

18-447

Computer Architecture

Lecture 16: SIMD and GPUs

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2014, 2/24/2014

Readings for Today

- SIMD Processing
- Basic GPU Architecture
- Other execution models: VLIW, Dataflow
- Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.
- Fatahalian and Houston, "A Closer Look at GPUs," CACM 2008.
- Stay tuned for more readings...

SIMD Processing: Exploiting Regular (Data) Parallelism

Flynn's Taxonomy of Computers

- Mike Flynn, “[Very High-Speed Computing Systems](#),” Proc. of IEEE, 1966

- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
 - Array processor
 - Vector processor
- **MISD**: Multiple instructions operate on single data element
 - Closest form: systolic array processor, streaming processor
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
 - Multiprocessor
 - Multithreaded processor

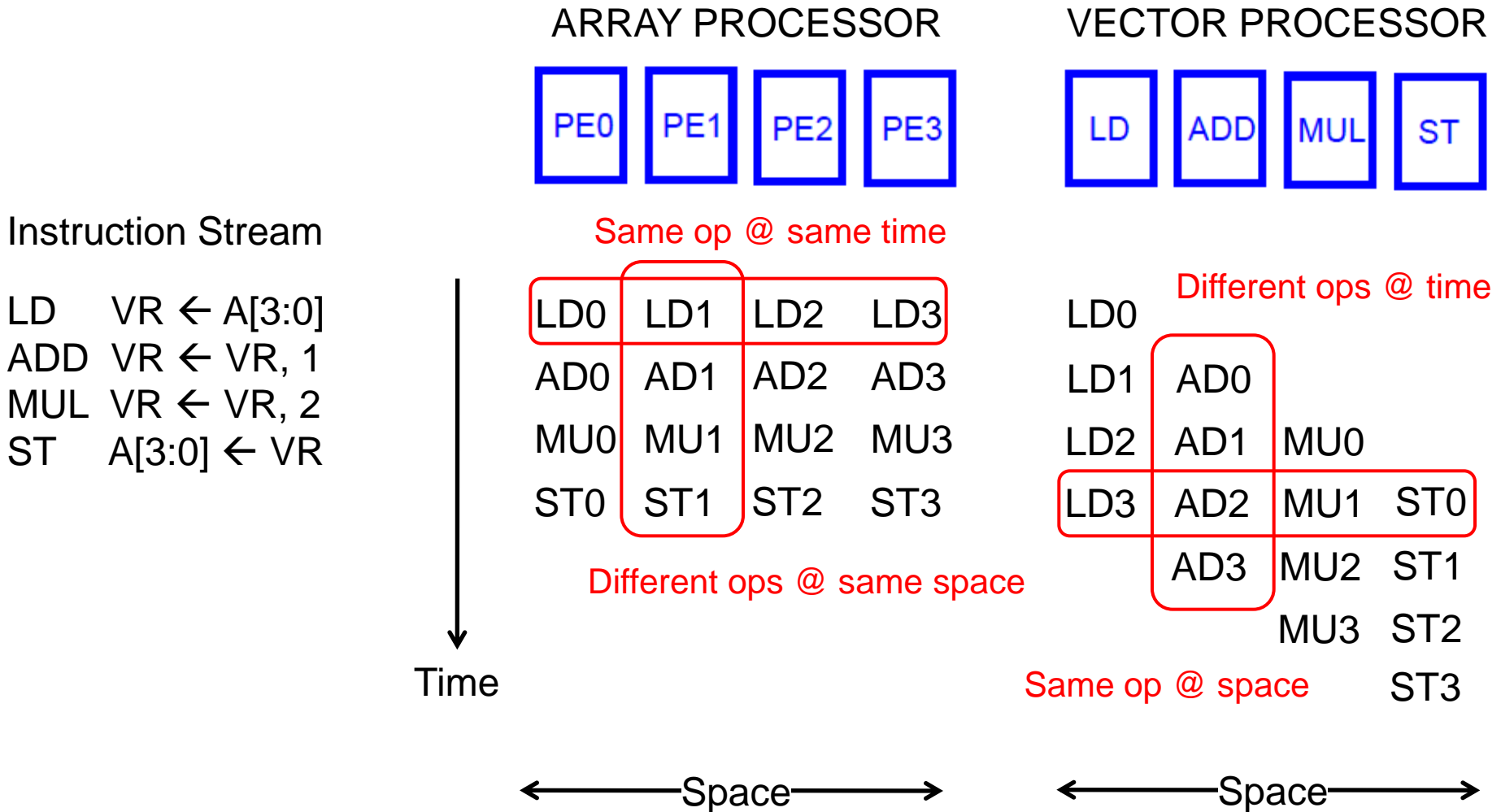
Data Parallelism

- Concurrency arises from performing the **same operations on different pieces of data**
 - Single instruction multiple data (SIMD)
 - E.g., dot product of two vectors
- Contrast with data flow
 - Concurrency arises from executing different operations in parallel (in a data driven manner)
- Contrast with thread (“control”) parallelism
 - Concurrency arises from executing different threads of control in parallel
- SIMD exploits instruction-level parallelism
 - Multiple “instructions” (more appropriately, operations) are concurrent: instructions happen to be the same

SIMD Processing

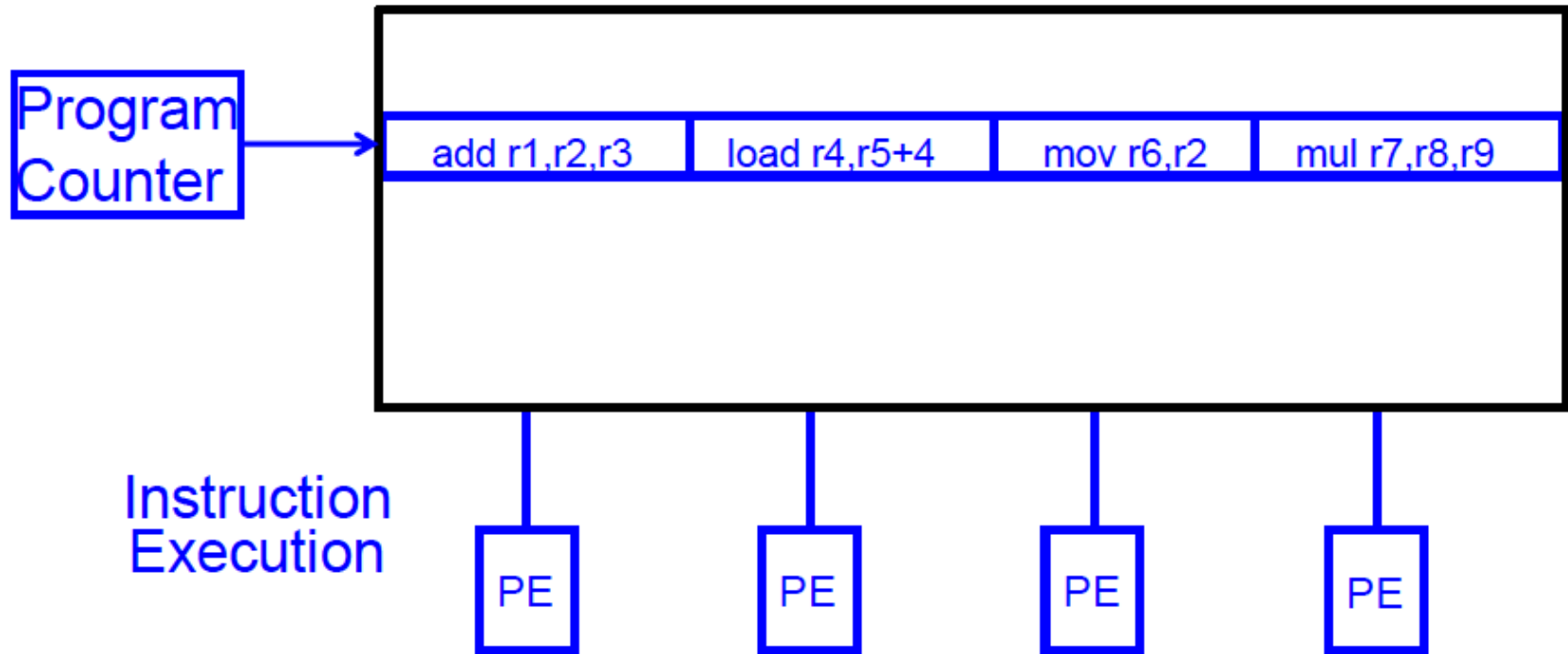
- Single instruction operates on multiple data elements
 - In time or in space
- Multiple processing elements
- Time-space duality
 - **Array processor**: Instruction operates on multiple data elements at the same time
 - **Vector processor**: Instruction operates on multiple data elements in consecutive time steps

Array vs. Vector Processors



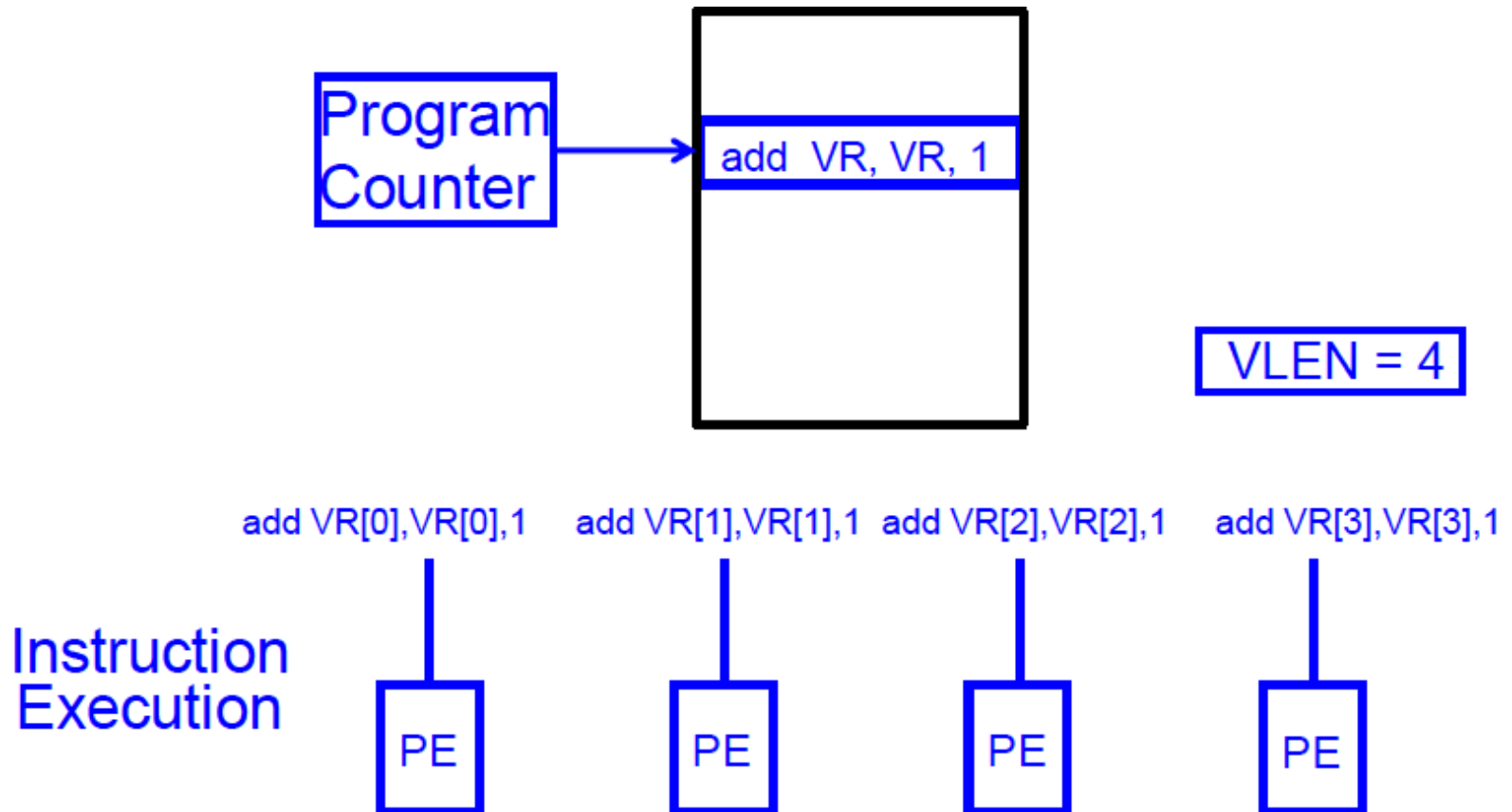
SIMD Array Processing vs. VLIW

■ VLIW



SIMD Array Processing vs. VLIW

- Array processor



Vector Processors

- A vector is a one-dimensional array of numbers
- Many scientific/commercial programs use vectors
 - for (i = 0; i<=49; i++)
C[i] = (A[i] + B[i]) / 2
- A vector processor is one whose instructions operate on vectors rather than scalar (single data) values
- Basic requirements
 - Need to load/store vectors → vector registers (contain vectors)
 - Need to operate on vectors of different lengths → vector length register (VLEN)
 - Elements of a vector might be stored apart from each other in memory → vector stride register (VSTR)
 - Stride: distance between two elements of a vector

Vector Processors (II)

- A vector instruction performs an operation on each element in consecutive cycles
 - Vector functional units are pipelined
 - Each pipeline stage operates on a different data element
- Vector instructions allow deeper pipelines
 - No intra-vector dependencies → no hardware interlocking within a vector
 - No control flow within a vector
 - Known stride allows prefetching of vectors into cache/memory

Vector Processor Advantages

+ No dependencies within a vector

- ❑ Pipelining, parallelization work well
- ❑ Can have very deep pipelines, no dependencies!

+ Each instruction generates a lot of work

- ❑ Reduces instruction fetch bandwidth

+ Highly regular memory access pattern

- ❑ Interleaving multiple banks for higher memory bandwidth
- ❑ Prefetching

+ No need to explicitly code loops

- ❑ Fewer branches in the instruction sequence

Vector Processor Disadvantages

- Works (only) if parallelism is regular (data/SIMD parallelism)
 - ++ Vector operations
 - Very inefficient if parallelism is irregular
 - How about searching for a key in a linked list?

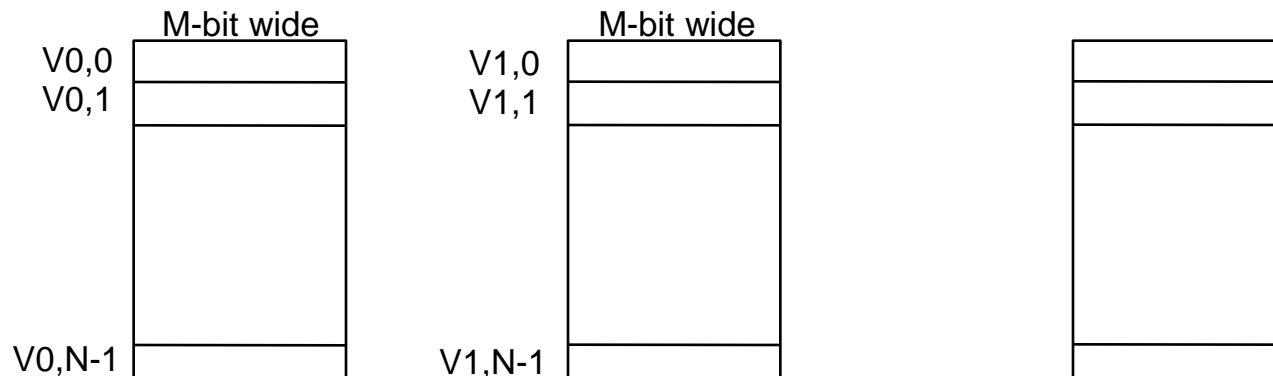
To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That's hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtleties of the application area. Often the rewriting is

Vector Processor Limitations

- Memory (bandwidth) can easily become a bottleneck, especially if
 1. compute/memory operation balance is not maintained
 2. data is not mapped appropriately to memory banks

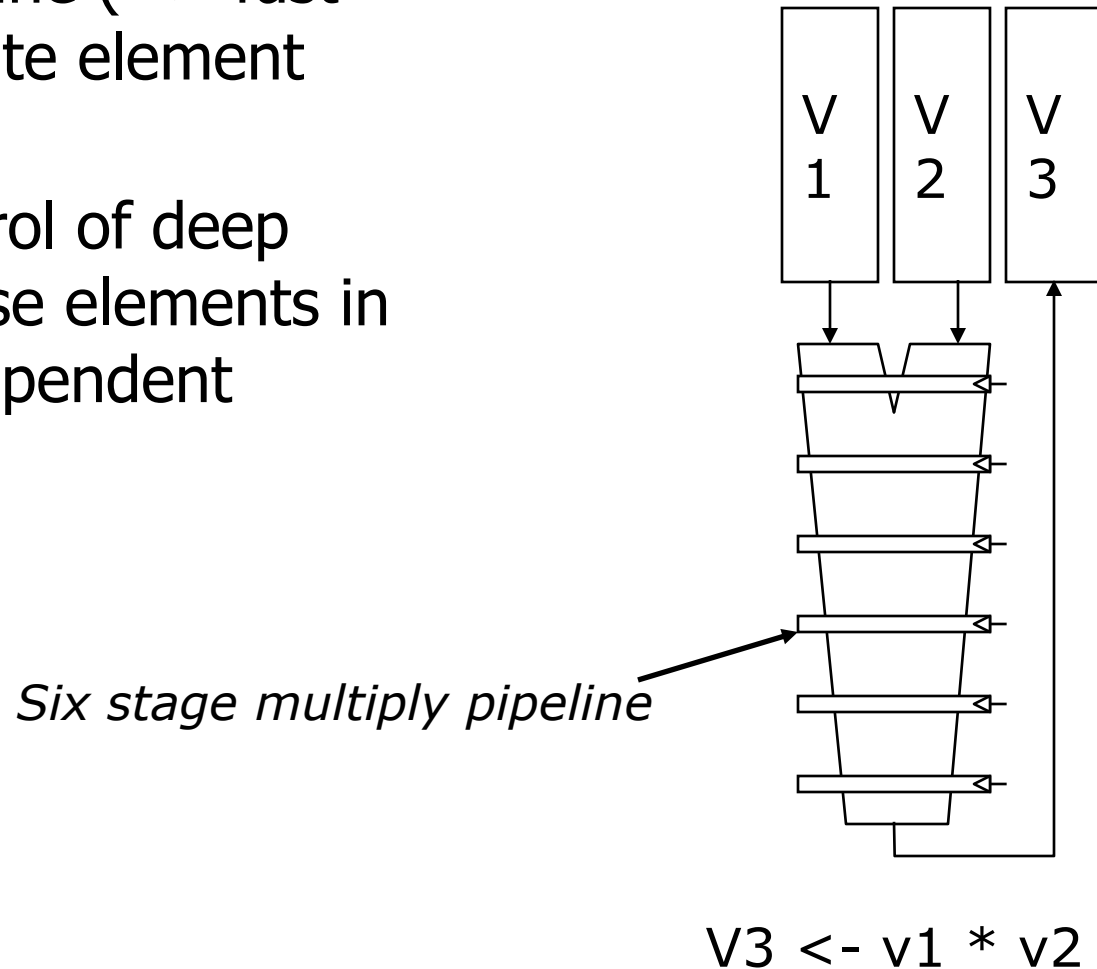
Vector Registers

- Each **vector data register** holds N M-bit values
- **Vector control registers**: VLEN, VSTR, VMASK
- **Vector Mask Register (VMASK)**
 - Indicates which elements of vector to operate on
 - Set by vector test instructions
 - e.g., $\text{VMASK}[i] = (\text{V}_k[i] == 0)$
- Maximum VLEN can be N
 - Maximum number of elements stored in a vector register

