

15-740/18-740
Computer Architecture
Lecture 28: SIMD and GPUs

Prof. Onur Mutlu
Carnegie Mellon University

Announcements

- Project Poster Session
 - December 10
 - NSH Atrium
 - 2:30-6:30pm

- Project Report Due
 - December 12
 - The report should be like a good conference paper

- Focus on Projects
 - All group members should contribute
 - Use the milestone feedback from the TAs

Final Project Report and Logistics

- Follow the guidelines in project handout for report
 - We will provide the Latex format
- Good papers should be similar to the best conference papers you have been reading throughout the semester
- Submit all code, documentation, supporting documents and data
 - Provide instructions as to how to compile and use your code in a README file
 - This will determine part of your grade
 - We will provide the directory to upload
- This is the single most important part of the project

Best Projects

- Best projects will be encouraged for a top conference submission
 - Talk with me if you are interested in this
- Examples from past:
 - Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter, "[ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers](#)," HPCA 2010 Best Paper Session
 - George Nychis, Chris Fallin, Thomas Moscibroda, and Onur Mutlu, "[Next Generation On-Chip Networks: What Kind of Congestion Control Do We Need?](#)," HotNets 2010.
 - Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter, "[Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior](#)," MICRO 2010. (IEEE Micro Top Picks 2010)

Please Fill Out Course Evaluations

- Please fill them out online until December 14, 4pm
- Very important for feedback, course development/improvement, administration
 - I read each of these carefully to improve the future course contents, logistics, etc.
- <http://cmu.onlinecourseevaluations.com>

TA Evaluations

- Please fill them out online until December 10, 5pm
- Vivek
Seshadri: <http://www.surveymonkey.com/s/PRW7DDJ>
- Lavanya
Subramanian: <http://www.surveymonkey.com/s/PRTCNBD>
- Evangelos Vlachos:
<http://www.surveymonkey.com/s/XZ88LVF>

Last Time

- VLIW
 - Concepts and Philosophy
 - Encoding and NOPs
 - Static Scheduling Concepts
 - Trace Scheduling
 - Superblock Scheduling
 - Hyperblock Scheduling

- EPIC: Explicitly Parallel Instruction Computing
 - IA-64
 - Static store-load scheduling

Today

- Data Parallel (SIMD) Execution Model
- GPU Basics
- GPU Programming → 18-742

Readings: SIMD and GPUs

- Required
 - Lindholm et al., “NVIDIA Tesla: A Unified Graphics and Computing Architecture,” IEEE Micro 2008.
 - Russell, “The CRAY-1 computer system,” CACM 1978.
- Recommended
 - Fung et al., “Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow,” MICRO 2007.
 - Luk et al., “Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping,” MICRO 2009.

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 thread (“control”) parallelism
 - Concurrency arises from executing different threads of control in parallel
- Contrast with data flow
 - Concurrency arises from executing different operations in parallel (in a data driven manner)
- SIMD exploits instruction-level parallelism
 - Multiple instructions concurrent: instructions happen to be the same

SIMD Processing

- Single instruction operates on multiple data elements
 - In time or in space
- Multiple processing elements

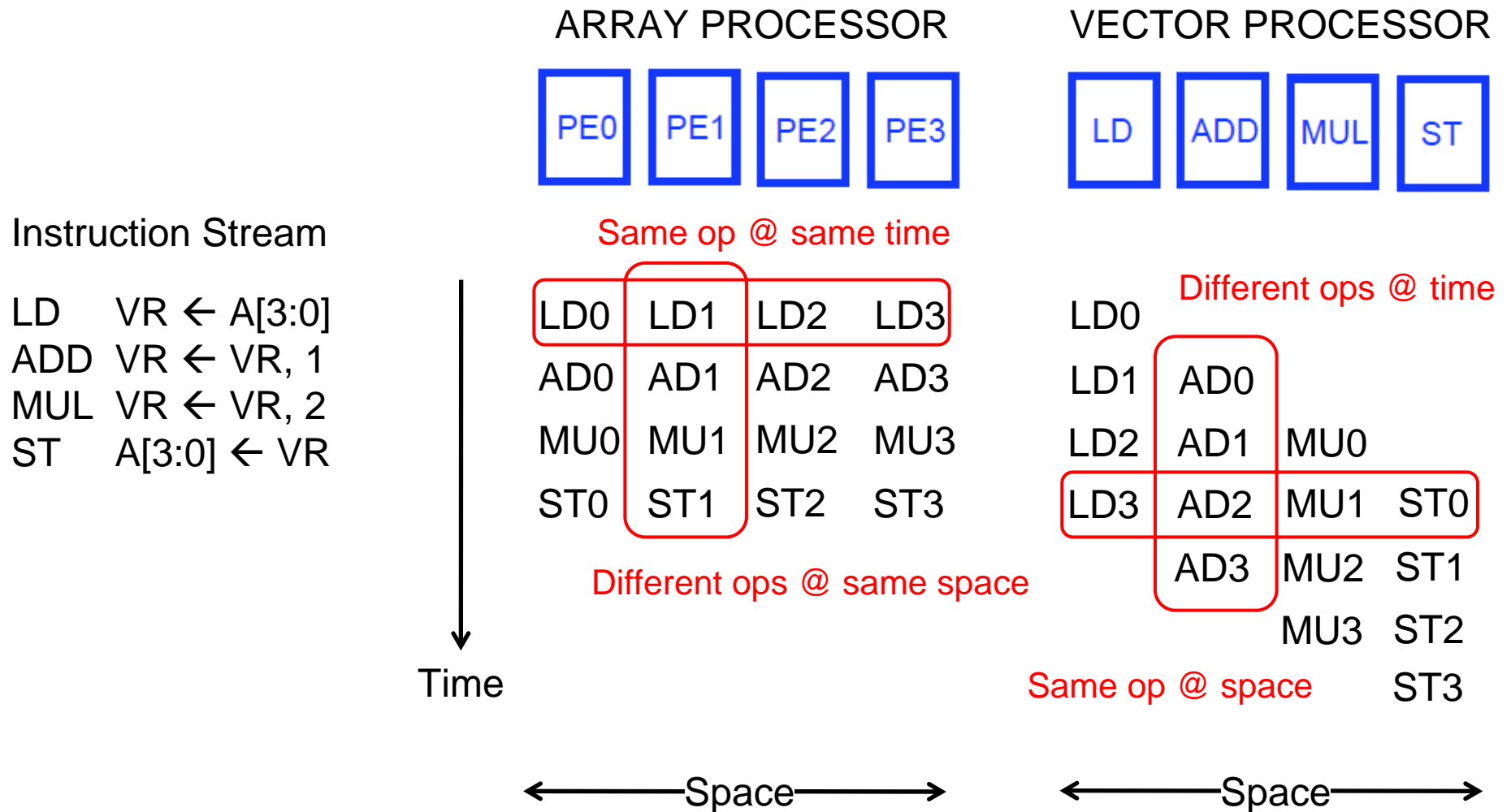
- Time-space duality
 - **Array processor**: Instruction operates on multiple data elements at the same time
 - **Vector processor**: Instruction operates on multiple data elements in consecutive time steps

SIMD Processing

- Single instruction operates on multiple data elements
 - In time or in space
- Multiple processing elements

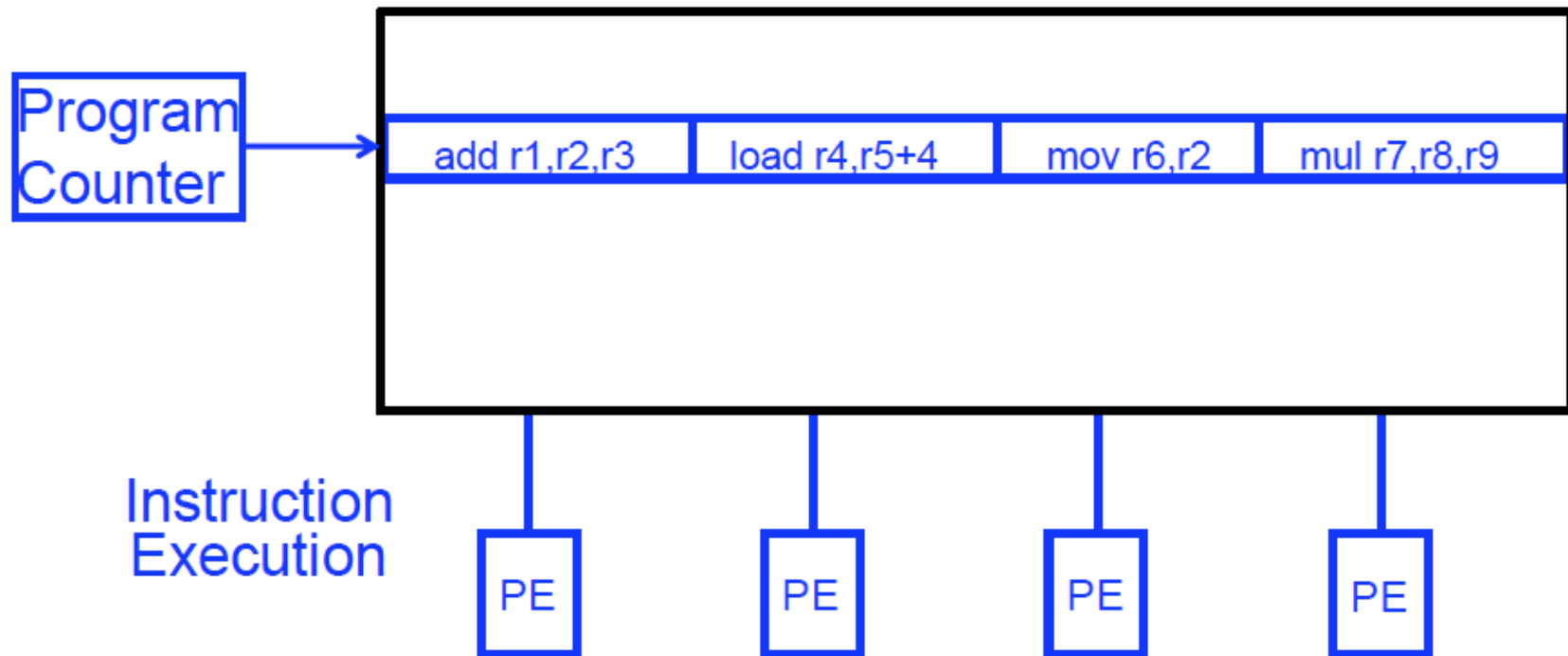
- Time-space duality
 - **Array processor**: Instruction operates on multiple data elements at the same time
 - **Vector processor**: Instruction operates on multiple data elements in consecutive time steps

Array vs. Vector Processors



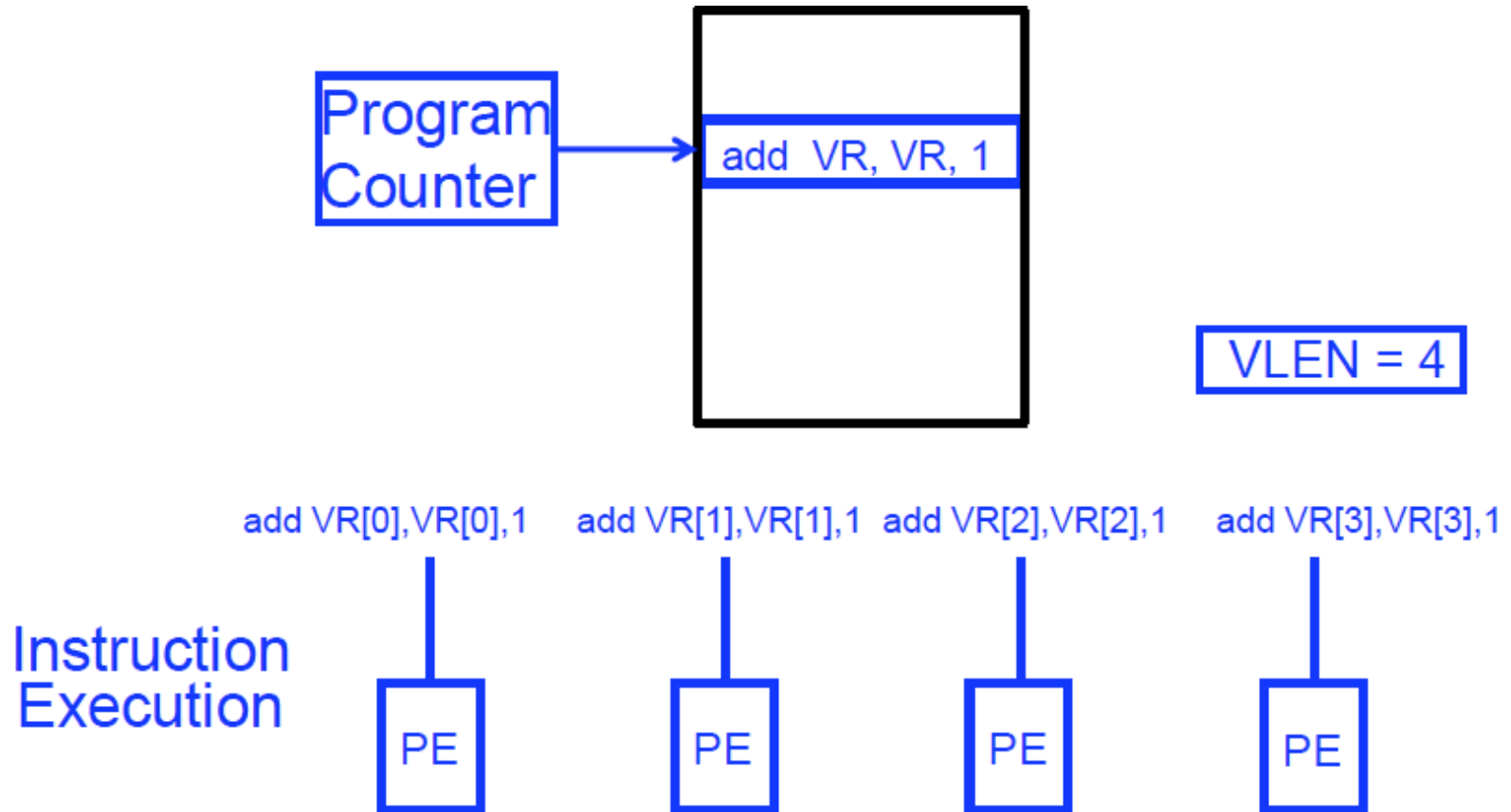
SIMD Array Processing vs. VLIW

- 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 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 ISA Advantages

