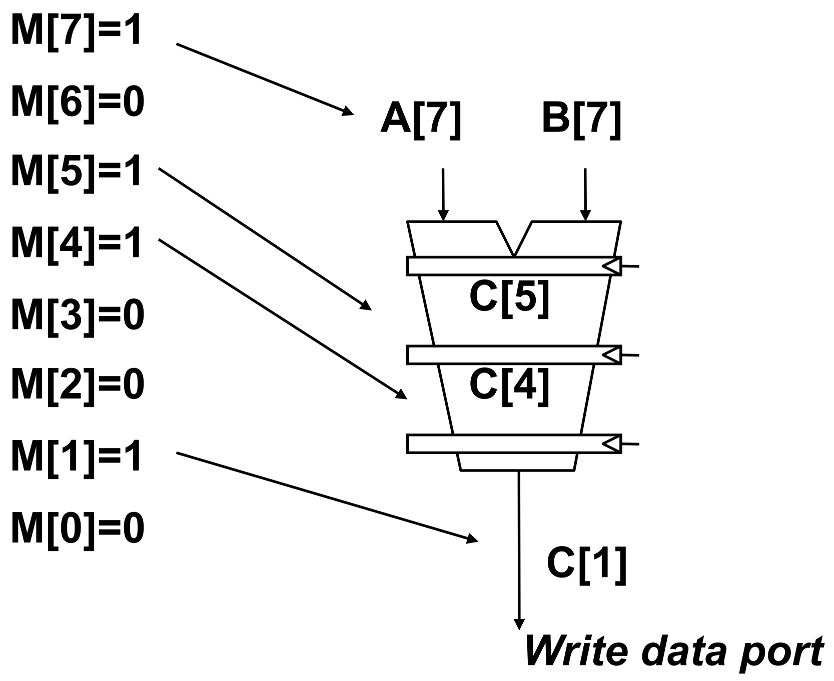
Simple Implementation

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



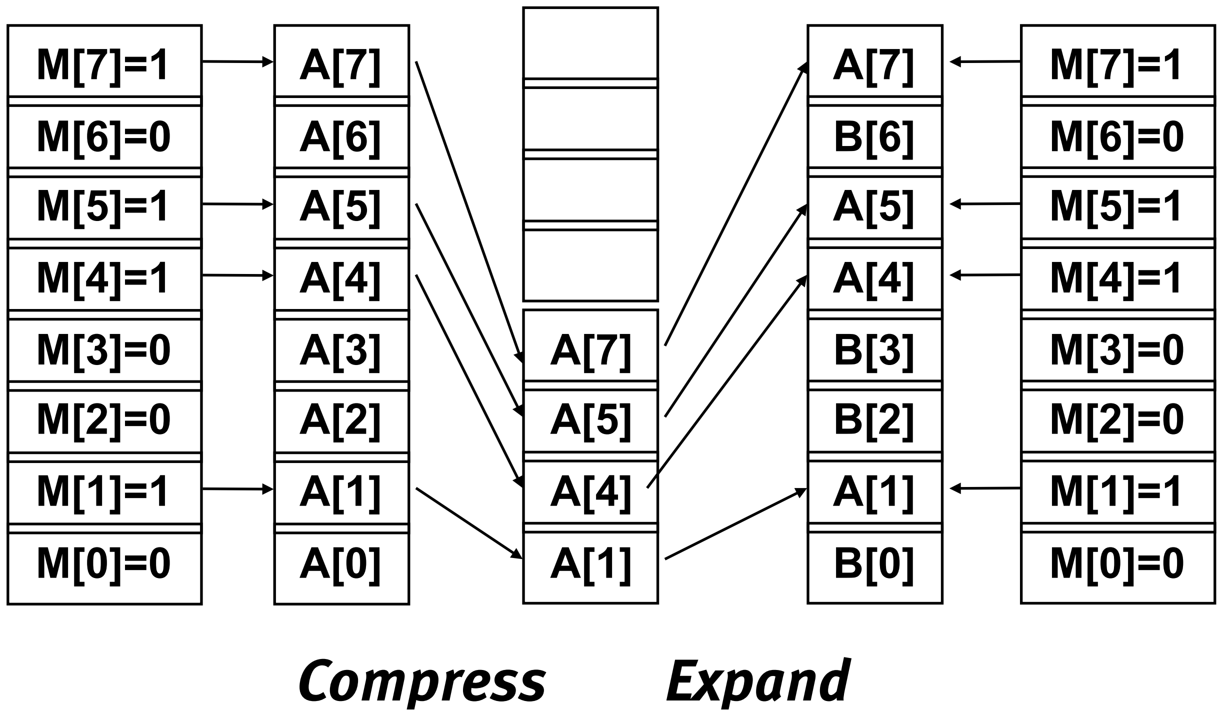
Density-Time Implementation

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



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)