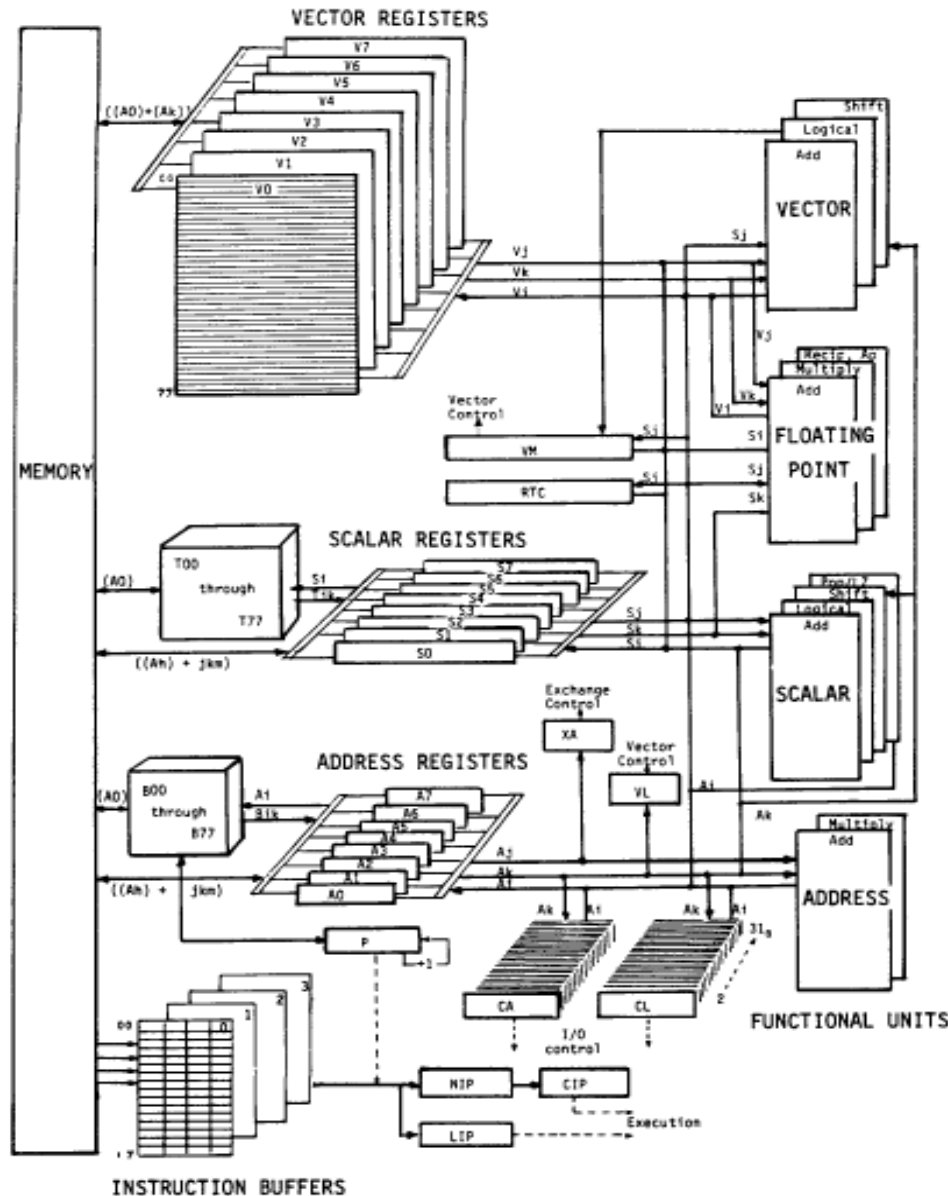


Vector Functional Units

- Use deep pipeline (\Rightarrow fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent



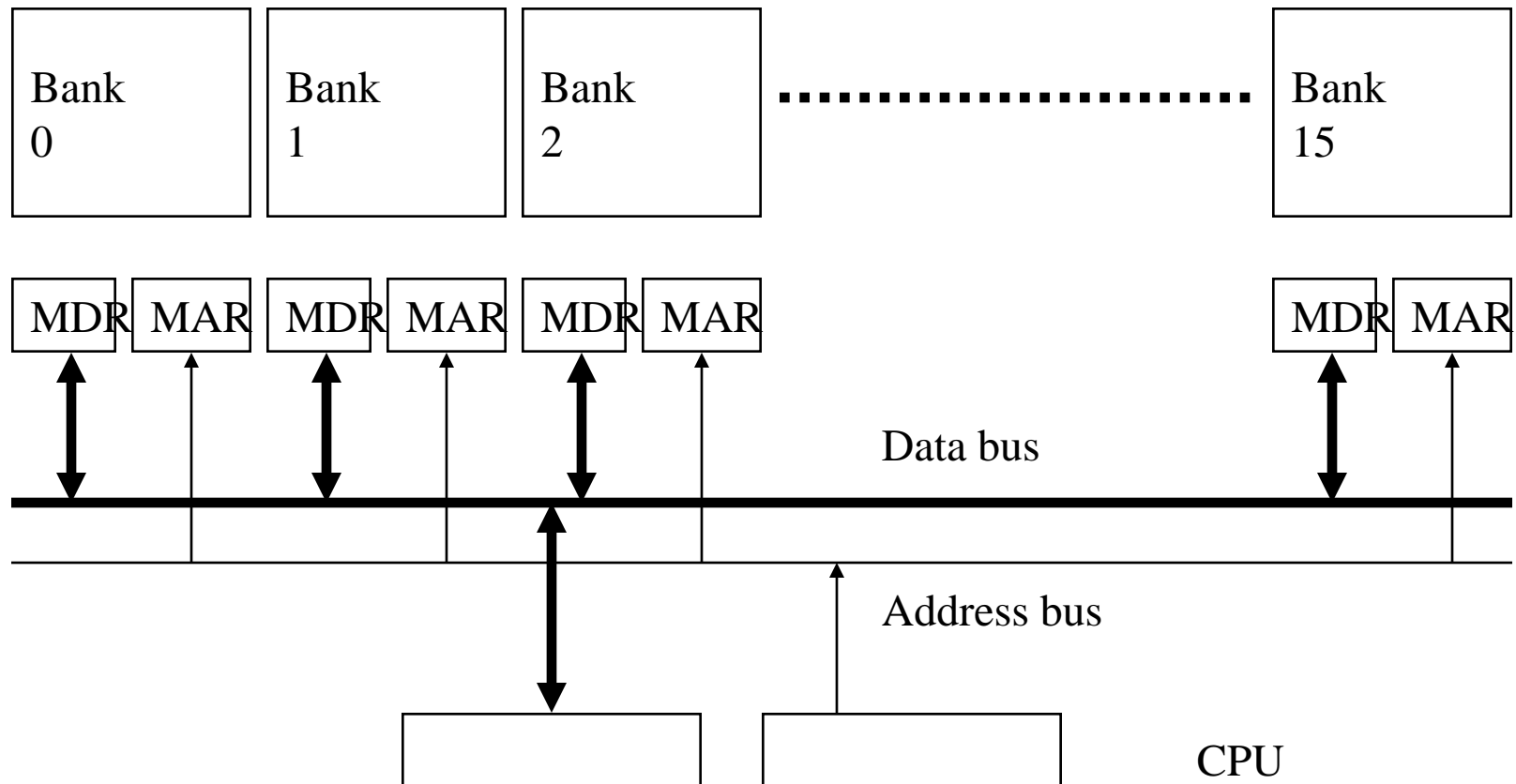
Vector Machine Organization (CRAY-1)



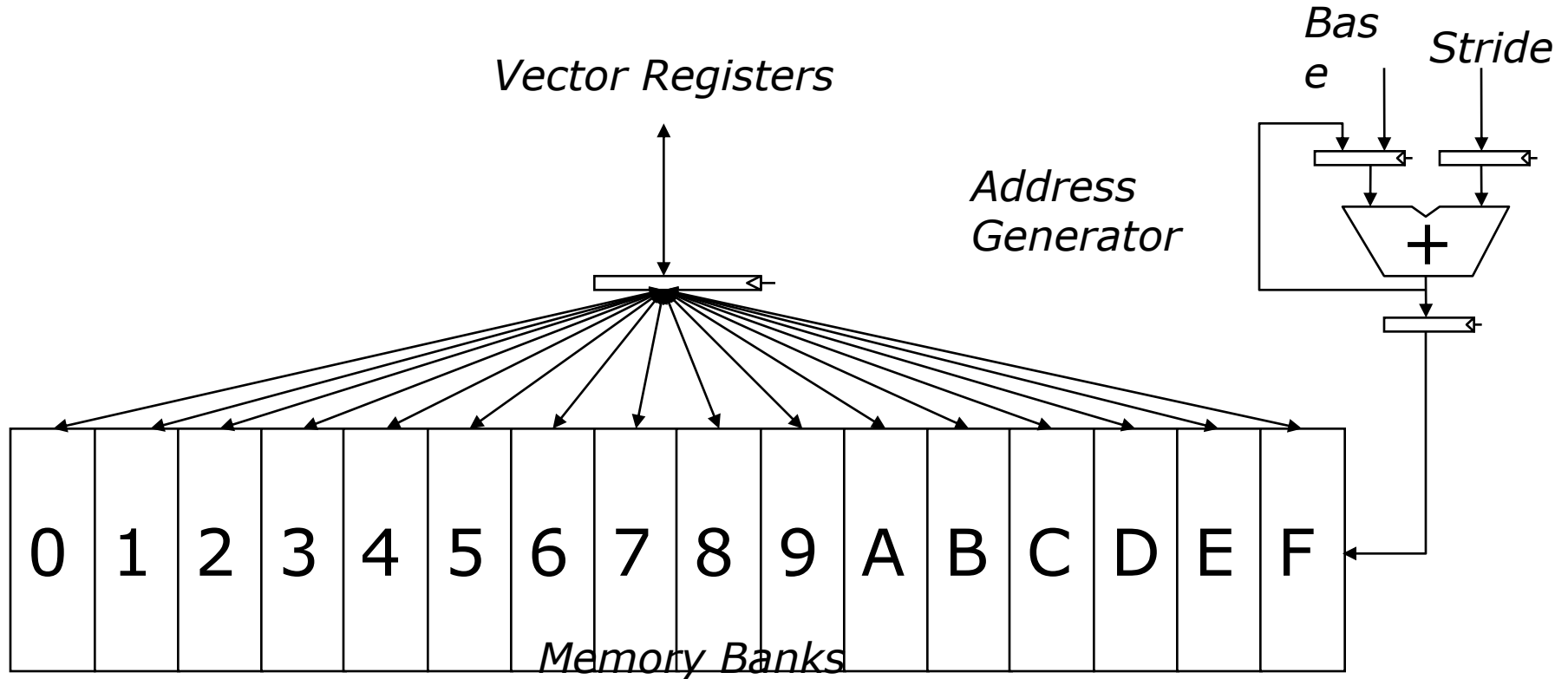
- CRAY-1
- Russell, “The CRAY-1 computer system,” CACM 1978.
- Scalar and vector modes
- 8 64-element vector registers
- 64 bits per element
- 16 memory banks
- 8 64-bit scalar registers
- 8 24-bit address registers

Memory Banking

- Example: 16 banks; can start one bank access per cycle
- Bank latency: 11 cycles
- Can sustain 16 parallel accesses if they go to different banks



Vector Memory System



Scalar Code Example

- For I = 0 to 49
 - ▣ $C[i] = (A[i] + B[i]) / 2$

- Scalar code

MOVI R0 = 50	1	
MOVA R1 = A	1	304 dynamic instructions
MOVA R2 = B	1	
MOVA R3 = C	1	
X: LD R4 = MEM[R1++]	11	
LD R5 = MEM[R2++]	11	
ADD R6 = R4 + R5	4	
SHFR R7 = R6 >> 1	1	
ST MEM[R3++] = R7	11	
DECBNZ --R0, X	2	;decrement and branch if NZ

Scalar Code Execution Time

- Scalar execution time on an in-order processor with 1 bank
 - First two loads in the loop cannot be pipelined: 2×11 cycles
 - $4 + 50 \times 40 = 2004$ cycles
- Scalar execution time on an in-order processor with 16 banks (word-interleaved)
 - First two loads in the loop can be pipelined
 - $4 + 50 \times 30 = 1504$ cycles
- Why 16 banks?
 - 11 cycle memory access latency
 - Having 16 (>11) banks ensures there are enough banks to overlap enough memory operations to cover memory latency

Vectorizable Loops

- A loop is **vectorizable** if each iteration is independent of any other
 - For I = 0 to 49
 - $C[i] = (A[i] + B[i]) / 2$
- 7 dynamic instructions

- Vectorized loop:

MOVI VLEN = 50	1
MOVI VSTR = 1	1
VLD V0 = A	11 + VLN - 1
VLD V1 = B	11 + VLN - 1
VADD V2 = V0 + V1	4 + VLN - 1
VSHFR V3 = V2 >> 1	1 + VLN - 1
VST C = V3	11 + VLN - 1

Vector Code Performance

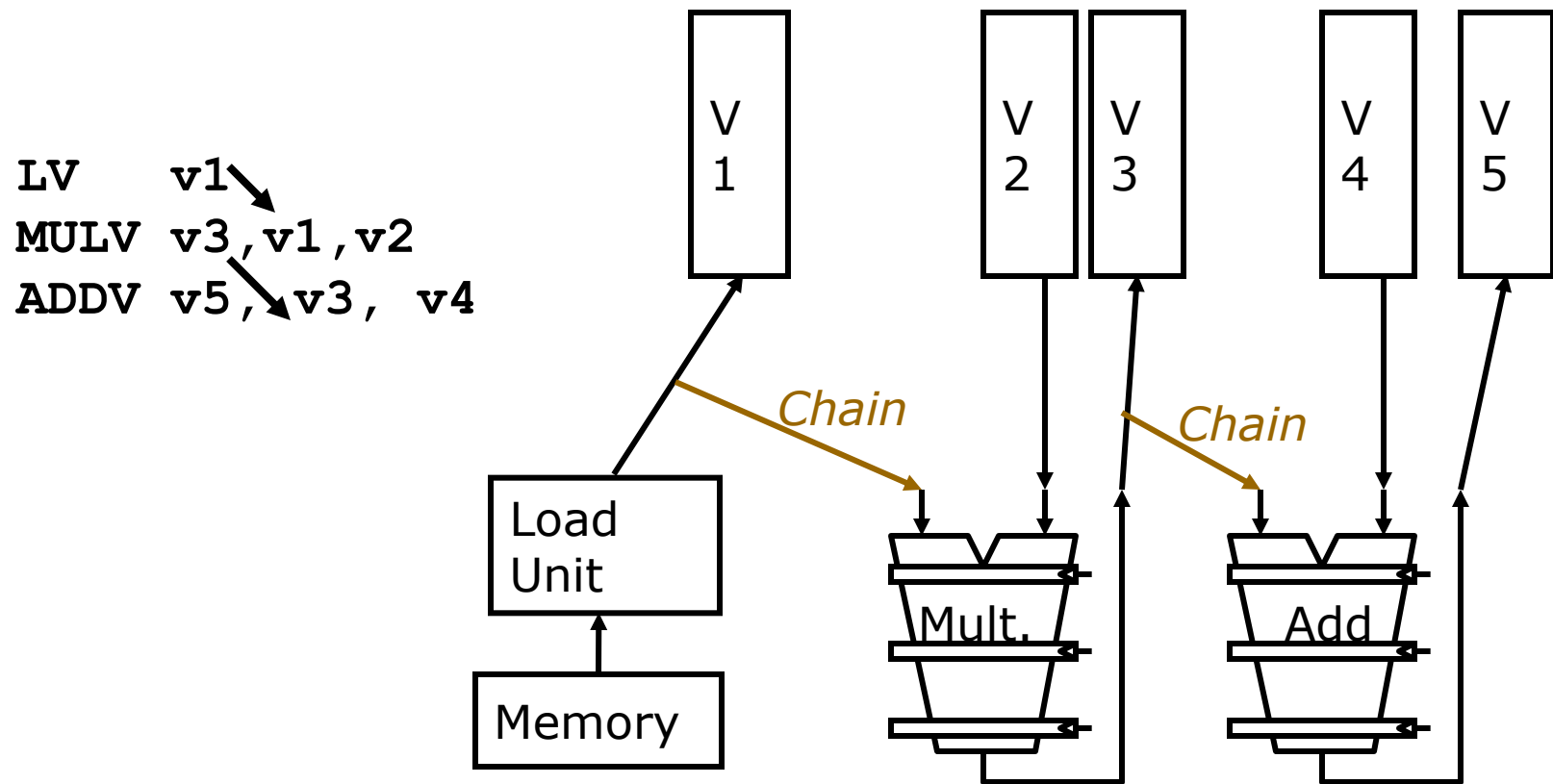
- No chaining
 - i.e., output of a vector functional unit cannot be used as the input of another (i.e., no vector data forwarding)
- One memory port (one address generator)
- 16 memory banks (word-interleaved)



- 285 cycles

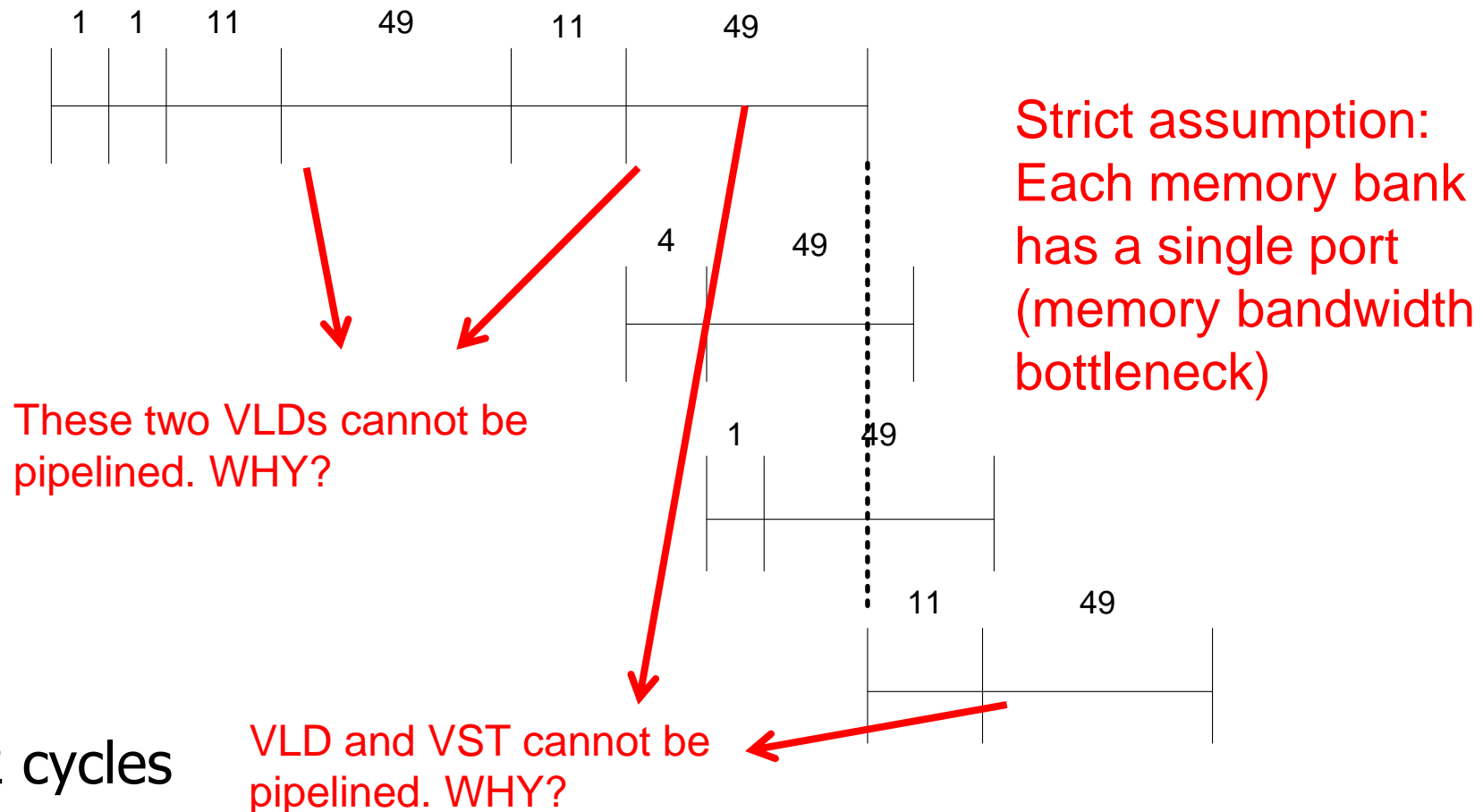
Vector Chaining

- **Vector chaining:** Data forwarding from one vector functional unit to another



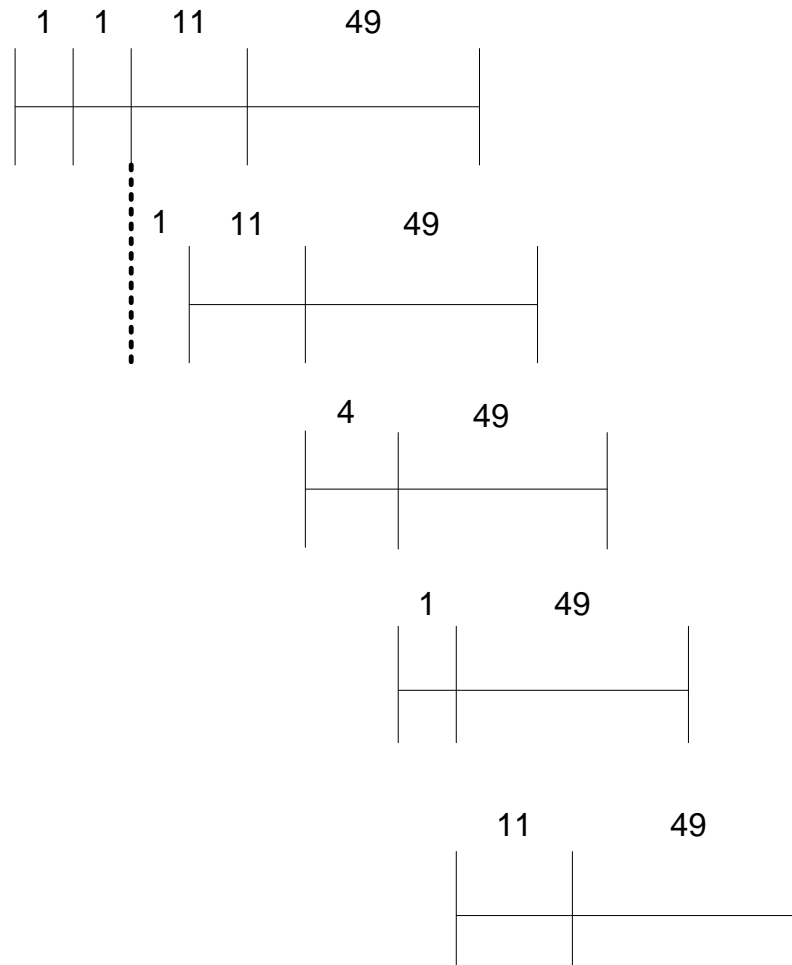
Vector Code Performance - Chaining

- **Vector chaining:** Data forwarding from one vector functional unit to another



Vector Code Performance – Multiple Memory Ports

- Chaining and 2 load ports, 1 store port in each bank



- 79 cycles

Questions (I)

- What if # data elements > # elements in a vector register?
 - Need to break loops so that each iteration operates on # elements in a vector register
 - E.g., 527 data elements, 64-element VREGs
 - 8 iterations where $VLEN = 64$
 - 1 iteration where $VLEN = 15$ (need to change value of $VLEN$)
 - Called **vector stripmining**

- What if vector data is not stored in a strided fashion in memory? (irregular memory access to a vector)
 - Use indirection to combine elements into vector registers
 - Called **scatter/gather operations**

Gather/Scatter Operations

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
```

Gather/Scatter Operations

- Gather/scatter operations often implemented in hardware to handle sparse matrices
- Vector loads and stores use an index vector which is added to the base register to generate the addresses

Index Vector	Data Vector	Equivalent
1	3.14	3.14
3	6.5	0.0
7	71.2	6.5
8	2.71	0.0
		0.0
		0.0
		0.0
		0.0
		71.2
		2.7

Conditional Operations in a Loop

- What if some operations should not be executed on a vector (based on a dynamically-determined condition)?