- Compact encoding
 - 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 same pattern as previous instructions
 - access a contiguous block of memory (unit-stride load/store)
 - access memory in a known pattern (strided load/store)
- Scalable
 - can run the same code in parallel pipelines (*lanes*)

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

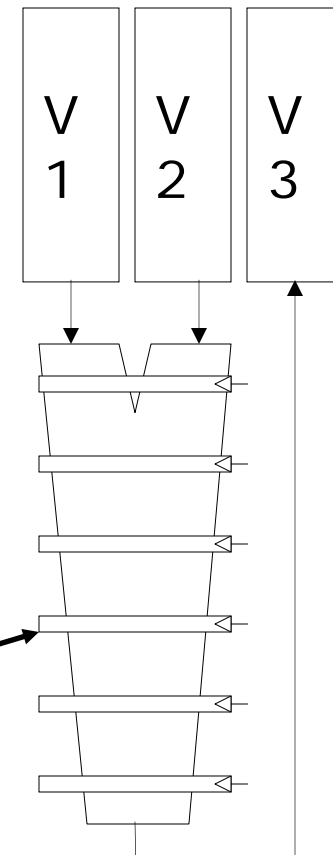
Vector Processor Limitations

- 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 Functional Units

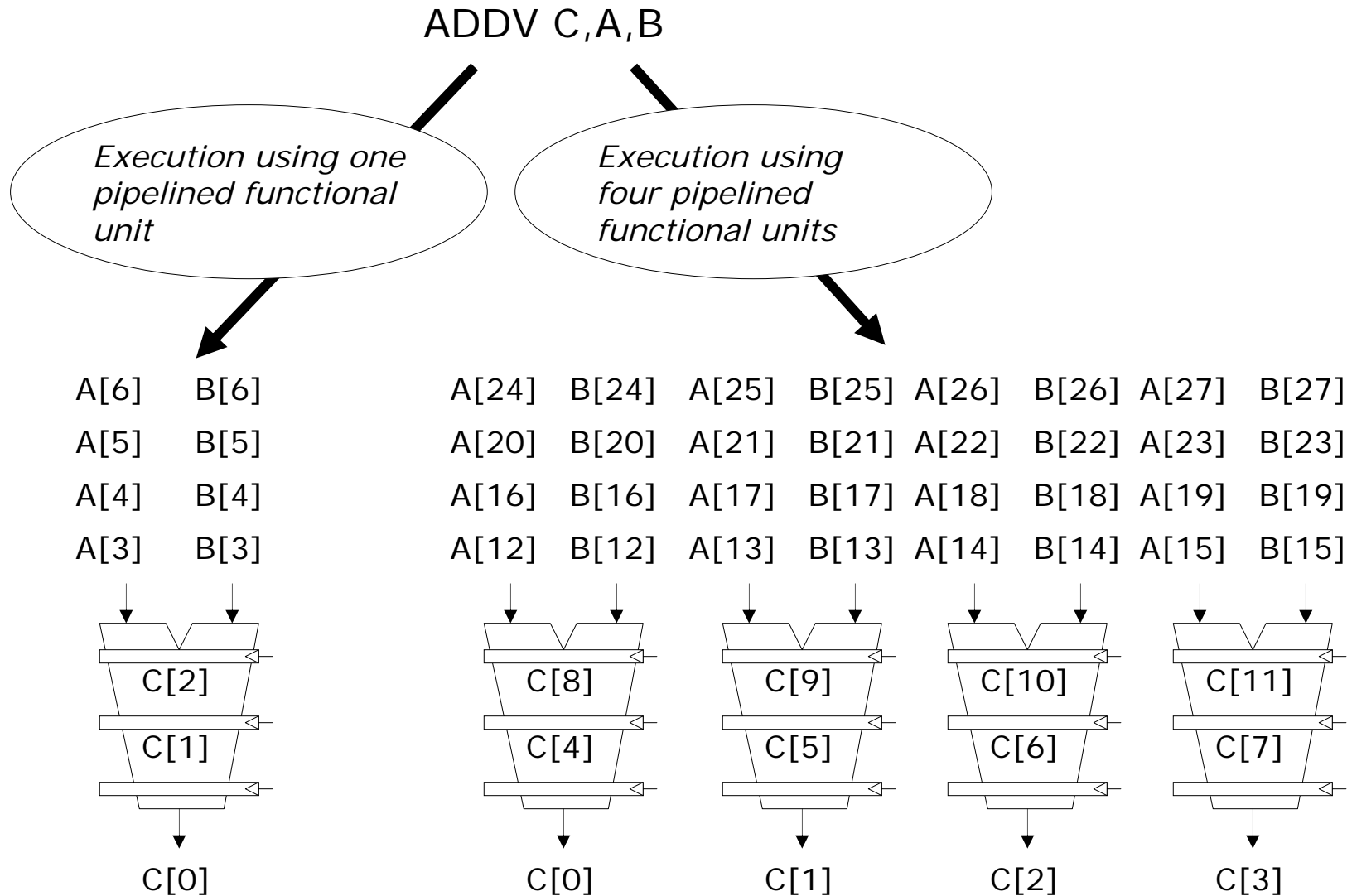
- Use deep pipeline (\Rightarrow fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (\Rightarrow no hazards!)