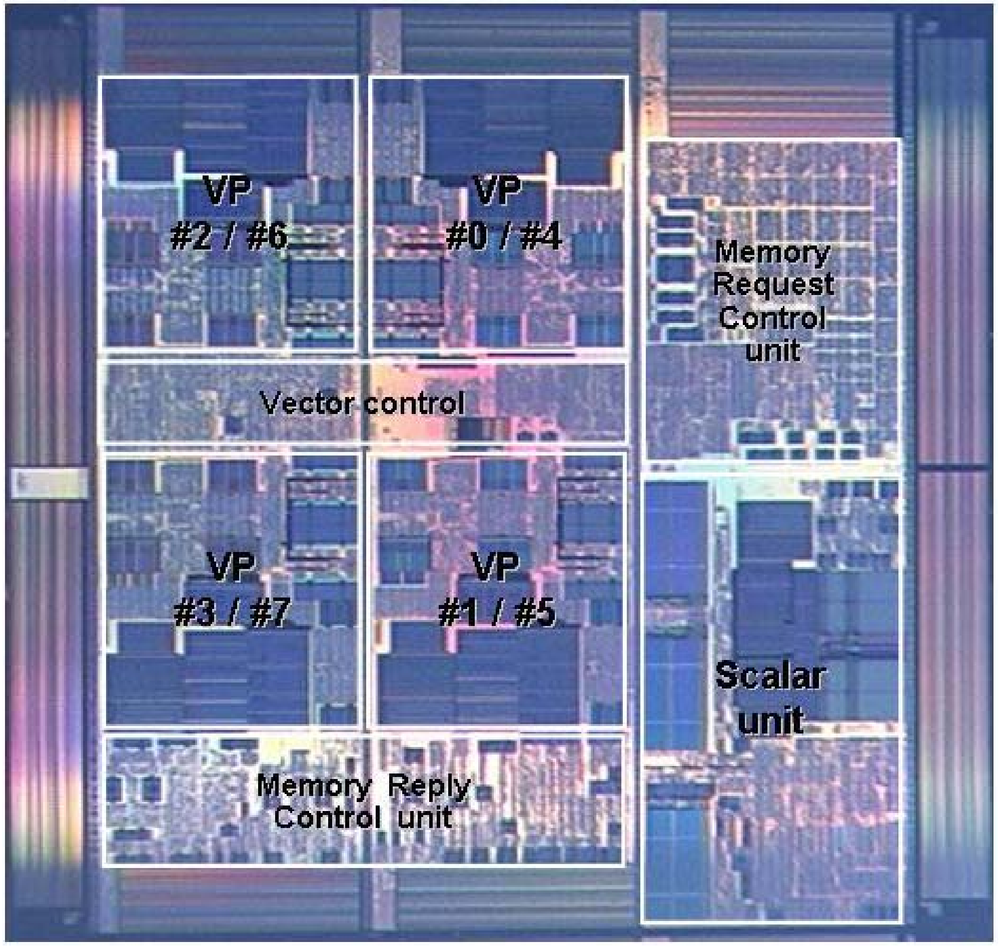
- CMOS Technology
  - 500 MHz CPU, fits on single chip
  - SDRAM main memory (up to 64 GB)
- Scalar unit
  - 4-way superscalar with out-of-order and speculative execution
  - 64 KB I-cache and 64 KB data cache

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

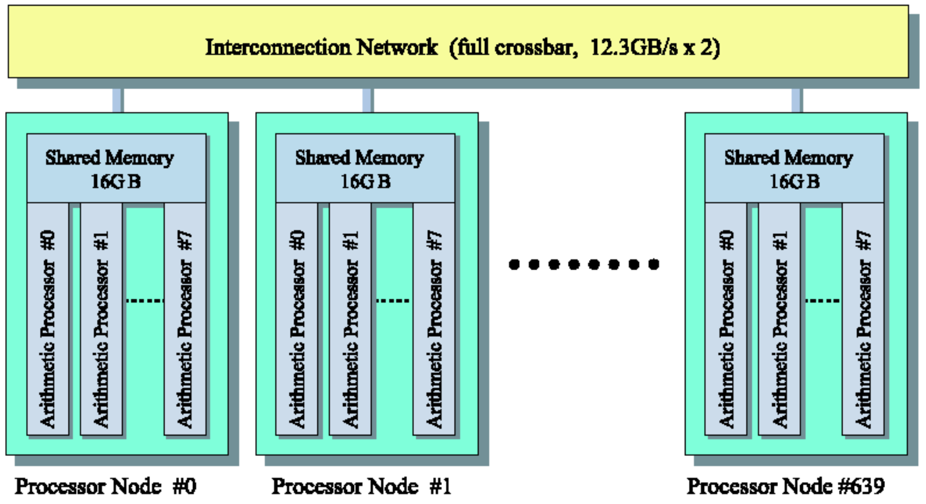
- 0.15 µm CMOS
- 60M transistors
- 432 mm^2
- 500 MHz scalar, 1 GHz vector