```
loop:      if (a[i] != 0) then b[i]=a[i]*b[i]
           goto loop
```

- Idea: **Masked operations**

- VMASK register is a bit mask determining which data element should not be acted upon

VLD V0 = A

VLD V1 = B

VMASK = (V0 != 0)

VMUL V1 = V0 * V1

VST B = V1

- Does this look familiar? This is essentially **predicated execution**.

Another Example with Masking

```
for (i = 0; i < 64; ++i)
  if (a[i] >= b[i]) then c[i] = a[i]
  else c[i] = b[i]
```

A	B	VMASK
1	2	0
2	2	1
3	2	1
4	10	0
-5	-4	0
0	-3	1
6	5	1
-7	-8	1

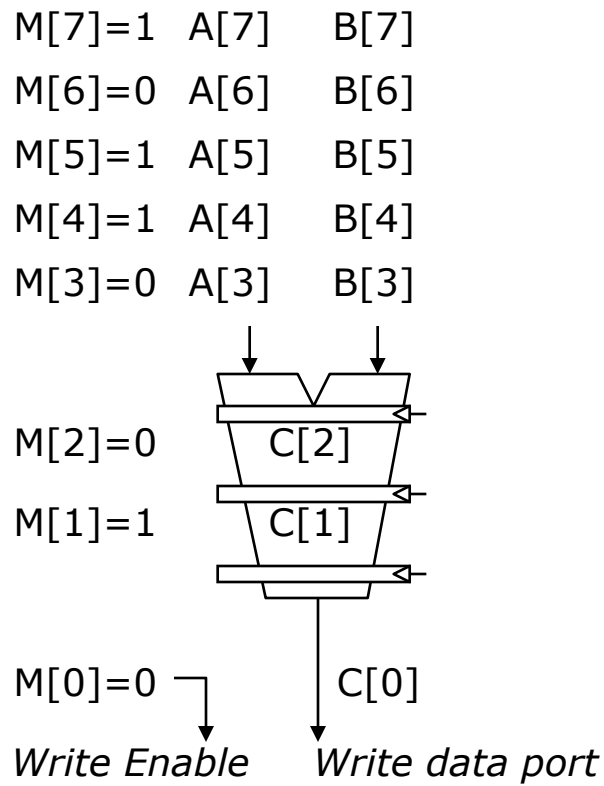
Steps to execute loop

1. Compare A, B to get VMASK
2. Masked store of A into C
3. Complement VMASK
4. Masked store of B into C

Masked Vector Instructions

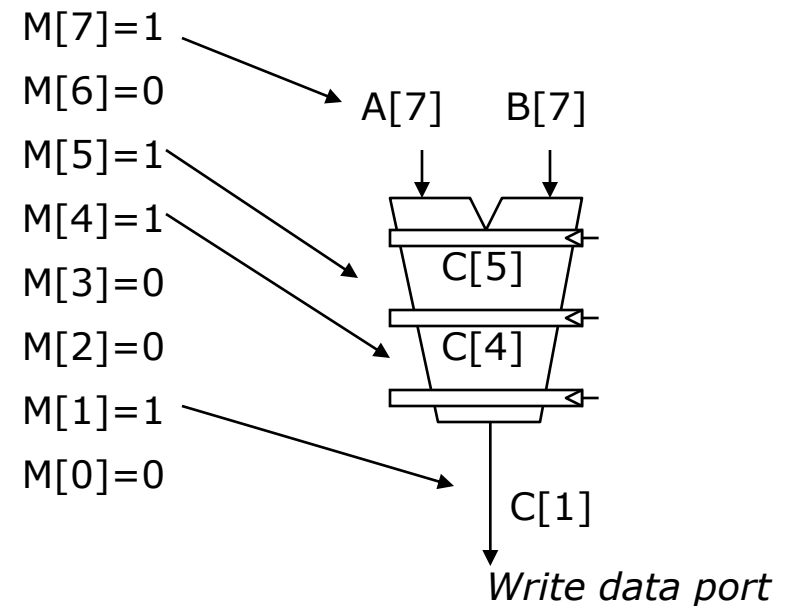
Simple Implementation

- 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



Some Issues

- Stride and banking
 - As long as they are relatively prime to each other and there are enough banks to cover bank access latency, consecutive accesses proceed in parallel
- Storage of a matrix
 - **Row major**: Consecutive elements in a row are laid out consecutively in memory
 - **Column major**: Consecutive elements in a column are laid out consecutively in memory
 - You need to change the stride when accessing a row versus column

Matrix multiplication

A & B, both in row major order

A₀

0	1	2	3	4	5
6	7	8	9	10	11

B₀

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20									
30									
40									
50									

$A_{6 \times 6} B_{6 \times 10} \rightarrow C_{6 \times 10}$ (dot products of rows & columns of A & B)

A: Load A₀ into a vector register V1

→ each time you need to increment the address by 1 to access the next column

→ First matrix accesses have a stride of 1

B: Load B₀ into a vector register V2

→ each time you need to increment by 10

→ stride of 10

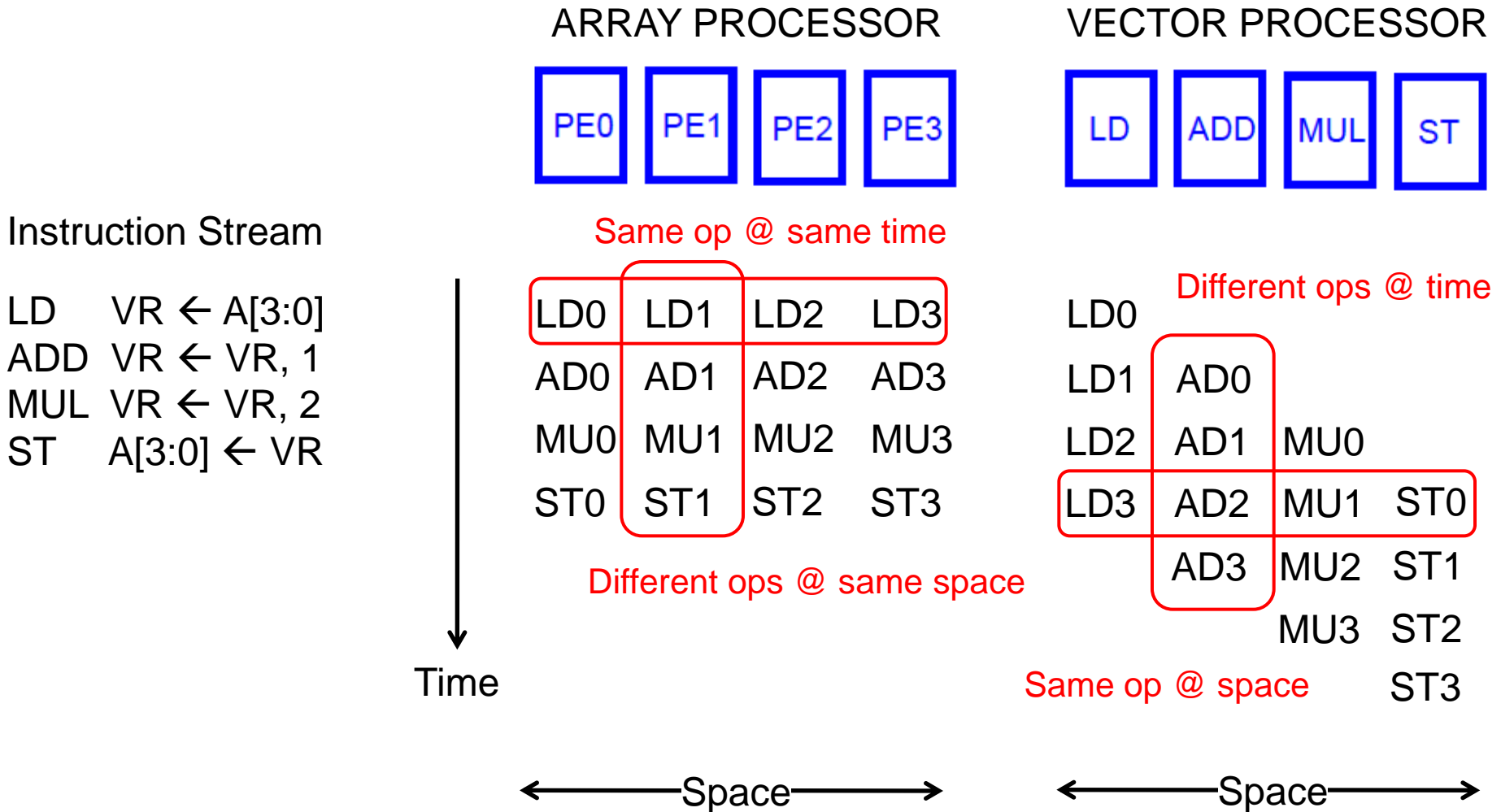
Different strides can lead to bank conflicts.

→ How do you minimize them?

Array vs. Vector Processors, Revisited

- Array vs. vector processor distinction is a “purist’s” distinction
- Most “modern” SIMD processors are a combination of both
 - They exploit data parallelism in both time and space

Remember: Array vs. Vector Processors

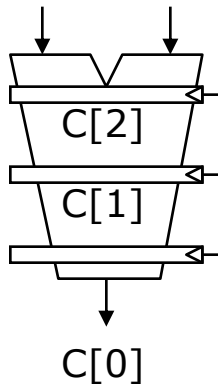


Vector Instruction Execution

ADDV C,A,B

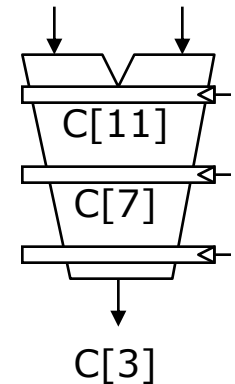
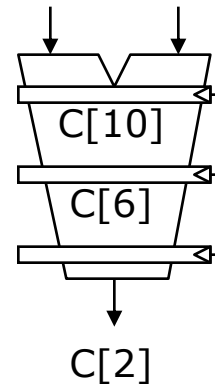
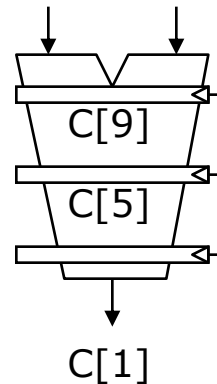
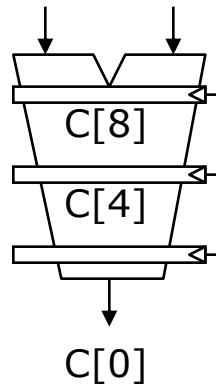
*Execution using
one pipelined
functional unit*

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

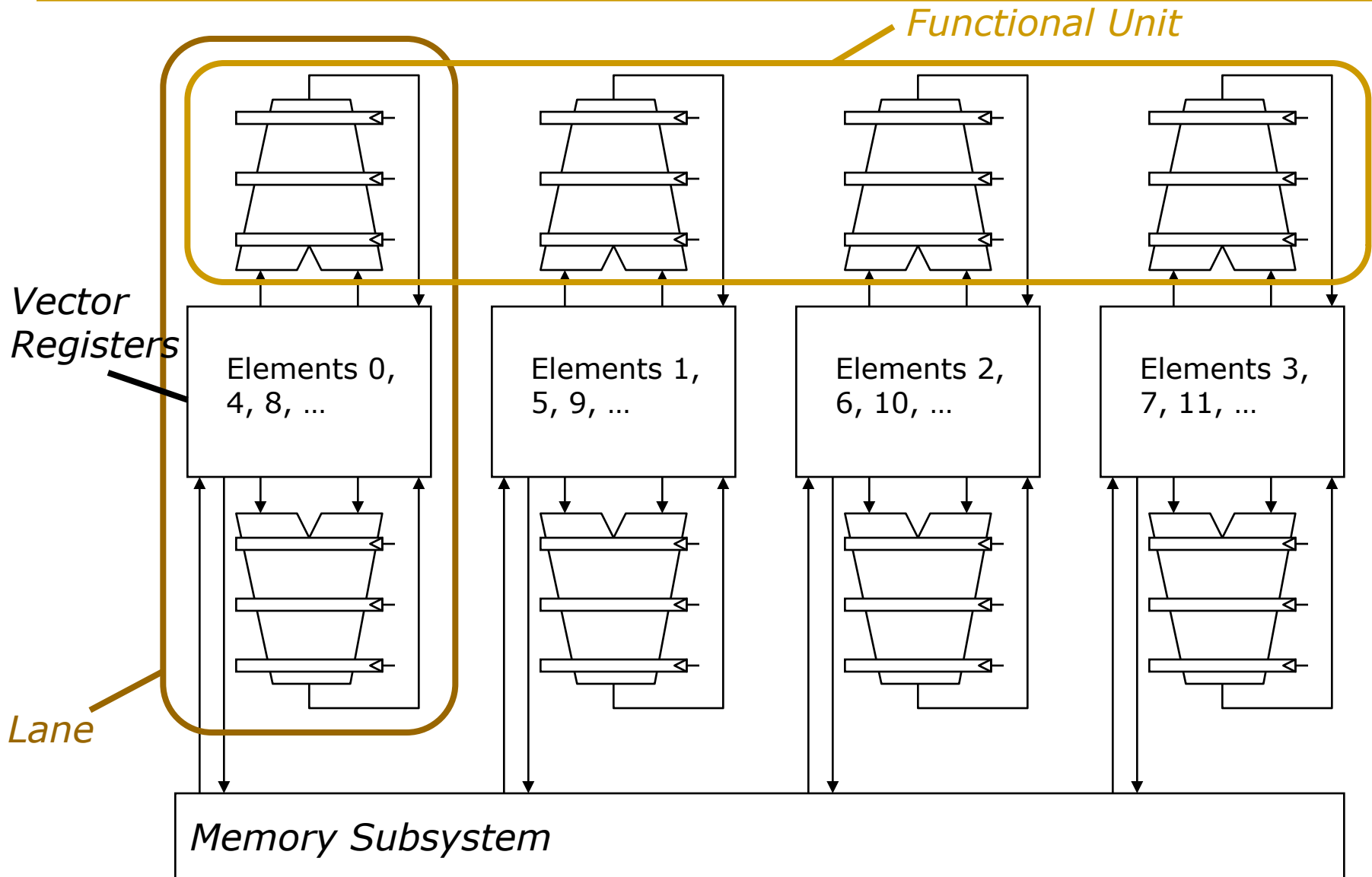


*Execution using
four pipelined
functional units*

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]



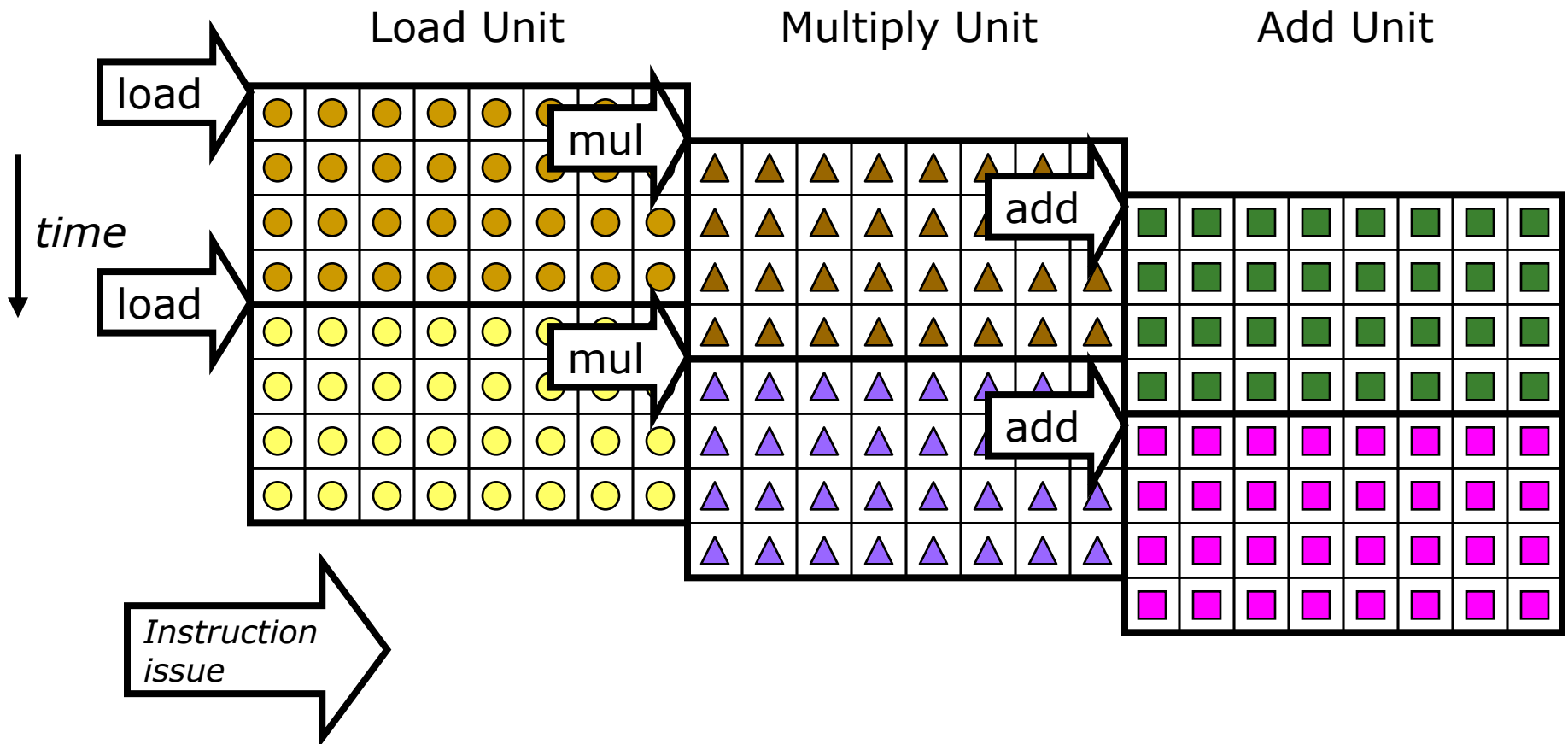
Vector Unit Structure



Vector Instruction Level Parallelism

Can overlap execution of multiple vector instructions

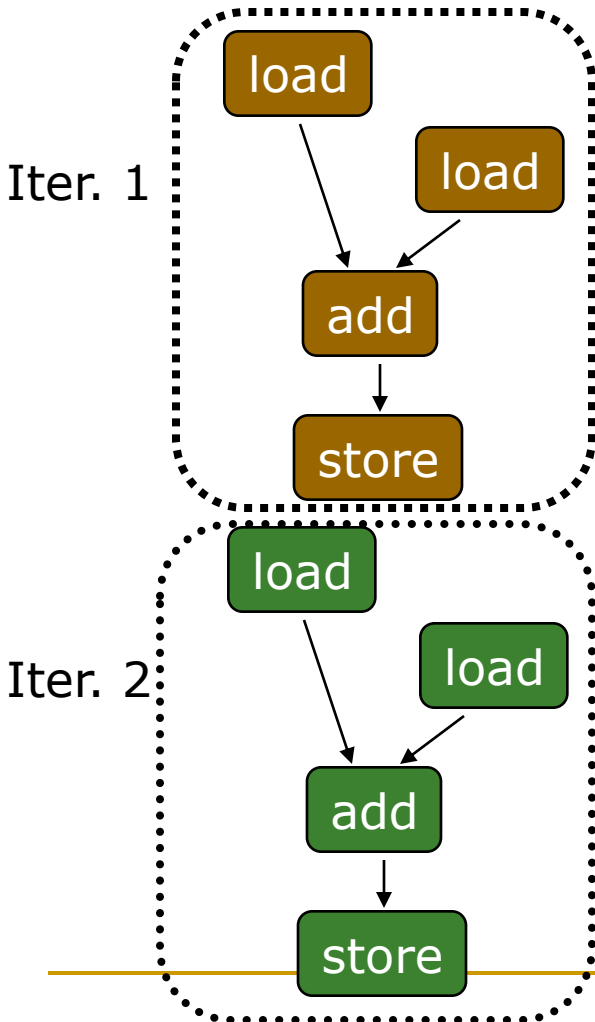
- example machine has 32 elements per vector register and 8 lanes
- Complete 24 operations/cycle while issuing 1 short instruction/cycle



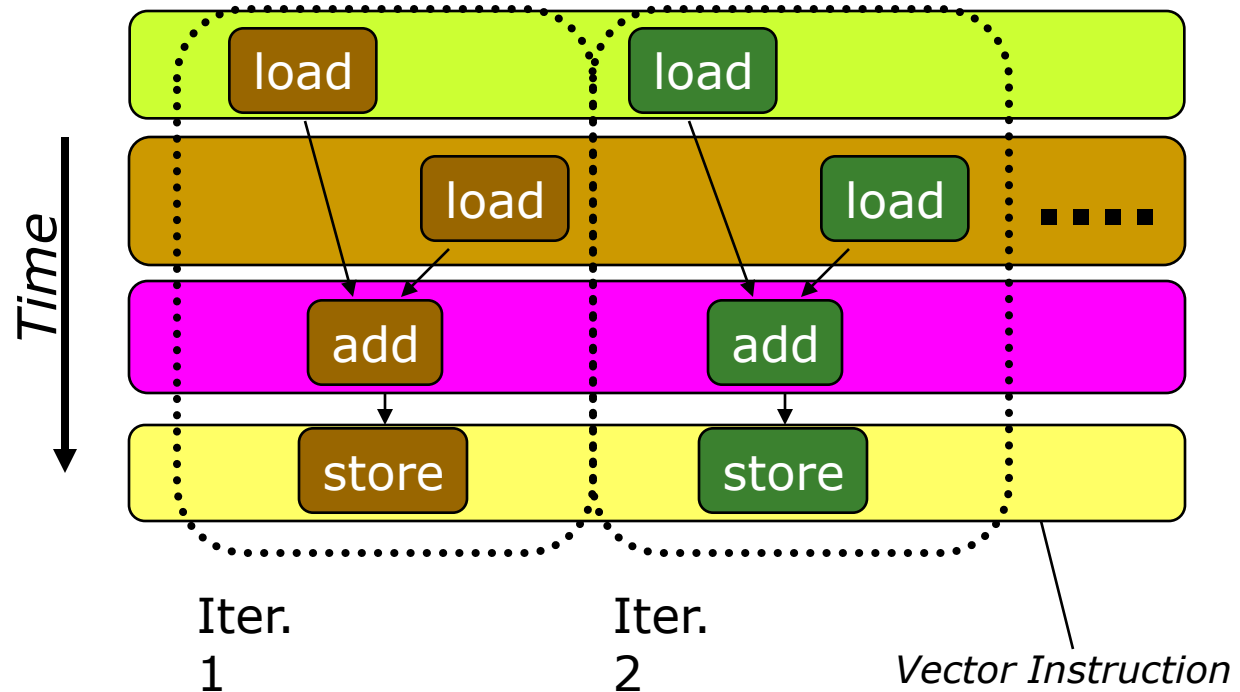
Automatic Code Vectorization

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

Scalar Sequential Code



Vectorized Code



Vectorization is a compile-time reordering of operation sequencing
⇒ requires extensive loop dependence analysis

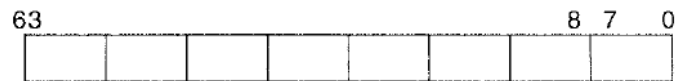
Vector/SIMD Processing Summary

- Vector/SIMD machines good at exploiting **regular data-level parallelism**
 - Same operation performed on many data elements
 - Improve performance, simplify design (no intra-vector dependencies)
- **Performance improvement limited by vectorizability** of code
 - Scalar operations limit vector machine performance
 - Amdahl's Law
 - CRAY-1 was the fastest SCALAR machine at its time!
- Many existing ISAs include (vector-like) SIMD operations
 - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD

SIMD Operations in Modern ISAs

Intel Pentium MMX Operations

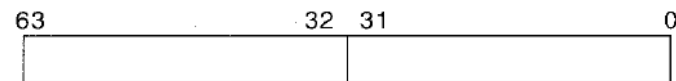
- Idea: One instruction operates on multiple data elements **simultaneously**
 - ❑ Ala array processing (yet much more limited)
 - ❑ Designed with multimedia (graphics) operations in mind



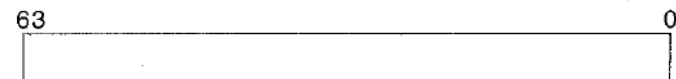
(a)



(b)



(c)



(d)

No VLEN register

Opcode determines data type:

8 8-bit bytes

4 16-bit words

2 32-bit doublewords

1 64-bit quadword

Stride always equal to 1.

Peleg and Weiser, “**MMX Technology Extension to the Intel Architecture**,”
IEEE Micro, 1996.

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

MMX Example: Image Overlaying (I)



Figure 8. Chroma keying: image overlay using a background color.

PCMPEQB MM1, MM3

MM1	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue
MM3	X7!=blue	X6!=blue	X5=blue	X4=blue	X3!=blue	X2!=blue	X1=blue	X0=blue
MM1	0x0000	0x0000	0xFFFF	0xFFFF	0x0000	0x0000	0xFFFF	0xFFFF



Bitmask

Figure 9. Generating the selection bit mask.

MMX Example: Image Overlaying (II)

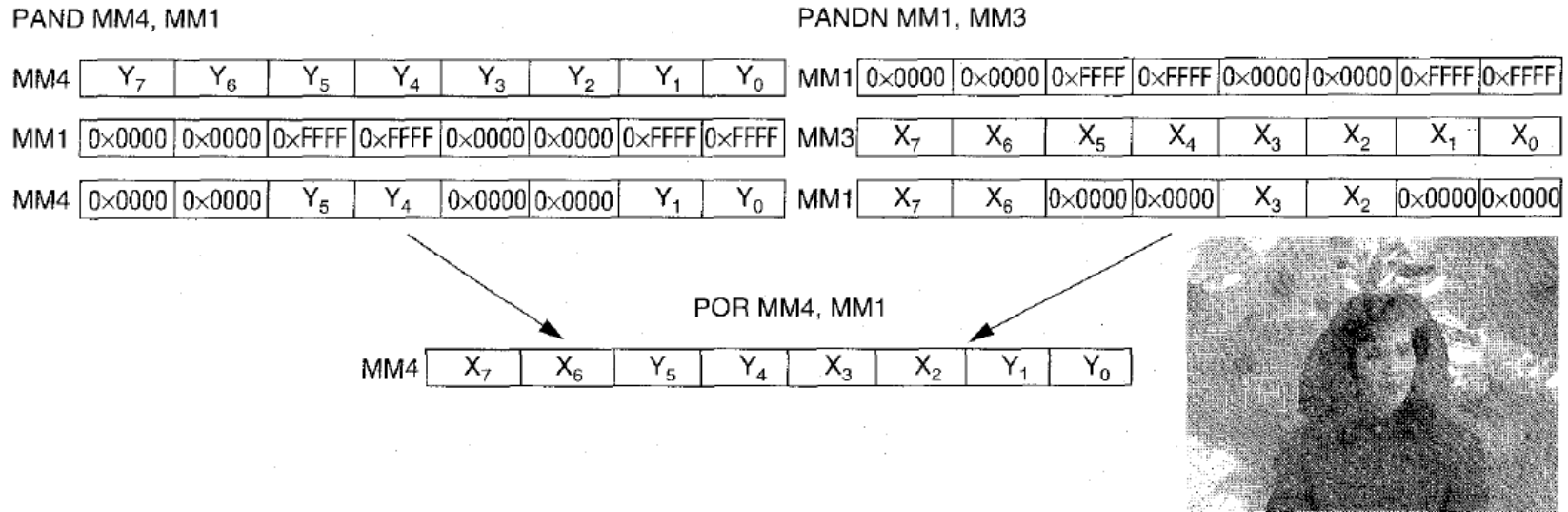


Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

