

Lecture 13 (part 2)

Data Level Parallelism (1)

EEC 171 Parallel Architectures

John Owens

UC Davis

Credits

- © John Owens / UC Davis 2007–17.
- 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

- SIMD instructions and instruction sets
- Vector machines (Cray 1)
- Vector complexities
- Massively parallel machines
(Thinking Machines CM-2)

Cook analogy

- We want to prepare food for several banquets, each of which requires many dinners.
- We have two positions we can fill:
 - The boss (control), who gets all the ingredients and tells the chef what to do
 - The chef (datapath), who does all the cooking

Cook analogy

- ILP is analogous to:
 - One ultra-talented boss with many hands
 - One ultra-talented chef with many hands
- TLP is analogous to:
 - Building multiple kitchens, each with a boss and a chef
 - Putting more bosses and chefs in the same kitchen

Cook analogy

- DLP is analogous to:
 - One boss
 - Lots of cloned chefs

Flynn's Classification Scheme



- SISD – single instruction, single data stream
 - Uniprocessors
- SIMD – single instruction, multiple data streams
 - single control unit broadcasting operations to multiple datapaths
- MISD – multiple instruction, single data
 - no such machine (although some people put vector machines in this category)
- MIMD – multiple instructions, multiple data streams
 - aka multiprocessors (SMPs, MPPs, clusters, NOWs)

Continuum of Granularity

- “Coarse”

- Each processor is more powerful
- Usually fewer processors
- Communication is more expensive between processors
- Processors are more loosely coupled
- Tend toward MIMD

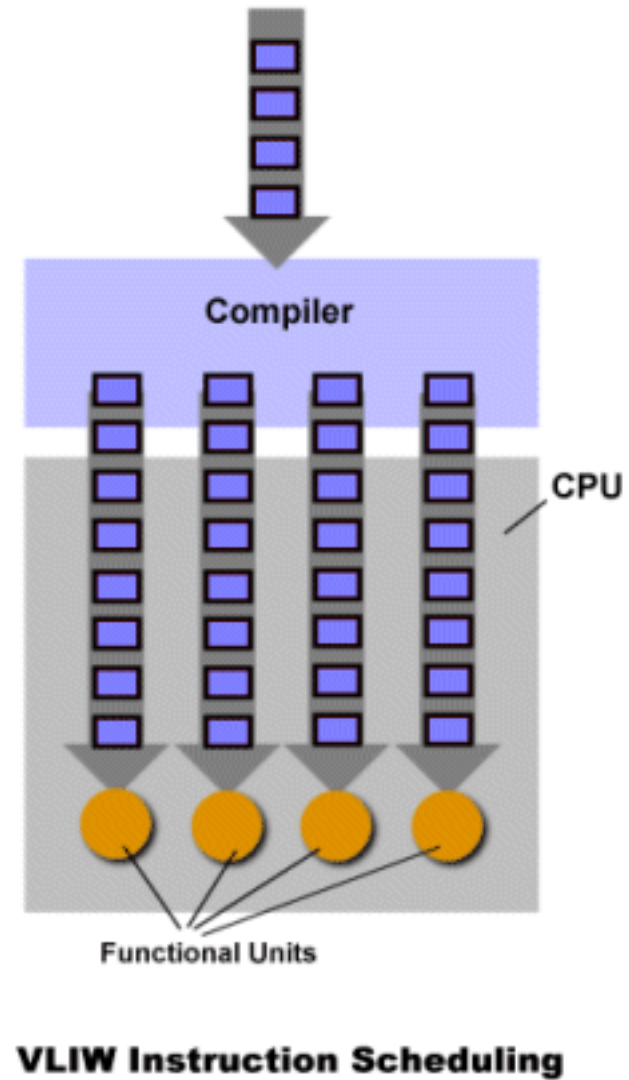
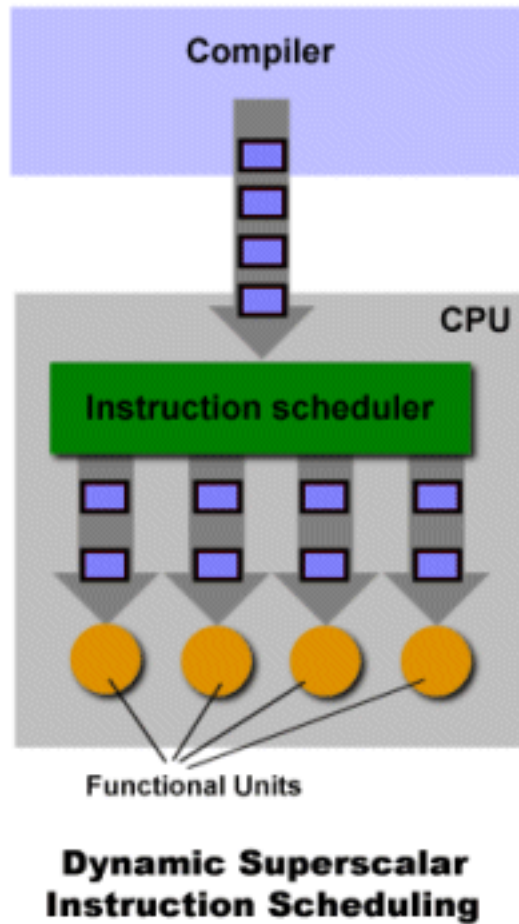
- “Fine”

- Each processor is less powerful
- Usually more processors
- Communication is cheaper between processors
- Processors are more tightly coupled
- Tend toward SIMD

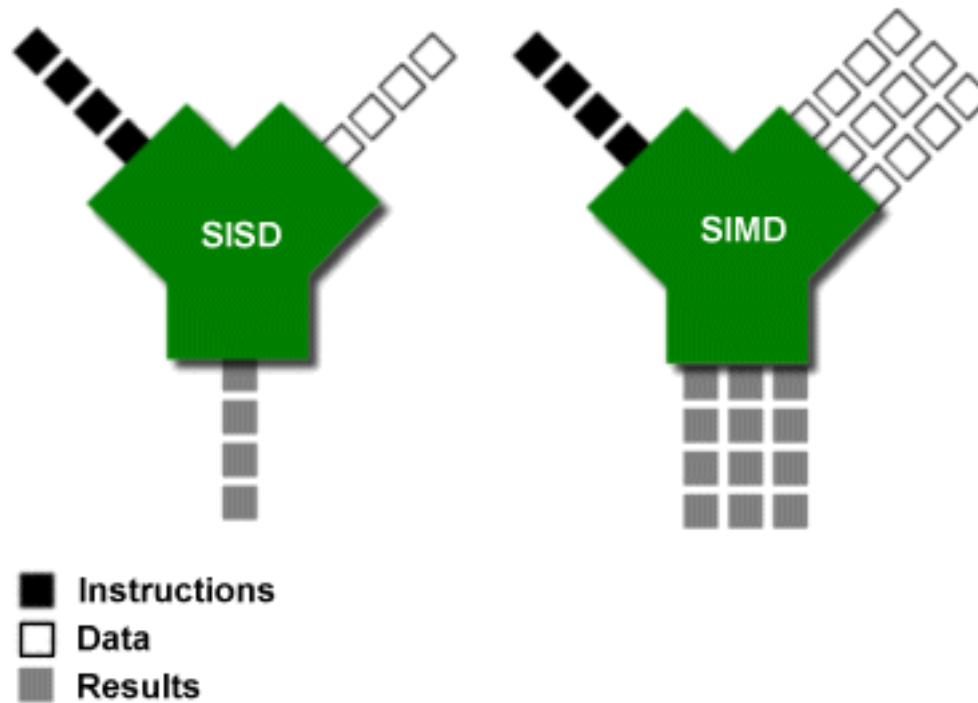
ILP vs. DLP

- “SIMD is about exploiting parallelism in the data stream, while superscalar SISD is about exploiting parallelism in the instruction stream.”

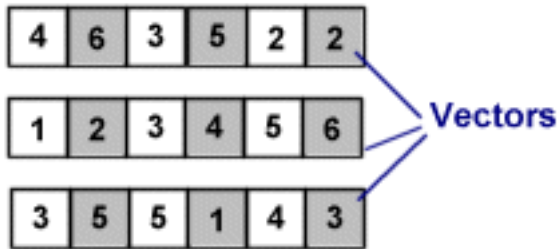
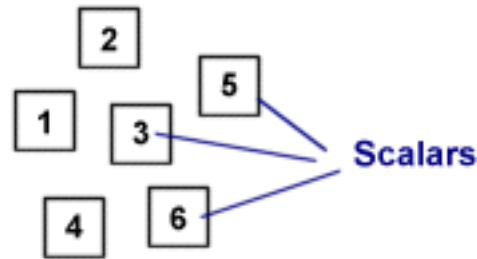
What We Know



What's New



Scalar vs. Vector



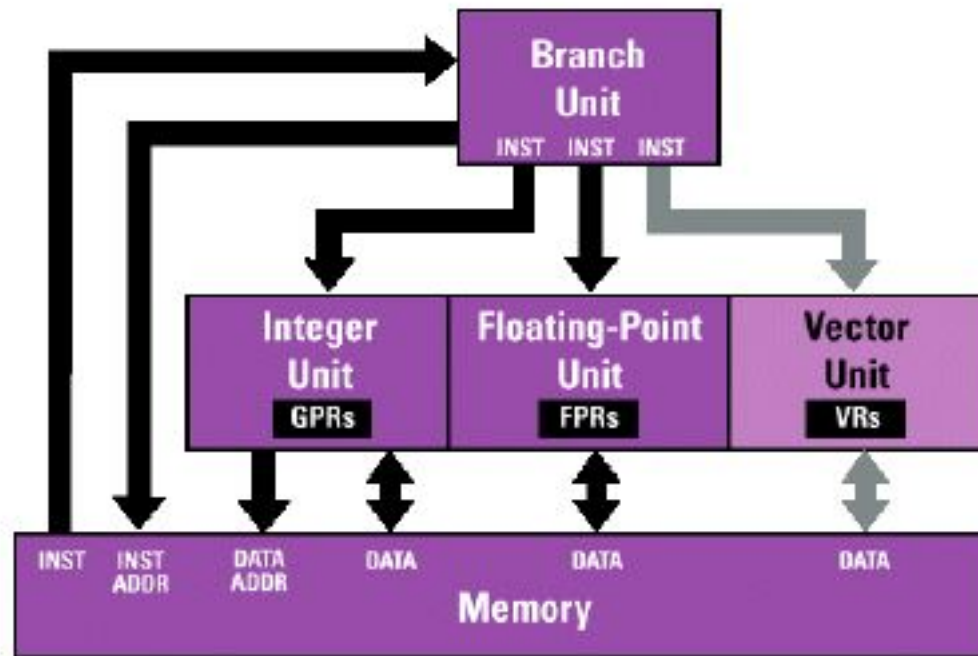
- “The basic unit of SIMD love is the vector, which is why SIMD computing is also known as vector processing. A vector is nothing more than a row of individual numbers, or scalars.”

Representing Vectors

- Multiple items within same data word
- Multiple data words

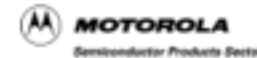
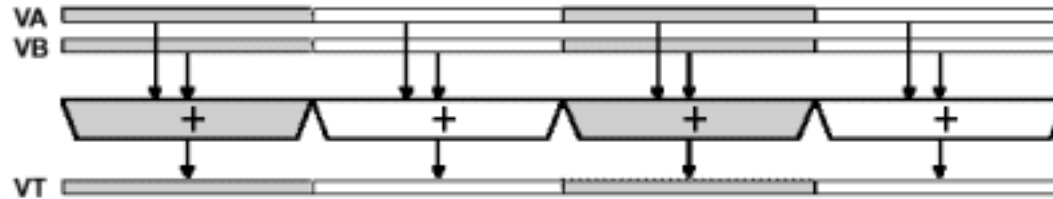
Motorola AltiVec