Six stage multiply pipeline



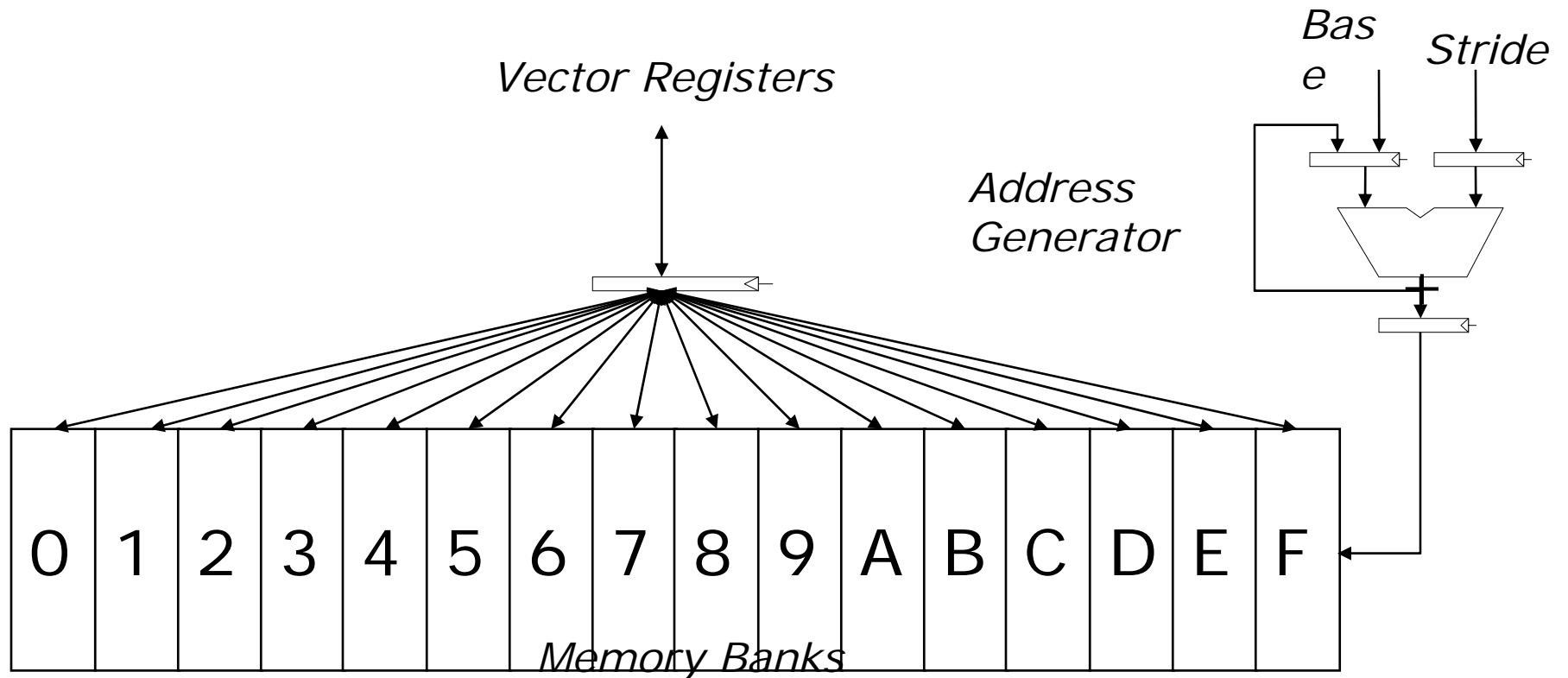
$$V3 \leftarrow v1 * v2$$

Vector Instruction Execution

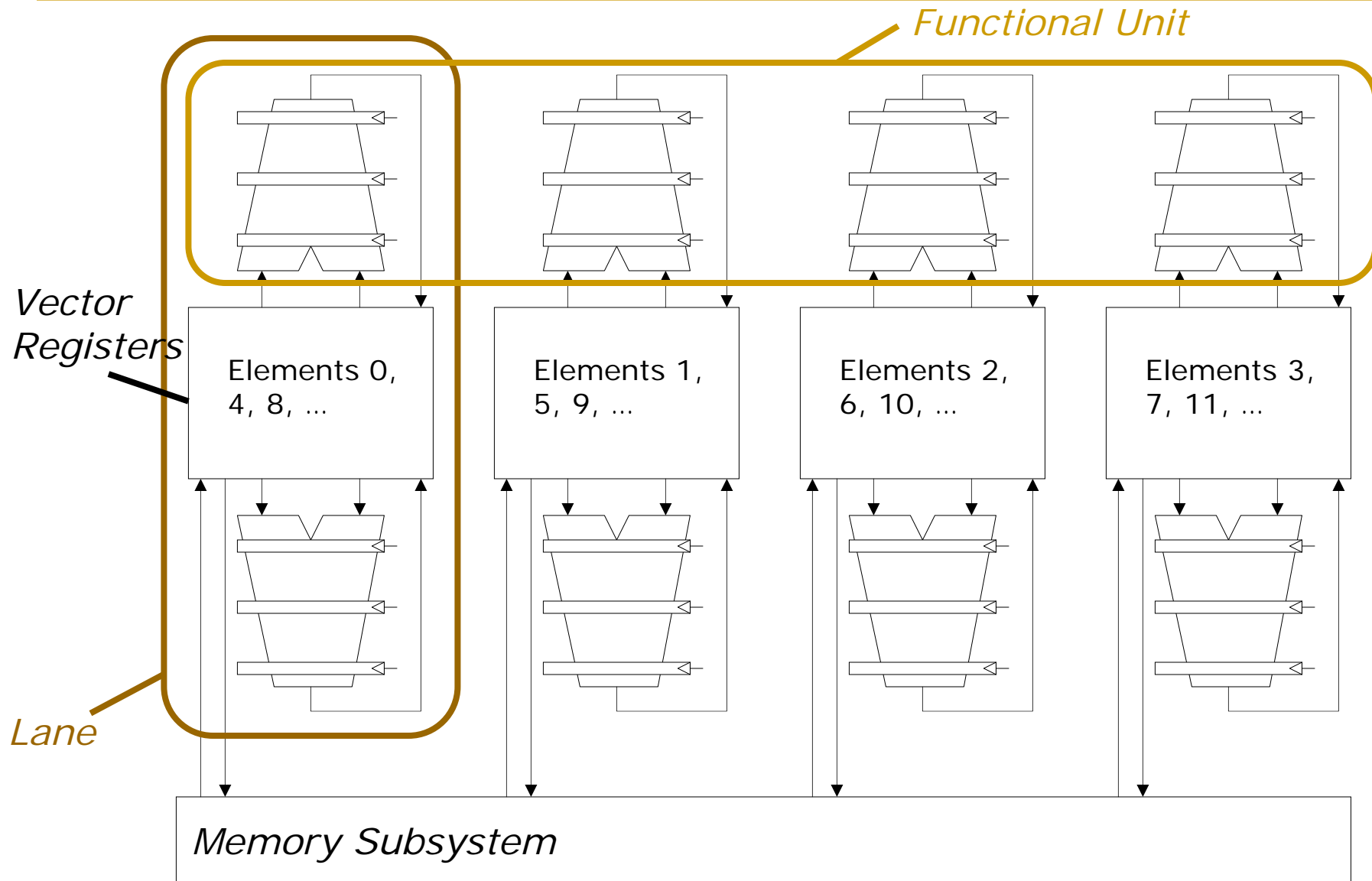


Vector Memory System

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



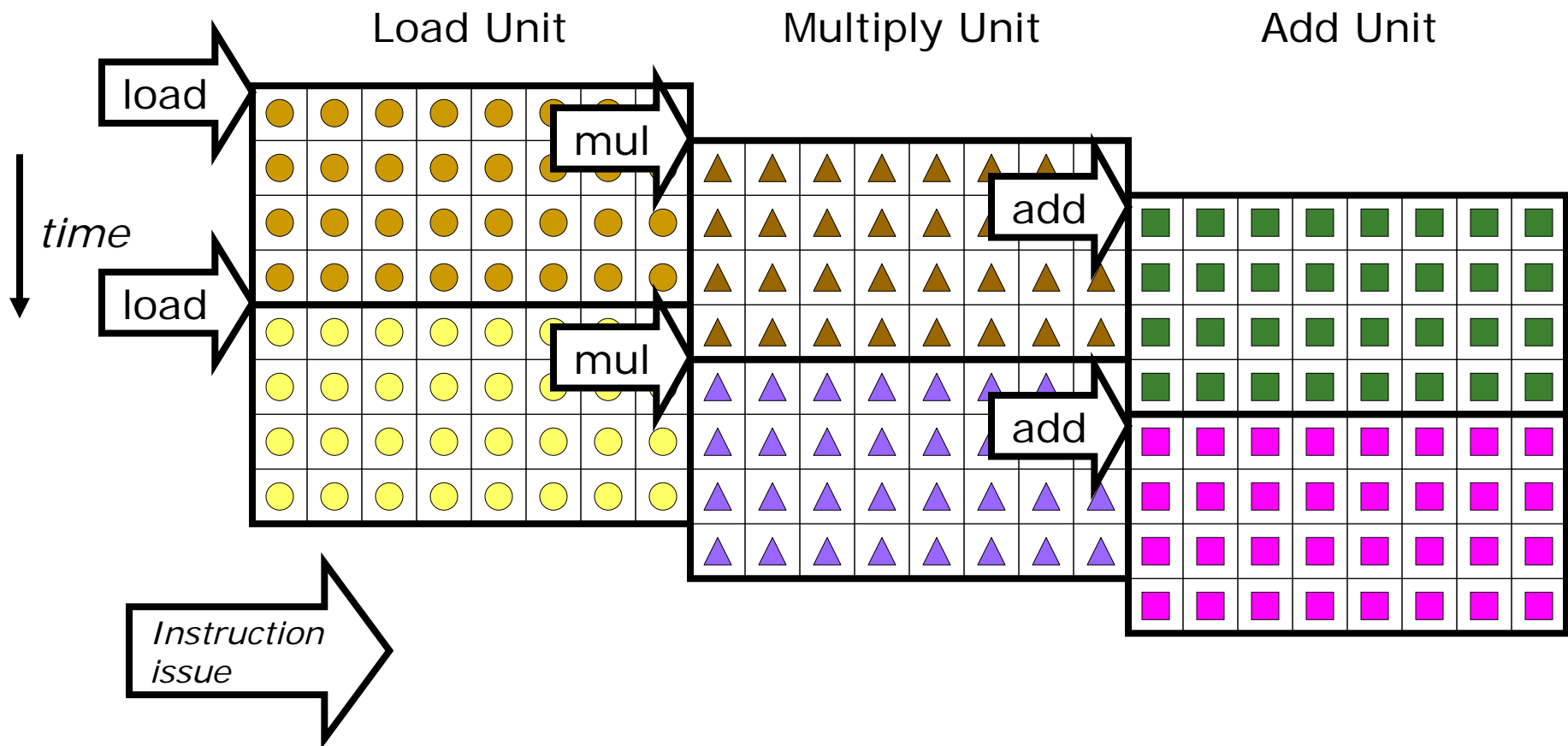
Vector Unit Structure



Vector Instruction Level Parallelism

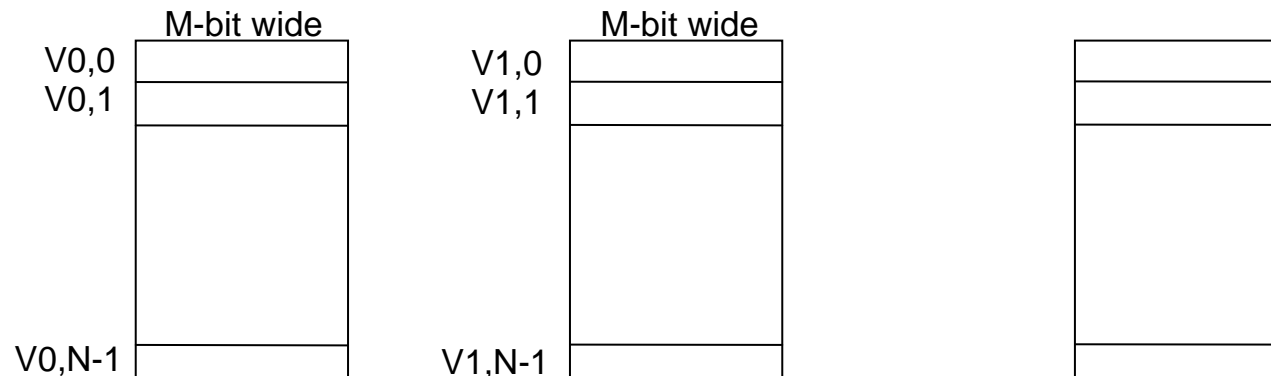
Can overlap execution of multiple vector instructions

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

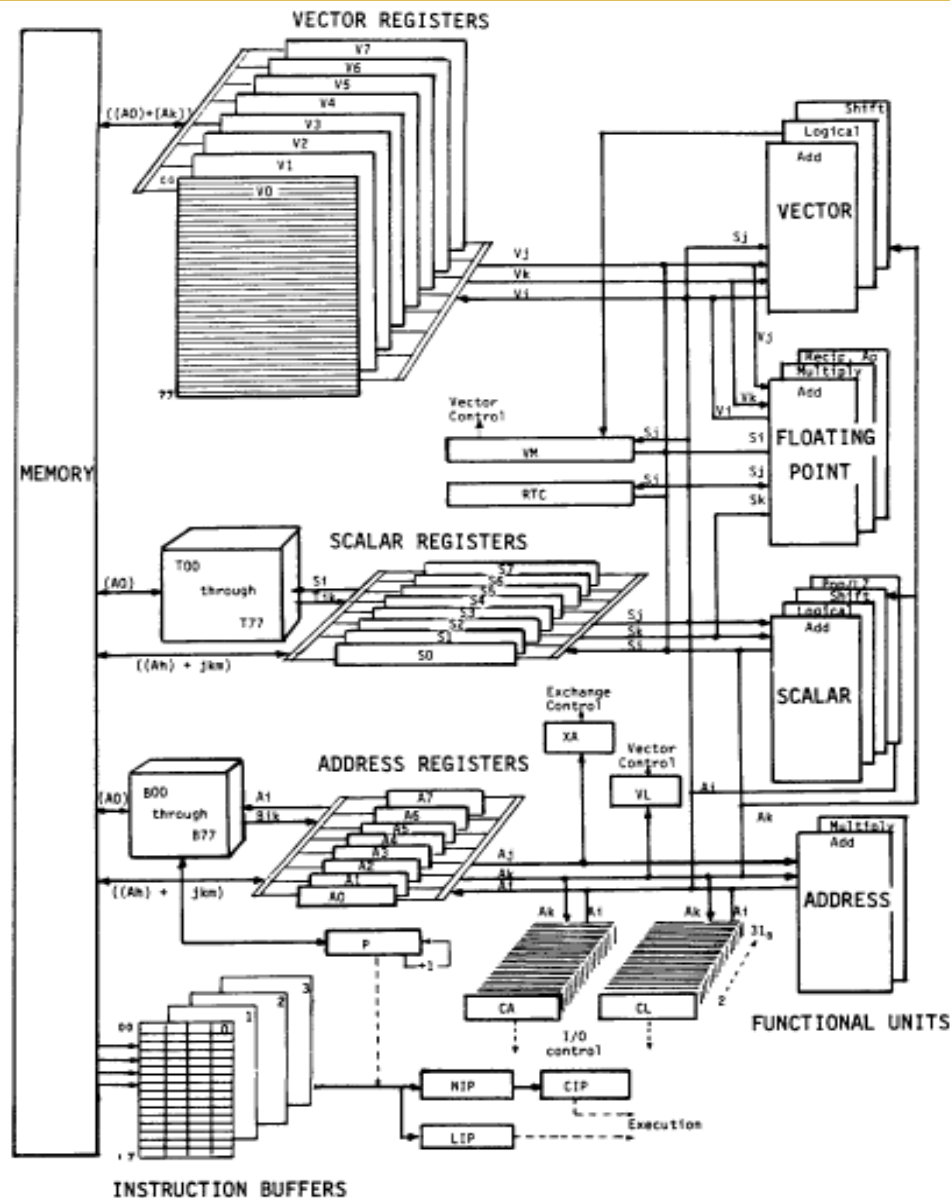


Vector Registers

- 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., $VMASK[i] = (V_k[i] == 0)$
- Maximum VLEN can be N
 - Maximum number of elements stored in a vector register

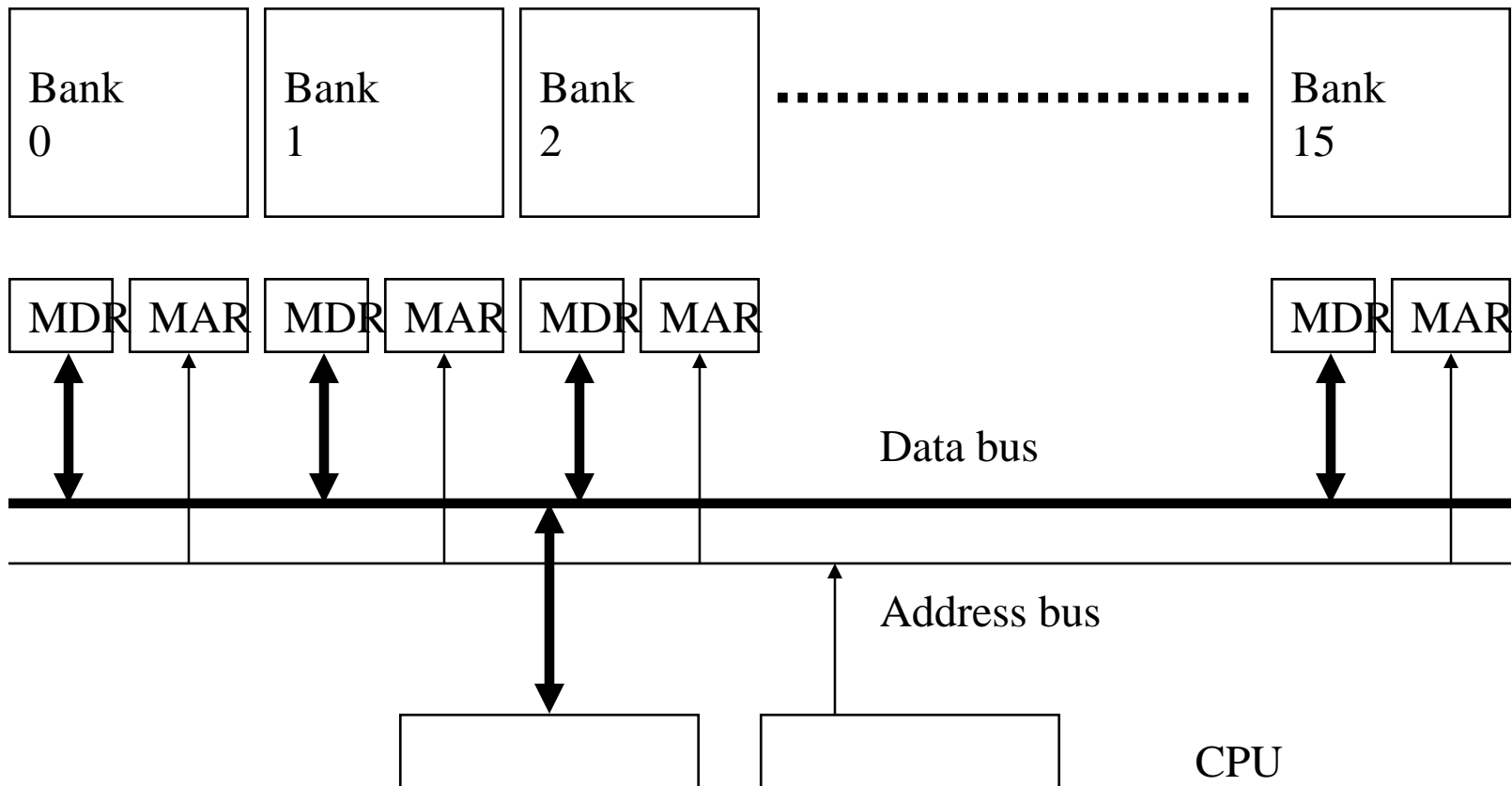


Vector Machine Organization (CRAY-1)



- 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 in CRAY-1



Scalar Code Example

- For I = 1 to 50
 - $C[i] = (A[i] + B[i]) / 2$

- 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	;autoincrement addressing
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

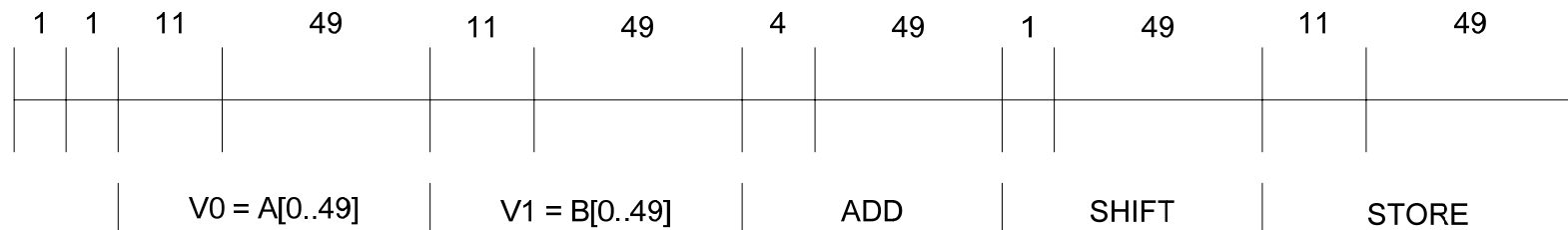
- A loop is **vectorizable** if each iteration is independent of any other
- For $I = 0$ to 49
 - $C[i] = (A[i] + B[i]) / 2$ 7 dynamic instructions

- Vectorized loop:

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

Vector Code Performance

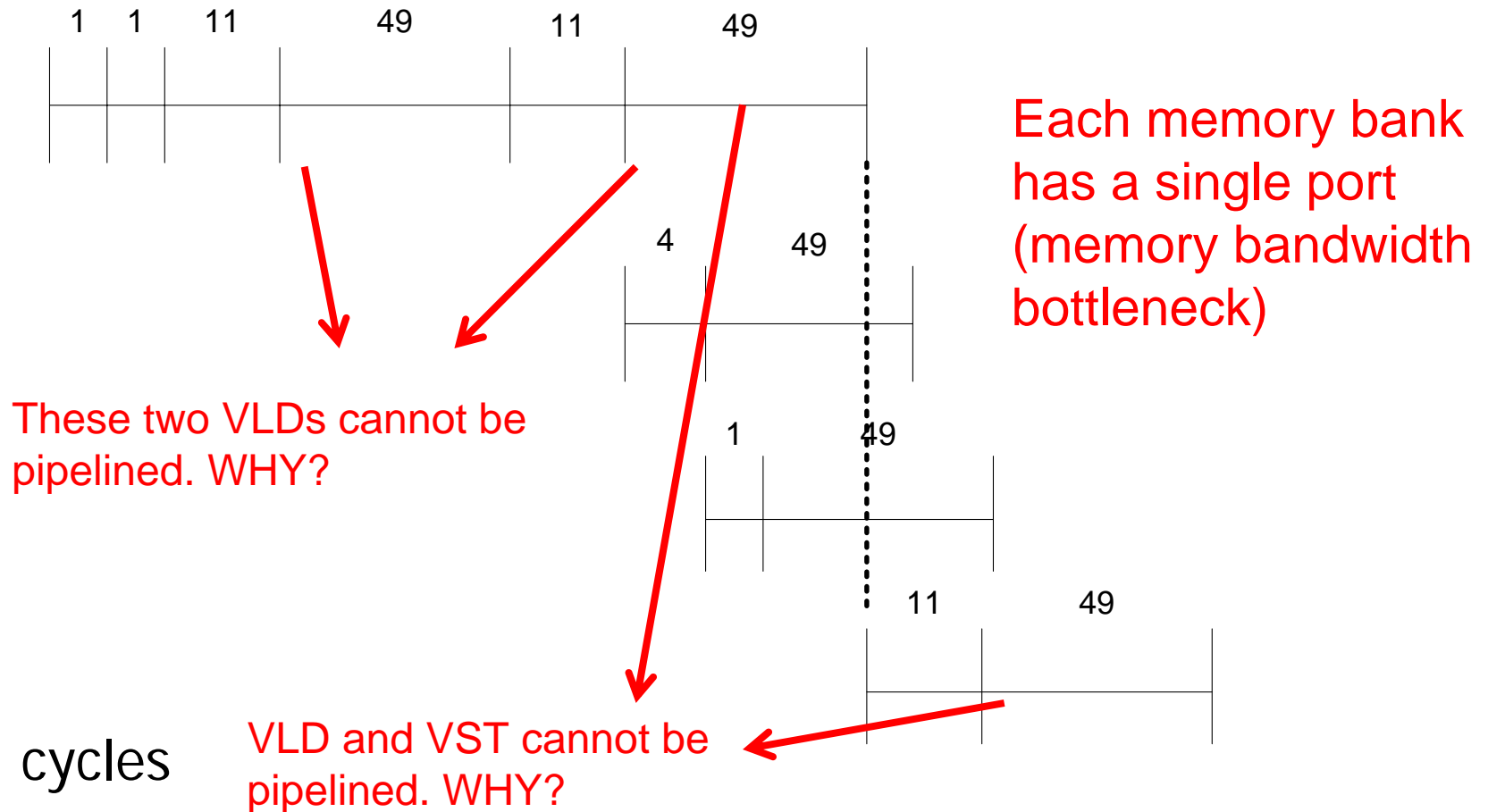
- No chaining
 - i.e., output of a vector functional unit cannot be used as the input of another (i.e., no vector data forwarding)
- 16 memory banks (word-interleaved)



- 285 cycles

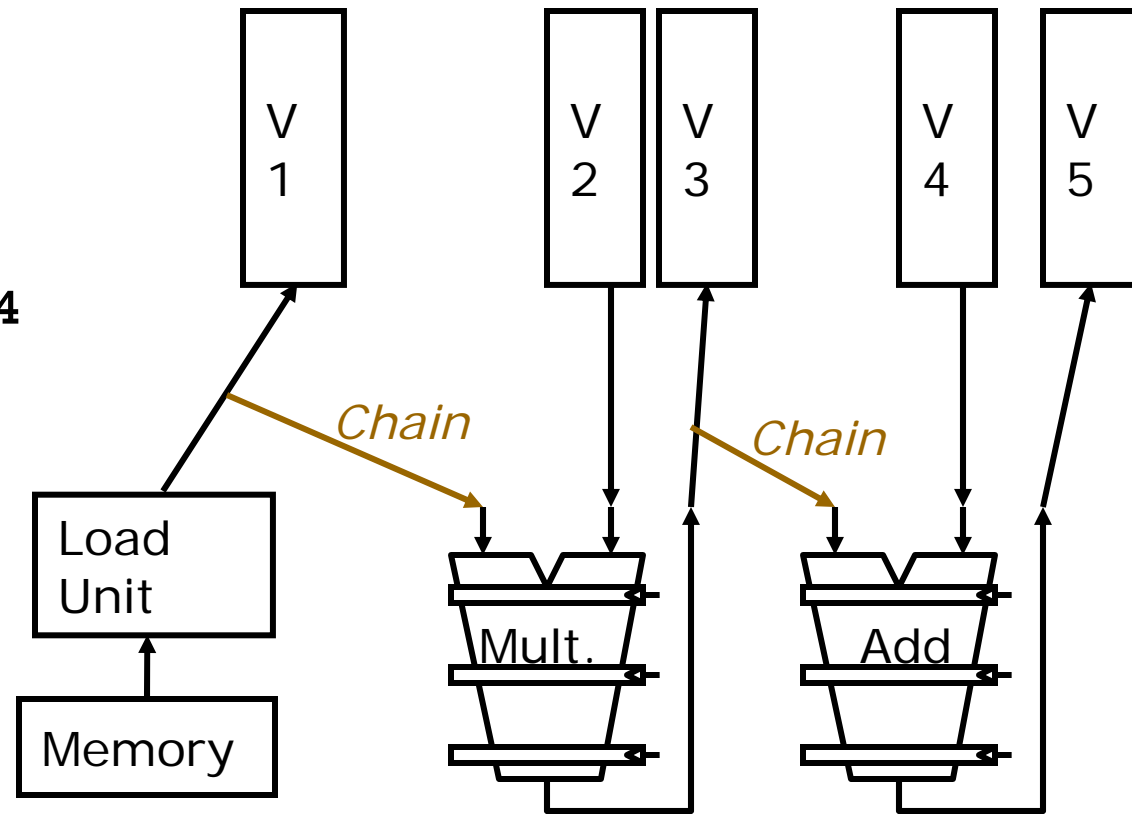
Vector Code Performance - Chaining

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



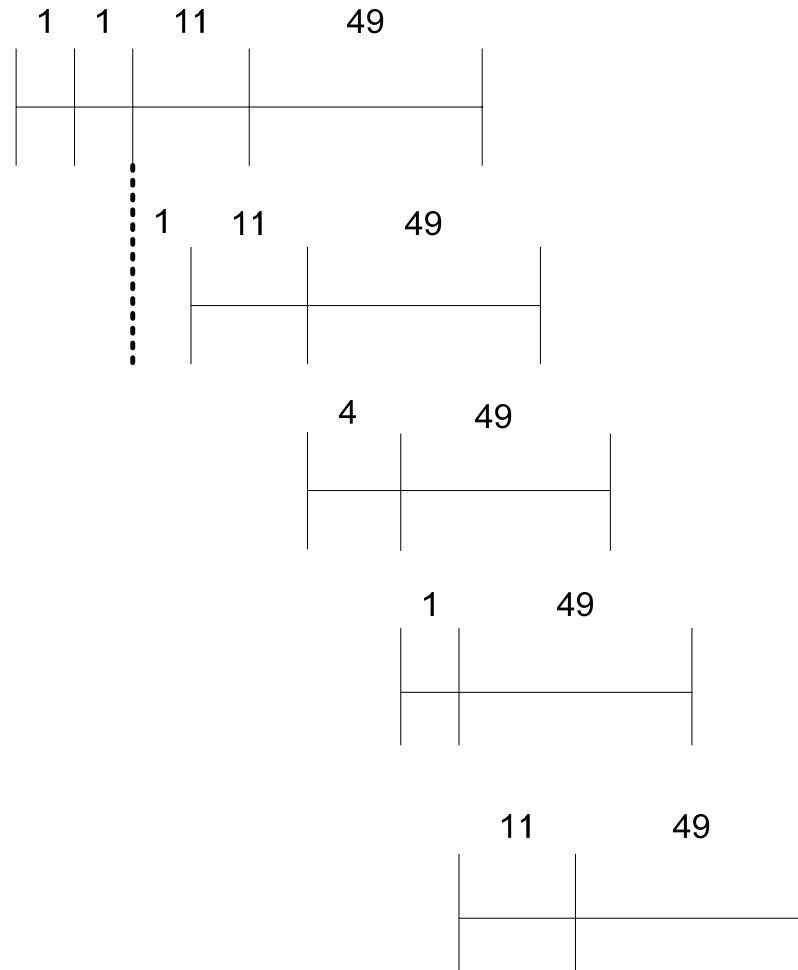
Vector Chaining

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



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](#)

Scatter/Gather 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
```

Scatter/Gather Operations

- Scatter/Gather 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
		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] 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. Selective store of A, VMASK into C
3. Complement VMASK
4. Selective store of B, VMASK into C

Masked Vector Instructions

Simple Implementation

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

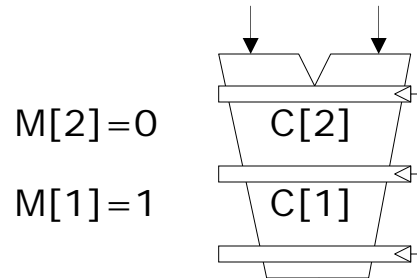
M[7]=1 A[7] B[7]

M[6]=0 A[6] B[6]

M[5]=1 A[5] B[5]

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

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



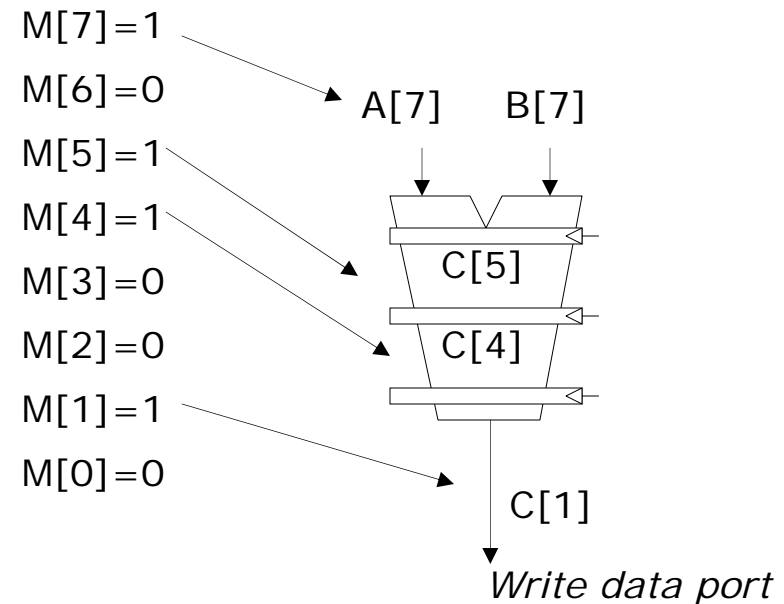
M[2]=0
M[1]=1
M[0]=0

Write Enable

Write data port

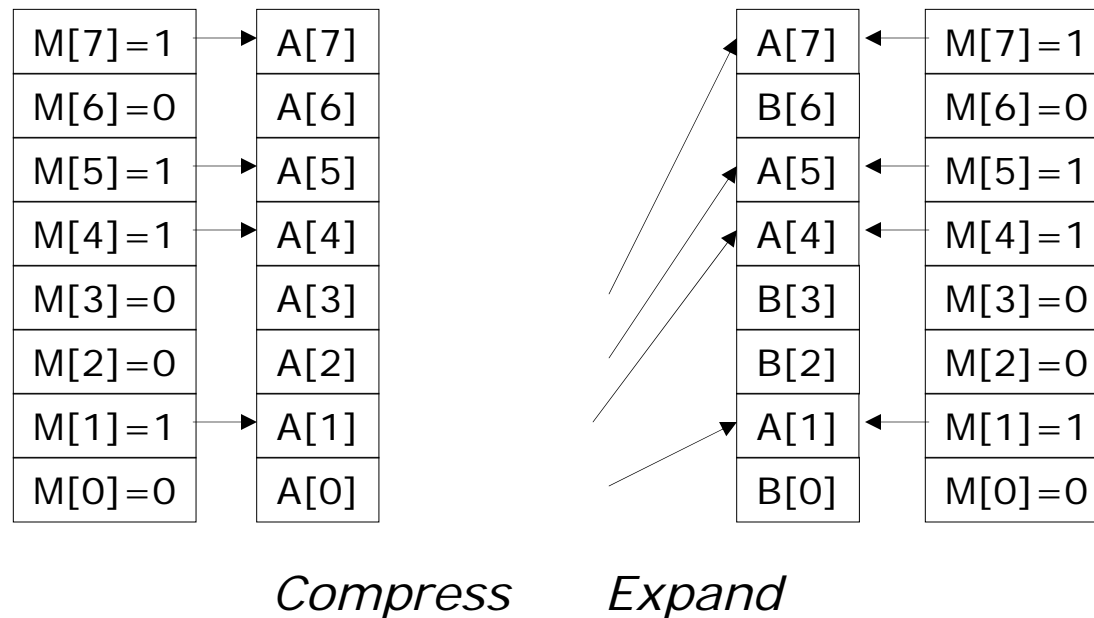
Density-Time Implementation

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



Compress and 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



Reduction Operations

Problem: Loop-carried dependence on reduction variables