```

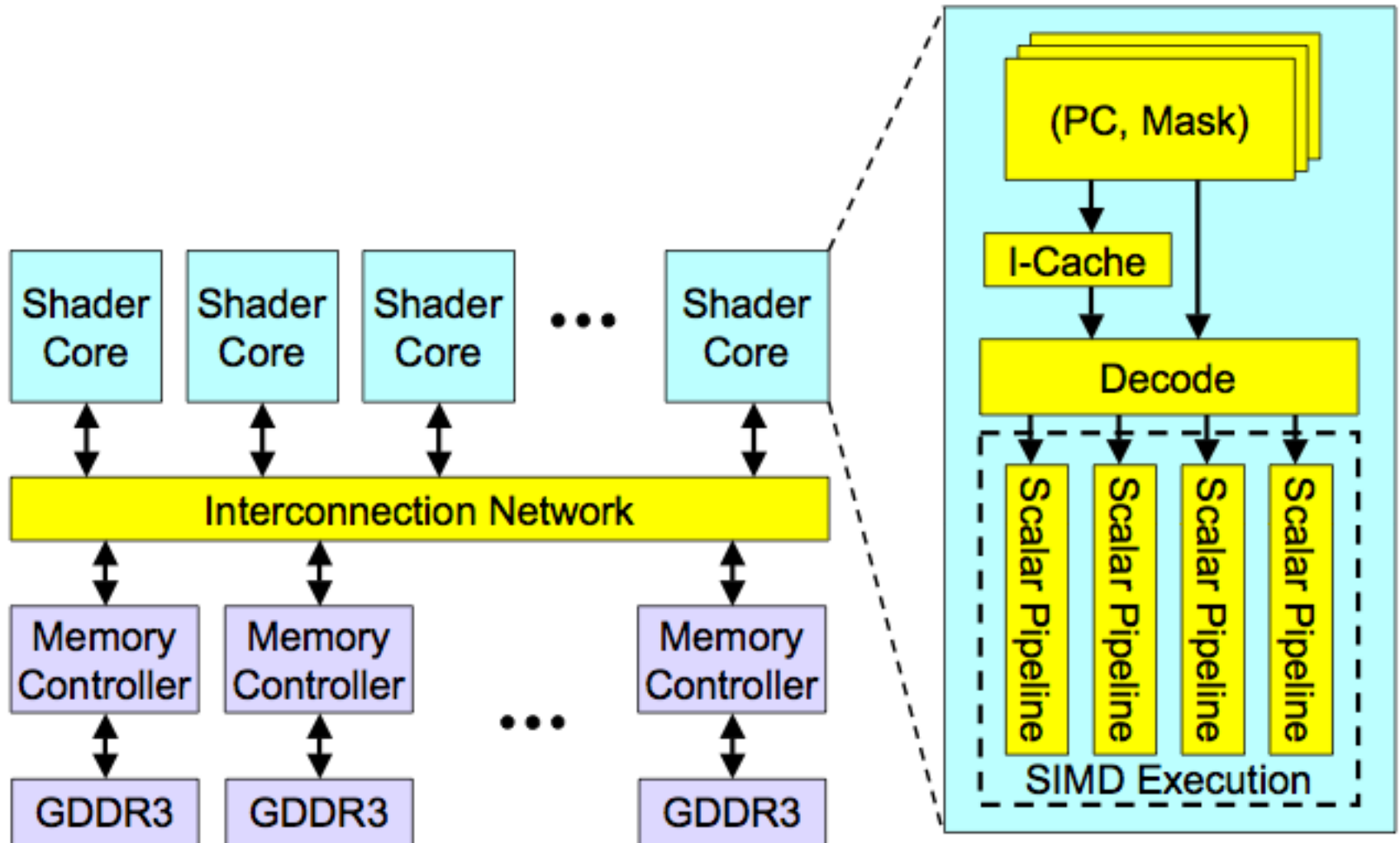
Movq    mm3, mem1    /* Load eight pixels from
                        woman's image
Movq    mm4, mem2    /* Load eight pixels from the
                        blossom image
Pcmpeqb mm1, mm3
Pand    mm4, mm1
Pandn   mm1, mm3
Por     mm4, mm1
    
```

Figure 11. MMX code sequence for performing a conditional select.

Graphics Processing Units

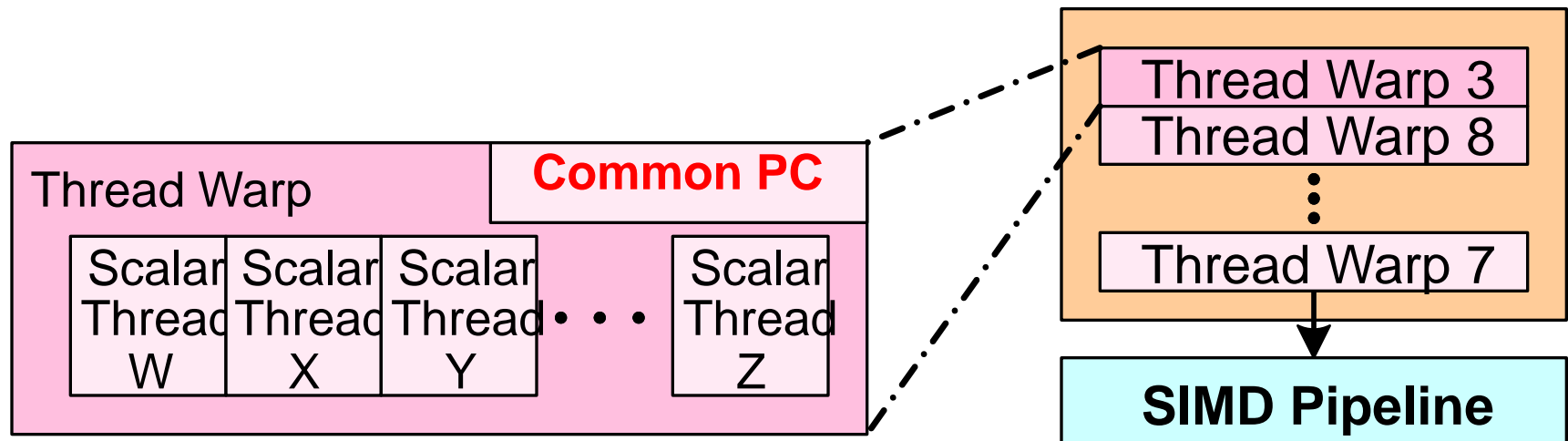
SIMD not Exposed to Programmer (SIMT)

High-Level View of a GPU



Concept of “Thread Warps” and SIMT

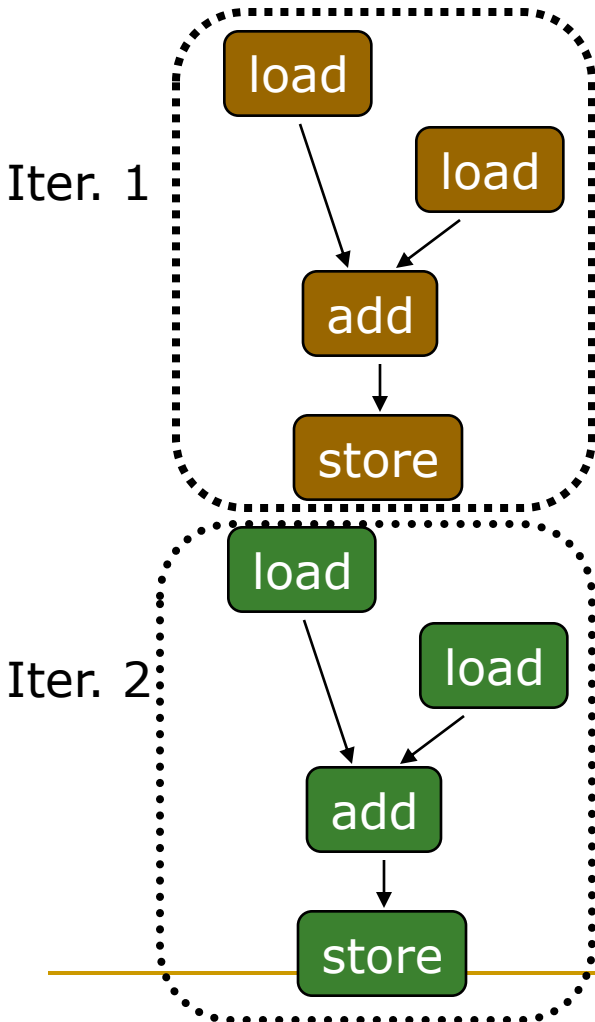
- Warp: A set of threads that execute the same instruction (on different data elements) → SIMT (Nvidia-speak)
- All threads run the same kernel
- Warp: The threads that run lengthwise in a woven fabric ...



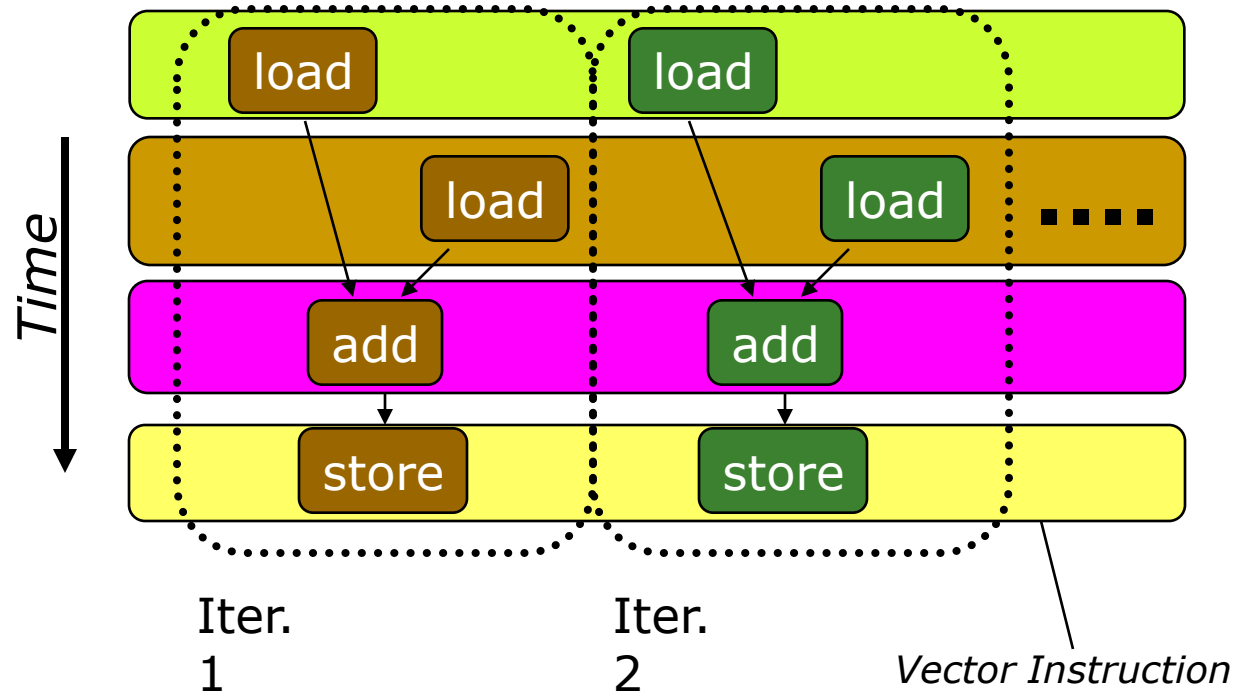
Loop Iterations as Threads

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

Scalar Sequential Code



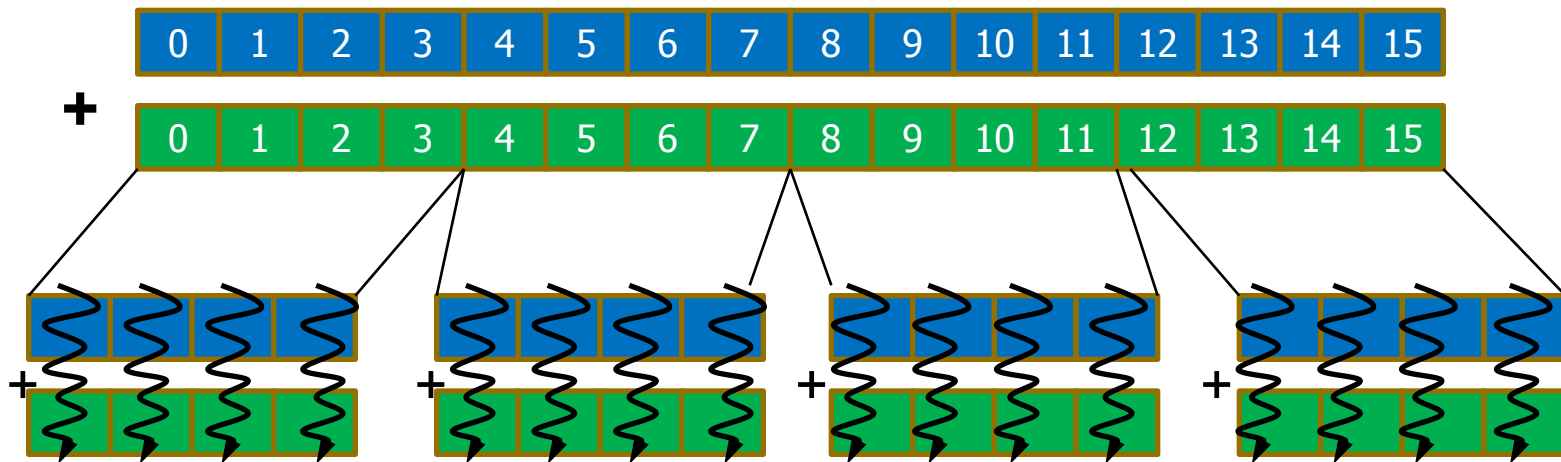
Vectorized Code



SIMT Memory Access

- Same instruction in different threads uses thread id to index and access different data elements

Let's assume $N=16$, $\text{blockDim}=4 \rightarrow 4$ blocks



Sample GPU SIMT Code (Simplified)

CPU code