High-level structural overview for
PowerPC with AltiVec technology



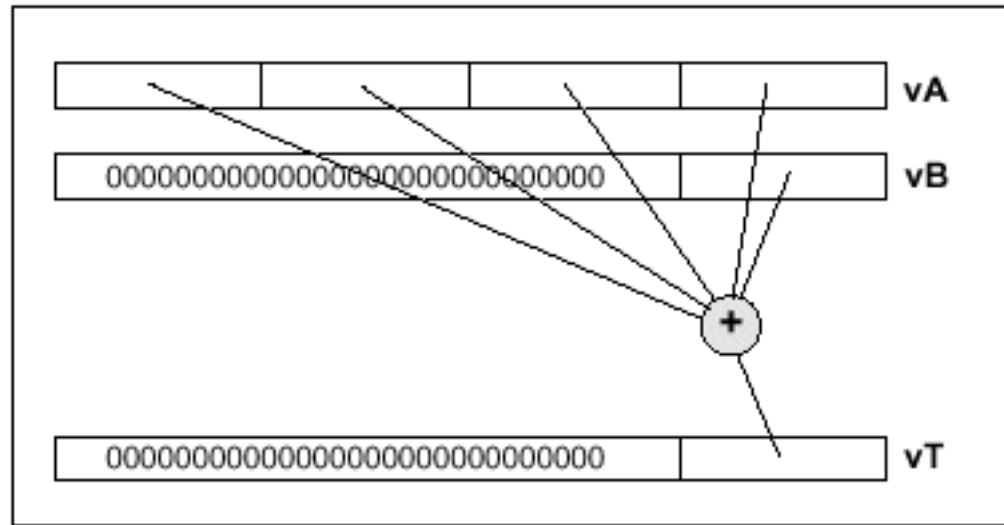
Motorola AltiVec

4 x 32-bit elements



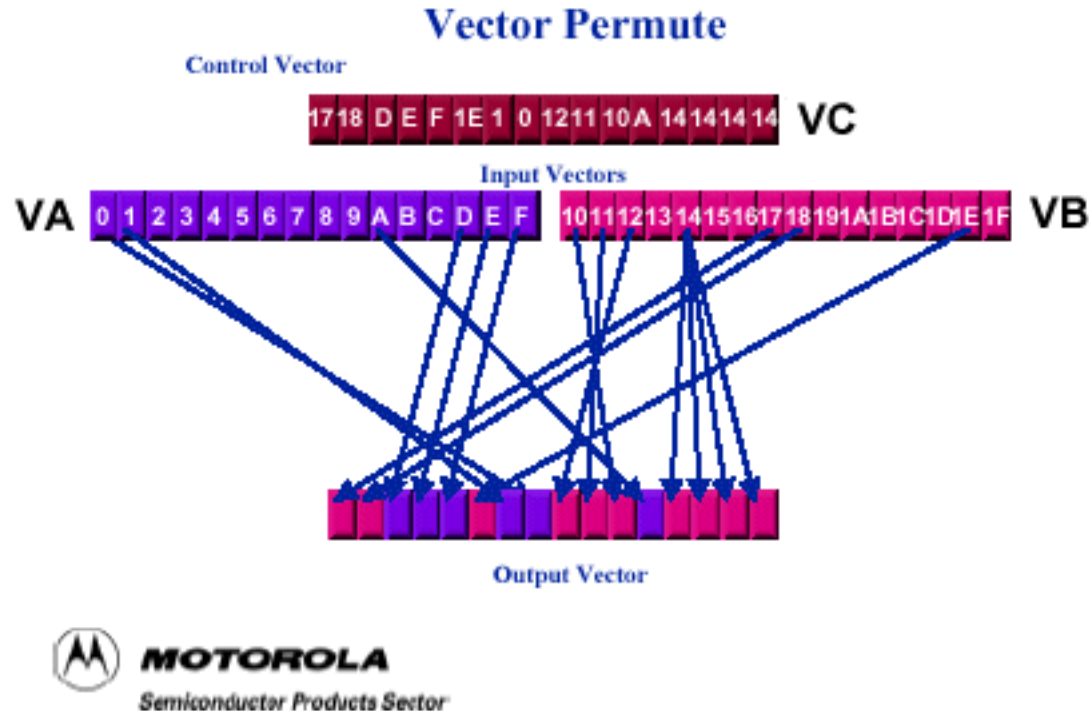
- Intra element arithmetic and non-arithmetic functions.
 - integer instructions
 - integer arithmetic instructions
 - integer compare instructions
 - integer rotate and shift instructions
 - floating-point instructions
- floating-point arithmetic instructions
- floating-point rounding and conversion instructions
- floating-point compare instruction
- floating-point estimate instructions
- memory access instructions

Motorola AltiVec



- Inter Element Arithmetic: Between elements in vector
- Example: Sum elements across vector, store in accumulator
- alignment support instructions
- permutation and formatting instructions
- pack instructions
- unpack instructions
- merge instructions
- splat instructions
- permute instructions
- shift left/right instructions

Motorola AltiVec



- Inter Element Non-Arithmetic (vector permute)

AltiVec Architecture

- 32 128b-wide new architectural registers
 - 16 8b elements, 8 16b elements, 4 32b (FP/int) elements
- 2 fully-pipelined, parallel AltiVec units:
 - Vector Permute Unit (pack, unpack, permute, load/store)
 - Vector ALU (all math)
 - Latency: 1 cycle for simple instrs, 3–4 for complex
 - No interrupts except load and store

MMX Instructions

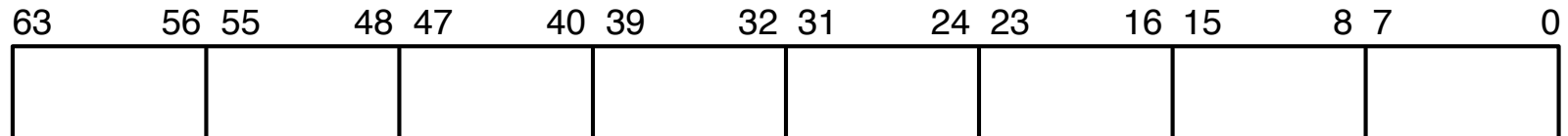
2 x 32-bit elements



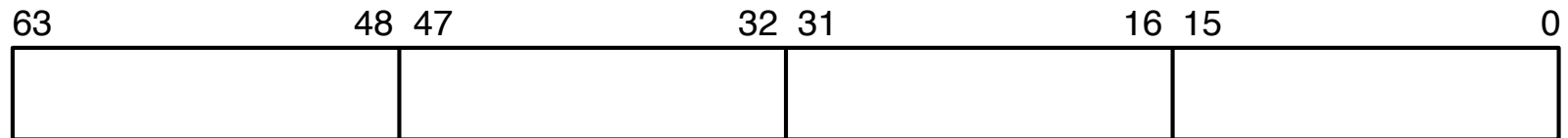
- 57 new instructions
- 2 operands per instruction (like x86)
 - No single-cycle permutes

Intel MMX Datatypes

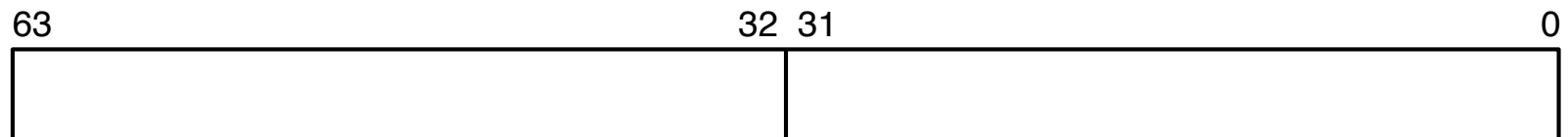
Packed bytes (8x8 bits)



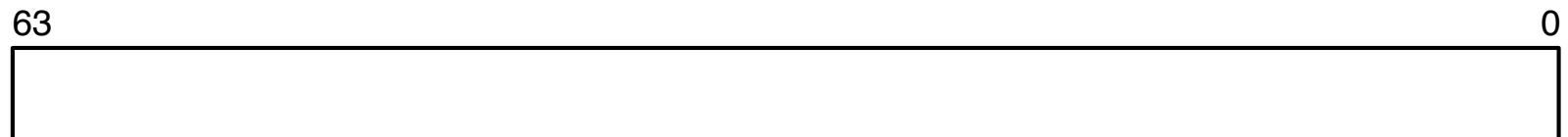
Packed word (4x16 bits)



Packed doublewords (2x32 bits)



Quadword (64 bits)



3006002

Figure 8-2. MMX™ Data Types

Intel MMX

Table 8-2. MMX™ Instruction Set Summary

Category		Wraparound	Signed Saturation		Unsigned Saturation
Arithmetic	Addition	PADDB, PADDW, PADDD	PADDSB, PADDSW		PADDUSB, PADDUSW
	Subtraction	PSUBB, PSUBW, PSUBD	PSUBSB, PSUBSW		PSUBUSB, PSUBUSW
	Multiplication	PMULL, PMULH			
	Multiply and Add	PMADD			
Comparison	Compare for Equal	PCMPEQB, PCMPEQW, PCMPEQD			
	Compare for Greater Than	PCMPGTPB, PCMPGTPW, PCMPGTPD			
Conversion	Pack		PACKSSWB, PACKSSDW		PACKUSWB
	Unpack High	PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ			
	Unpack Low	PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ			
Logical	And And Not Or Exclusive OR	Packed		Full Quadword	
				PAND PANDN POR PXOR	
Shift	Shift Left Logical Shift Right Logical Shift Right Arithmetic	PSLLW, PSLLD PSRLW, PSRLD PSRAW, PSRAD		PSLLQ PSRLQ	
Data Transfer	Register to Register Load from Memory Store to Memory	Doubleword Transfers		Quadword Transfers	
		MOVD MOVD MOVD		MOVQ MOVQ MOVQ	
Empty MMX™ State		EMMS			

MMX Details

- 8 MMX registers (64b each)
- Aliased with FP registers
 - “EMMS” (exit MMX) switches between them
 - Why is this a good idea?
 - Why is this a bad idea?
- Supports saturating arithmetic
- Programmed with intrinsics

MMX Example

```
; DWORD LerpARGB(DWORD a, DWORD b, DWORD f);  
global _LerpARGB
```

```
_LerpARGB:
```

```
; load the pixels and expand to 4 words  
movd      mm1, [esp+4]      ; mm1 = 0 0 0 0 aA aR aG aB  
movd      mm2, [esp+8]      ; mm2 = 0 0 0 0 bA bR bG bB  
pxor      mm5, mm5          ; mm5 = 0 0 0 0 0 0 0 0  
punpcklbw mm1, mm5          ; mm1 = 0 aA 0 aR 0 aG 0 aB  
punpcklbw mm2, mm5          ; mm2 = 0 bA 0 bR 0 bG 0 bB  
  
; load the factor and increase range to [0-256]  
movd      mm3, [esp+12]     ; mm3 = 0 0 0 0 faA faR faG faB  
punpcklbw mm3, mm5          ; mm3 = 0 faA 0 faR 0 faG 0 faB  
movq      mm6, mm3          ; mm6 = faA faR faG faB [0 - 255]  
psrlw     mm6, 7             ; mm6 = faA faR faG faB [0 - 1]  
paddw     mm3, mm6          ; mm3 = faA faR faG faB [0 - 256]  
  
; fb = 256 - fa  
pcmpeqw   mm4, mm4          ; mm4 = 0xFFFF 0xFFFF 0xFFFF 0xFFFF  
psrlw     mm4, 15            ; mm4 = 1 1 1 1  
psllw     mm4, 8             ; mm4 = 256 256 256 256  
psubw     mm4, mm3          ; mm4 = fbA fbR fbG fbB  
  
; res = (a*fa + b*fb)/256  
pmullw    mm1, mm3          ; mm1 = aA aR aG aB  
pmullw    mm2, mm4          ; mm2 = bA bR bG bB  
paddw     mm1, mm2          ; mm1 = rA rR rG rB  
psrlw     mm1, 8             ; mm1 = 0 rA 0 rR 0 rG 0 rB  
  
; pack into eax  
packuswb  mm1, mm1          ; mm1 = 0 0 0 0 rA rR rG rB  
movd      eax, mm1          ; eax = rA rR rG rB  
  
ret
```

To give you a better understanding of what can be done with MMX I've written a small function that blends two 32-bit ARGB pixels using 4 8-bit factors, one for each channel. To do this in C++ you would have to do the blending channel by channel. But with MMX we can blend all channels at once.

The blending factor is a one byte value between 0 and 255, as is the channel components. Each channel is blended using the following formula.

$$\text{res} = (a*fa + b*(255-fa))/255$$

OpenEXR

Original OpenEXR Image

... with original Exposure setting of zero (0):



(click for larger image)

Adjust 3 Stops Brighter

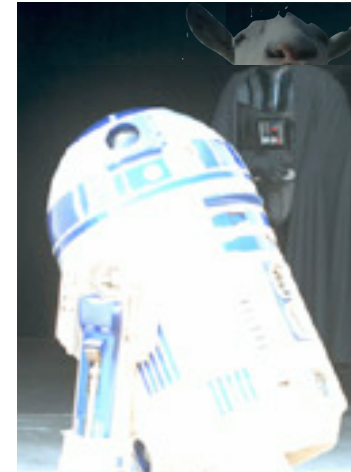
... details emerge from the shadows.



(click for larger image)

Adjust 5 Stops Brighter

... and even more details emerge from the shadows.



(click for larger image)

<http://www.openexr.com/>

- 16-bit FP format (1 sign, 10 mantissa, 5 exponent)
- Natively supported (paired “half”) in NVIDIA GPUs

SSE (Streaming SIMD Extensions)

- Intel 4-wide floating point
- Pentium III: 2 fully-pipelined, SIMD, single-precision FP units
- 8 128-bit registers added to ISA
- In HW, (historically) broke 4-wide instructions into 2 2-wide instructions
 - Interrupts are a problem
- MMX + SSE: added 10% to PIII die

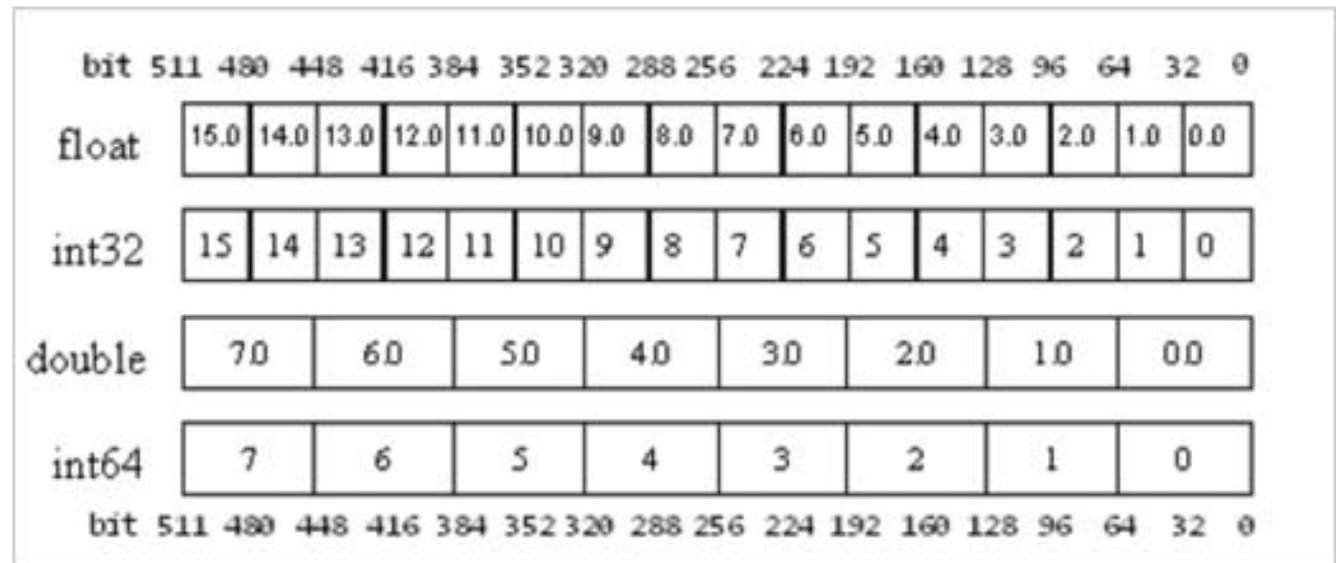
AVX (Advanced Vector Extensions)

- Proposed March 2008 by Intel; shipped Q1 2011 by Intel (Sandy Bridge), Q3 2011 by AMD (Bulldozer)
- Intel 8-wide floating point (256 bits, contrast to 128 bits in SSE)
 - SSE operations operate on lower 128 bits
- AVX2 (Haswell, 2013)
 - 3-operand instructions ($a=b+c$); Vector gather; Integer ops now support 256 bit-wide registers (presumably they didn't in AVX?)
- AVX-512 (Intel Knight's Landing [Xeon Phi], 2015; Skylake, 2017)
 - “4 operands, 7 new 64-bit opmask registers, scalar memory mode with automatic broadcast, explicit rounding control, and compressed displacement memory addressing mode. The width of the register file is increased to 512 bits and total register count increased to 32 (registers ZMM₀-ZMM₃₁) in x86-64 mode.”