Die photo and photos on next page courtesy of Don Alpert

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



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**

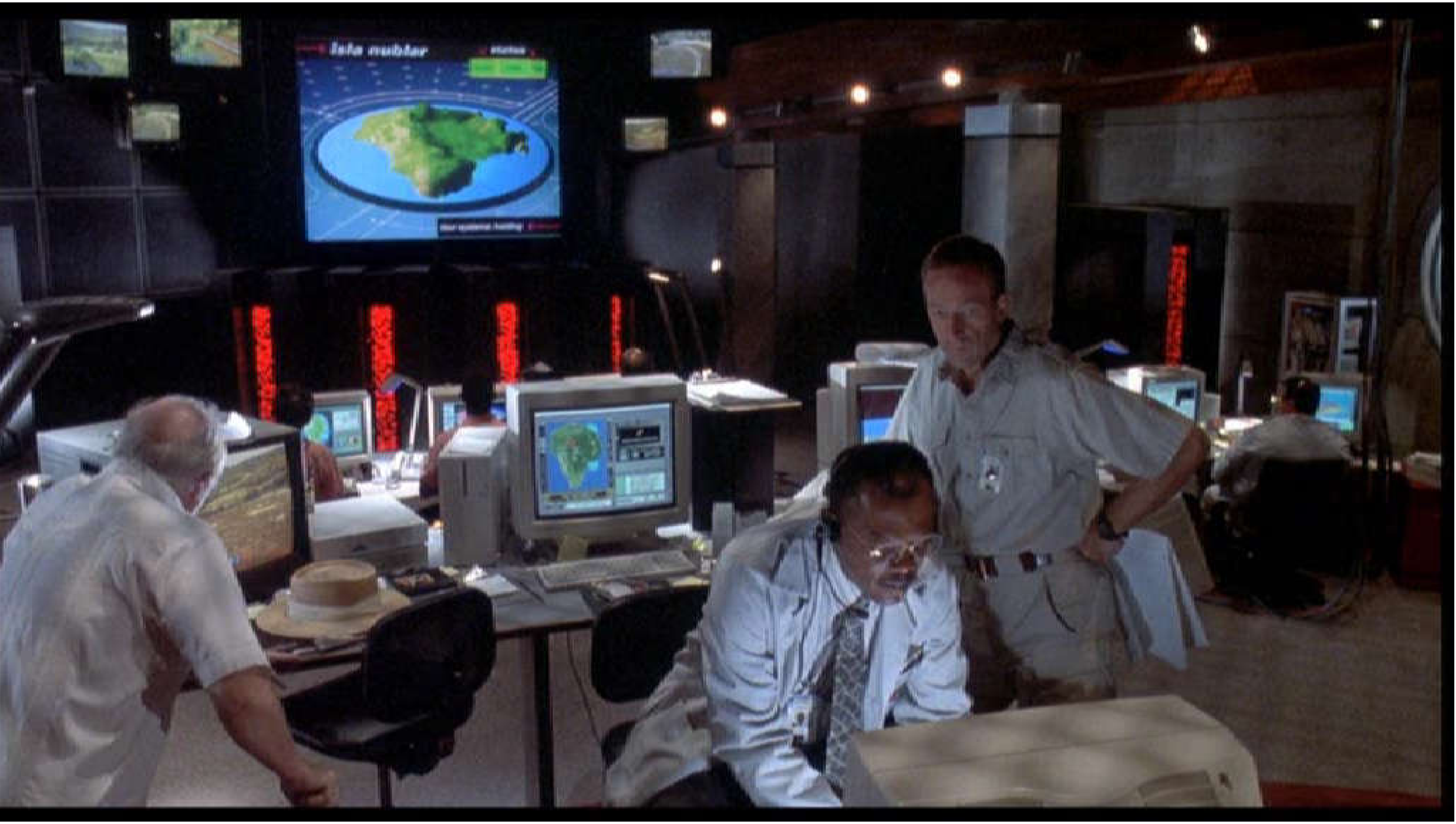
What we've learned

- 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

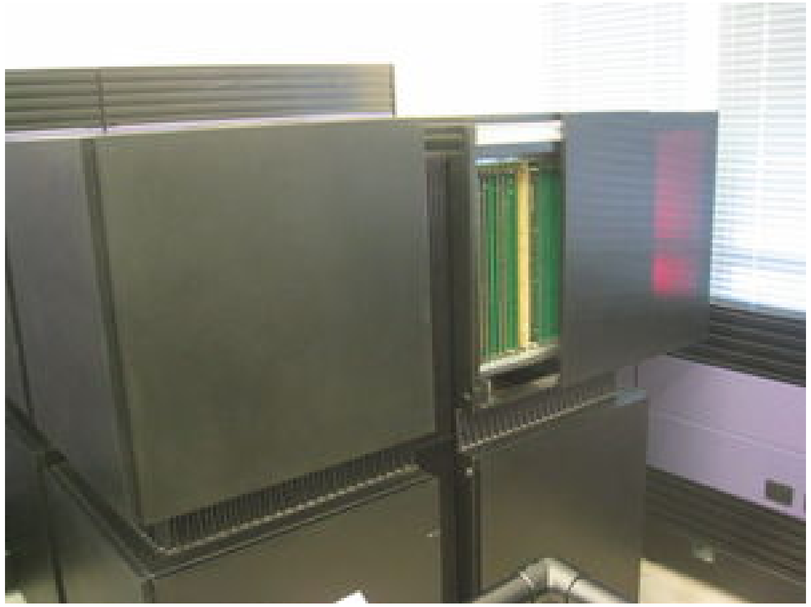
- Massively parallel machines
- Big idea: Write one program, run it on lots of processors
  - Now we’re going to look at hardware
    - Thinking Machines CM-2
  - We already looked at algorithms last week!

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



Danny Hillis …

- Q: What AI advances were made on TMC machines?
- A: “Thanks for the note. Of course, general AI did not make a lot of progress. The machine was used for a lot of neural network modeling. The machine was also used for a lot for computer vision, for example by Poggio. There was real progress in both these areas. Semantic networks also made some progress. (Lenat started the first version of his Cyc project at Thinking Machines.) Also one of the first internet search engines (WAIS) was built on it. And some of the first real applications of genetic algorithms.”