```
for (ii = 0; ii < 100; ++ii) {  
    C[ii] = A[ii] + B[ii];  
}
```



CUDA code

```
// there are 100 threads  
__global__ void KernelFunction(...) {  
    int tid = blockDim.x * blockIdx.x + threadIdx.x;  
    int varA = aa[tid];  
    int varB = bb[tid];  
    C[tid] = varA + varB;  
}
```

Sample GPU Program (Less Simplified)

CPU Program

```
void add matrix
( float *a, float* b, float *c, int N) {
    int index;
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j) {
            index = i + j*N;
            c[index] = a[index] + b[index];
        }
}

int main () {

    add matrix (a, b, c, N);
}
```

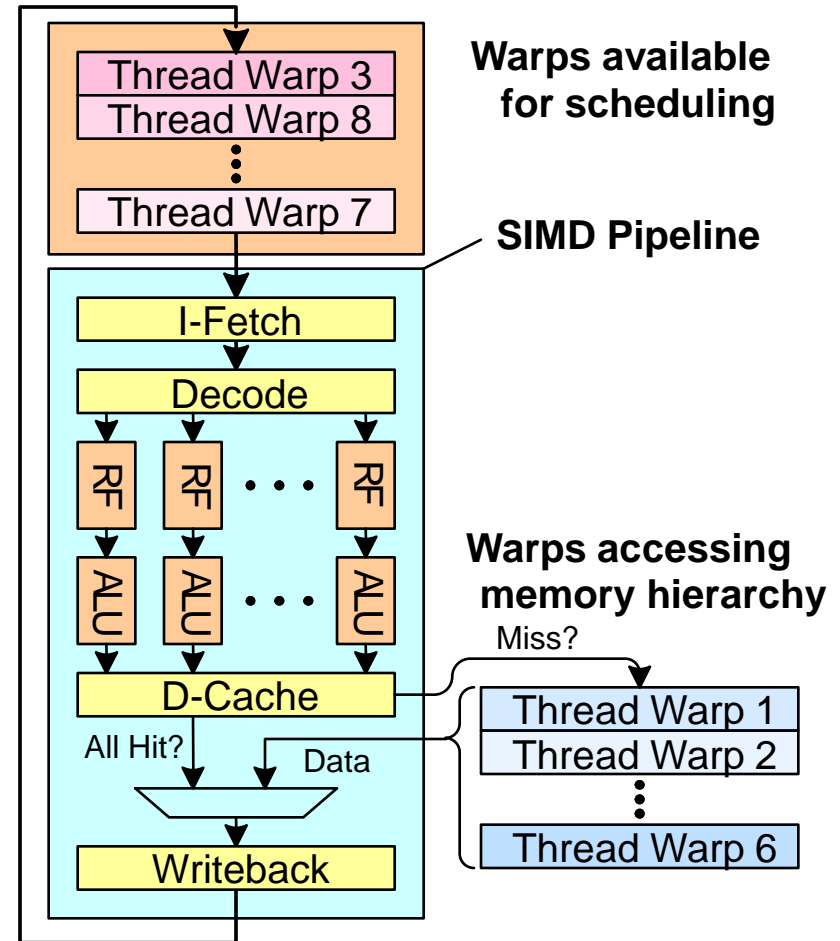
GPU Program

```
__global__ add_matrix
( float *a, float *b, float *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if (i < N && j < N)
        c[index] = a[index]+b[index];
}

int main() {
    dim3 dimBlock( blocksize, blocksize) ;
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N);
}
```

Latency Hiding with “Thread Warps”

- Warp: A set of threads that execute the same instruction (on different data elements)
- Fine-grained multithreading
 - One instruction per thread in pipeline at a time (No branch prediction)
 - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- No OS context switching
- Memory latency hiding
 - Graphics has millions of pixels



Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD contains a single thread
 - Lock step
 - Programming model is SIMD (no threads) → SW needs to know vector length
 - ISA contains vector/SIMD instructions

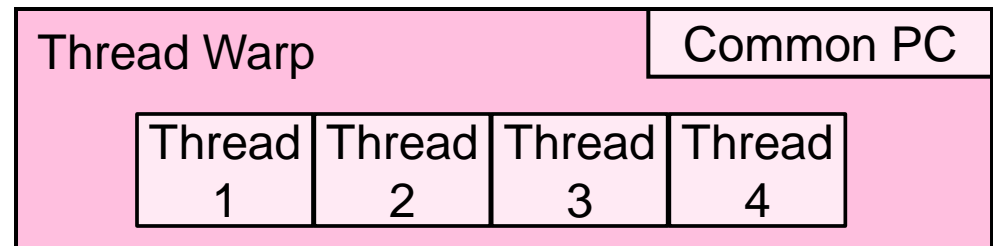
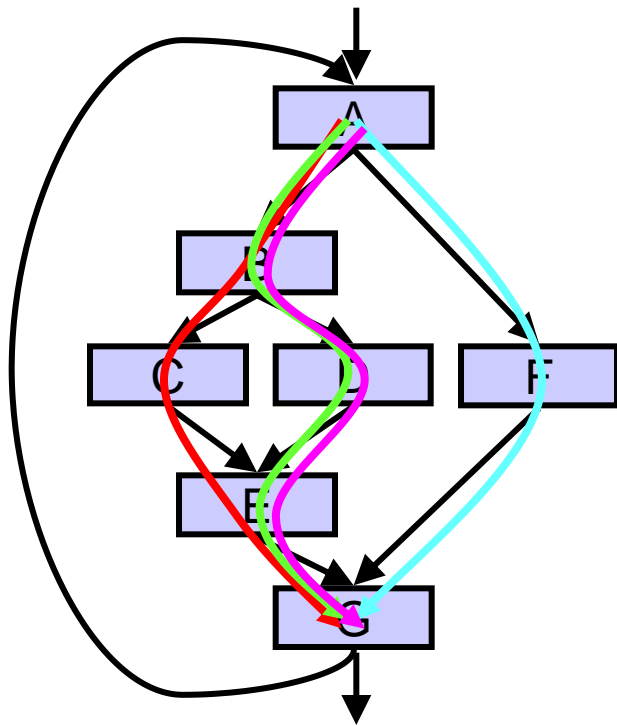
- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
 - Does not have to be lock step
 - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
 - SW does not need to know vector length
 - Enables memory and branch latency tolerance
 - ISA is scalar → vector instructions formed dynamically
 - Essentially, it is SPMD programming model implemented on SIMD hardware

SPMD

- Single procedure/program, multiple data
 - This is a programming model rather than computer organization
- Each processing element executes the same procedure, except on different data elements
 - Procedures can synchronize at certain points in program, e.g. barriers
- Essentially, multiple instruction streams execute the same program
 - Each program/procedure can 1) execute a different control-flow path, 2) work on different data, at run-time
 - Many scientific applications programmed this way and run on MIMD computers (multiprocessors)
 - Modern GPUs programmed in a similar way on a SIMD computer

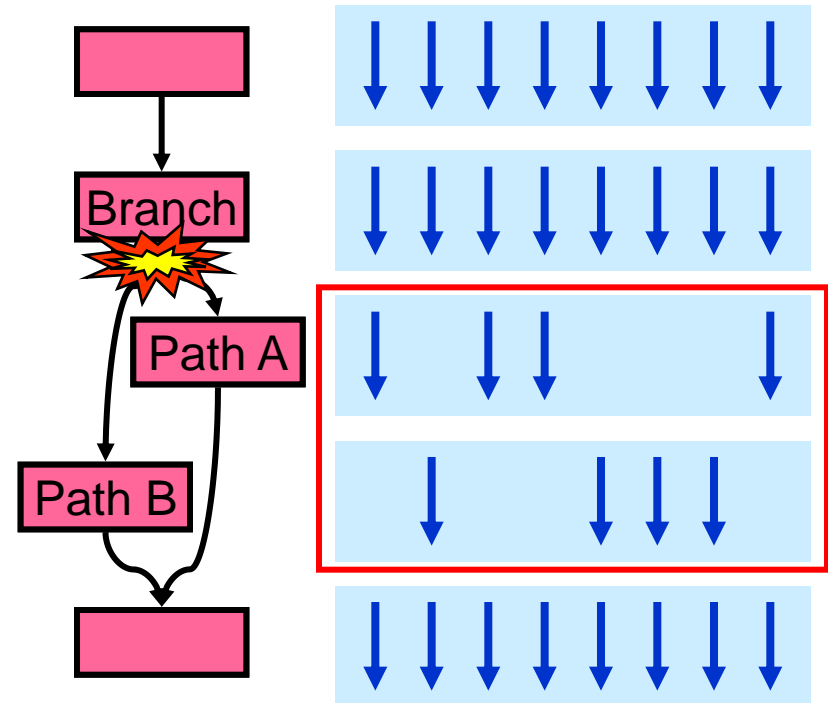
Branch Divergence Problem in Warp-based SIMD

- SPMD Execution on SIMD Hardware
 - NVIDIA calls this “Single Instruction, Multiple Thread” (“SIMT”) execution

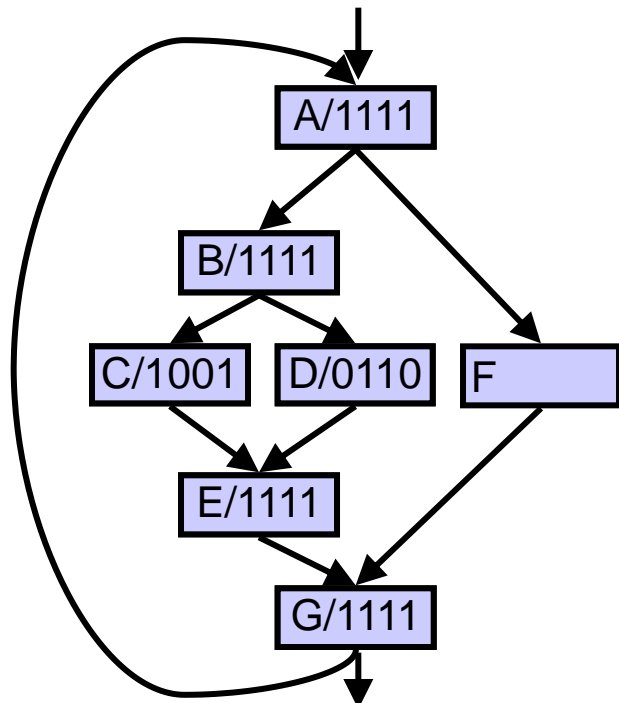


Control Flow Problem in GPUs/SIMD

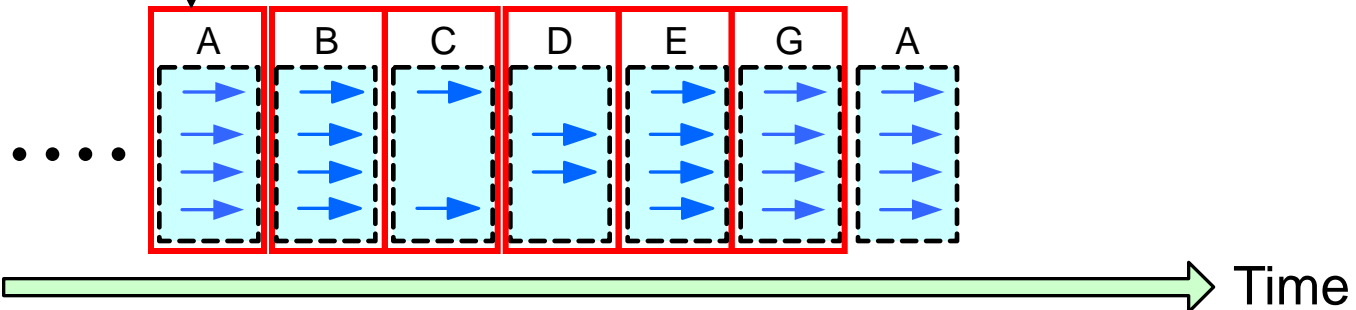
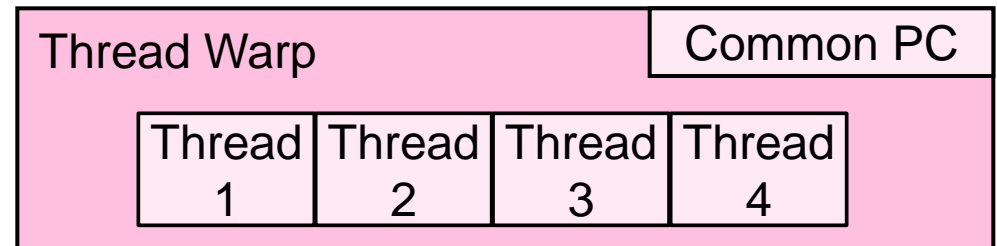
- GPU uses SIMD pipeline to save area on control logic.
 - Group scalar threads into warps
- Branch divergence occurs when threads inside warps branch to different execution paths.



Branch Divergence Handling (I)

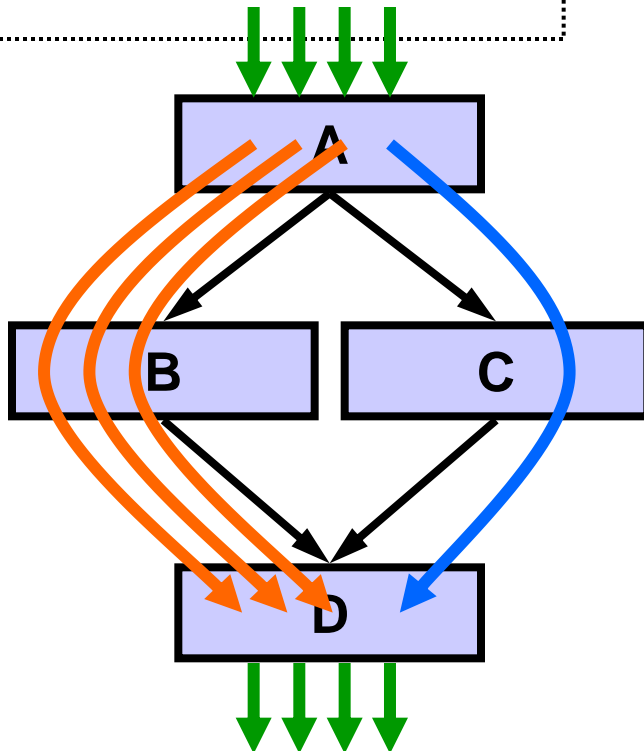


Stack			
	Reconv. PC	Next PC	Active Mask
TOS →	-	E	1111
TOS →	E	D	0110
TOS →	E	E	1001



Branch Divergence Handling (II)