AVX (Advanced Vector Extensions)

- Requires OS support for context switch:
 - Win7 SP1
 - OS X 10.6.8 (Snow Leopard)
 - Linux 2.6.30 (2009)

Larrabee Vector Instructions



- 32 512b-wide new registers
- 8 16b-wide vector mask registers
- Register ops generally ternary ($a=b \text{ op } c$)
- Extensive use of masks

LRB examples

`vmadd231ps v0 {k1}, v5, v6`

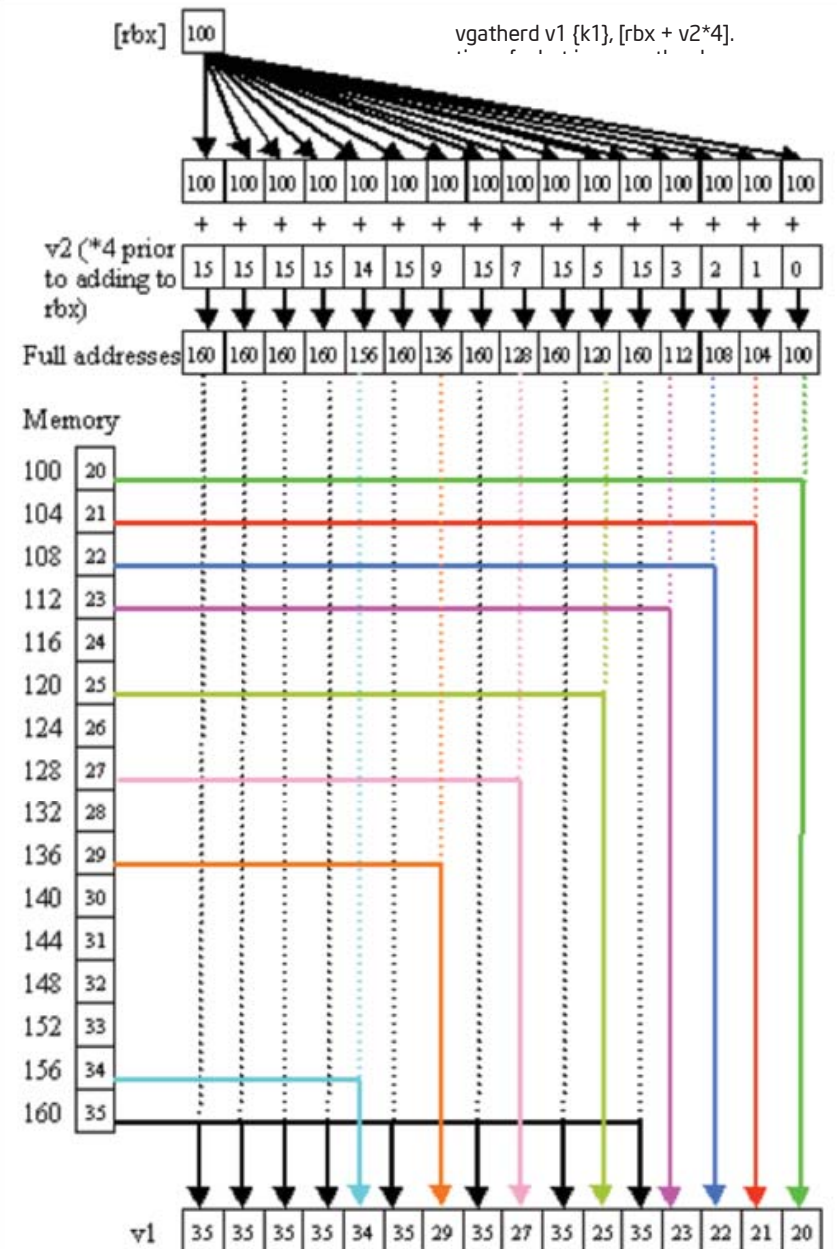
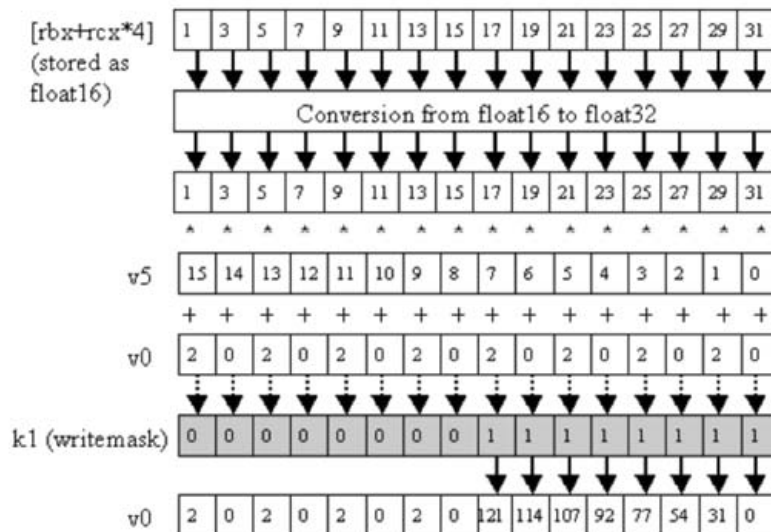
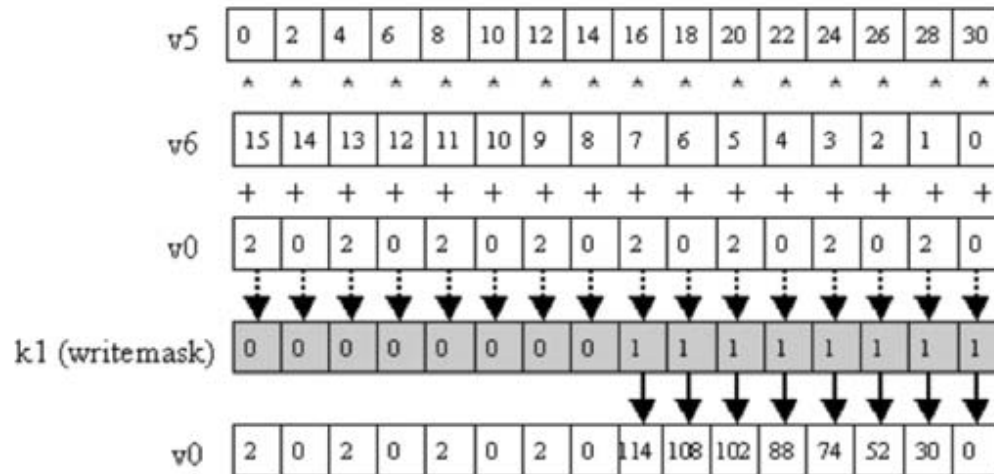


Figure 9: `vmadd231ps v0 {k1}, v5, [rbx+rcx*4] {float16}`.

Vector Processing

- Appendix G & 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

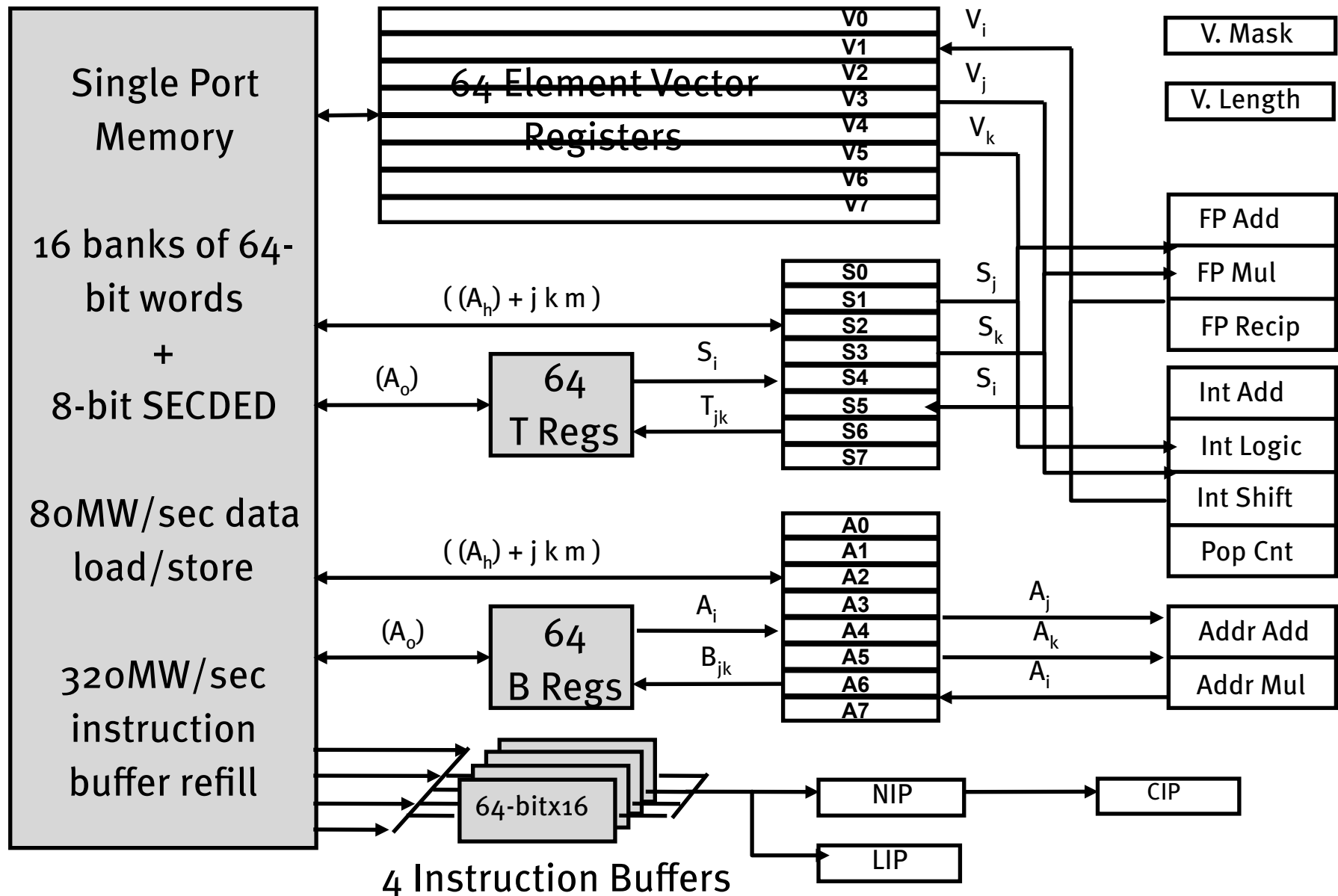
- Epitomized by Cray-1, 1976:
 - Scalar Unit + Vector Extensions
 - Load/Store Architecture
 - *Vector Registers*
 - *Vector Instructions*
 - Hardwired Control
 - Highly Pipelined Functional Units
 - Interleaved Memory System
 - No Data Caches
 - No Virtual Memory

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



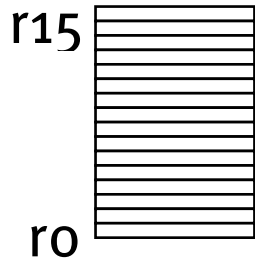
Cray-1 (1976)



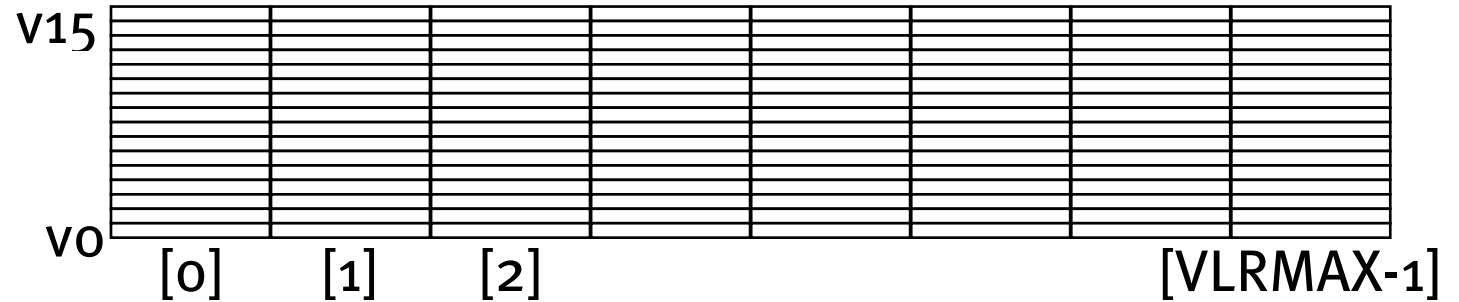
memory bank cycle 50 ns processor cycle 12.5 ns (80 MHz)