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

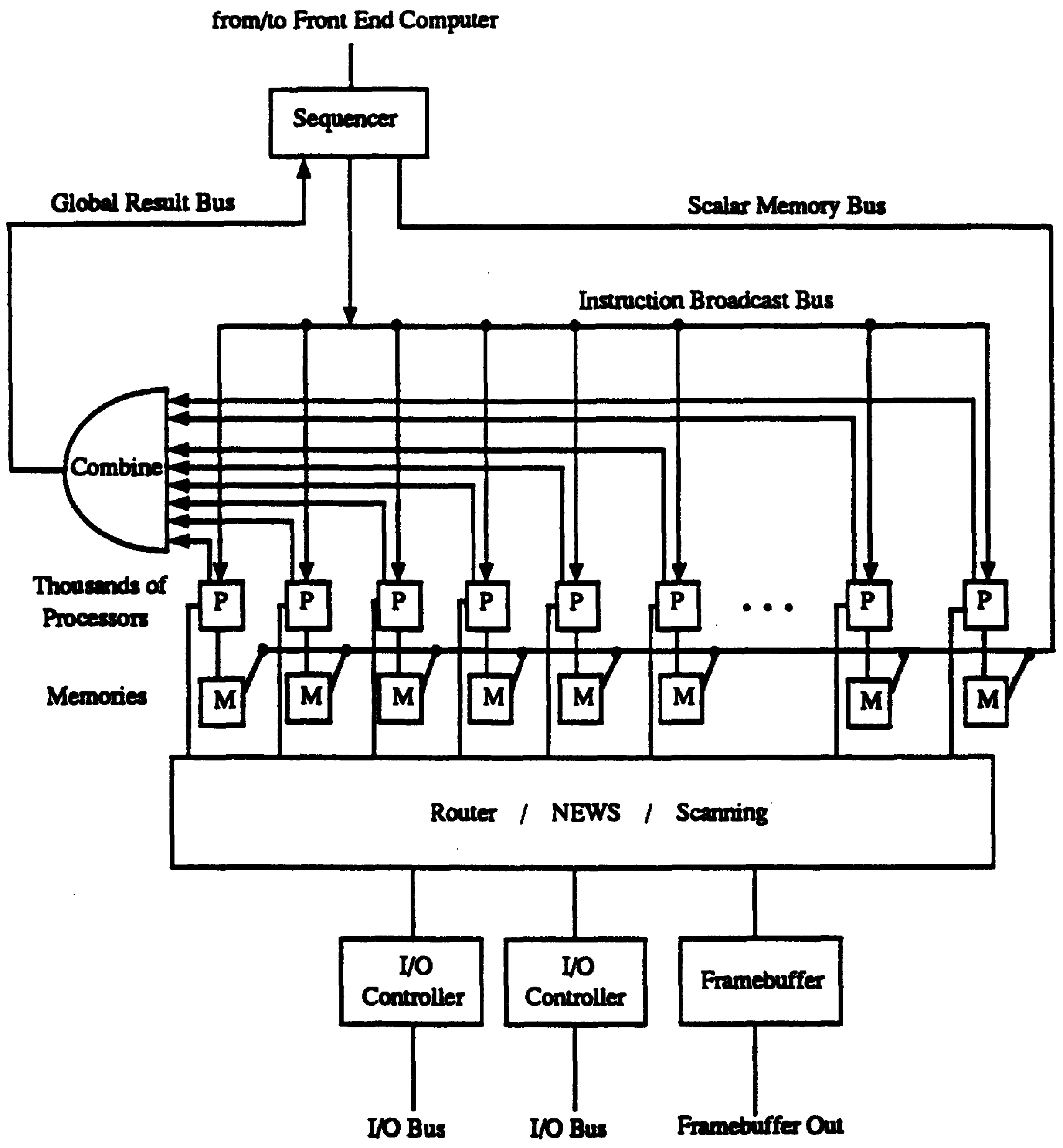
Communication Primitives

- 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

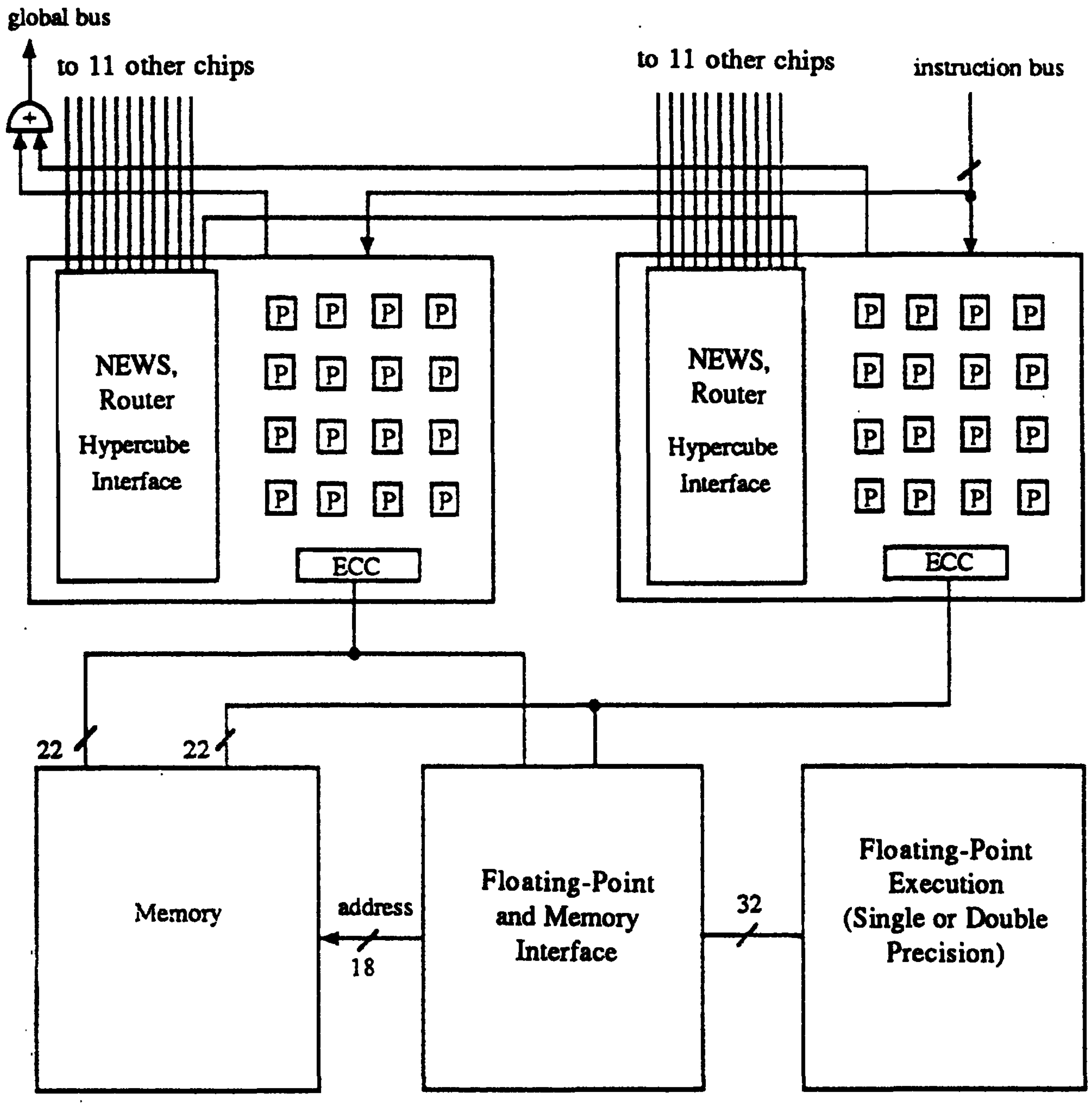
Computation + Communication Primitives

- 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


global bus
to 11 other chips
to 11 other chips
instruction bus
NEWS, Router Hypercube Interface
P
ECC
NEWS, Router Hypercube Interface
ECC
22
22
Memory
address
18
Floating-Point and Memory Interface
32
Floating-Point Execution (Single or Double Precision)