```
A;  
if (some condition) {  
    B;  
} else {  
    C;  
}  
D;
```

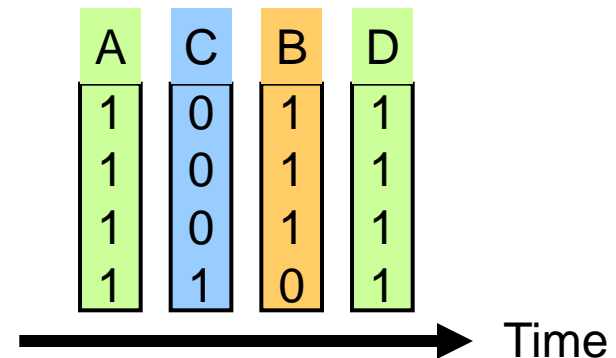


One per warp

Control Flow Stack

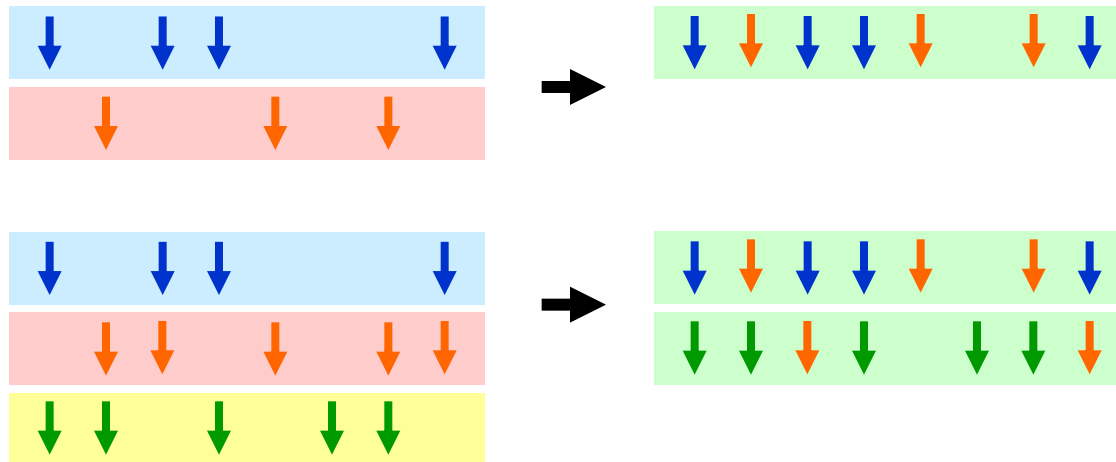
	Next PC	Recv PC	Amask
TOS →	D	--	1111
	B	D	1110
	D	D	0001

Execution Sequence



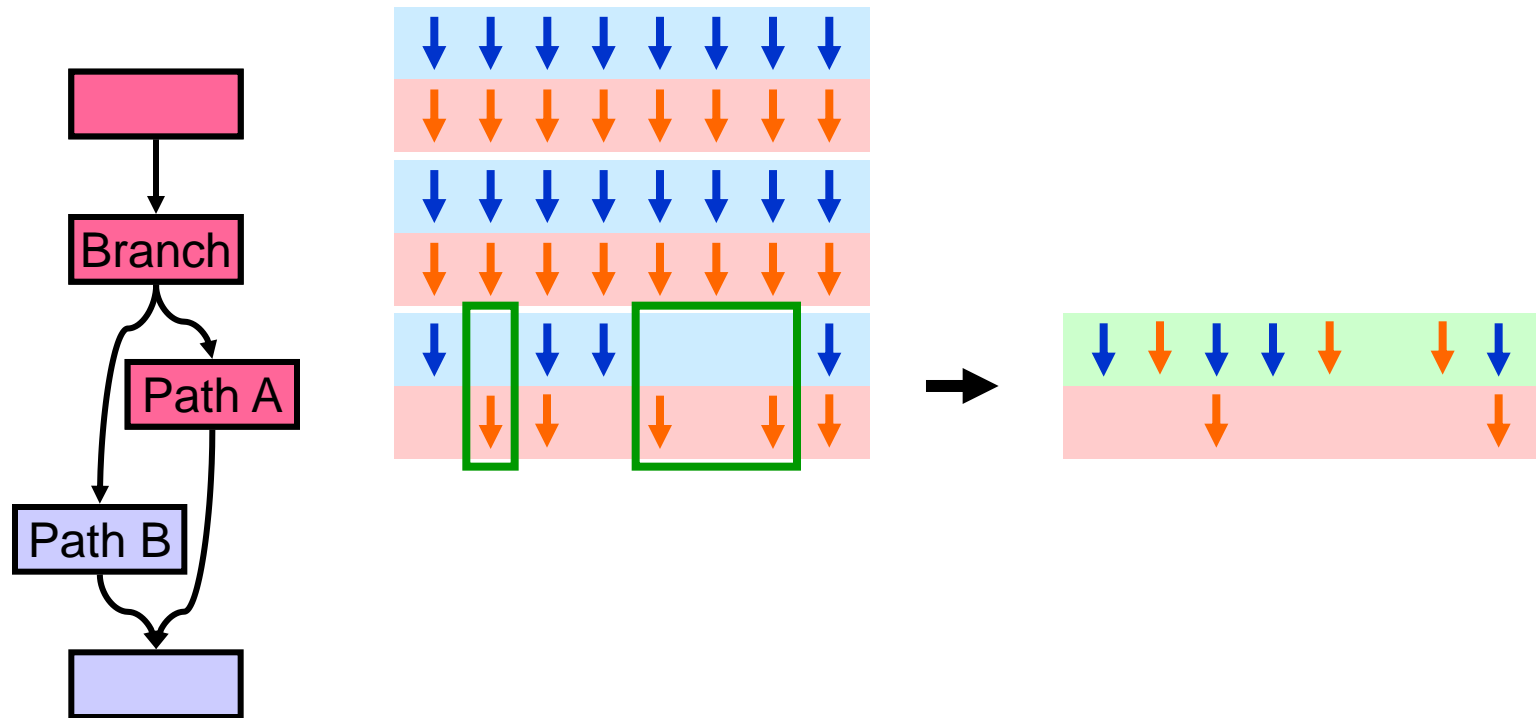
Dynamic Warp Formation

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)
- Form new warp at divergence
 - Enough threads branching to each path to create full new warps



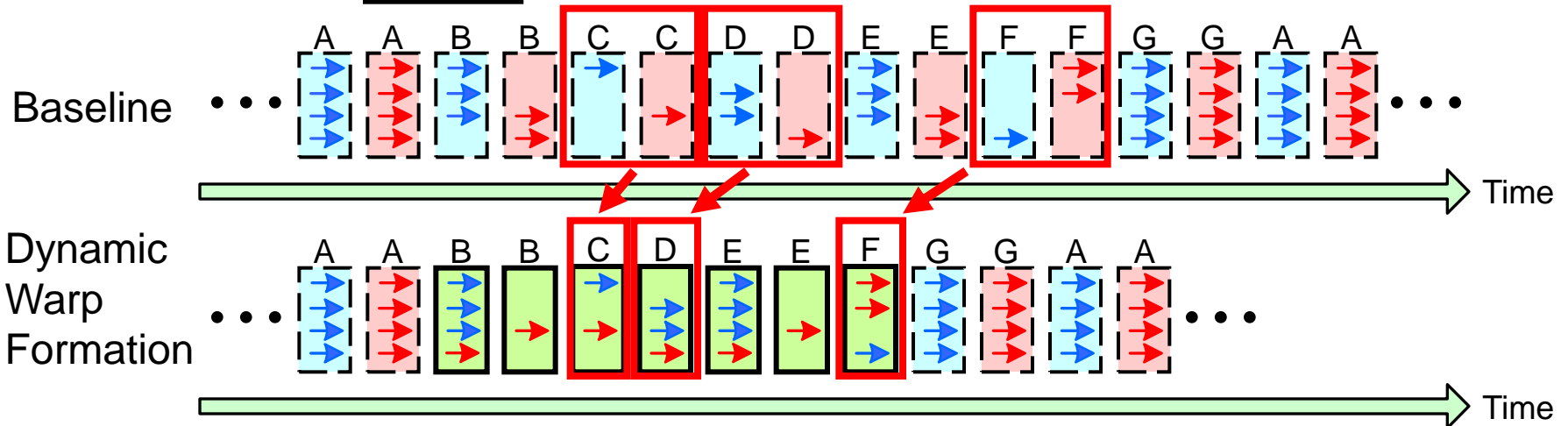
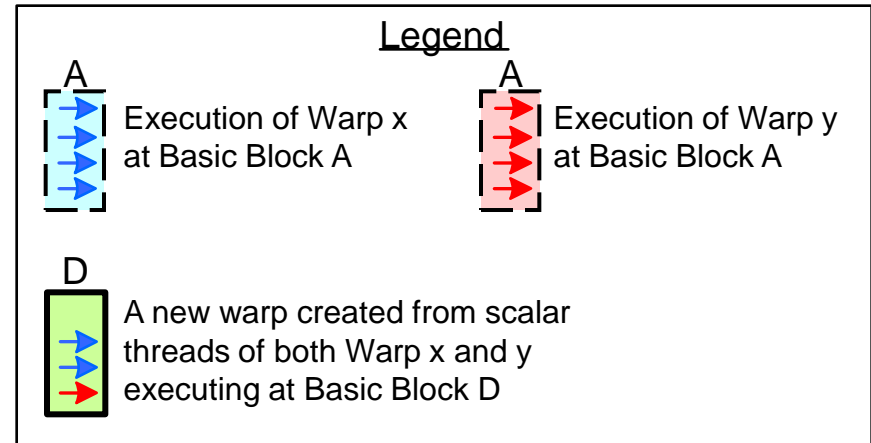
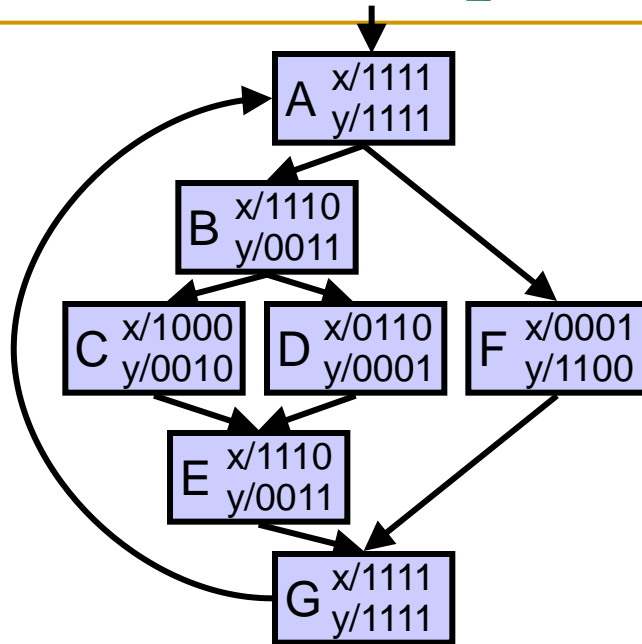
Dynamic Warp Formation/Merging

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)



- Fung et al., “Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow,” MICRO 2007.

Dynamic Warp Formation Example



What About Memory Divergence?

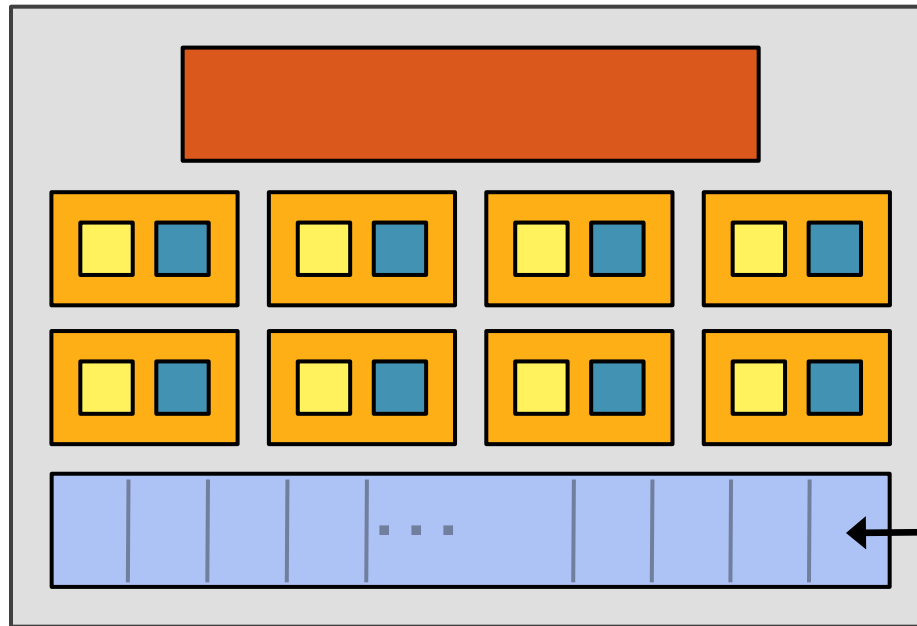
- Modern GPUs have caches
- Ideally: Want all threads in the warp to hit (without conflicting with each other)
- Problem: One thread in a warp can stall the entire warp if it misses in the cache.
- Need techniques to
 - Tolerate memory divergence
 - Integrate solutions to branch and memory divergence

NVIDIA GeForce GTX 285

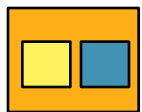
- NVIDIA-speak:
 - ❑ 240 stream processors
 - ❑ “SIMT execution”
- Generic speak:
 - ❑ 30 cores
 - ❑ 8 SIMD functional units per core



NVIDIA GeForce GTX 285 “core”



64 KB of storage
for fragment
contexts (registers)



= SIMD functional unit, control
shared across 8 units



= multiply-add



= multiply

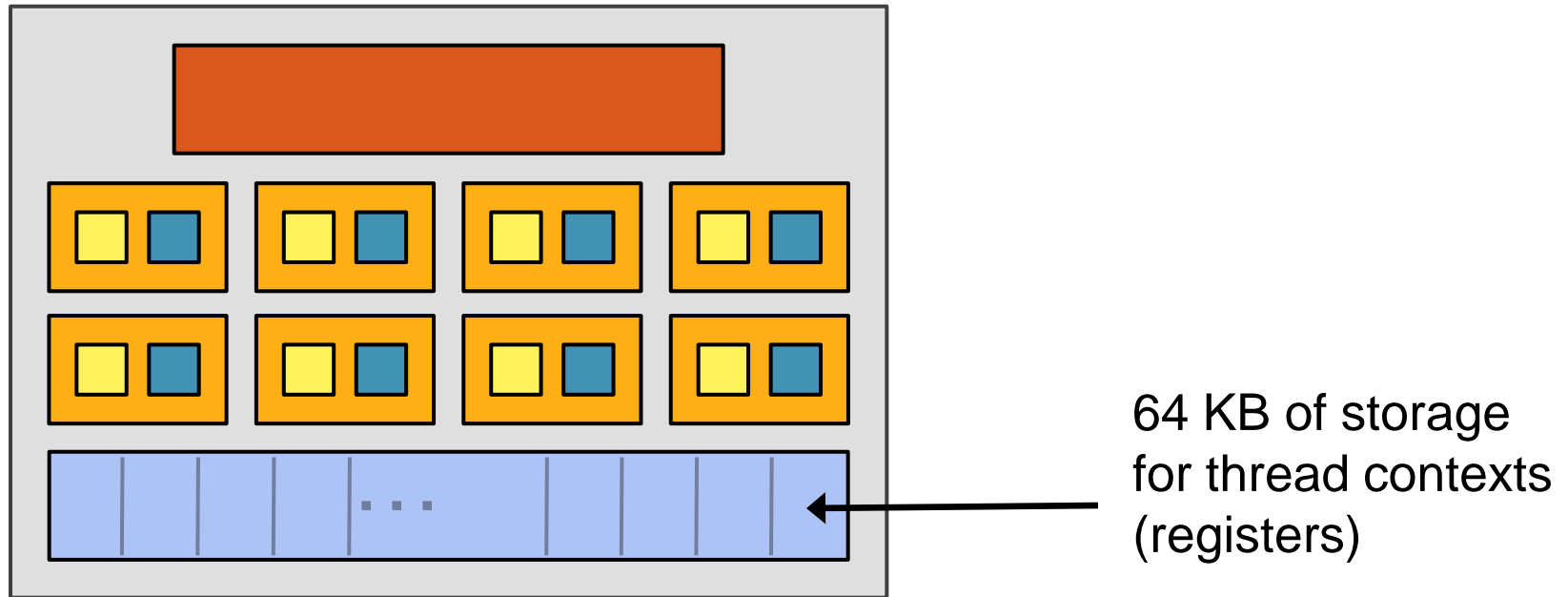


= instruction stream decode



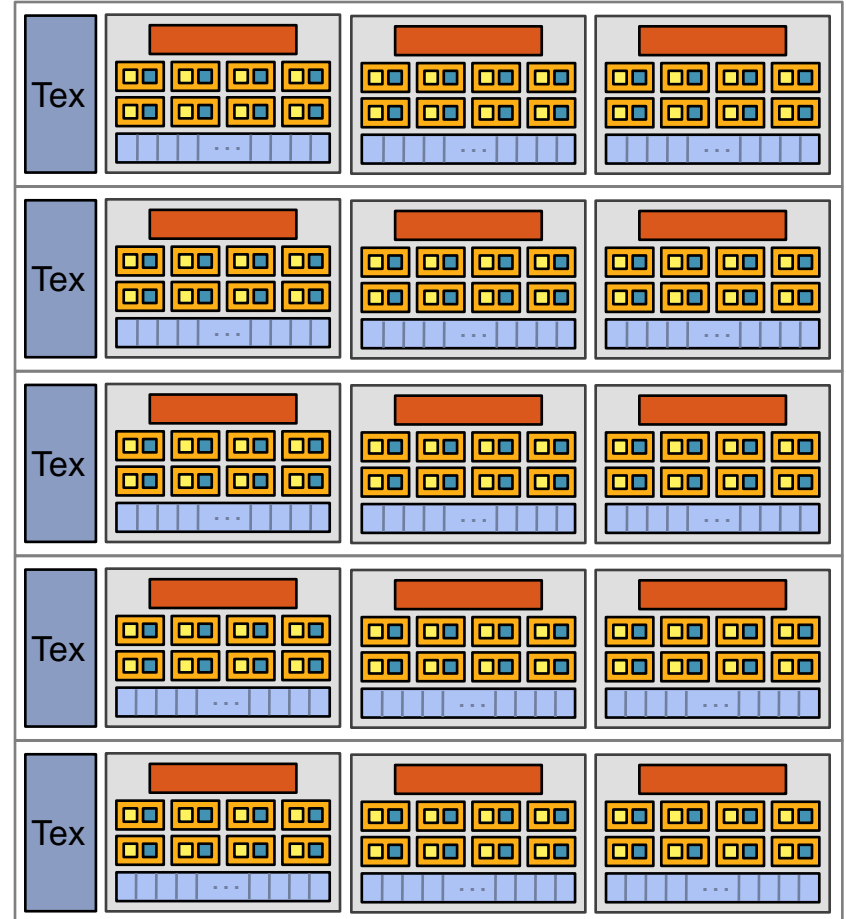
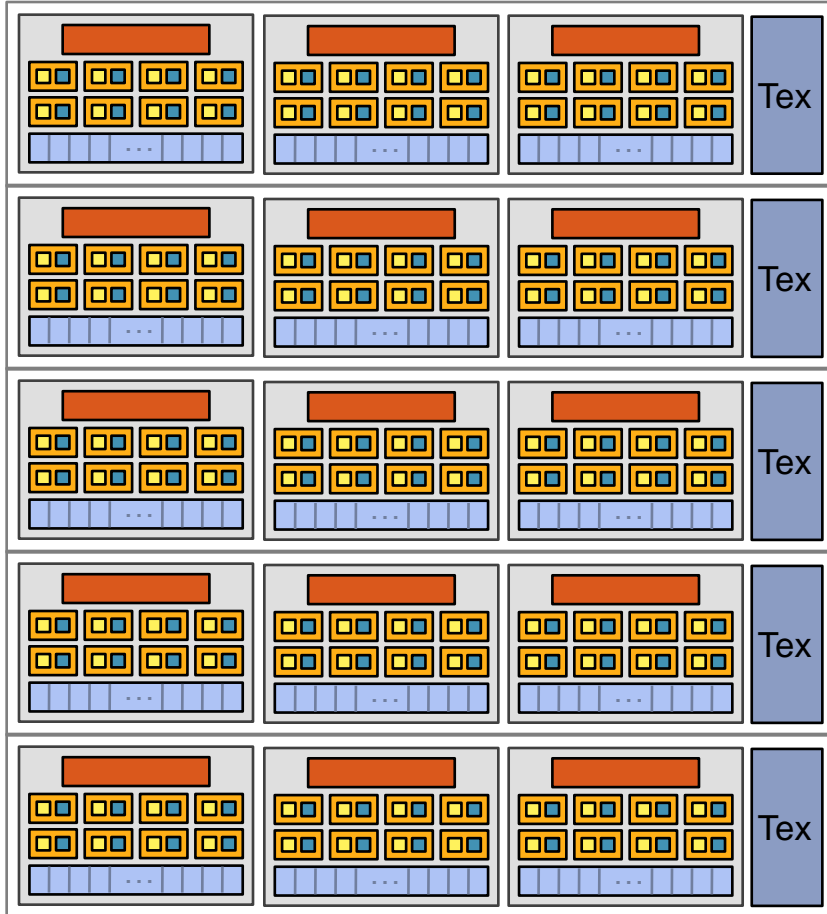
= execution context storage

NVIDIA GeForce GTX 285 “core”



- Groups of 32 **threads** share instruction stream (each group is a Warp)
- Up to 32 warps are simultaneously interleaved
- Up to 1024 thread contexts can be stored

NVIDIA GeForce GTX 285



30 cores on the GTX 285: 30,720 threads

VLIW and DAE

Remember: SIMD/MIMD Classification of Computers

- Mike Flynn, “**Very High Speed Computing Systems**,” Proc. of the IEEE, 1966
- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
 - Array processor
 - Vector processor
- **MISD**? Multiple instructions operate on single data element
 - Closest form: systolic array processor?
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
 - Multiprocessor
 - Multithreaded processor

SISD Parallelism Extraction Techniques

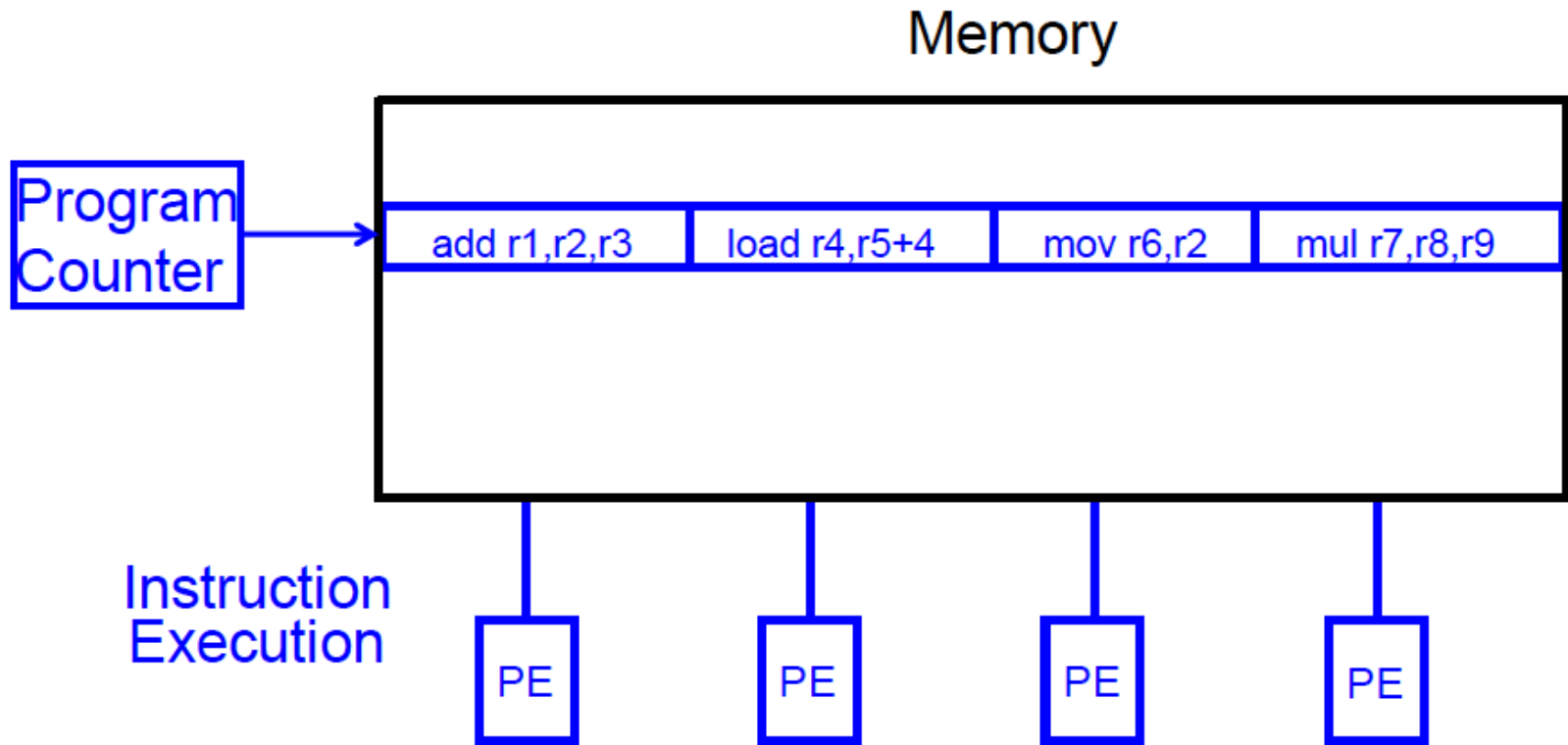
- We have already seen
 - Superscalar execution
 - Out-of-order execution
- Are there simpler ways of extracting SISD parallelism?
 - VLIW (Very Long Instruction Word)
 - Decoupled Access/Execute

VLIW

VLIW (Very Long Instruction Word)

- A very long instruction word consists of multiple independent instructions packed together by the compiler
 - Packed instructions can be logically unrelated (contrast with SIMD)
- Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction
- Traditional Characteristics
 - Multiple functional units
 - Each instruction in a bundle executed in lock step
 - Instructions in a bundle statically aligned to be directly fed into the functional units

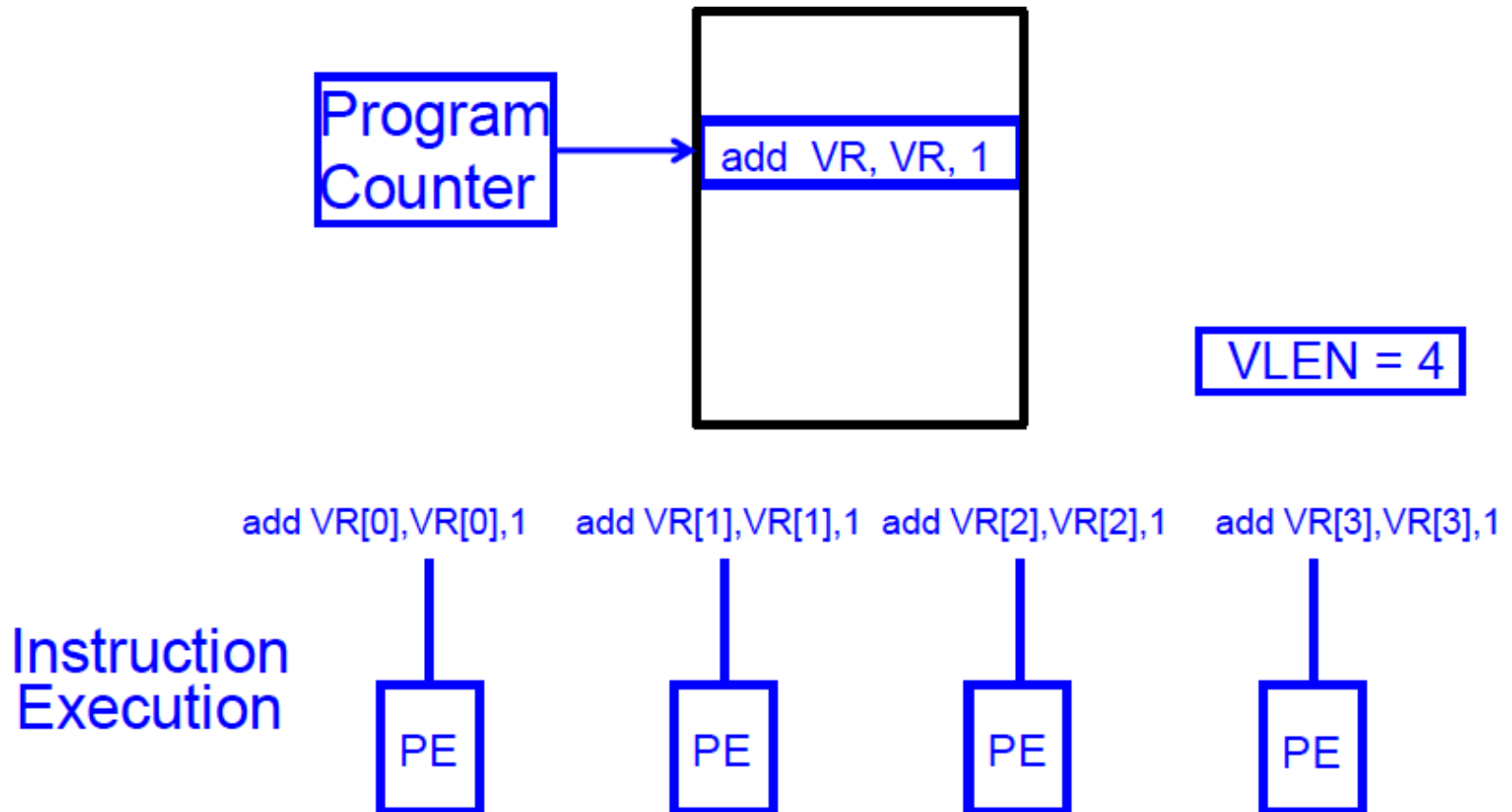
VLIW Concept



- Fisher, “**Very Long Instruction Word architectures and the ELI-512**,” ISCA 1983.
 - ELI: Enormously longword instructions (512 bits)

SIMD Array Processing vs. VLIW

- Array processor



VLIW Philosophy

- Philosophy similar to RISC (simple instructions and hardware)
 - Except multiple instructions in parallel

- RISC (John Cocke, 1970s, IBM 801 minicomputer)
 - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
 - And, to reorder simple instructions for high performance
 - Hardware does little translation/decoding → very simple

- VLIW (Fisher, ISCA 1983)
 - Compiler does the hard work to find instruction level parallelism
 - Hardware stays as simple and streamlined as possible
 - Executes each instruction in a bundle in lock step
 - Simple → higher frequency, easier to design

VLIW Philosophy (II)

More formally, VLIW architectures have the following properties:

There is one central control unit issuing a single long instruction per cycle.

Each long instruction consists of many tightly coupled independent operations.

Each operation requires a small, statically predictable number of cycles to execute.

Operations can be pipelined. These properties distinguish VLIWs from multiprocessors (with large asynchronous tasks) and dataflow machines (without a single flow of control, and without the tight coupling). VLIWs have none of the required regularity of a vector processor, or true array processor.

Commercial VLIW Machines

- Multiflow TRACE, Josh Fisher (7-wide, 28-wide)
- Cydrome Cydra 5, Bob Rau
- Transmeta Crusoe: x86 binary-translated into internal VLIW
- TI C6000, Trimedia, STMicro (DSP & embedded processors)
 - Most successful commercially

- Intel IA-64
 - Not fully VLIW, but based on VLIW principles
 - EPIC (Explicitly Parallel Instruction Computing)
 - Instruction bundles can have dependent instructions
 - A few bits in the instruction format specify explicitly which instructions in the bundle are dependent on which other ones

VLIW Tradeoffs

■ Advantages

- + No need for dynamic scheduling hardware → simple hardware
- + No need for dependency checking within a VLIW instruction → simple hardware for multiple instruction issue + no renaming
- + No need for instruction alignment/distribution after fetch to different functional units → simple hardware

■ Disadvantages

- Compiler needs to find N independent operations
 - If it cannot, inserts NOPs in a VLIW instruction
 - Parallelism loss AND code size increase
- Recompile required when execution width (N), instruction latencies, functional units change (Unlike superscalar processing)
- Lockstep execution causes independent operations to stall
 - No instruction can progress until the longest-latency instruction completes

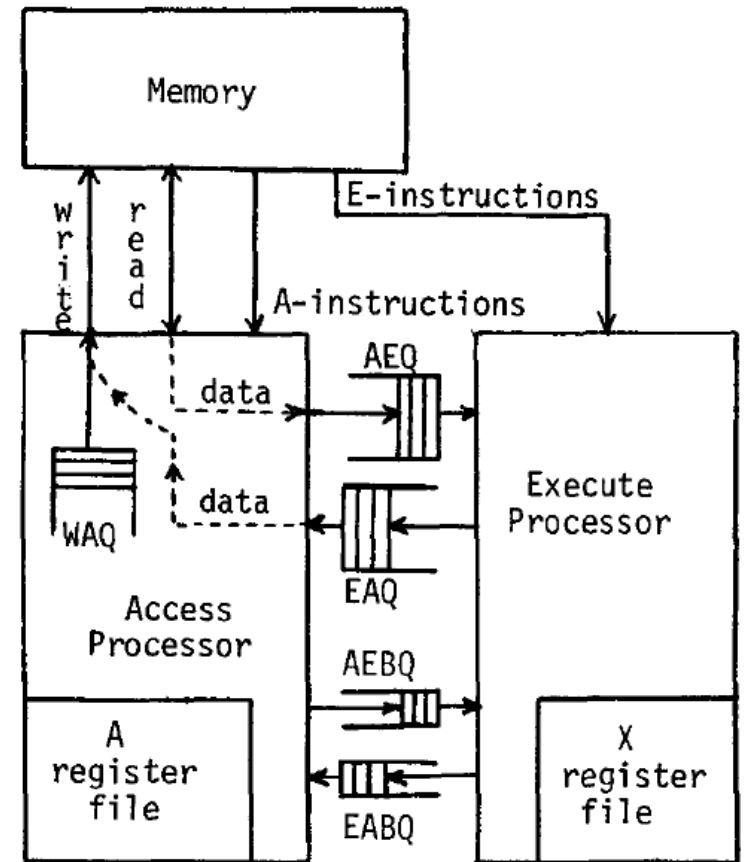
VLIW Summary

- VLIW simplifies hardware, but requires complex compiler techniques
 - Solely-compiler approach of VLIW has several downsides that reduce performance
 - Too many NOPs (not enough parallelism discovered)
 - Static schedule intimately tied to microarchitecture
 - Code optimized for one generation performs poorly for next
 - No tolerance for variable or long-latency operations (lock step)
- ++ Most compiler optimizations developed for VLIW employed in optimizing compilers (for superscalar compilation)
- Enable code optimizations
- ++ VLIW successful in embedded markets, e.g. DSP

DAE

Decoupled Access/Execute

- Motivation: Tomasulo's algorithm too complex to implement
 - 1980s before HPS, Pentium Pro
- Idea: Decouple operand access and execution via two separate instruction streams that communicate via ISA-visible queues.
- Smith, “Decoupled Access/Execute Computer Architectures,” ISCA 1982, ACM TOCS 1984.



Decoupled Access/Execute (II)

- Compiler generates two instruction streams (A and E)
 - Synchronizes the two upon control flow instructions (using branch queues)