Vector Programming Model

Scalar Registers



Vector Registers

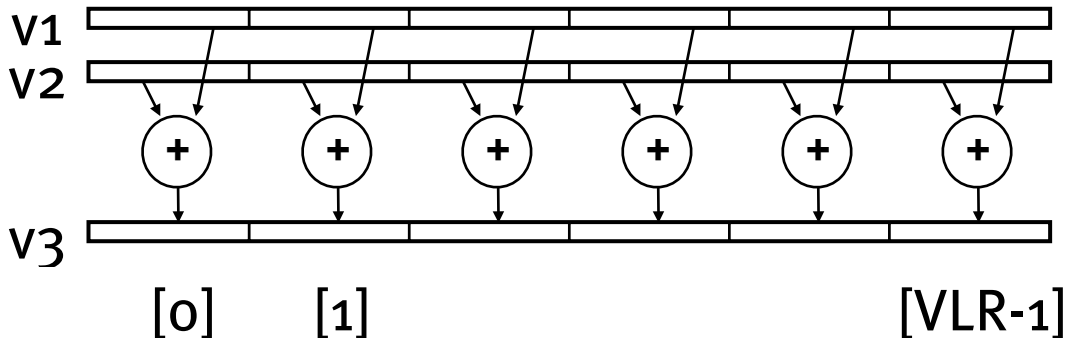


Vector Length Register

VLR

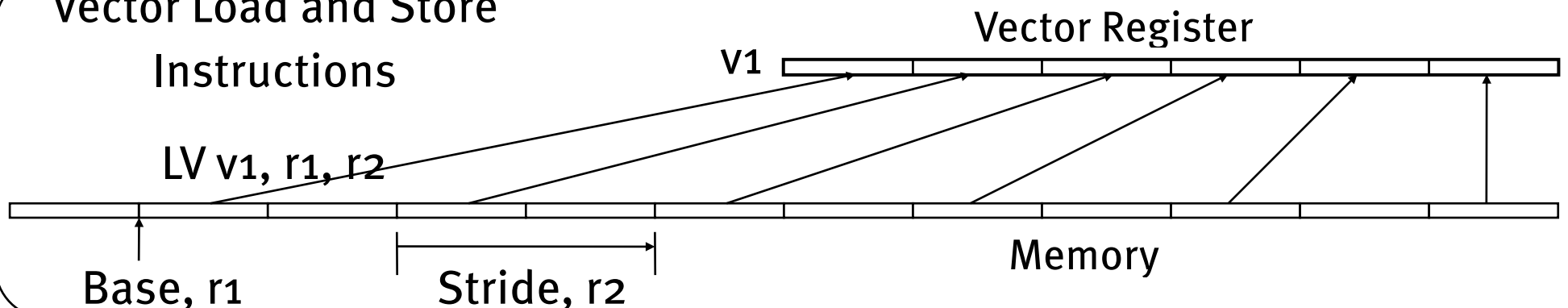
Vector Arithmetic
Instructions

ADDV v3, v1, v2



Vector Load and Store
Instructions

LV v1, r1, r2



Vector Code Example

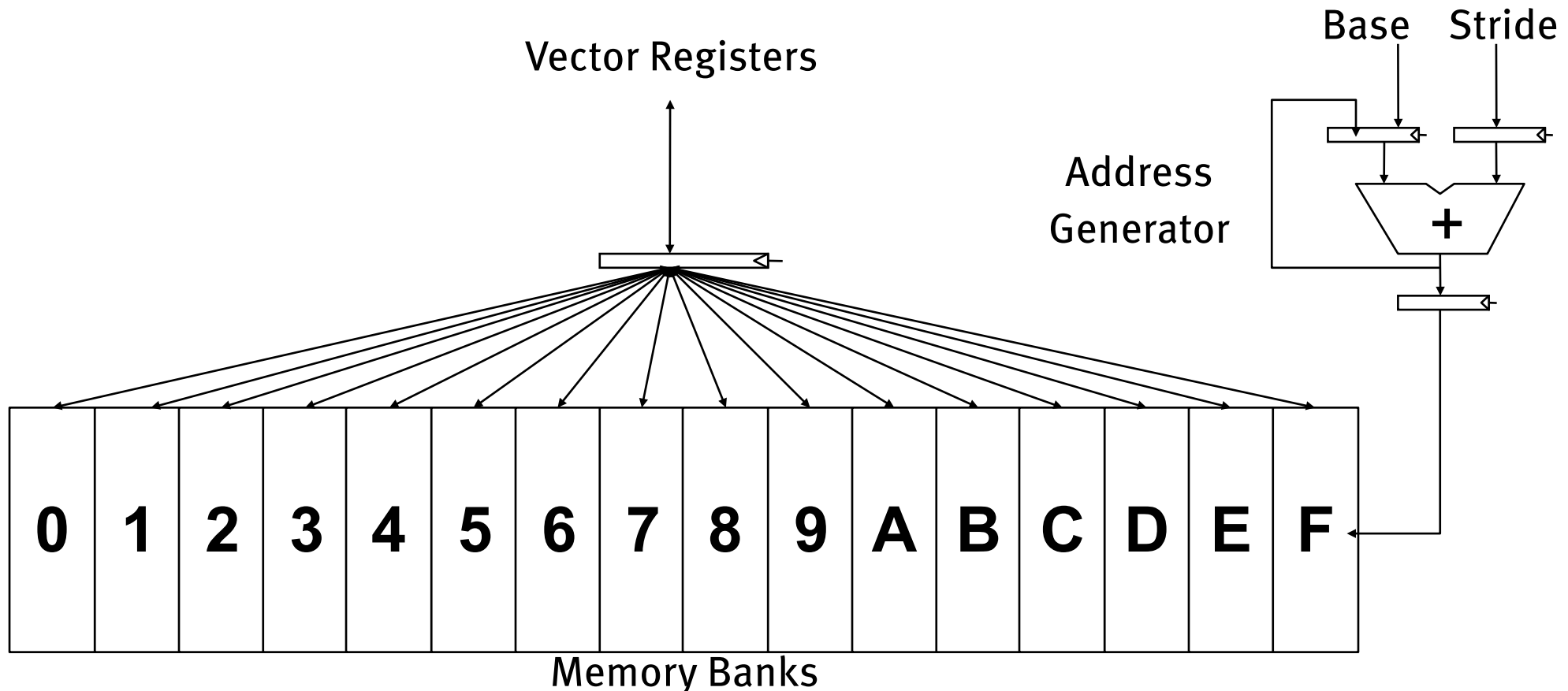
<pre># C code for (i=0; i<64; i++) C[i] = A[i] + B[i];</pre>	<pre># Scalar Code LI R4, 64 loop: L.D F0, 0(R1) L.D F2, 0(R2) ADD.D F4, F2, F0 S.D F4, 0(R3) DADDIU R1, 8 DADDIU R2, 8 DADDIU R3, 8 DSUBIU R4, 1 BNEZ R4, loop</pre>	<pre># Vector Code LI VLR, 64 LV V1, R1 LV V2, R2 ADDV.D V3, V1, V2 SV V3, R3</pre>
---	---	---

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 Memory System

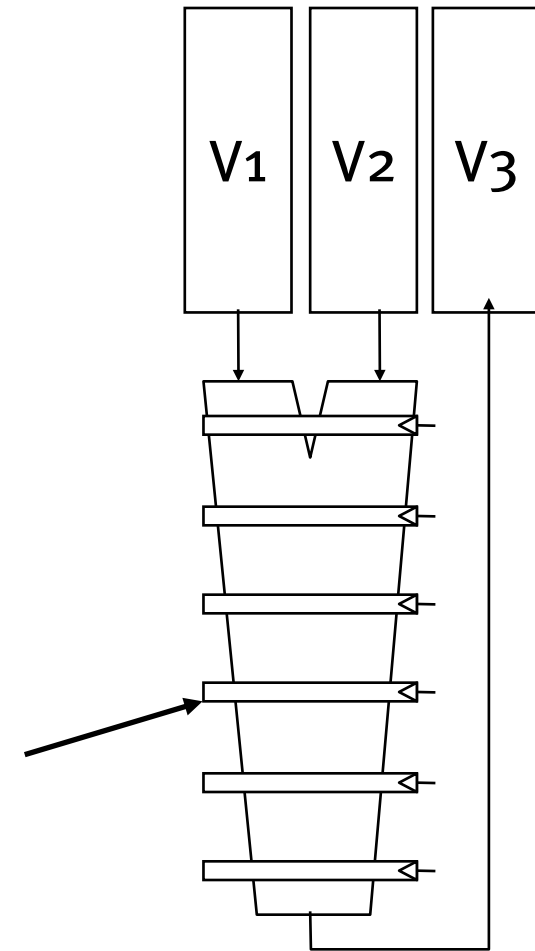
- Cray-1: 16 banks, 64b wide per bank, 4 cycle bank busy time, 12 cycle latency
- Bank busy time: Cycles between accesses to same bank



Vector 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



$$V_3 \leftarrow V_1 * V_2$$

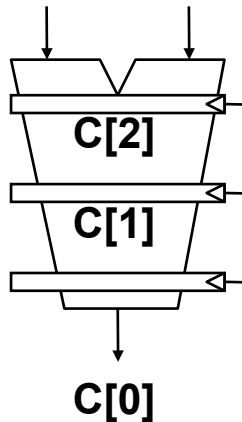
Vector Instruction Execution

ADDV C, A, B

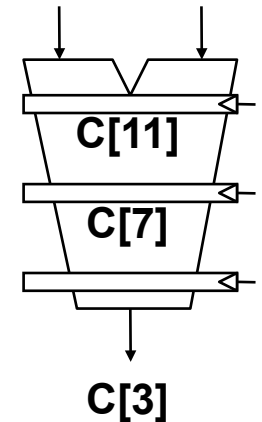
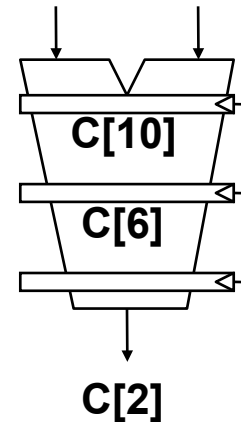
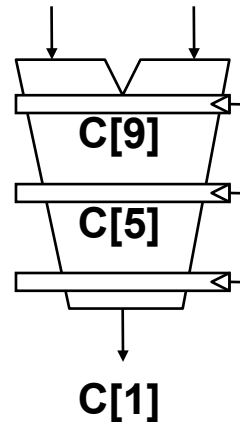
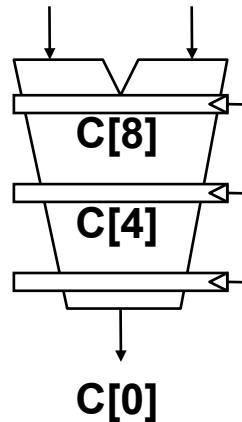
*Execution using one
pipelined functional
unit*

*Execution using
four pipelined
functional units*

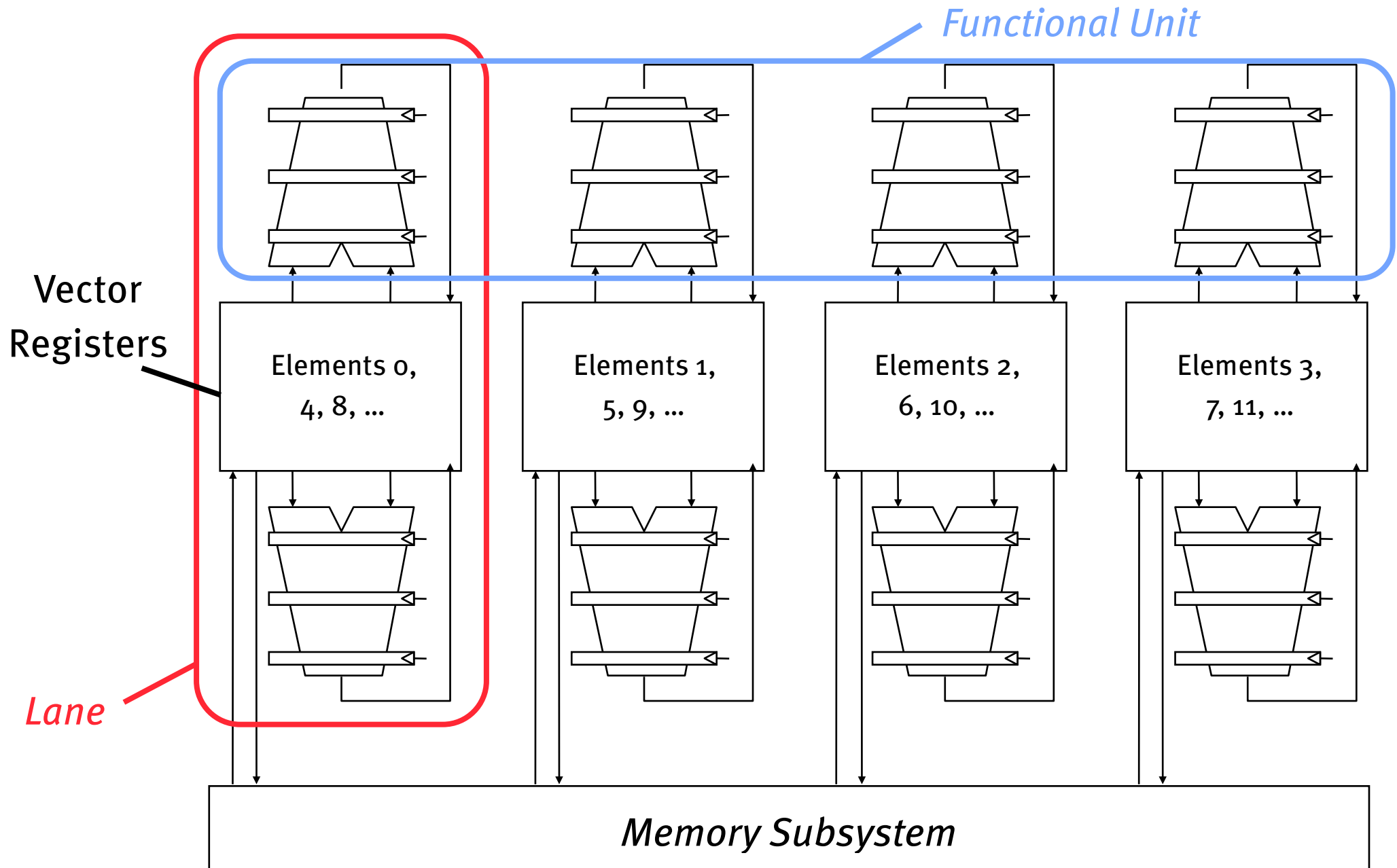
A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]