```
sum = 0;
for (i=0; i<N; i++)
    sum += A[i]; # Loop-carried dependence on sum
```

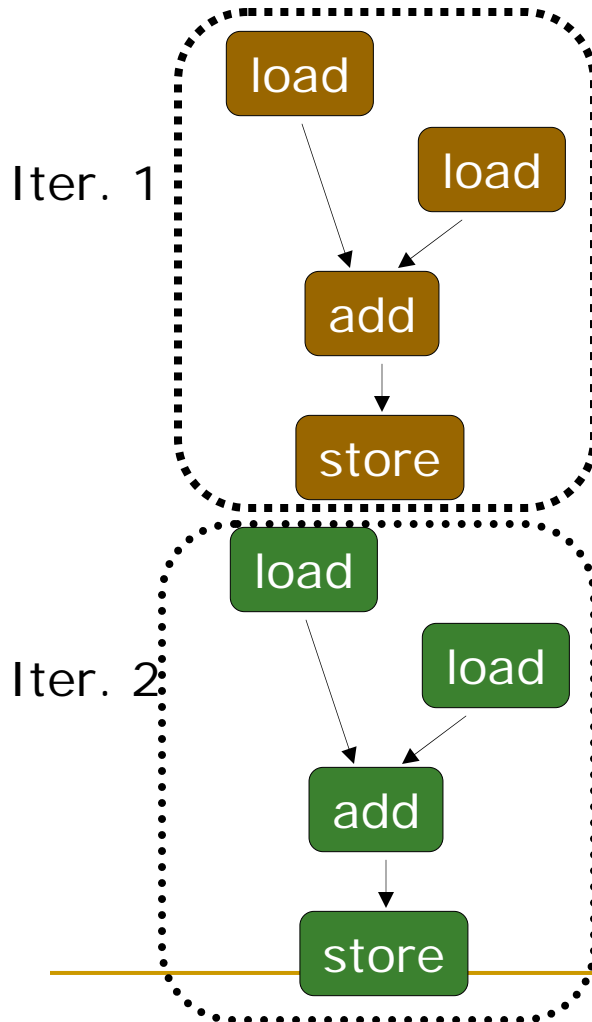
Solution: Re-associate operations if possible, use binary tree to perform reduction

```
# 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)
```

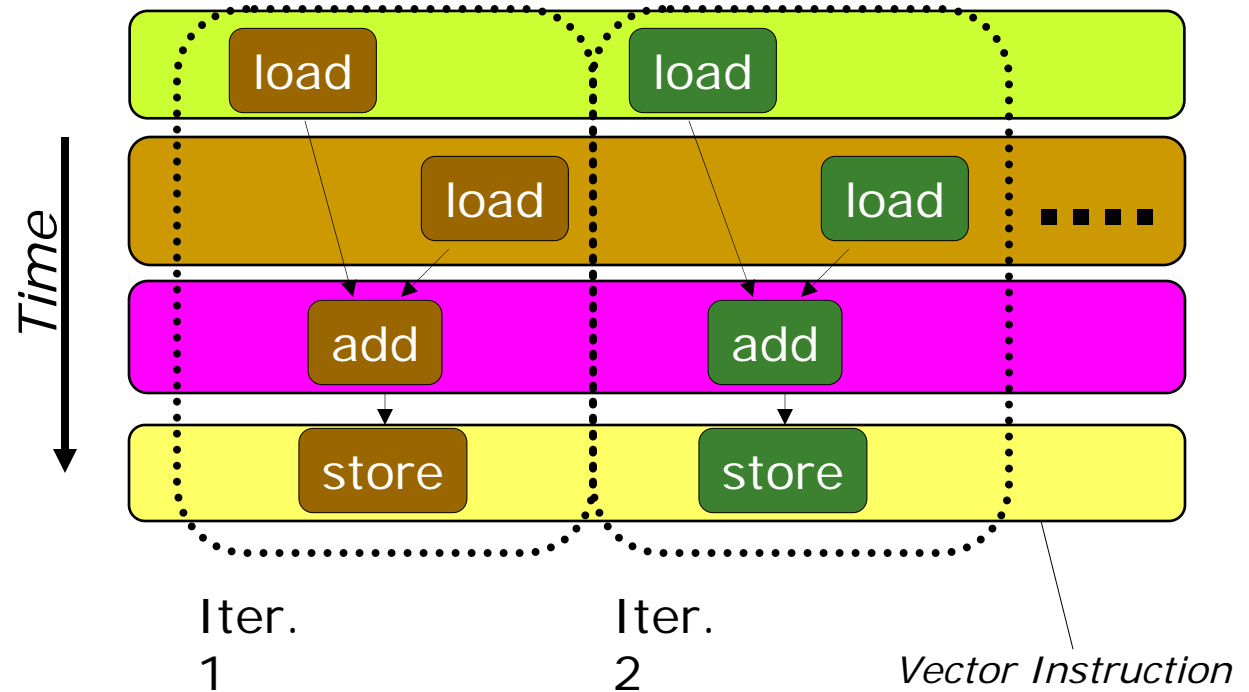
Automatic Code Vectorization

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

Scalar Sequential Code



Vectorized Code



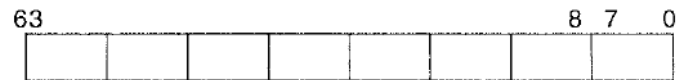
Vectorization is a compile-time reordering of operation sequencing
⇒ requires extensive loop dependence analysis

Vector Processing Summary

- Vector 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, PowerPC AltiVec, ARM Advanced SIMD

Intel Pentium MMX Operations

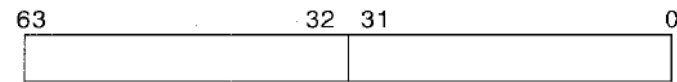
- Idea: One instruction operates on multiple data elements **simultaneously**
 - Ala array processing (yet much more limited)
 - Designed with multimedia (graphics) operations in mind



(a)



(b)



(c)



(d)

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

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)

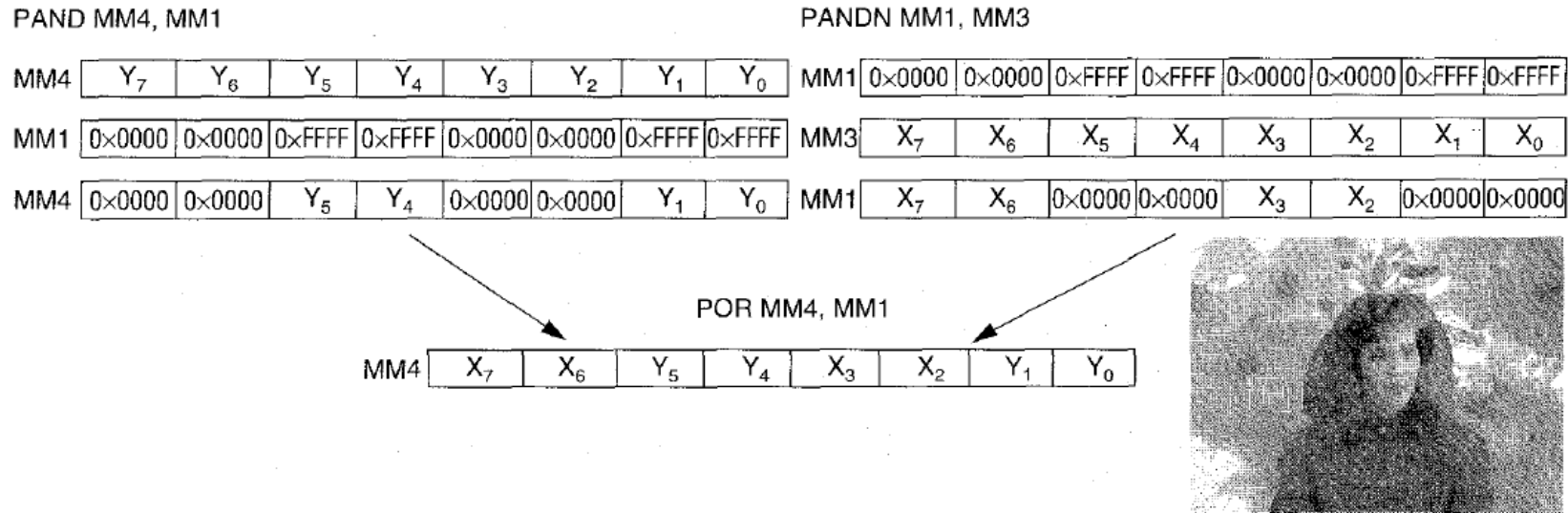


Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

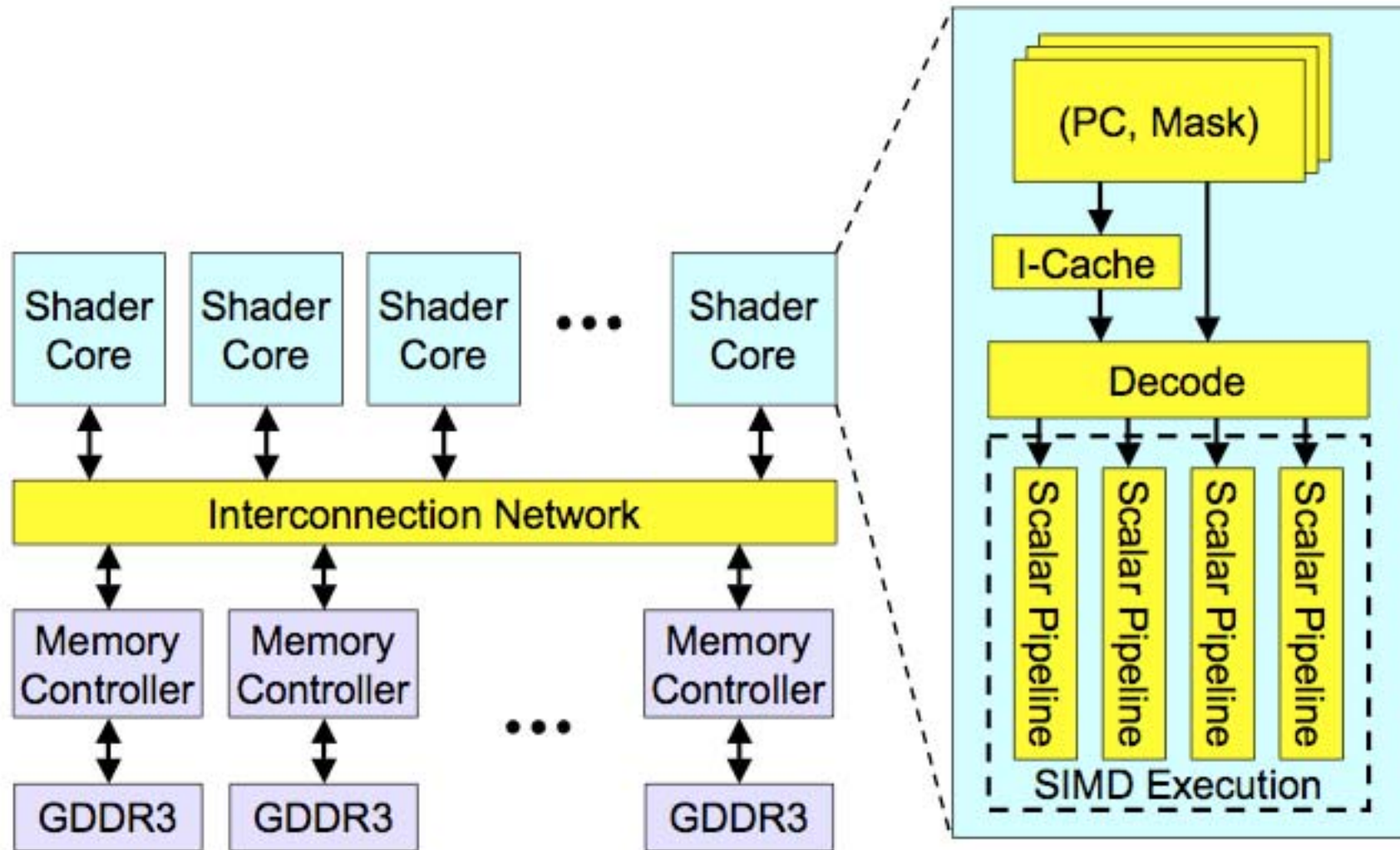
```

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.