```

q = 0.0
Do 1 k = 1, 400
1  x(k) = q + y(k) * (r * z(k+10) + t * z(k+11))
    
```

Fig. 2a. Lawrence Livermore Loop 1 (HYDRO EXCERPT)

	A7 ← -400	. negative loop count
	A2 ← 0	. initialize index
	A3 ← 1	. index increment
	X2 ← r	. load loop invariants
	X5 ← t	. into registers
loop:	X3 ← z + 10, A2	. load z(k+10)
	X7 ← z + 11, A2	. load z(k+11)
	X4 ← X2 *f X3	. r*z(k+10)-flt. mult.
	X3 ← X5 *f X7	. t * z(k+11)
	X7 ← y, A2	. load y(k)
	X6 ← X3 +f X4	. r*z(x+10)+t*z(k+11))
	X4 ← X7 *f X6	. y(k) * (above)
	A7 ← A7 + 1	. increment loop counter
	x, A2 ← X4	. store into x(k)
	A2 ← A2 + A3	. increment index
	JAM loop	. Branch if A7 < 0

Fig. 2b. Compilation onto CRAY-1-like architecture

<u>Access</u>	<u>Execute</u>
.	
.	
.	
AEQ ← z + 10, A2	X4 ← X2 *f AEQ
AEQ ← z + 11, A2	X3 ← X5 *f AEQ
AEQ ← y, A2	X6 ← X3 +f X4
A7 ← A7 + 1	EAQ ← AEQ *f X6
x, A2 ← EAQ	.
A2 ← A2+ A3	.
.	.
.	
.	

Fig. 2c. Access and execute programs for straight-line section of loop

Decoupled Access/Execute (III)

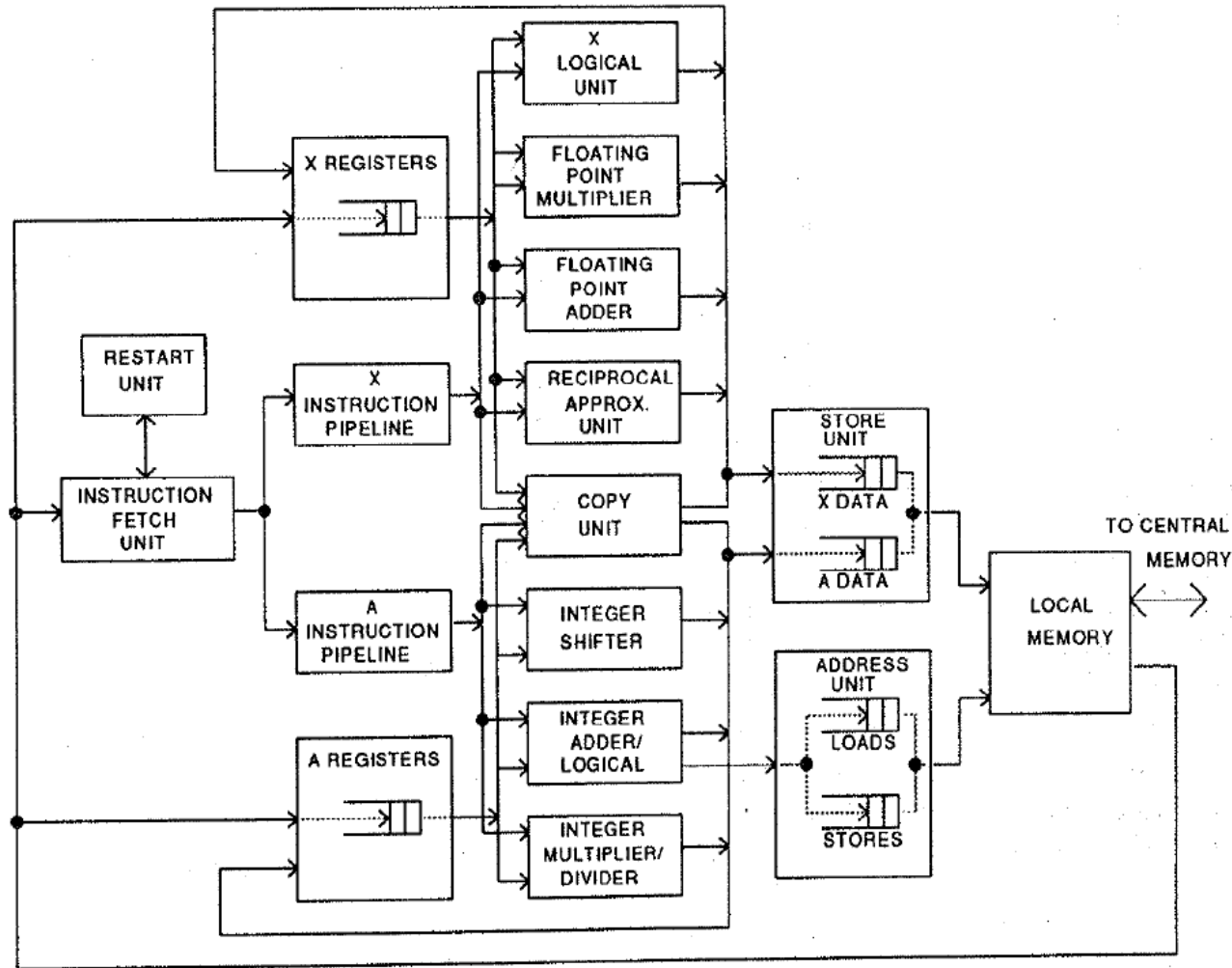
■ Advantages:

- + Execute stream can run ahead of the access stream and vice versa
 - + If A takes a cache miss, E can perform useful work
 - + If A hits in cache, it supplies data to lagging E
 - + Queues reduce the number of required registers
- + Limited out-of-order execution without wakeup/select complexity

■ Disadvantages:

- Compiler support to partition the program and manage queues
 - Determines the amount of decoupling
- Branch instructions require synchronization between A and E
- Multiple instruction streams (can be done with a single one, though)

Astronautics ZS-1



- Single stream steered into A and X pipelines
- Each pipeline in-order
- Smith et al., “**The ZS-1 central processor,**” ASPLOS 1987.
- Smith, “**Dynamic Instruction Scheduling and the Astronautics ZS-1,**” IEEE Computer 1989.

Astronautics ZS-1 Instruction Scheduling

■ Dynamic scheduling

- ❑ A and X streams are issued/executed independently
- ❑ Loads can bypass stores in the memory unit (if no conflict)
- ❑ Branches executed early in the pipeline
 - To reduce synchronization penalty of A/X streams
 - Works only if the register a branch sources is available

■ Static scheduling

- ❑ Move compare instructions as early as possible before a branch
 - So that branch source register is available when branch is decoded
- ❑ Reorder code to expose parallelism in each stream
- ❑ Loop unrolling:
 - Reduces branch count + exposes code reordering opportunities

Loop Unrolling

```
i = 1;
while ( i < 100 ) {
    a[i] = b[i+1] + (i+1)/m
    b[i] = a[i-1] - i/m
    i = i + 1
}
```

```
i = 1;
while ( i < 100 ) {
    a[i] = b[i+1] + (i+1)/m
    b[i] = a[i-1] - i/m