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]

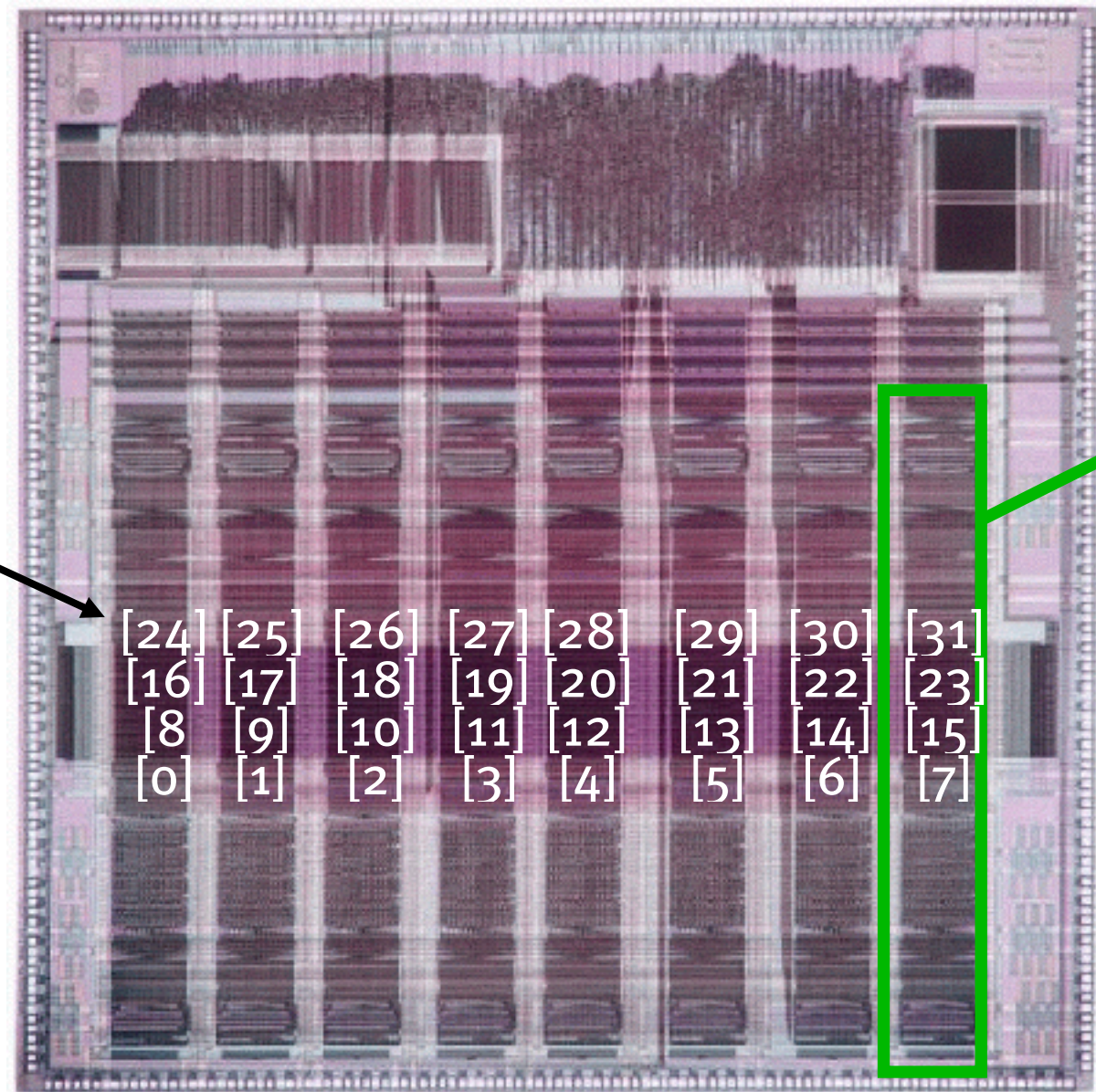


Vector Unit Structure



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)

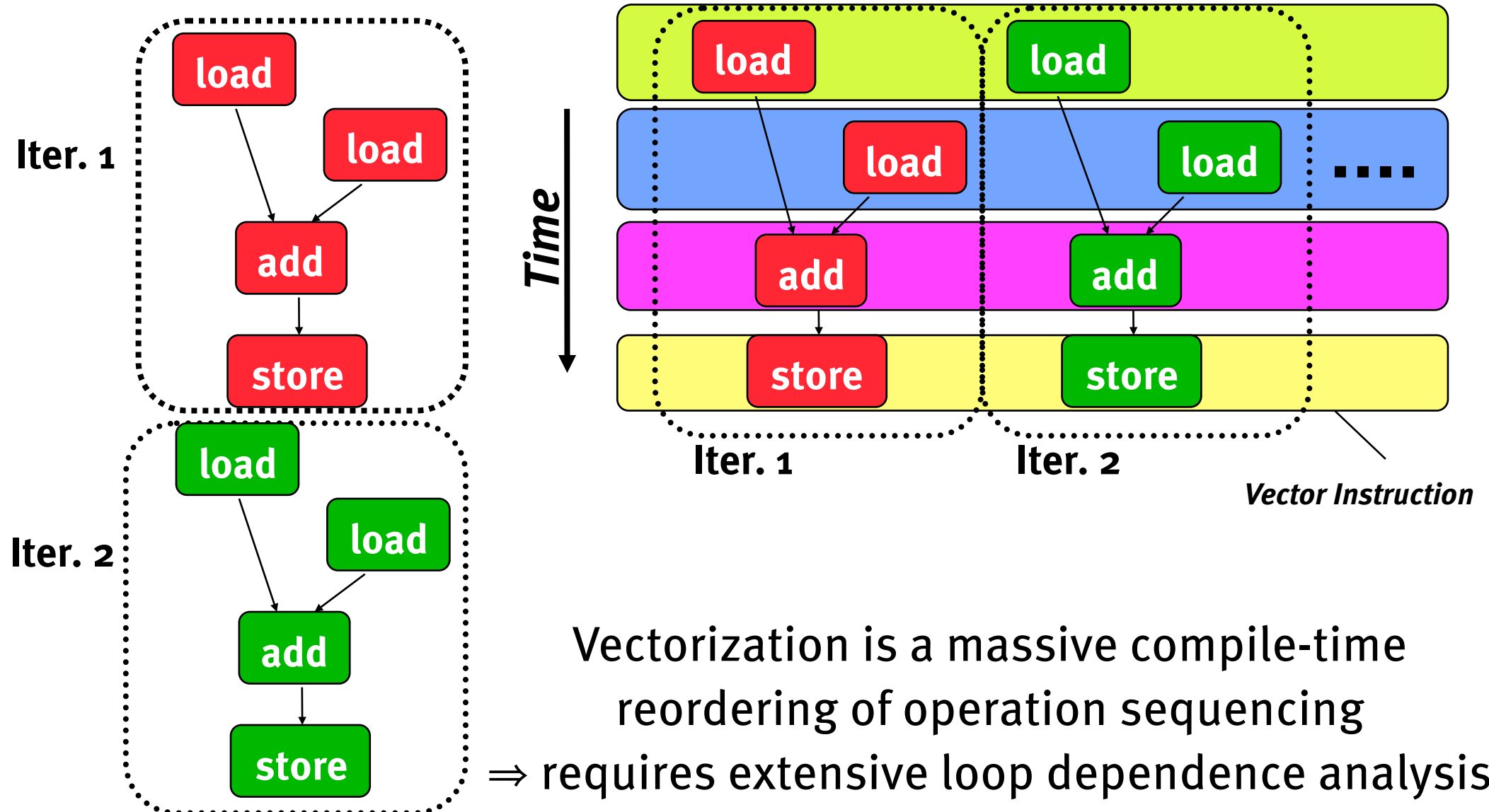
Automatic Code Vectorization

```
for (i=0; i < N; i++)
```

```
  C[i] = A[i] + B[i];
```

Scalar Sequential Code

Vectorized Code



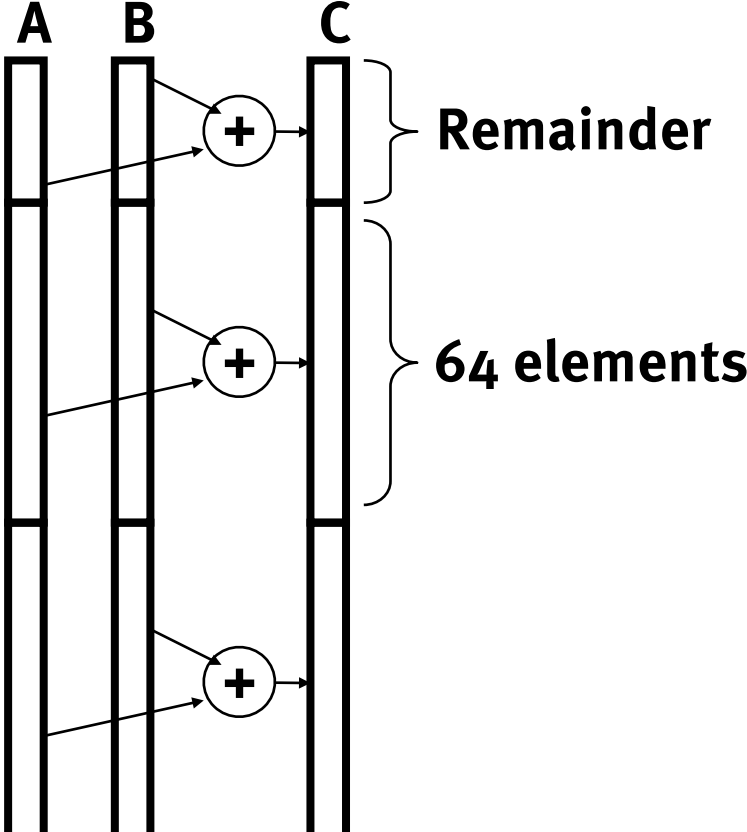
Guy Steele, Dr Dobbs Journal 24 Nov 2005