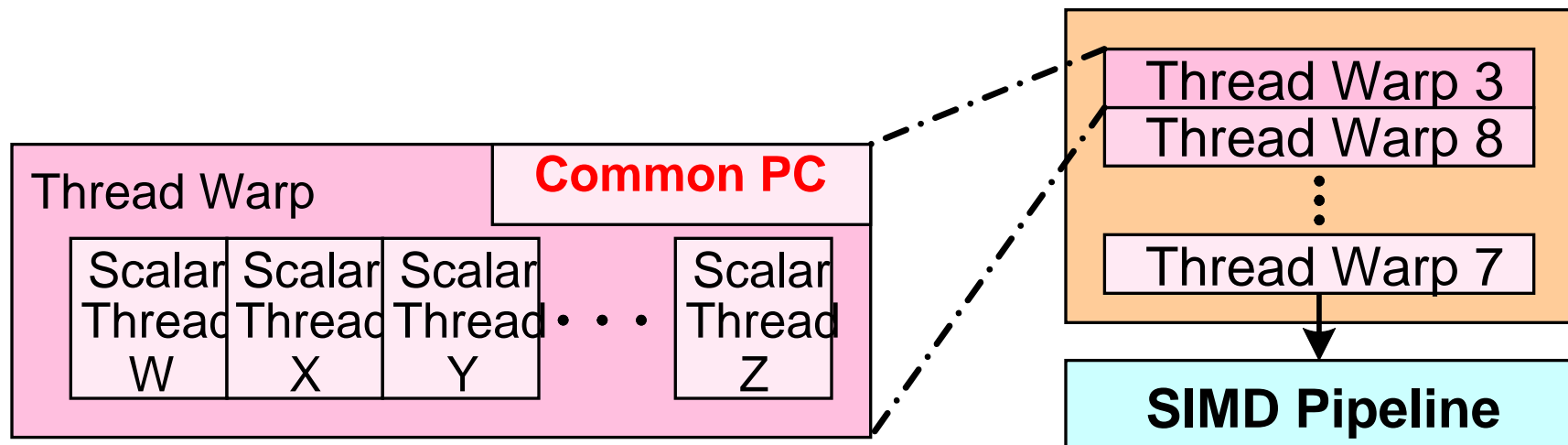
Graphics Processing Units

High-Level View of a GPU



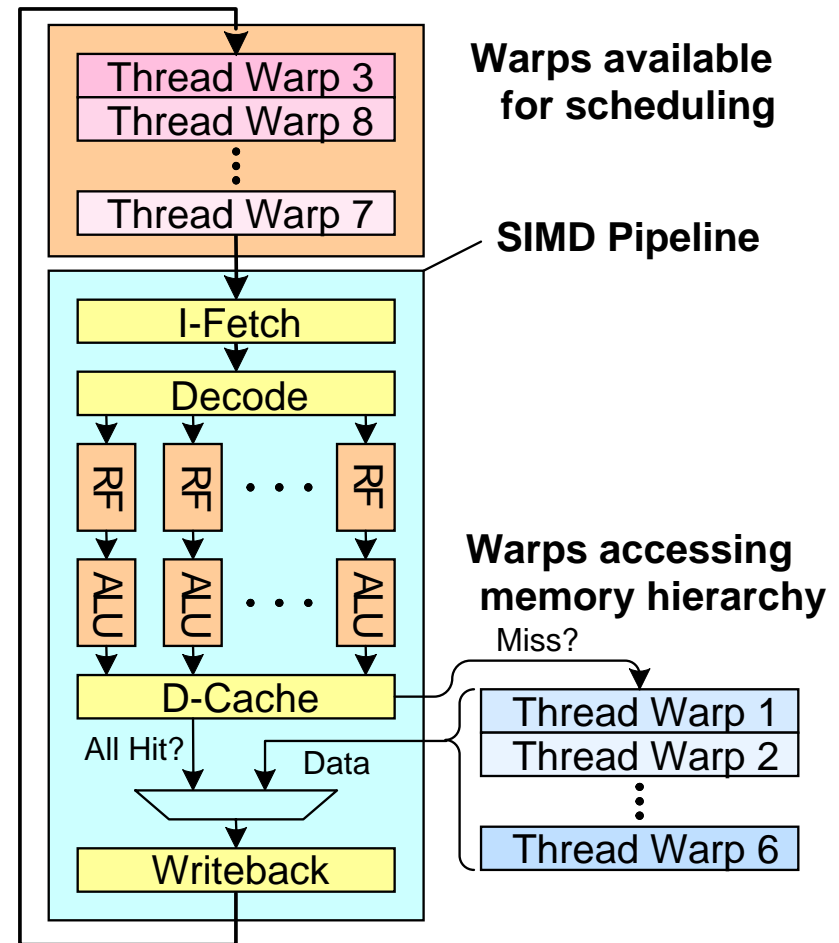
Concept of “Thread Warps”

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



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

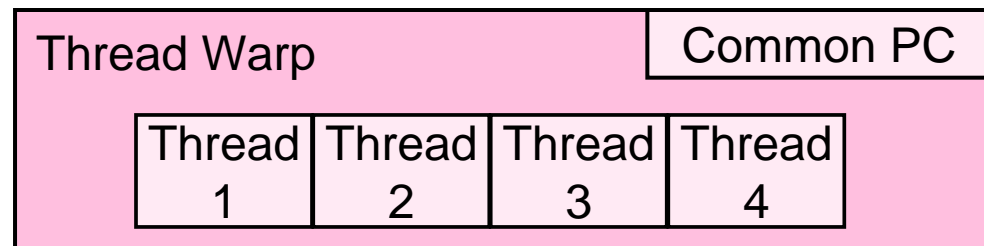
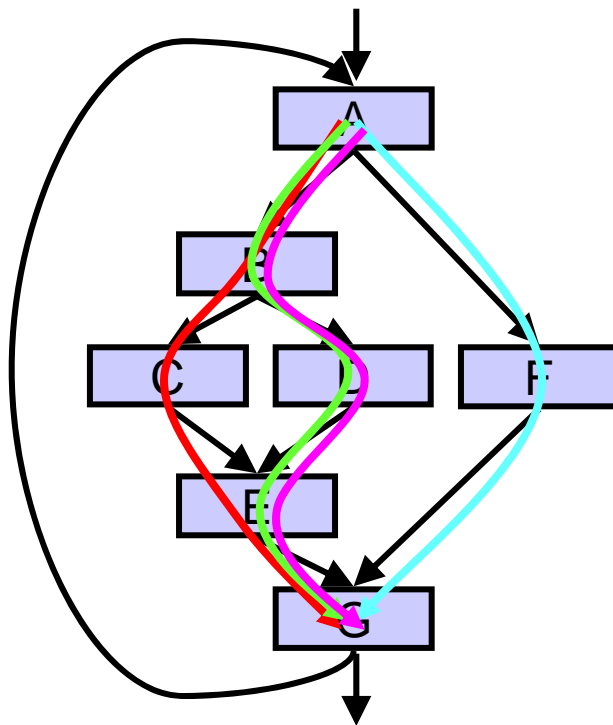


Warp-based SIMD vs. Traditional SIMD

- 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 MIMD/SPMD programming model implemented on SIMD hardware

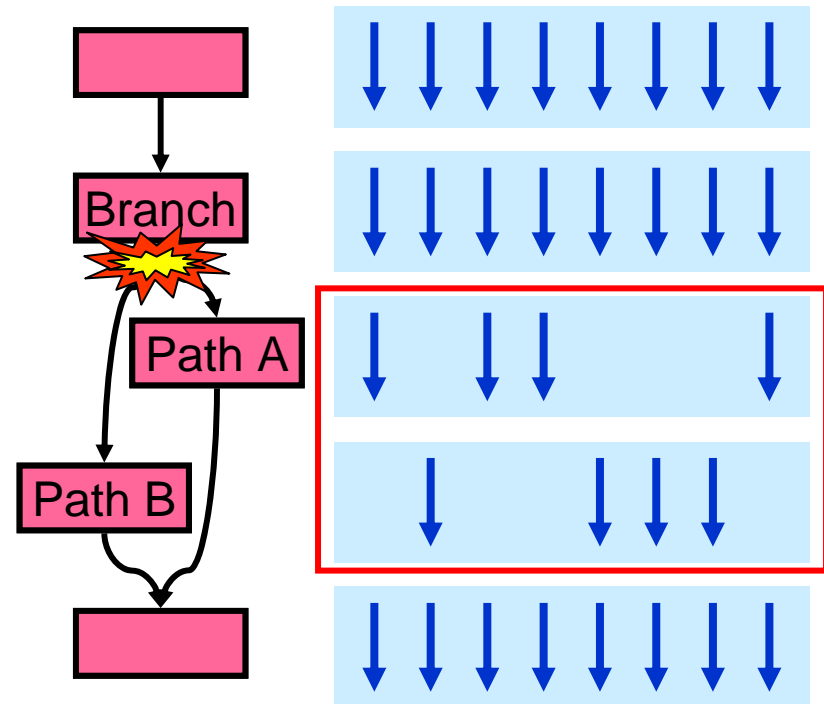
Branch Divergence Problem in Warp-based SIMD

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

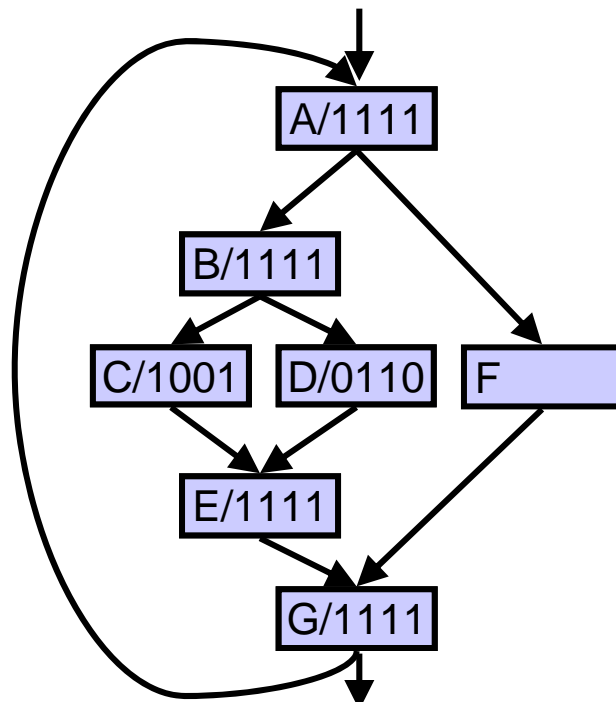


Control Flow Problem in GPUs/SIMD

- 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.

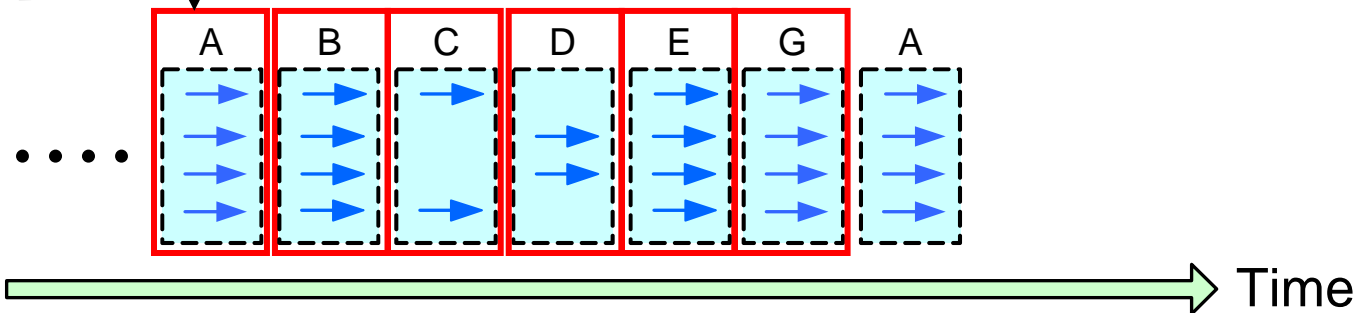
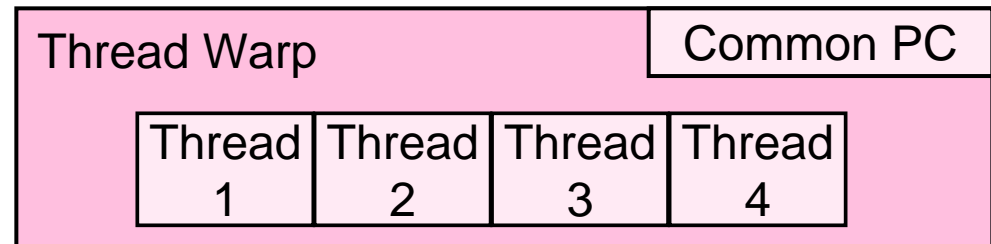


Branch Divergence Handling (I)



Stack

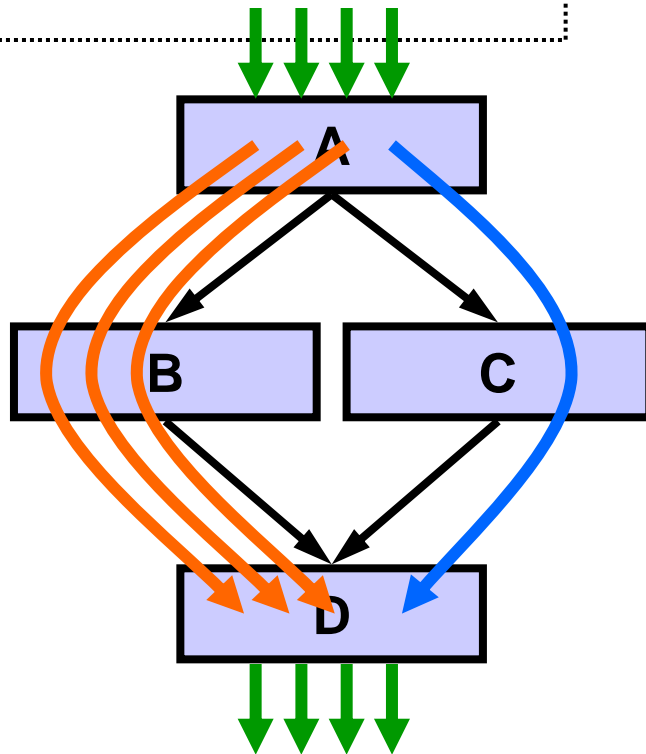
	Reconv. PC	Next PC	Active Mask
TOS →	-	E	1111
TOS →	E	D	0110
TOS →	E	E	1001