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

ispc: A SPMD Compiler for High-Performance CPU Programming

Matt Pharr and William R. Mark
Intel

14 May 2012

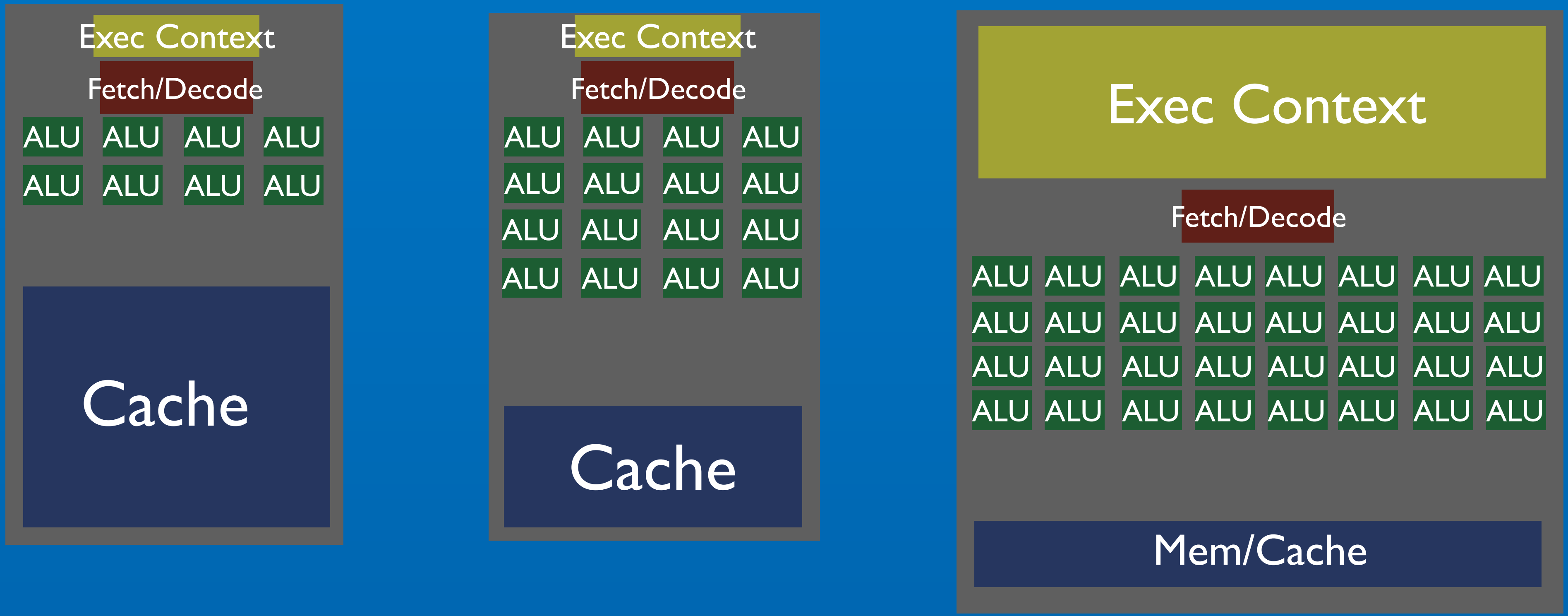
http://ispc.github.com

Motivation: 3 Modern Parallel Architectures

CPU:
2-10x

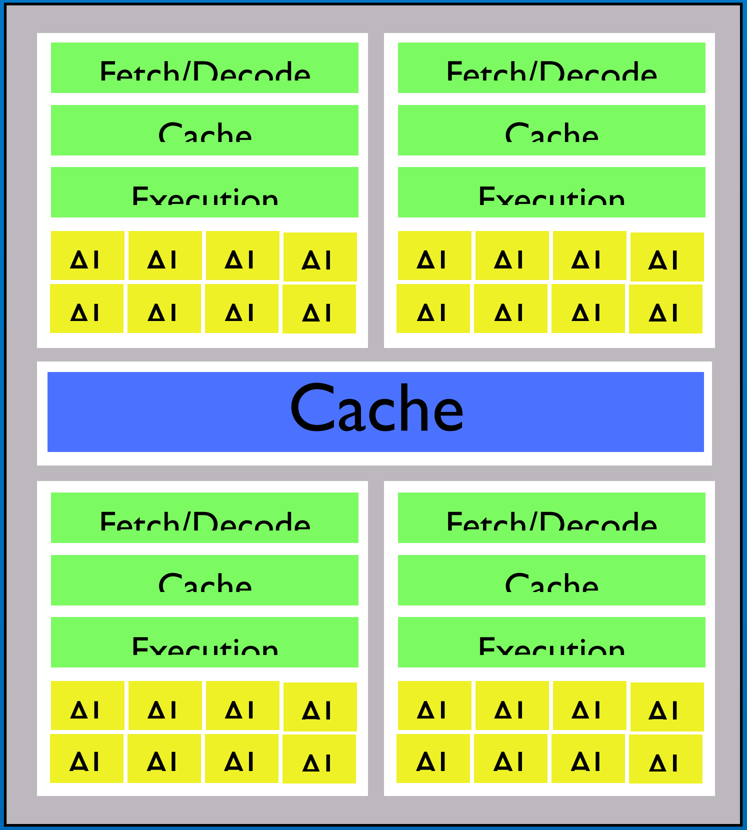
MIC:
50+x

GPU:
2-32x



Filling the Machine (CPU and GPU)

- *Task parallelism* across cores: run different programs (if wanted) on different cores
- *Data-parallelism* across SIMD lanes in a single core: run the same program on different input values



ispc: Key Features

- “SPMD on SIMD” on modern CPUs (coupled with task parallelism)
- Ease of adoption and integration
    - C syntax and feature set, single coherent address space