- “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  
    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?
```

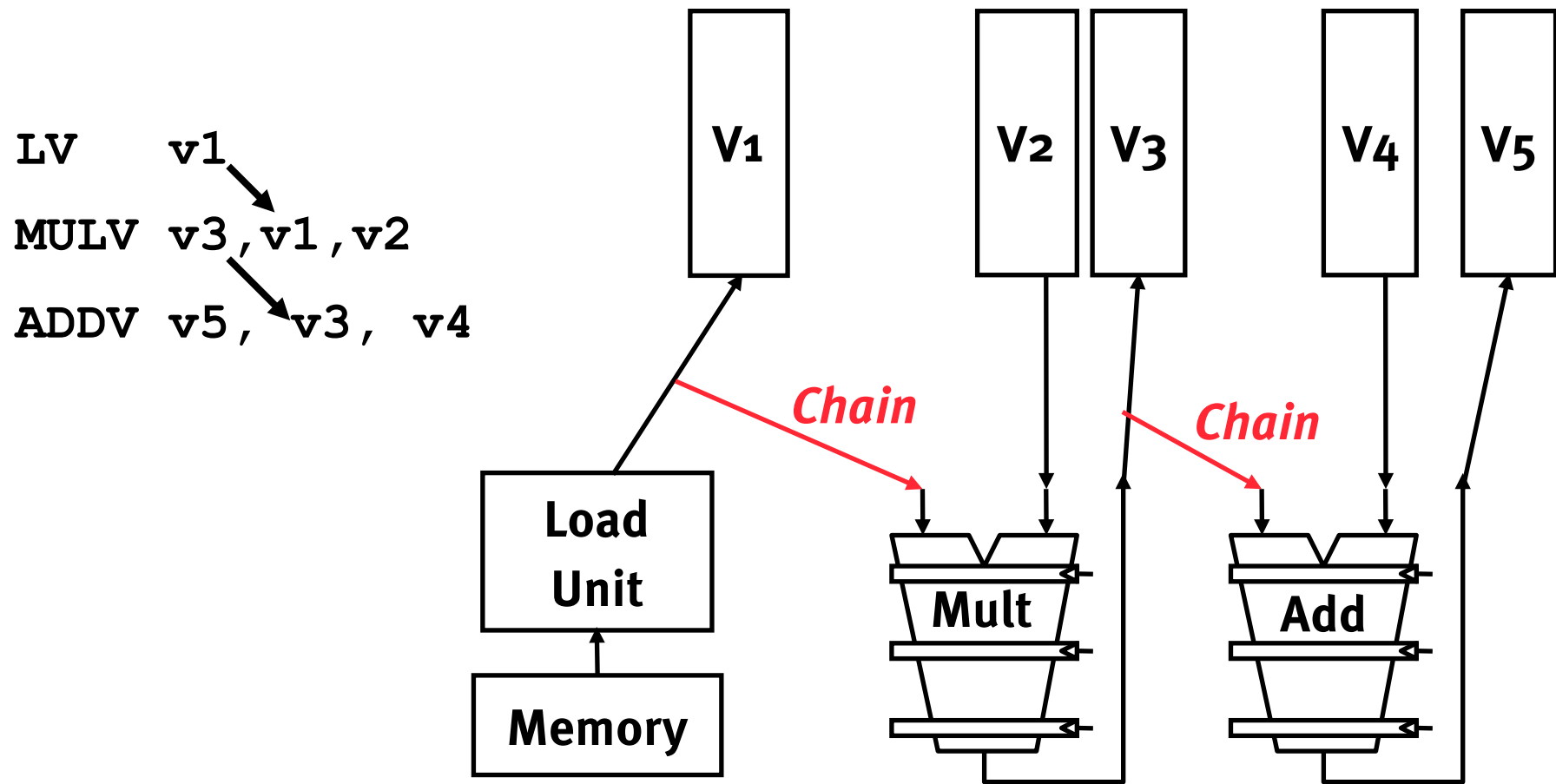
Vector Inefficiency

- 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

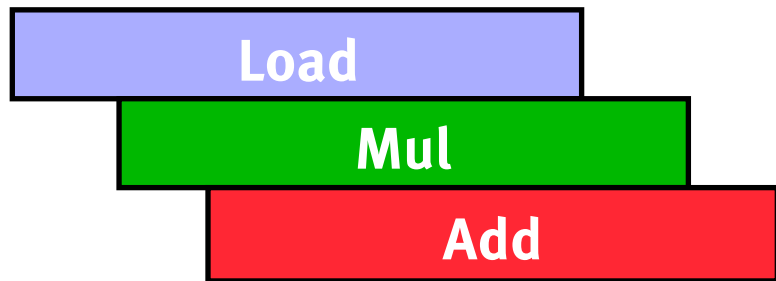


Vector Chaining Advantage

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

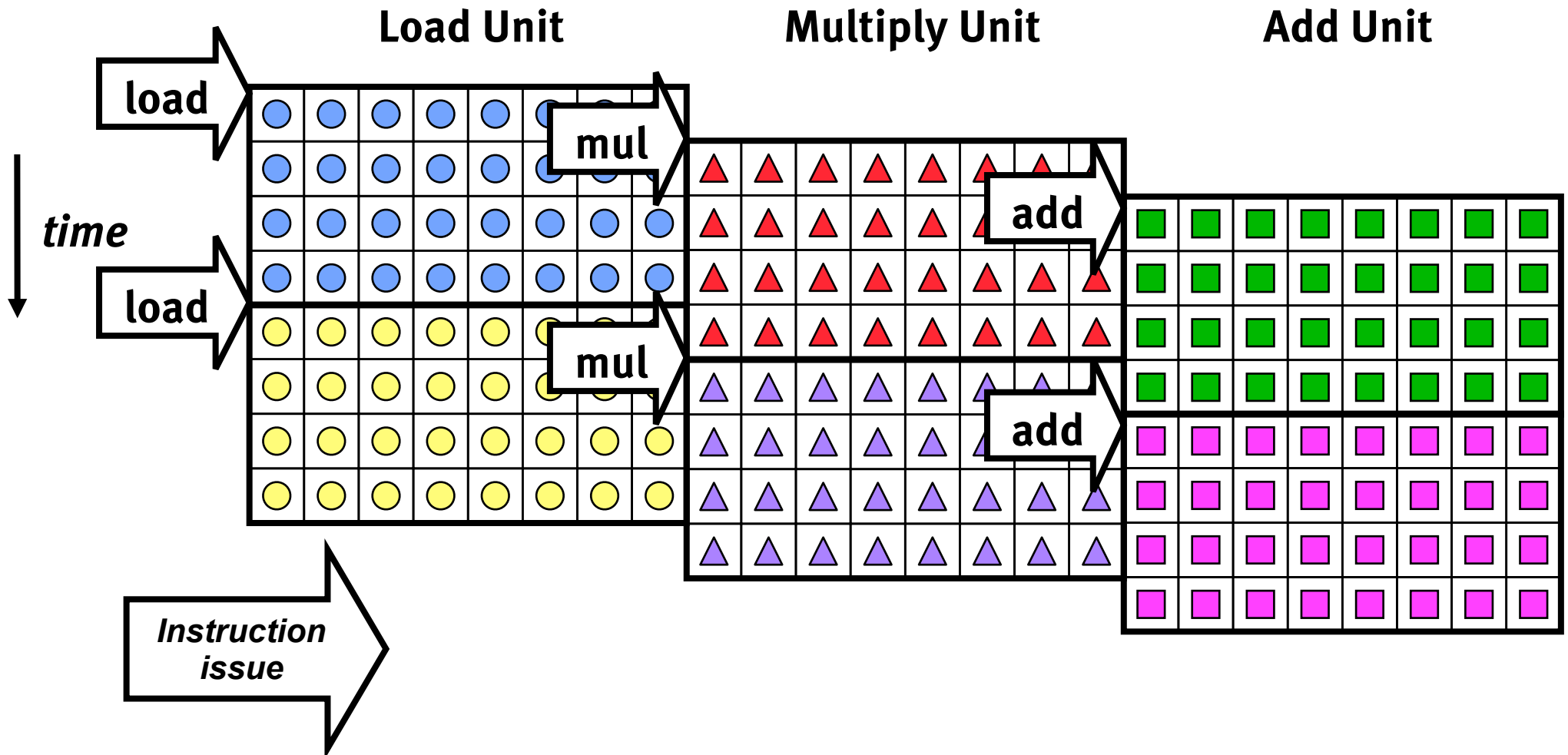


- With chaining, can start dependent instruction as soon as first result appears



Vector Instruction Parallelism

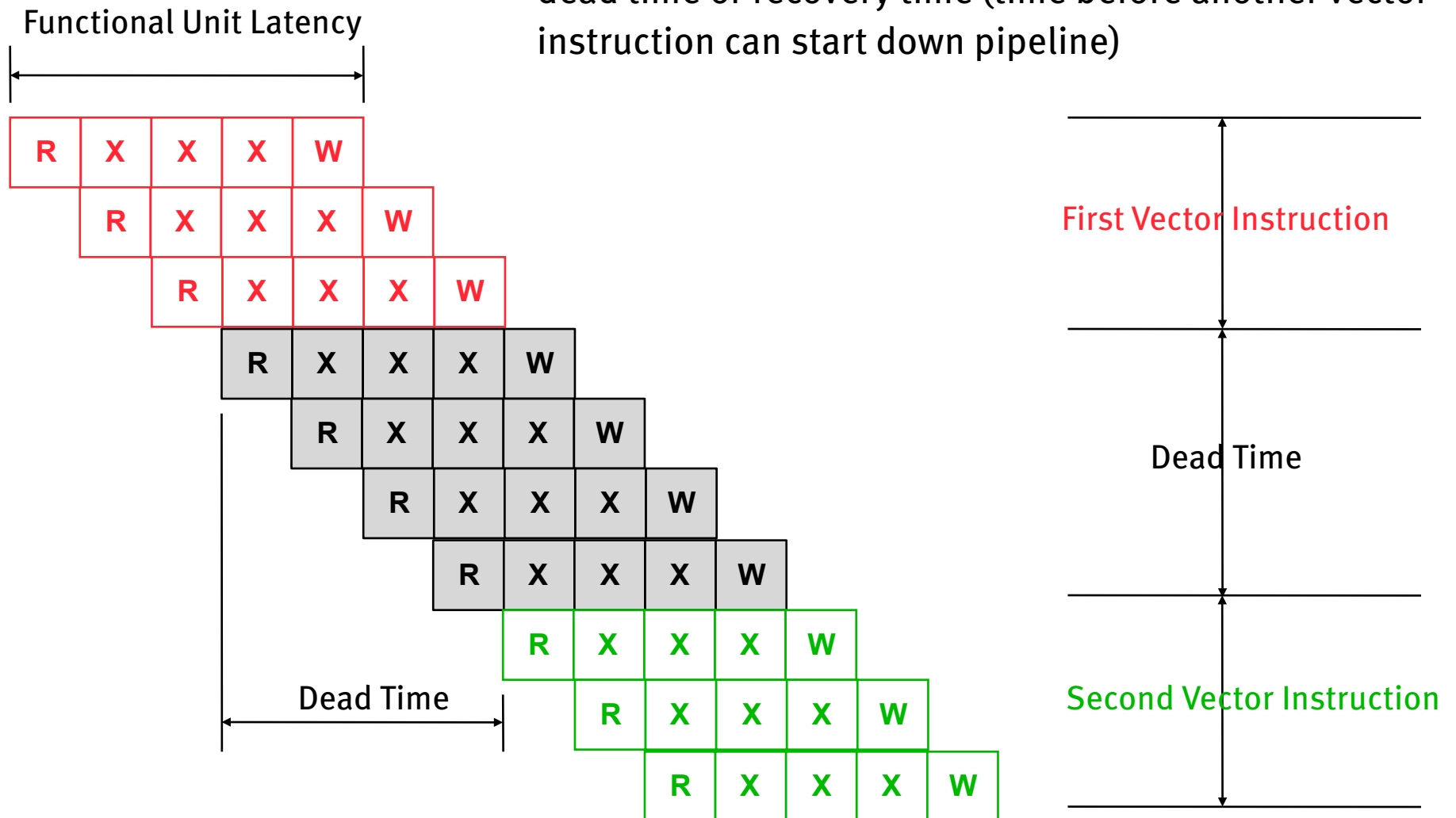
- Can overlap execution of multiple vector instructions
- example machine has 32 elements per vector register and 8 lanes



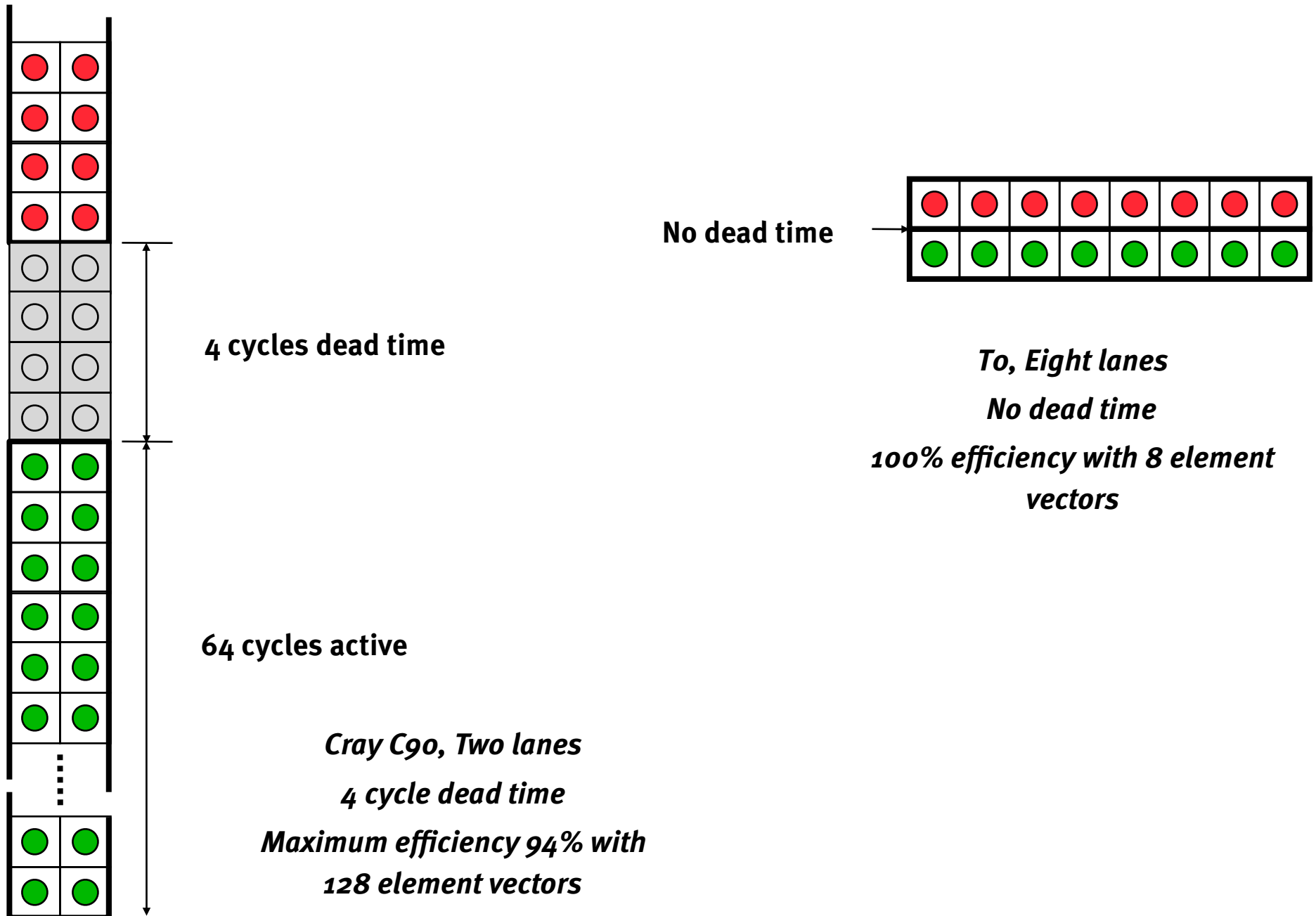
Complete 24 operations/cycle while issuing 1 short instruction/cycle

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



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];
```