Branch Divergence Handling (II)

```

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

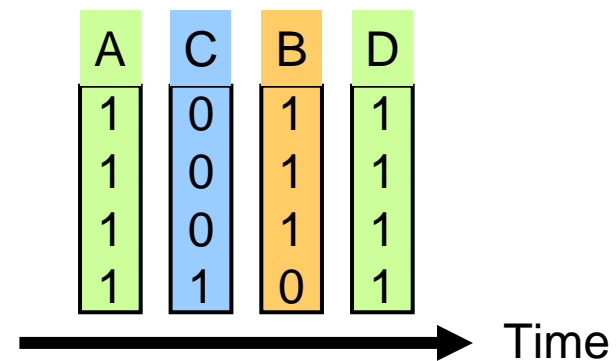


One per warp

Control Flow Stack

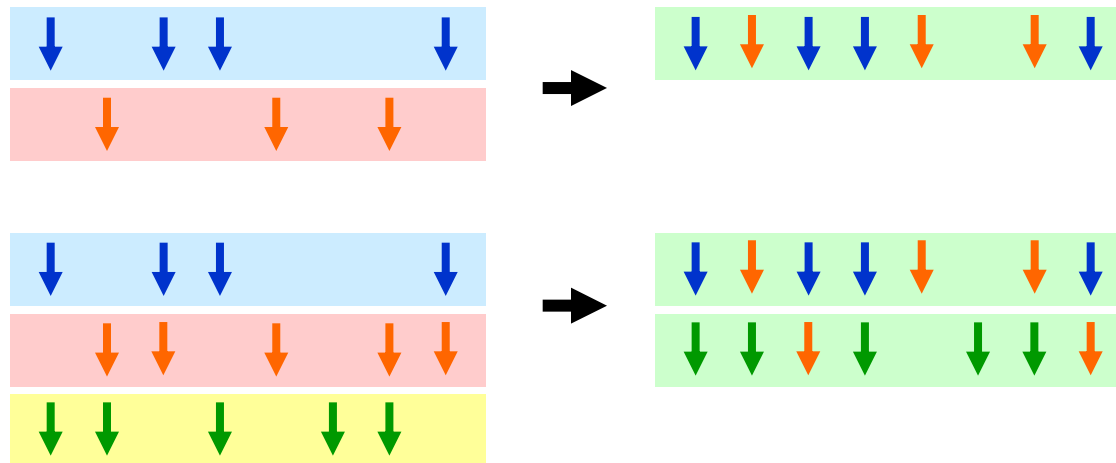
	Next PC	Recv PC	Amask
TOS →	D	--	1111
	B	D	1110
	D	D	0001

Execution Sequence



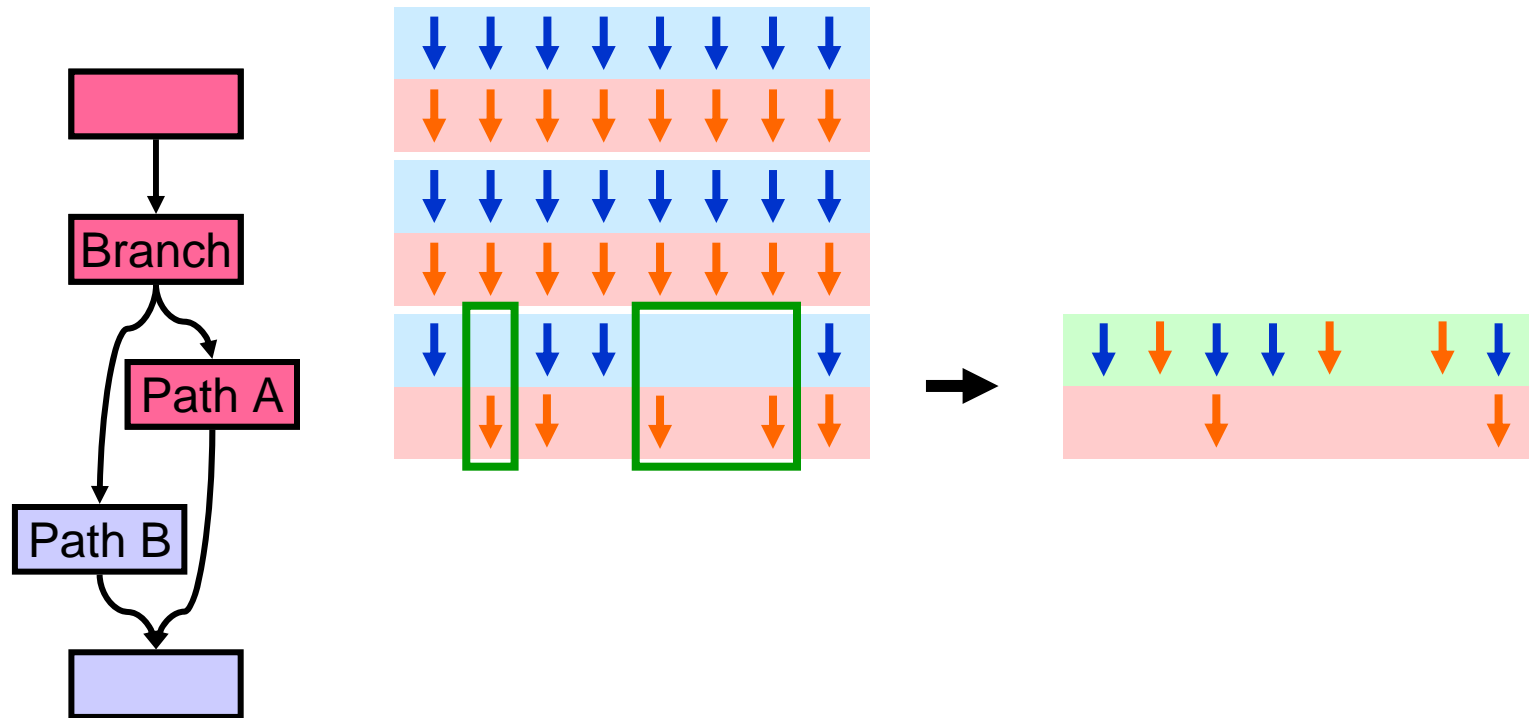
Dynamic Warp Formation

- 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



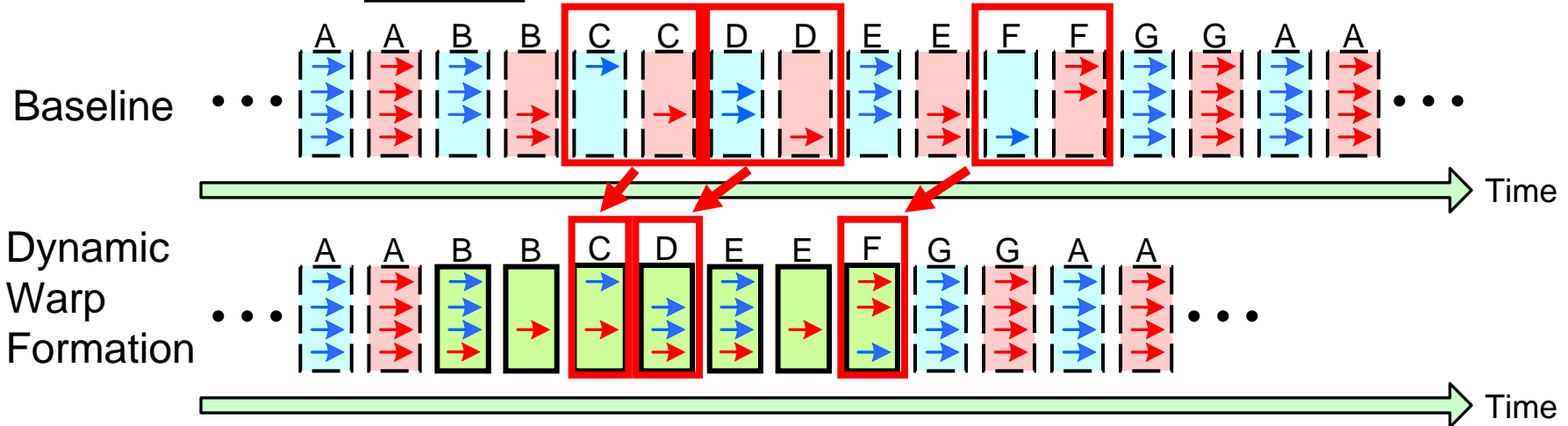
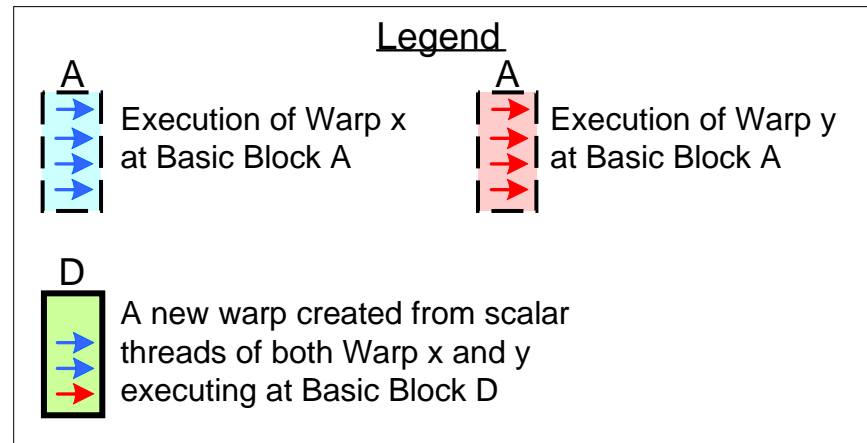
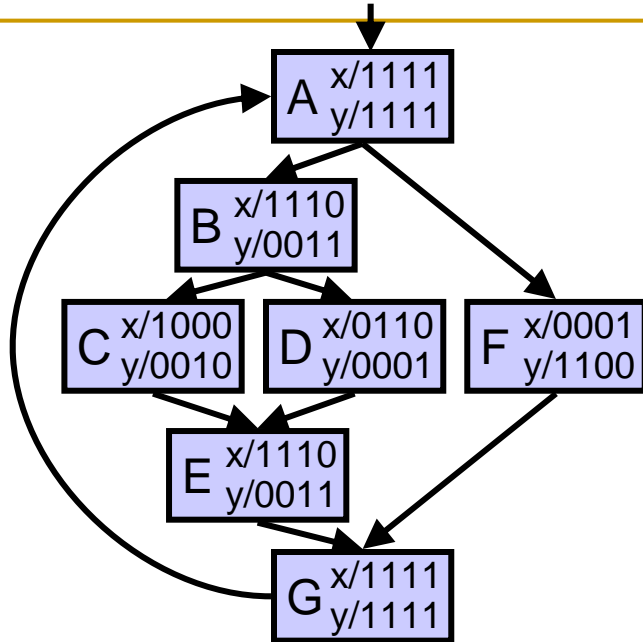
Dynamic Warp Formation/Merging

- 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



How to Fill Holes in Warps?

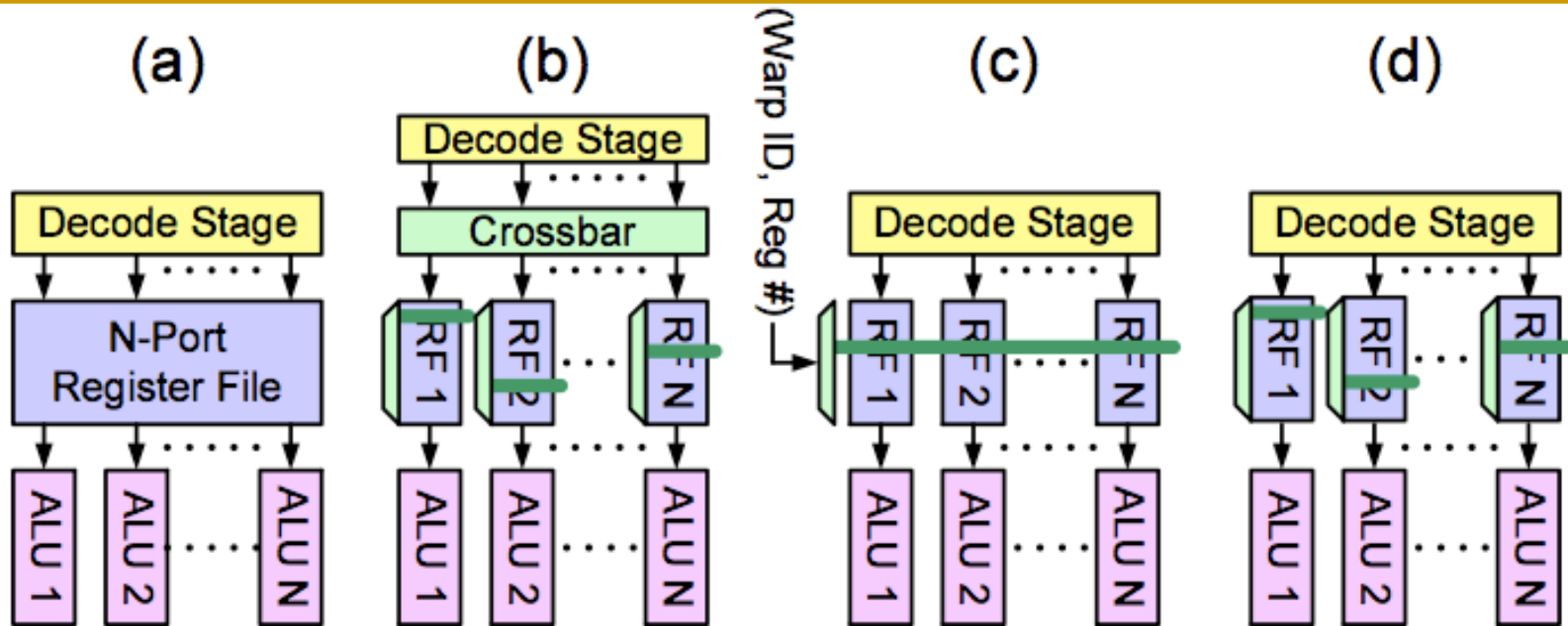


Figure 10. Register file configuration for (a) ideal dynamic warp formation and MIMD, (b) naïve dynamic warp formation, (c) static warp formation, (d) lane-aware dynamic warp formation.

Memory Access within A Warp

- “To improve memory bandwidth and reduce overhead, the local and global load/ store instructions coalesce individual parallel thread accesses from the same warp into fewer memory block accesses.”
- Highest efficiency achieved if individual threads within a warp access consecutive locations in memory → same row
- If threads within a warp conflict with each other, SIMD efficiency degrades significantly → similar to traditional SIMD machines