- Performance transparency
- Scalability (cores * SIMD width)

SPMD 101

- Run the same program concurrently with different inputs
  - Inputs = array/matrix elements, particles, pixels, ...

```
float func(float a, float b) {
    if (a < 0.) a = 0.;
    return a + b;
}
```

- The contract:
  Programmer guarantees independence across program instances; Compiler is free to run those instances in parallel

SPMD On A GPU SIMD Unit

~PTX

```
a = b + c;         fadd
if (a < 0)     cmp, jge l_a
    ++b;      fadd, jmp l_b
else               l_a:
    ++c;           fadd
                   l_b:
```

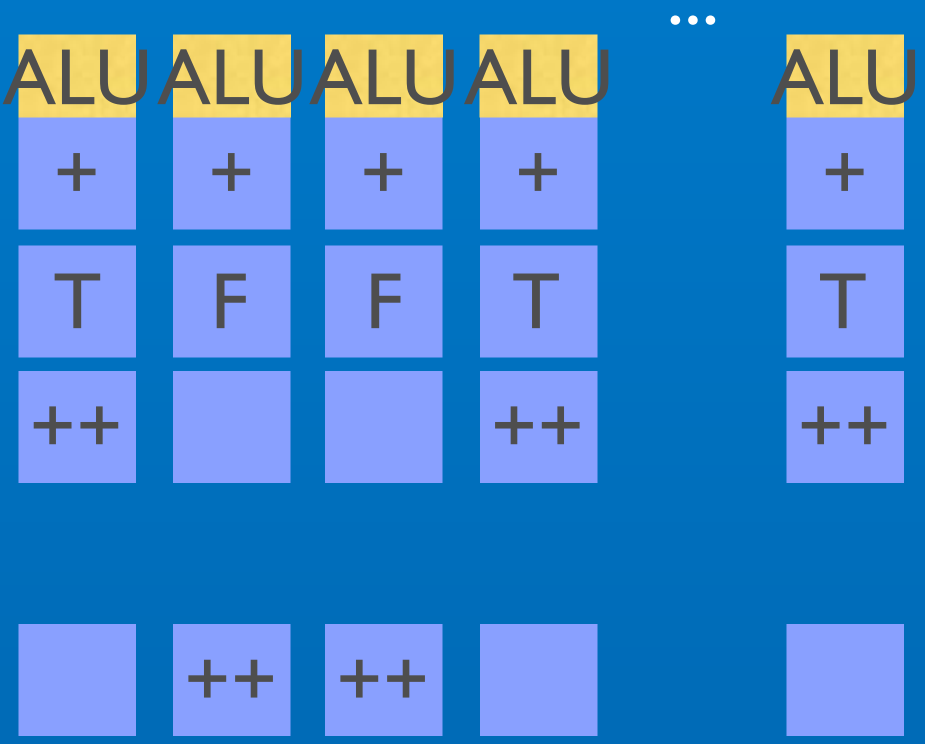
(Based on http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf)