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

- 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

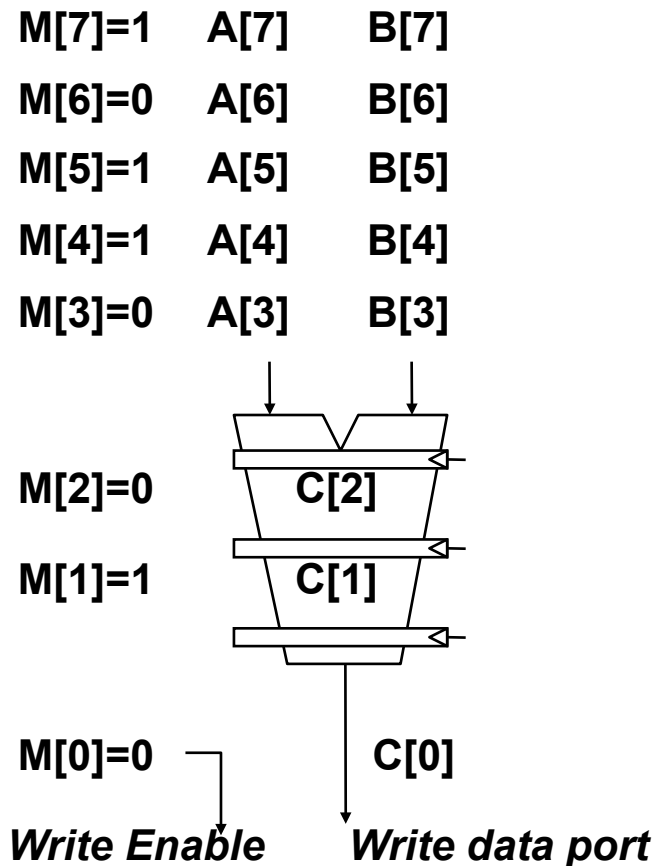
- Code example (vector mask is implicit in this instruction set):

```
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

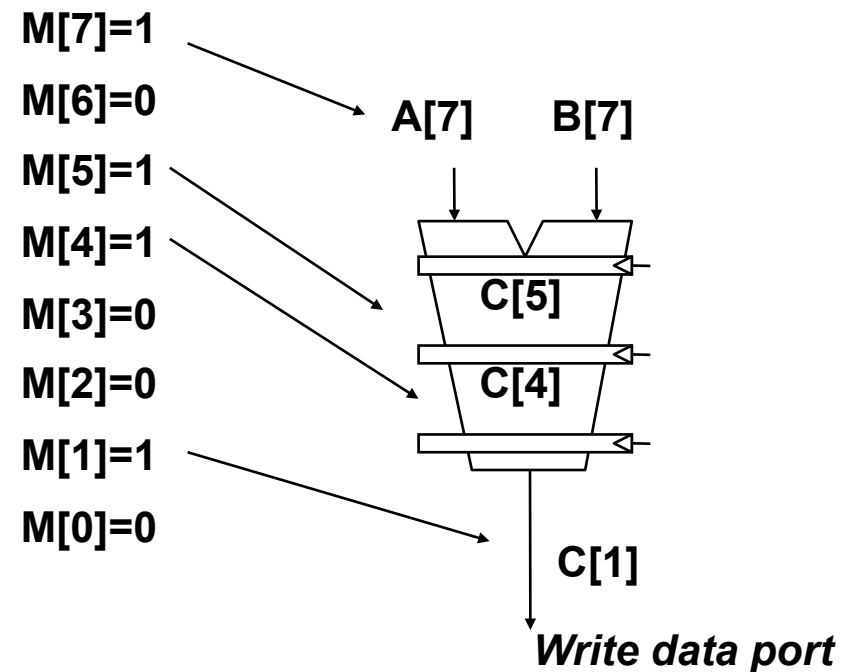
Simple Implementation

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



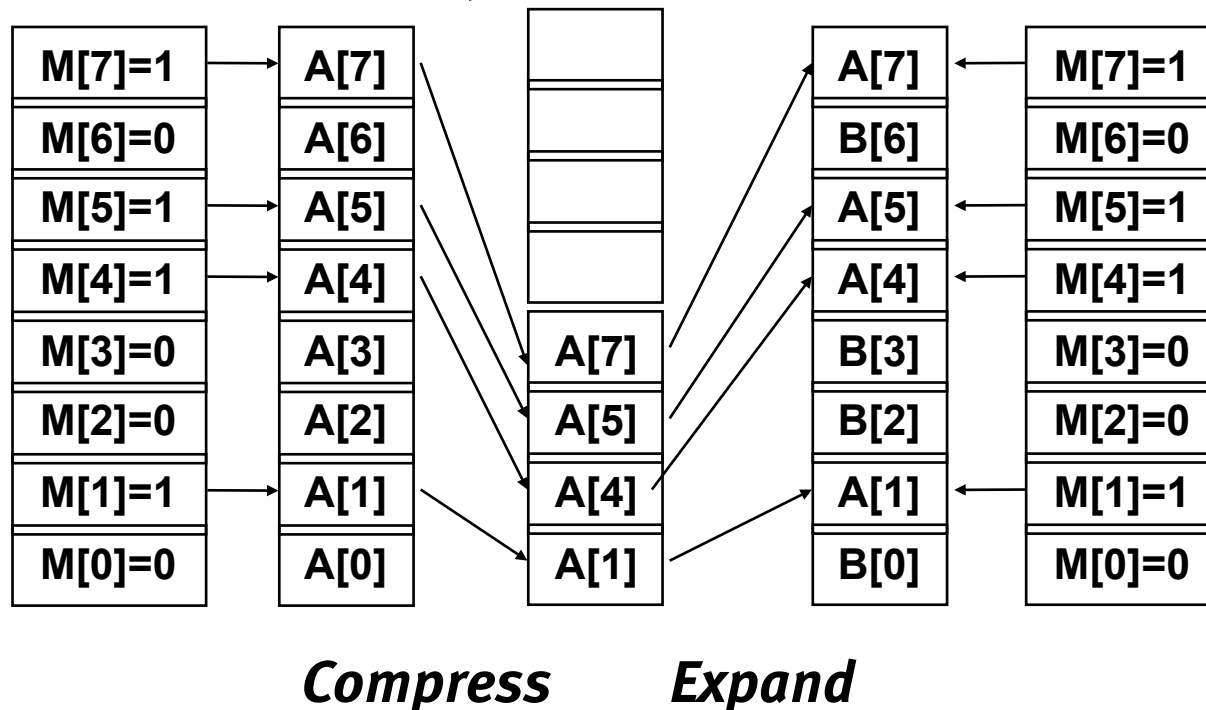
Density-Time Implementation

- 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



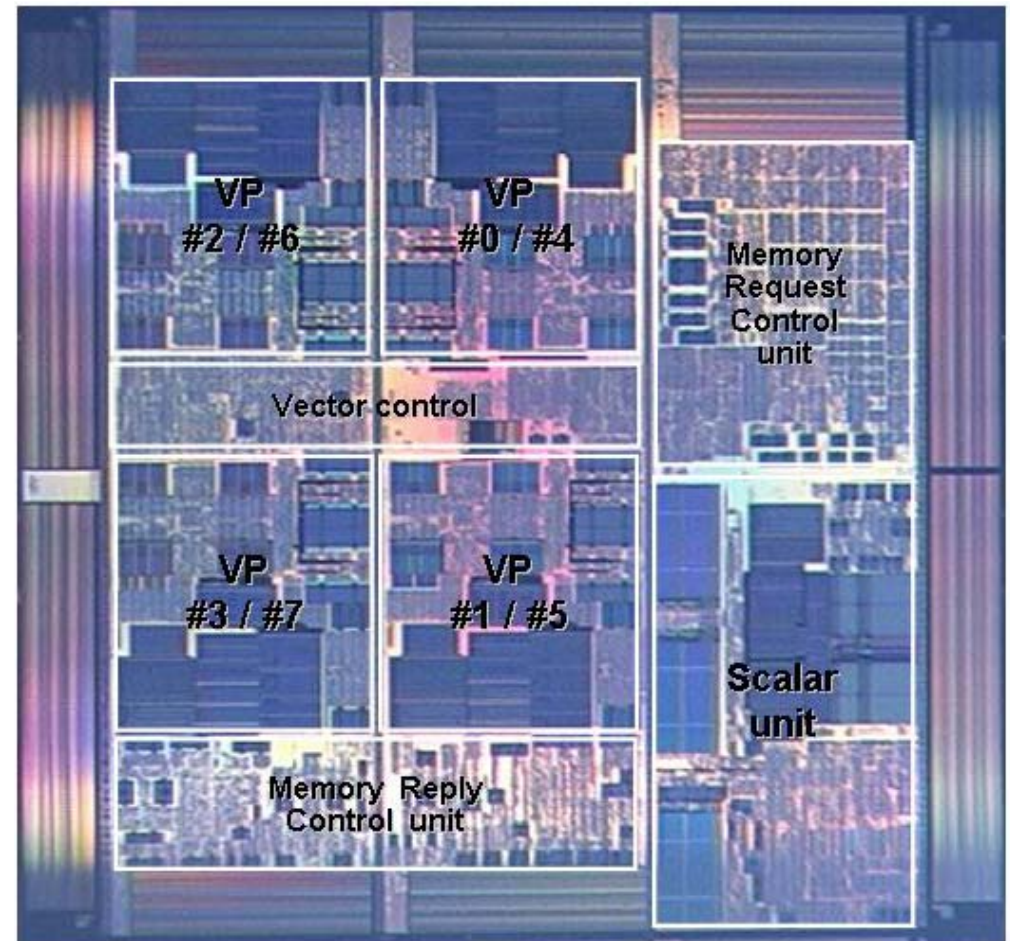
A Modern Vector Super: NEC SX-6 (2003)

- 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)



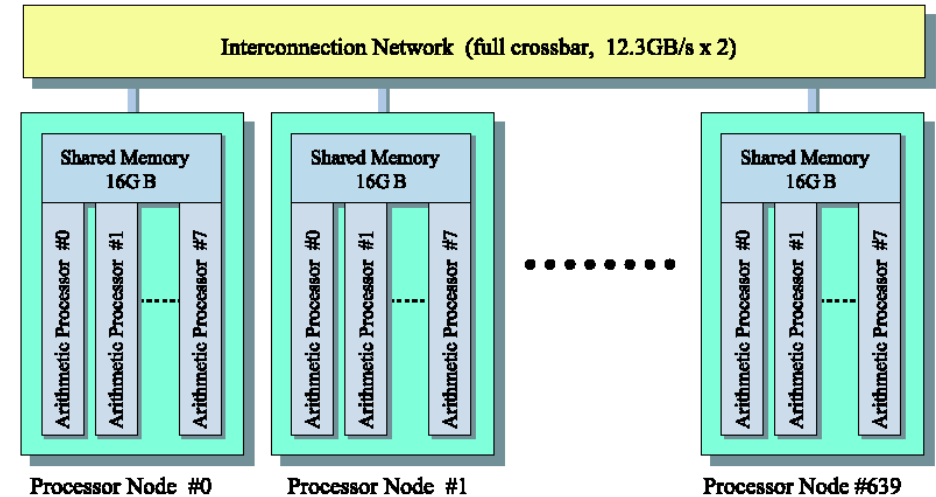
SX-6 Die Photo

- 0.15 μm CMOS
- 60M transistors
- 432 mm²
- 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
 - 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 dopi-est 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 512x512 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

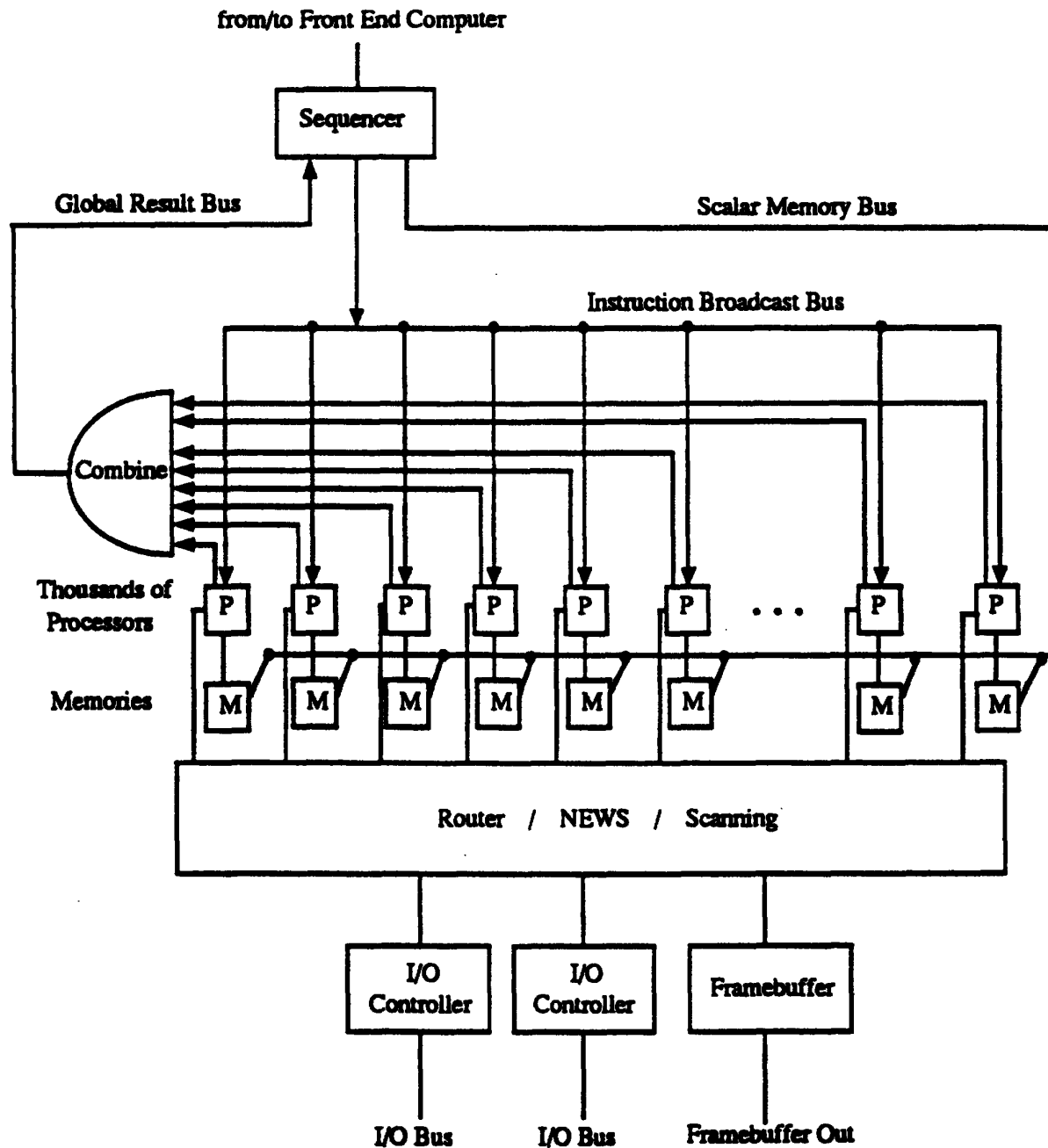
Communication Primitives

- 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

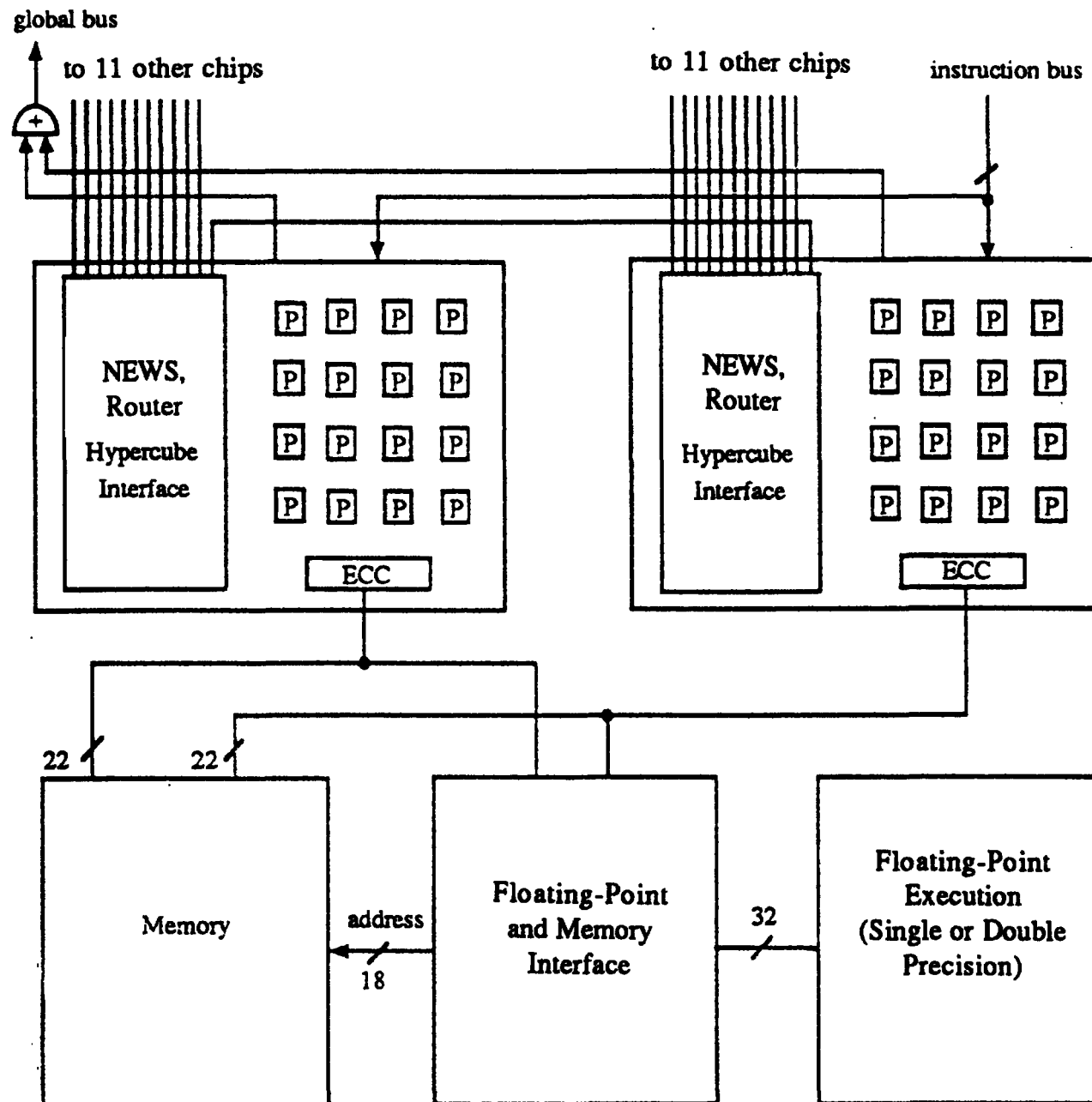
Computation + Communication Primitives

- 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

ispc: A SPMD Compiler for High-Performance CPU Programming

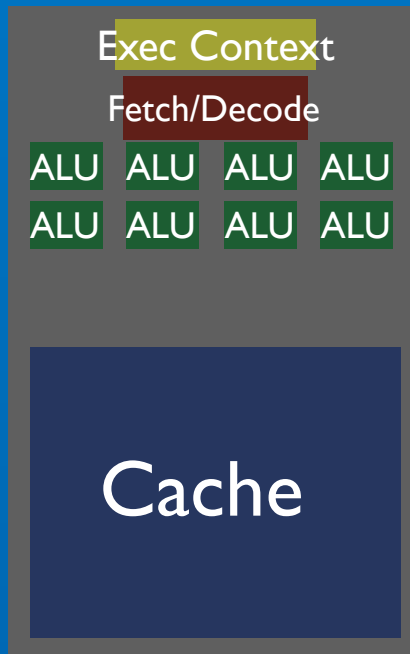
Matt Pharr and William R. Mark
Intel

14 May 2012

<http://ispc.github.com>

Motivation: 3 Modern Parallel Architectures

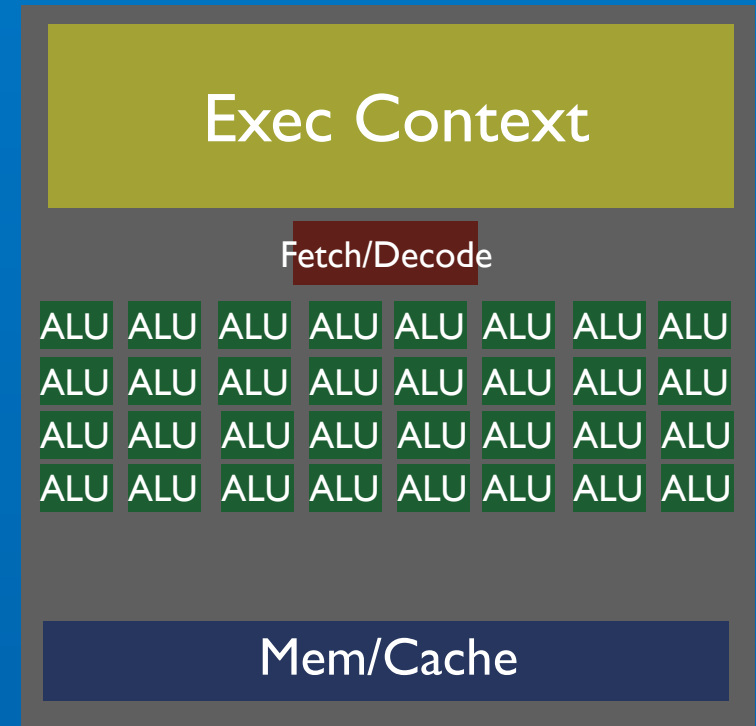
CPU:
2-10x



MIC:
50+x

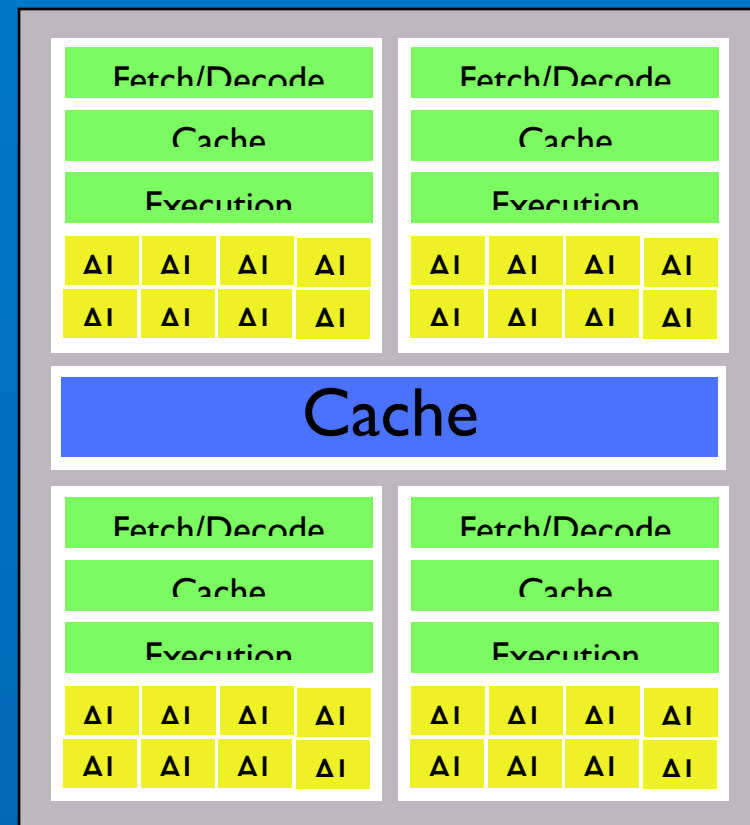


GPU:
2-32x



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 1_a
    ++b;        fadd, jmp 1_b
else
    ++c;        fadd
                1_a:
                1_b:
```