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.**
- Dynamic warp formation can cause bank conflicts between threads within a warp (if the warp is not formed in a bank-aware manner)
- Need techniques to
 - Tolerate memory divergence
 - Integrate solutions to branch and memory divergence

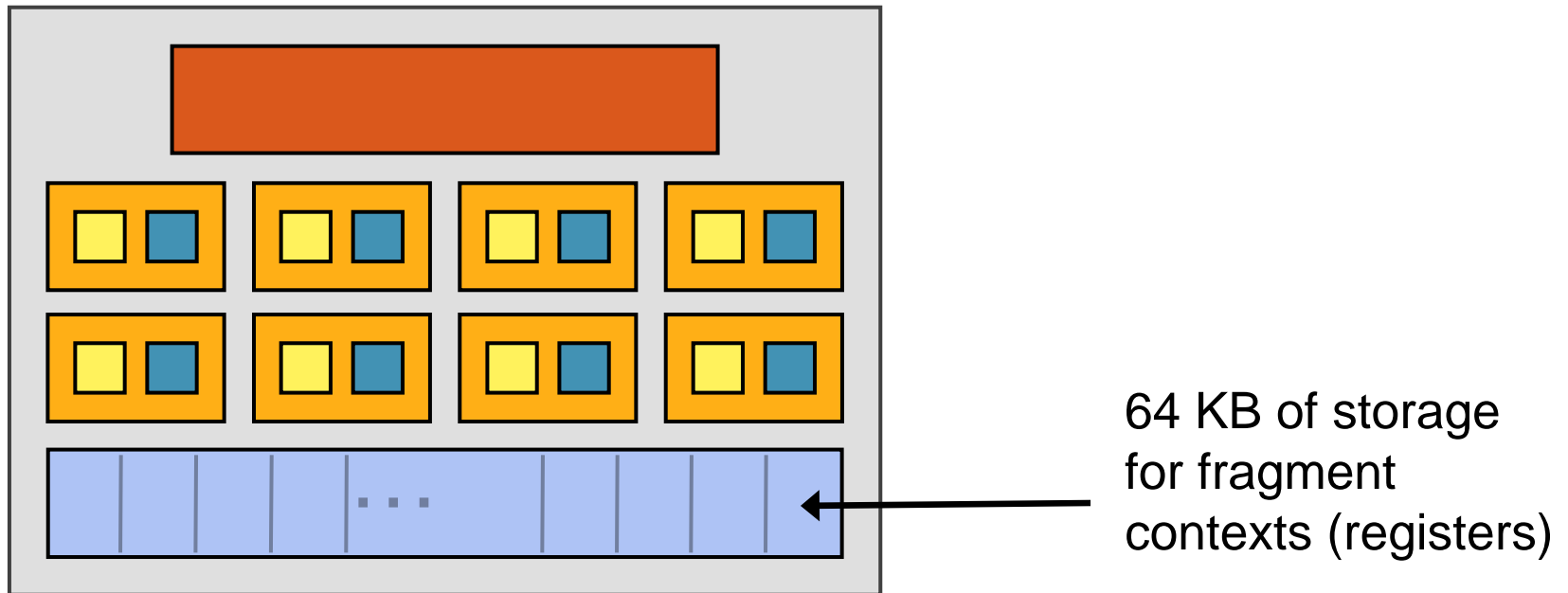
NVIDIA GeForce GTX 285

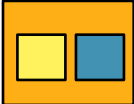
- NVIDIA-speak:
 - 240 stream processors
 - "SIMT execution"

- Generic speak:
 - 30 cores
 - 8 SIMD functional units per core



NVIDIA GeForce GTX 285 “core”



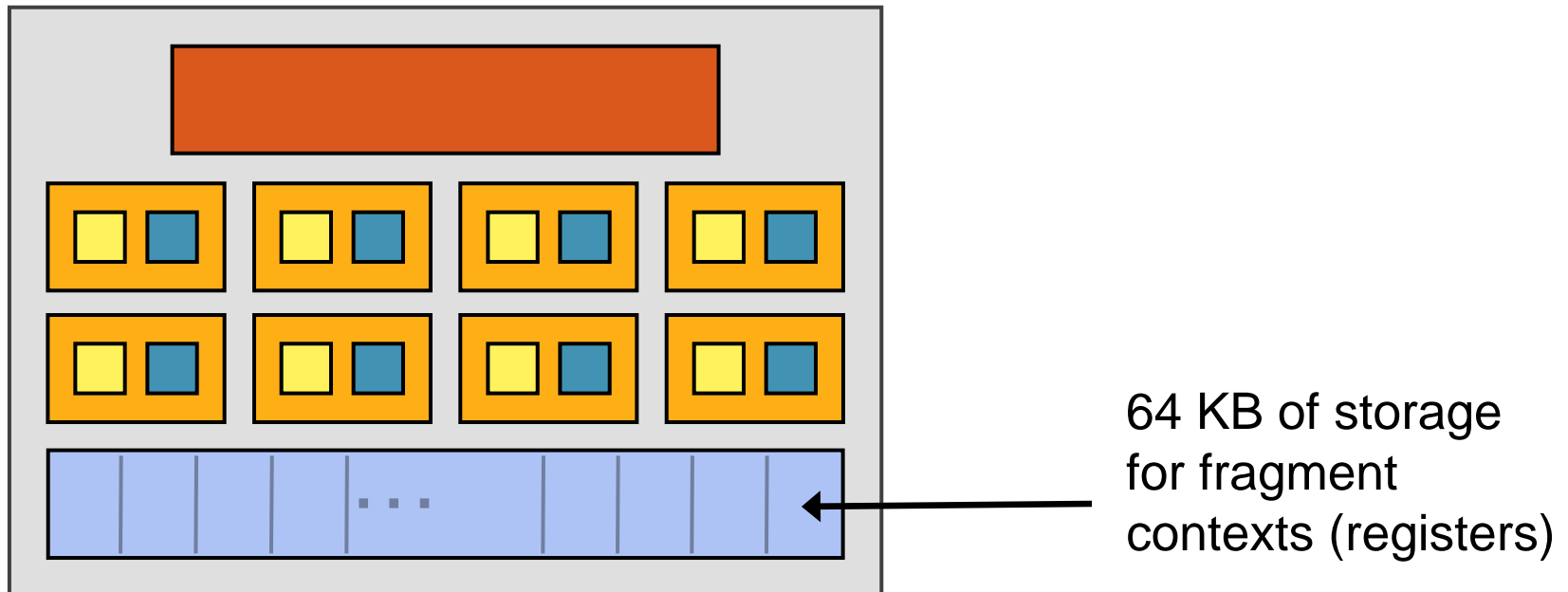
 = SIMD functional unit, control shared across 8 units

 = multiply-add
 = multiply

 = instruction stream decode

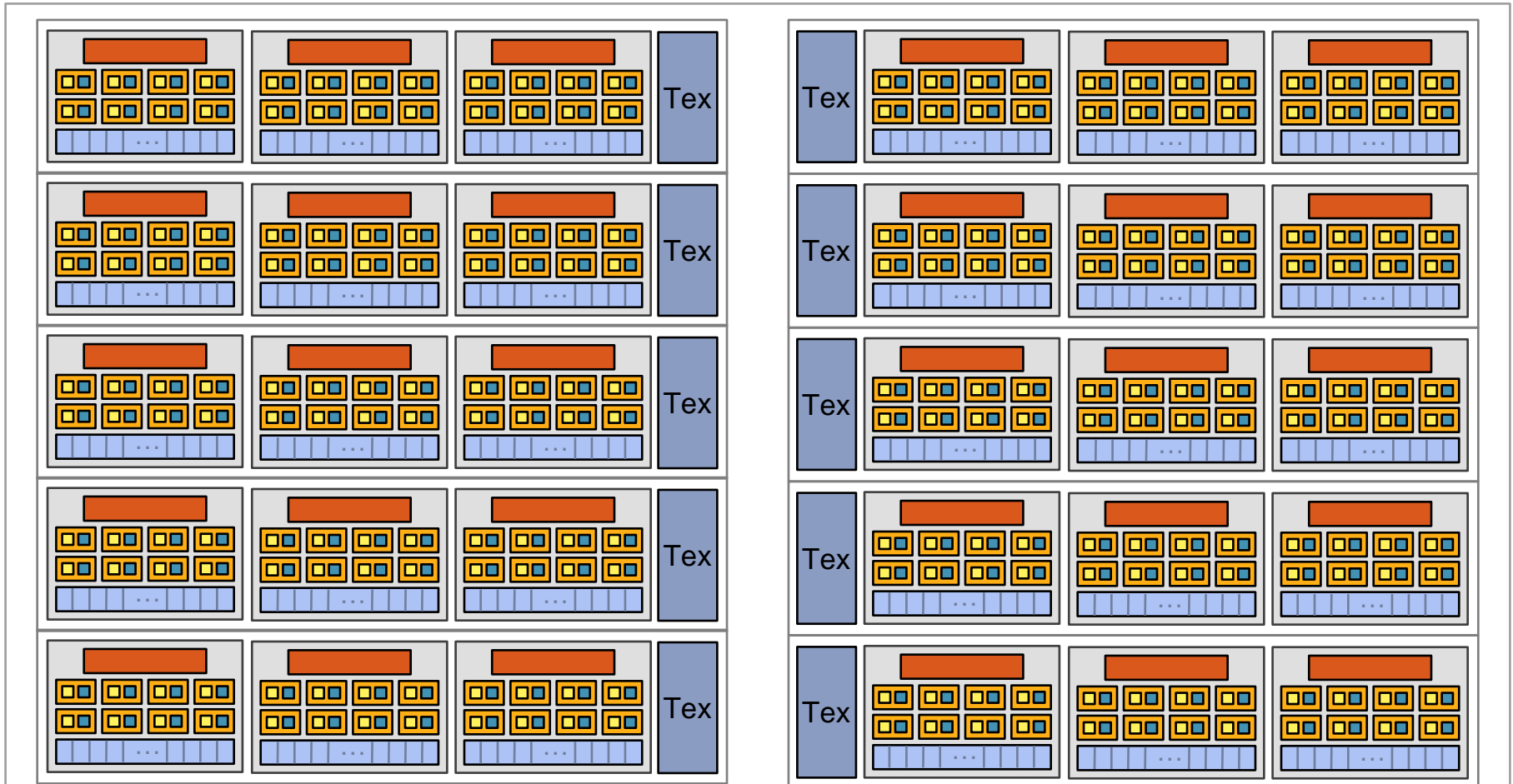
 = execution context storage

NVIDIA GeForce GTX 285 “core”



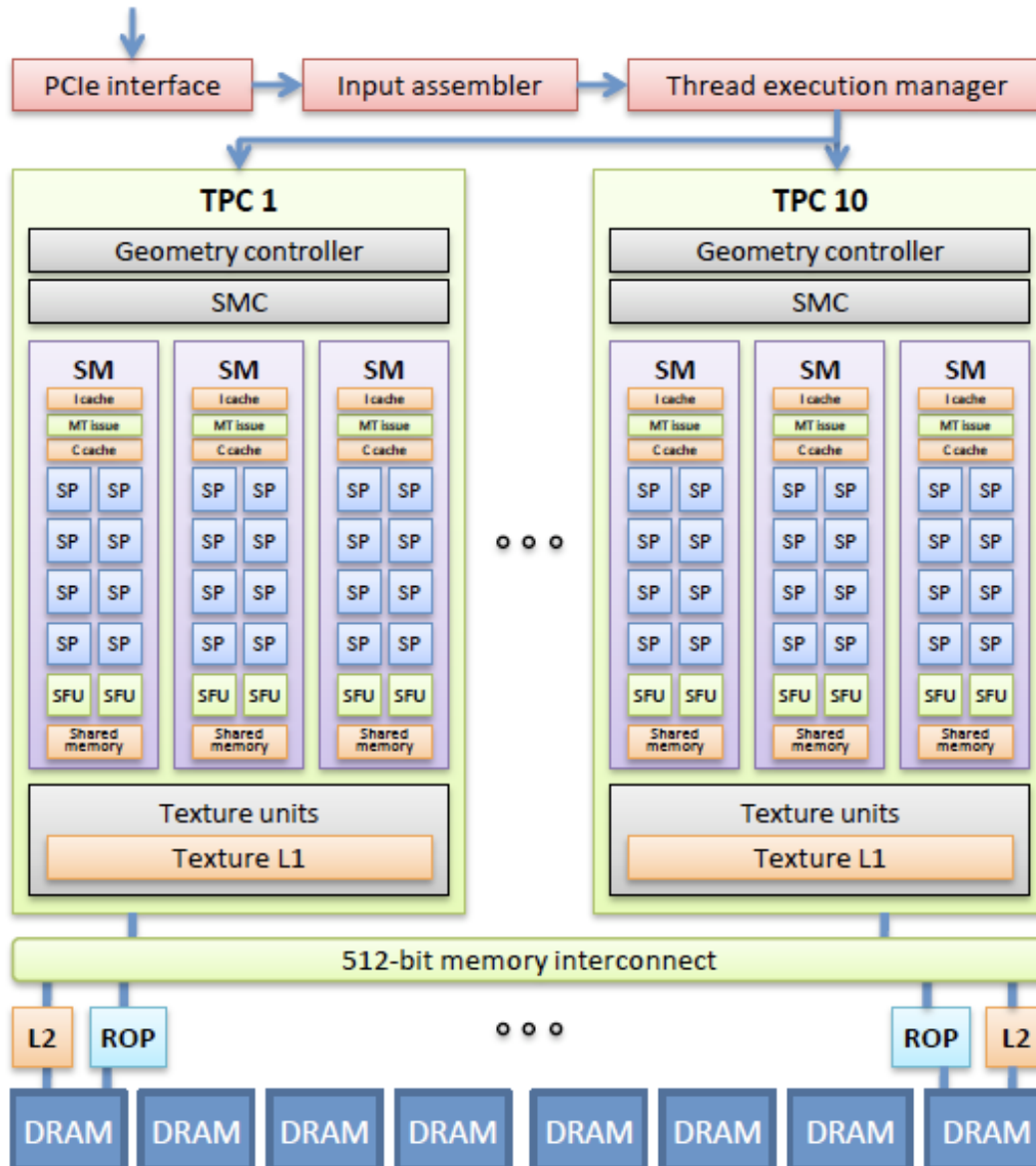
- Groups of 32 [fragments/vertices/**threads**/etc.] share instruction stream (each group is a Warp)
- Up to 32 warps are simultaneously interleaved
- Up to 1024 thread contexts can be stored

NVIDIA GeForce GTX 285



There are 30 of these things on the GTX 285: 30,720 threads

A More Detailed View



- Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.

NVIDIA GeForce GTX 285

- Generic speak:
 - 30 processing cores
 - 8 SIMD functional units per core
 - Best case: 240 mul-adds + 240 muls per clock

Food for Thought