    a[i+1] = b[i+2] + (i+2)/m
    b[i+1] = a[i] - (i+1)/m
    i = i + 2
}
```

- Idea: Replicate loop body multiple times within an iteration
- + Reduces loop maintenance overhead
 - Induction variable increment or loop condition test
- + Enlarges basic block (and analysis scope)
 - Enables code optimization and scheduling opportunities
- What if iteration count not a multiple of unroll factor? (need extra code to detect this)
- Increases code size

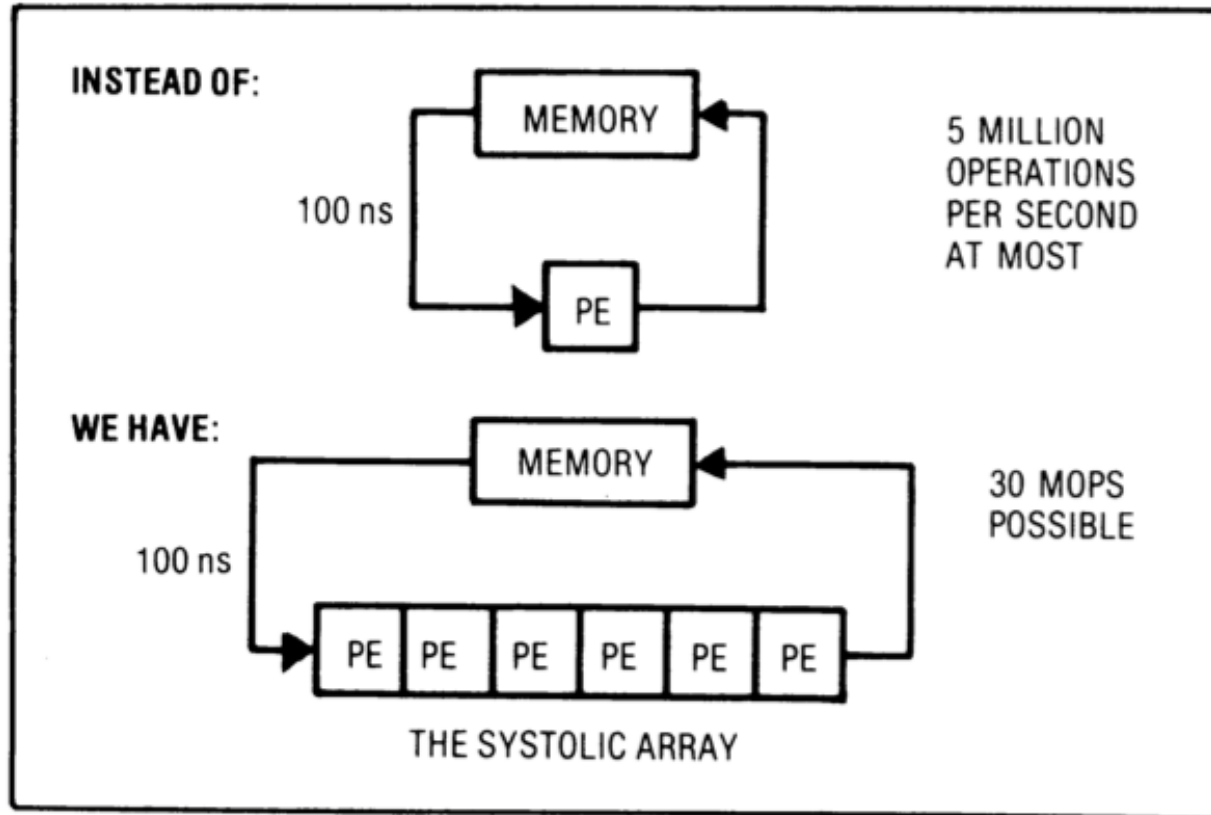
Systolic Arrays

Why Systolic Architectures?

- Idea: Data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory
- Similar to an assembly line
 - Different people work on the same car
 - Many cars are assembled simultaneously
 - Can be two-dimensional
- Why? Special purpose accelerators/architectures need
 - Simple, regular designs (keep # unique parts small and regular)
 - High concurrency → high performance
 - Balanced computation and I/O (memory access)

Systolic Architectures

- H. T. Kung, "[Why Systolic Architectures?](#)," IEEE Computer 1982.



Memory: heart
PEs: cells

Memory pulses
data through
cells

Figure 1. Basic principle of a systolic system.

Systolic Architectures

- Basic principle: Replace a single PE with a regular array of PEs and carefully orchestrate flow of data between the PEs
→ achieve high throughput w/o increasing memory bandwidth requirements

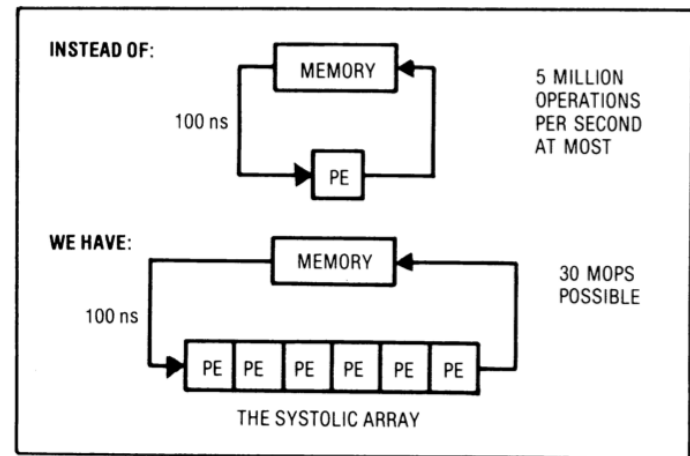


Figure 1. Basic principle of a systolic system.

- Differences from pipelining:
 - ❑ Array structure can be non-linear and multi-dimensional
 - ❑ PE connections can be multidirectional (and different speed)
 - ❑ PEs can have local memory and execute kernels (rather than a piece of the instruction)

Systolic Computation Example

■ Convolution

- Used in filtering, pattern matching, correlation, polynomial evaluation, etc ...
- Many image processing tasks

Given the sequence of weights $\{w_1, w_2, \dots, w_k\}$
and the input sequence $\{x_1, x_2, \dots, x_n\}$,

compute the result sequence $\{y_1, y_2, \dots, y_{n+1-k}\}$
defined by

$$y_i = w_1x_i + w_2x_{i+1} + \dots + w_kx_{i+k-1}$$

Systolic Computation Example: Convolution

- $y_1 = w_1x_1 + w_2x_2 + w_3x_3$
- $y_2 = w_1x_2 + w_2x_3 + w_3x_4$
- $y_3 = w_1x_3 + w_2x_4 + w_3x_5$

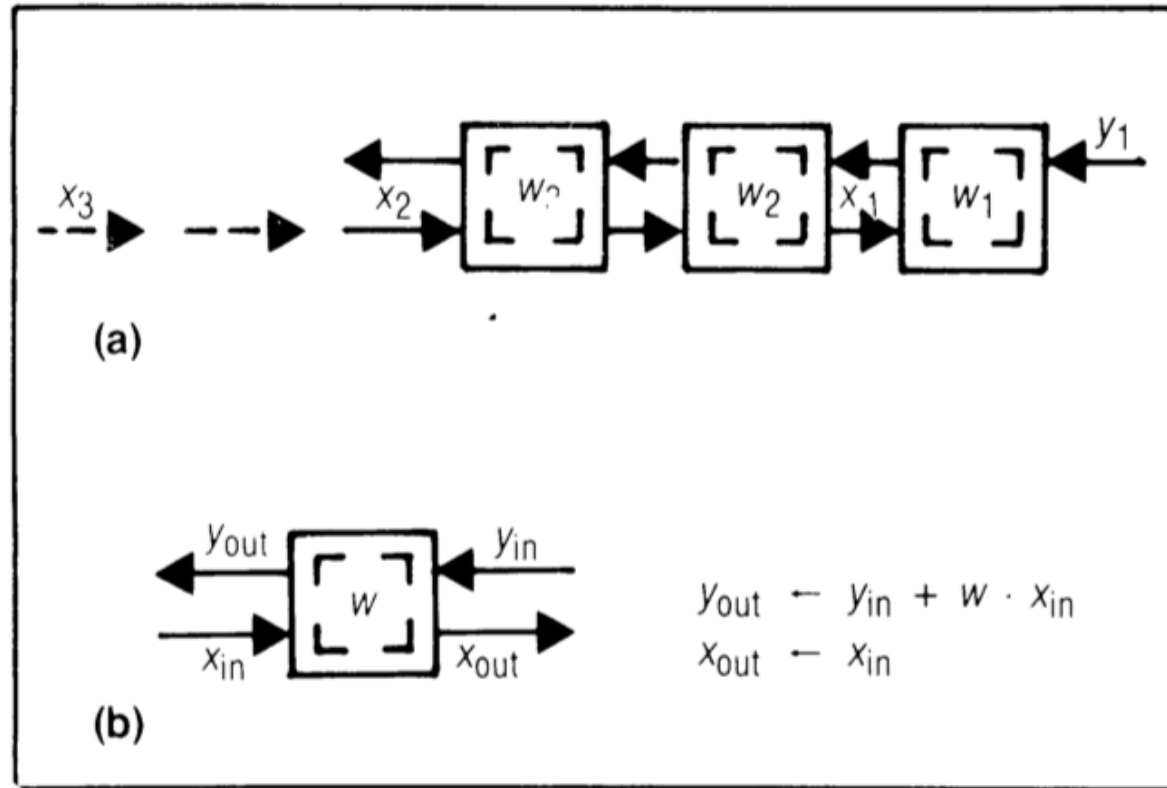


Figure 8. Design W1: systolic convolution array (a) and cell (b) where w_i 's stay and x_i 's and y_i 's move systolically in opposite directions.

Systolic Computation Example: Convolution

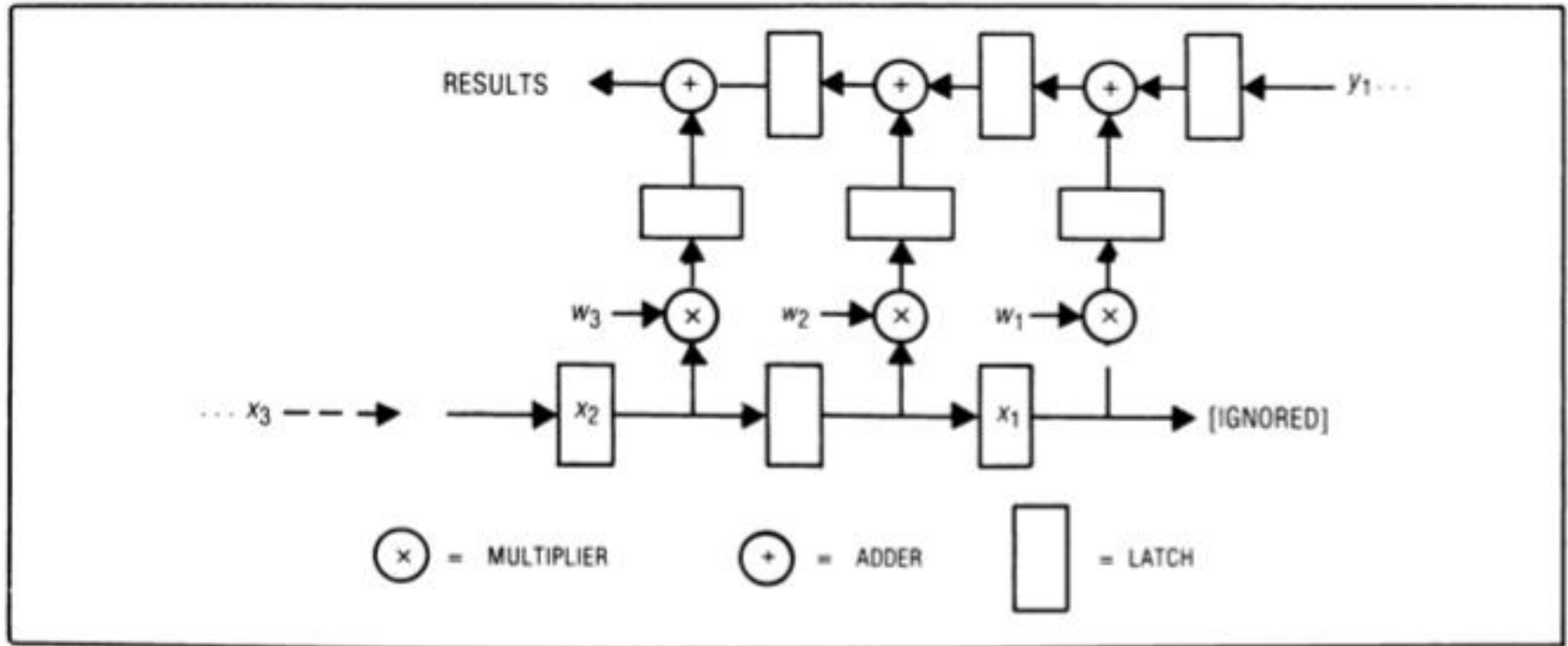


Figure 10. Overlapping the executions of multiply and add in design W1.

- Worthwhile to implement adder and multiplier separately to allow overlapping of add/mul executions

More Programmability

- Each PE in a systolic array
 - Can store multiple “weights”
 - Weights can be selected on the fly
 - Eases implementation of, e.g., adaptive filtering
- Taken further
 - Each PE can have its own data and instruction memory
 - Data memory → to store partial/temporary results, constants
 - Leads to **stream processing, pipeline parallelism**
 - More generally, **staged execution**

Pipeline Parallelism

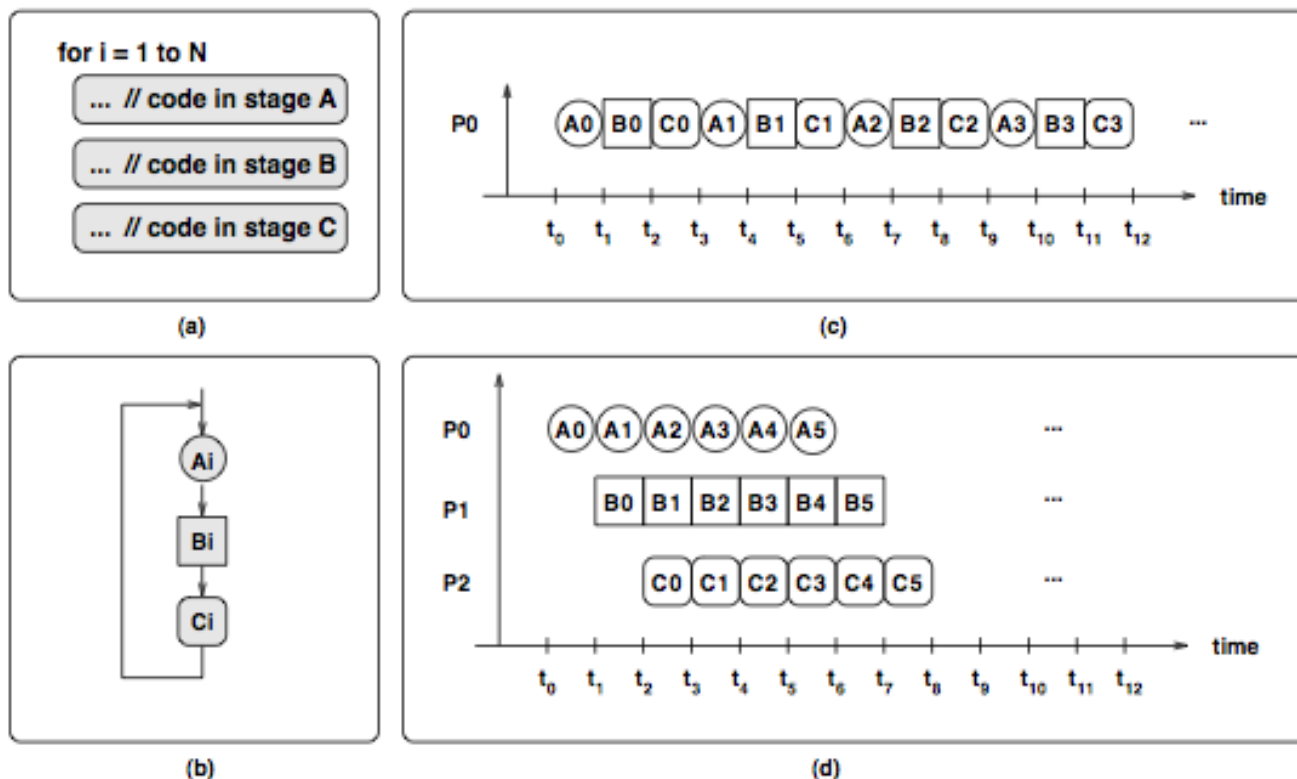


Figure 1. (a) The code of a loop, (b) Each iteration is split into 3 pipeline stages: A, B, and C. Iteration i comprises A_i , B_i , C_i . (c) Sequential execution of 4 iterations. (d) Parallel execution of 6 iterations using pipeline parallelism on a three-core machine. Each stage executes on one core.

File Compression Example

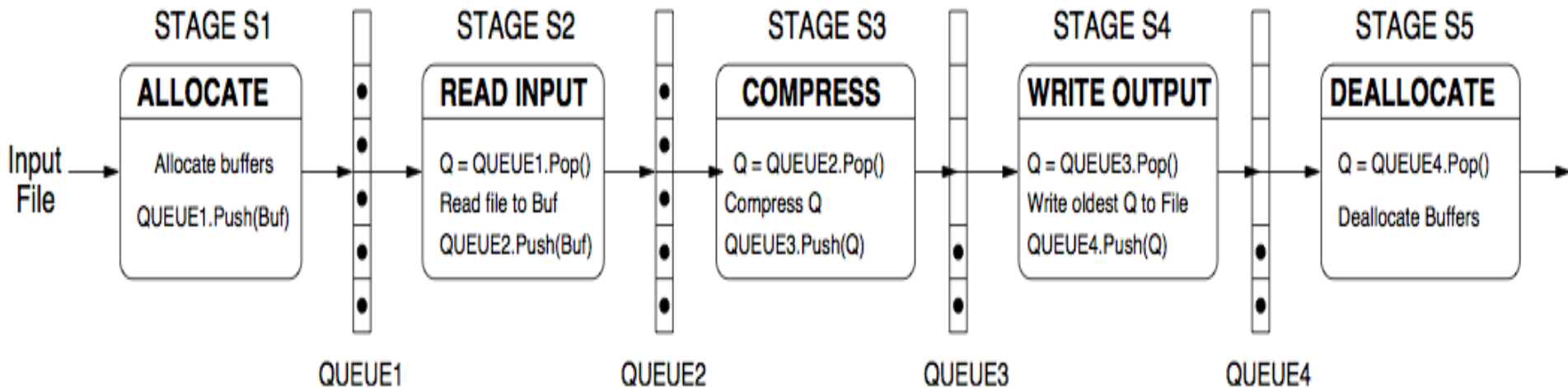


Figure 3. File compression algorithm executed using pipeline parallelism

Systolic Array

■ Advantages

- ❑ Makes multiple uses of each data item → reduced need for fetching/refetching
- ❑ High concurrency
- ❑ Regular design (both data and control flow)

■ Disadvantages

- ❑ Not good at exploiting irregular parallelism
- ❑ Relatively special purpose → need software, programmer support to be a general purpose model

The WARP Computer

- HT Kung, CMU, 1984-1988
- Linear array of 10 cells, each cell a 10 Mflop programmable processor
- Attached to a general purpose host machine
- HLL and optimizing compiler to program the systolic array
- Used extensively to accelerate vision and robotics tasks
- Annaratone et al., “Warp Architecture and Implementation,” ISCA 1986.
- Annaratone et al., “The Warp Computer: Architecture, Implementation, and Performance,” IEEE TC 1987.

The WARP Computer

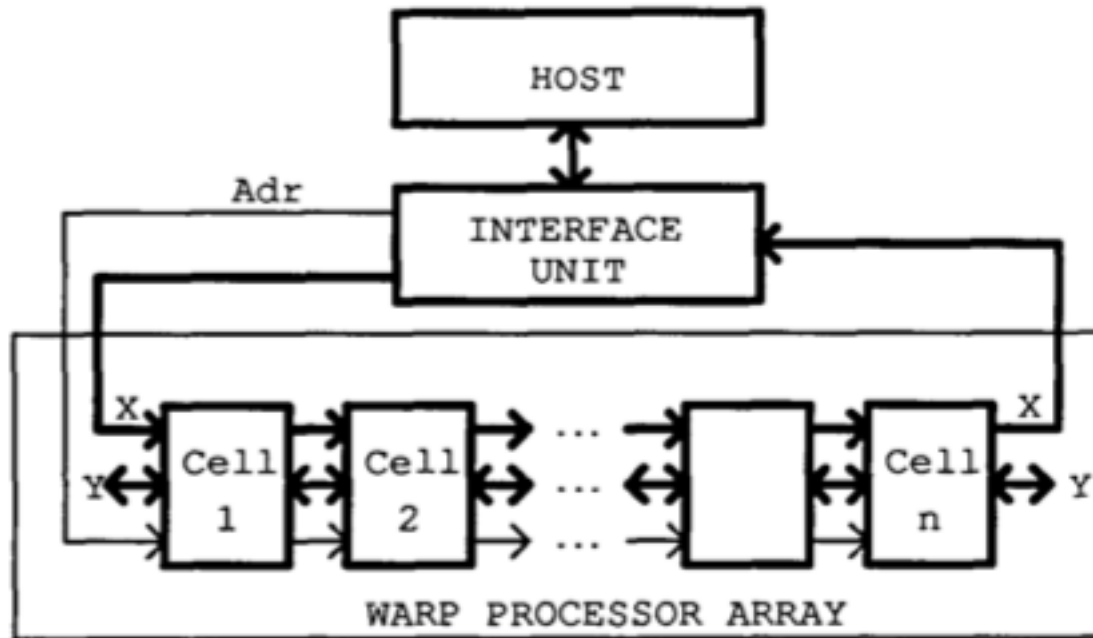


Figure 1: Warp system overview

The WARP Computer

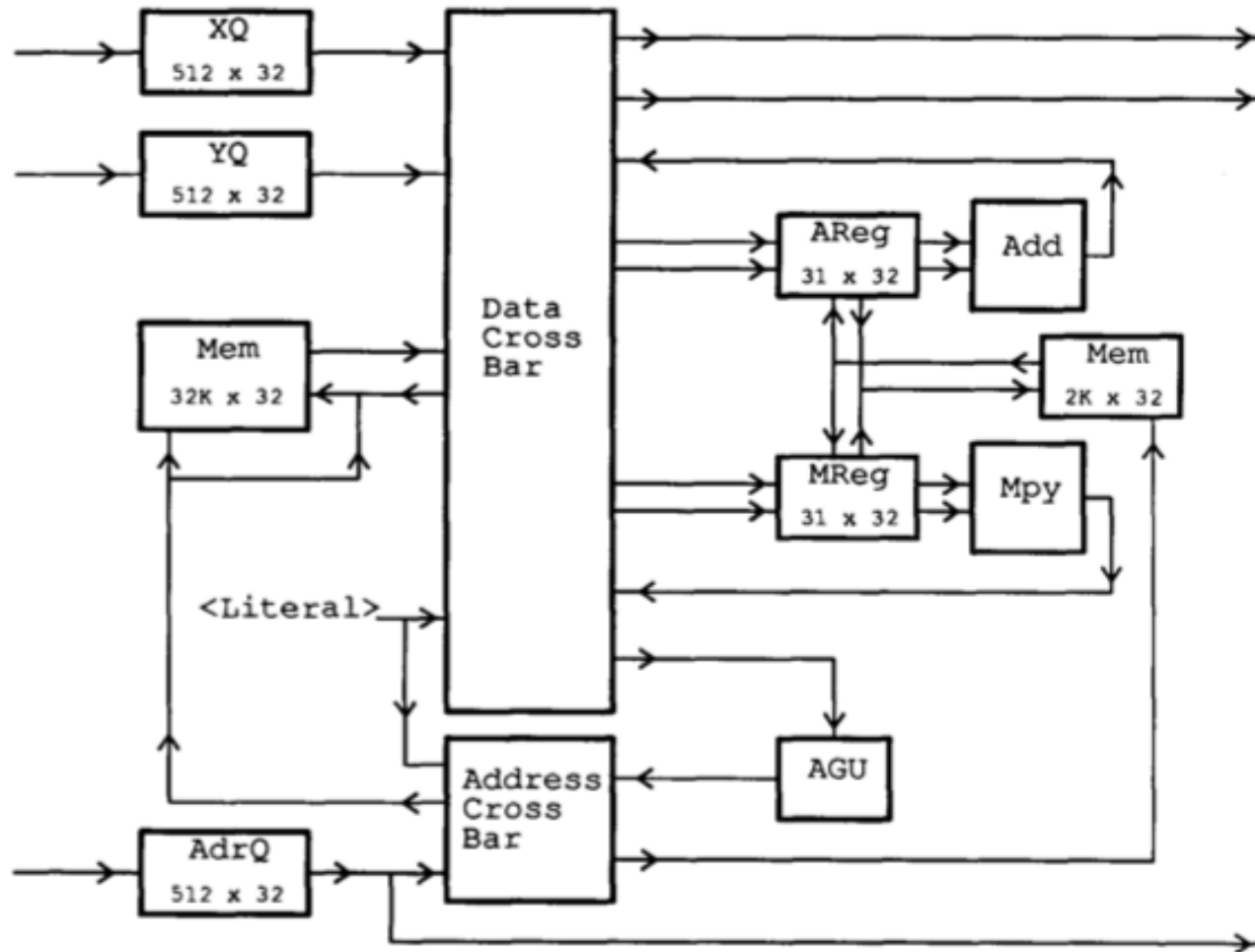


Figure 2: Warp cell data path

Systolic Arrays vs. SIMD

- Food for thought...

Some More Recommended Readings

■ Recommended:

- ❑ Fisher, “**Very Long Instruction Word architectures and the ELI-512,**” ISCA 1983.
- ❑ Huck et al., “**Introducing the IA-64 Architecture,**” IEEE Micro 2000.
- ❑ Russell, “**The CRAY-1 computer system,**” CACM 1978.
- ❑ Rau and Fisher, “**Instruction-level parallel processing: history, overview, and perspective,**” Journal of Supercomputing, 1993.
- ❑ Faraboschi et al., “**Instruction Scheduling for Instruction Level Parallel Processors,**” Proc. IEEE, Nov. 2001.