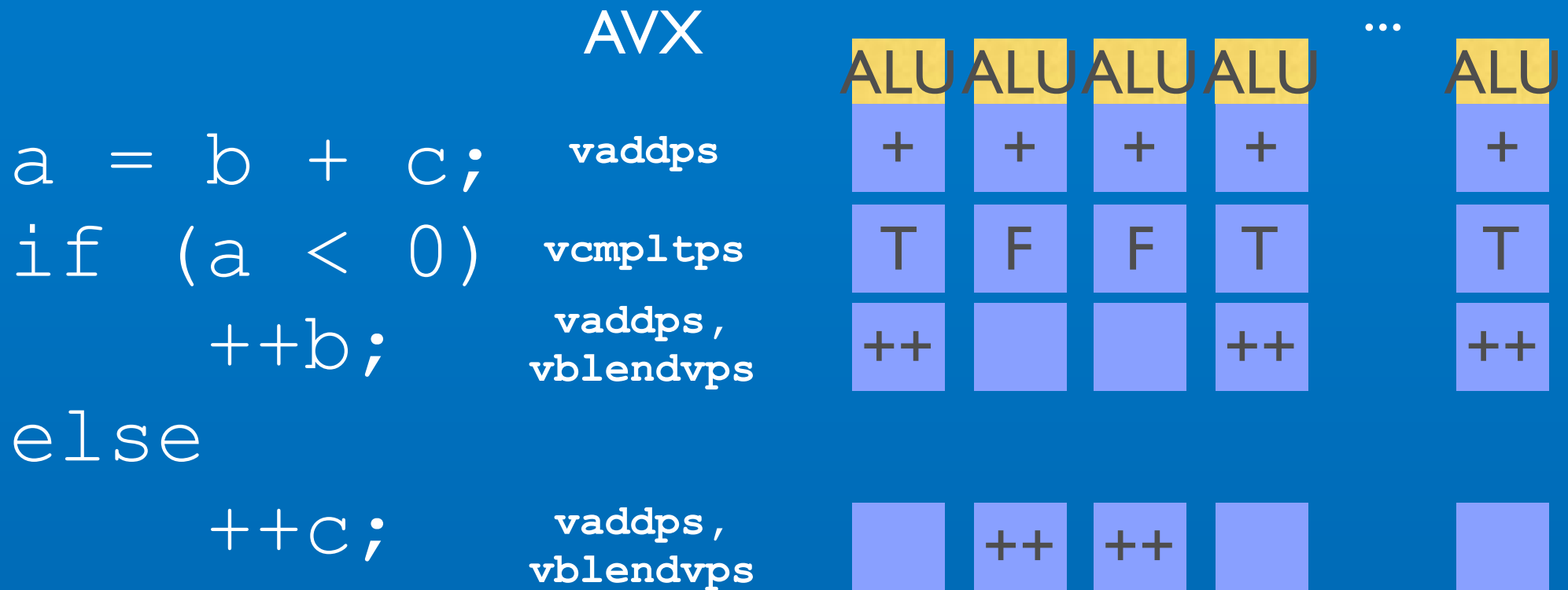
(Based on http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraFlop_BPS_SIGGRAPH2010.pdf)

SPMD On A GPU SIMD Unit

	~PTX	ALU	ALU	ALU	ALU	...	ALU
<code>a = b + c;</code>	<code>fadd</code>	+	+	+	+		+
<code>if (a < 0)</code>	<code>cmp, jge 1_a</code>	T	F	F	T		T
<code> ++b;</code>	<code>fadd, jmp 1_b</code>	++			++		++
<code>else</code>	<code>1_a:</code>						
<code> ++c;</code>	<code>fadd</code>		++	++			
	<code>1_b:</code>						

(Based on http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraFlop_BPS_SIGGRAPH2010.pdf)

SPMD On A CPU SIMD Unit



(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) {           old_mask = current_mask
    true stmts;       test_mask = evaluate test
}                     current_mask &= test_mask
else {                // emit true stmts, predicate with current_mask
    false stmts;      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

Multimedia Extensions

- Very short vectors added to existing ISAs for micros
- Usually 64-bit registers split into 2x32b or 4x16b or 8x8b
- Newer designs have 128-bit registers (AltiVec, SSE2)
- Limited instruction set:
 - no vector length control
 - no strided load/store or scatter/gather
 - unit-stride loads must be aligned to 64/128-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