SPMD On A GPU SIMD Unit

```
a = b + c;
if (a < 0)
      ++b;
else
      ++c;
```

~PTX

```
fadd
cmp, jge l_a
fadd, jmp l_b
l_a:
fadd
l_b:
```

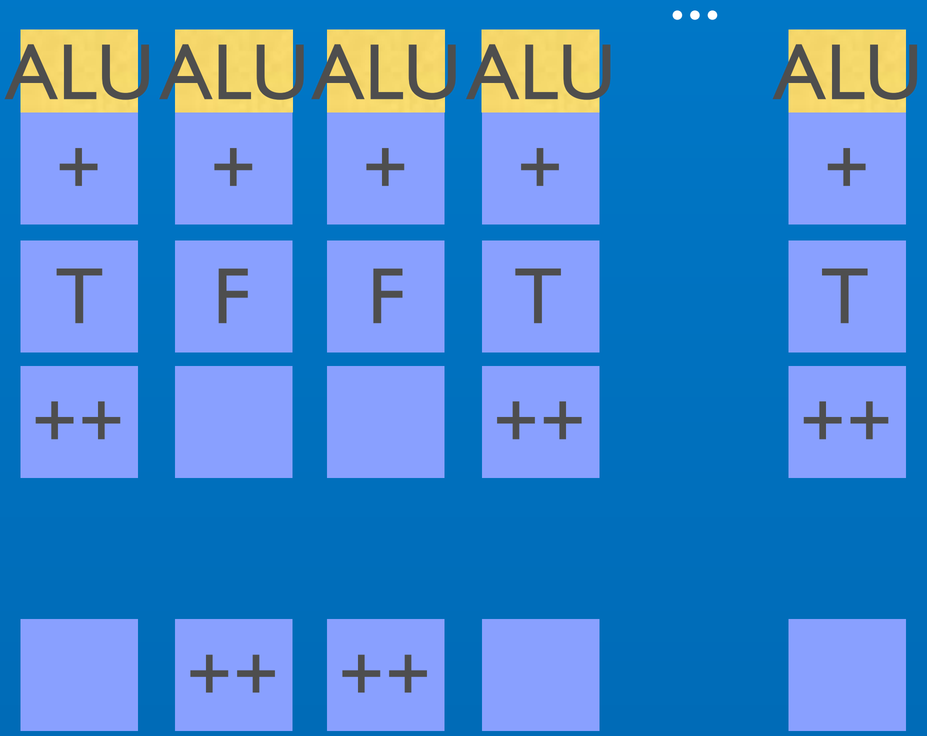


(Based on http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf)

SPMD On A CPU SIMD Unit

AVX

```
a = b + c;       vaddps
if (a < 0)       vcmpltps
    ++b;         vaddps, vblendvps
else
    ++c;         vaddps, vblendvps
```



(Based on http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf)

SPMD on SIMD Execution

Transform control-flow to data-flow

```
if (test) {
    true stmts;
}
else {
    false stmts;
}
```

```
old_mask = current_mask
test_mask = evaluate test
current_mask &= test_mask
// emit true stmts, predicate with current_mask
current_mask = old_mask & ~test_mask
// emit false stmts, predicate with current_mask
current_mask = old_mask
```

[Allen et al. 1983, Karrenberg and Hack 2011]

SPMD On SIMD in ispc

- Map *program instances* to individual lanes of the SIMD unit
  - e.g. 8 instances on 8-wide AVX SIMD unit
- A *gang* of program instances runs concurrently
  - One gang per hardware thread / execution context