18-447
Computer Architecture
Lecture 19: SIMD and GPUs

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2013, 3/18/2013

Reminder: Homework 4 Due Today

- Homework 4
  - Due Mar 18
  - Branch prediction, out-of-order execution, data flow, scoreboarding

Lab Assignment 4 Due March 22 (Friday)

- Lab Assignment 4
  - Due Friday, March 22
  - Branch prediction in MIPS implementation in Verilog
  - Global and hybrid branch predictors
  - All labs are individual assignments
  - No collaboration; please respect the honor code

  - Extra credit: Optimize for execution time!
    - Top assignments with lowest execution times will get extra credit.
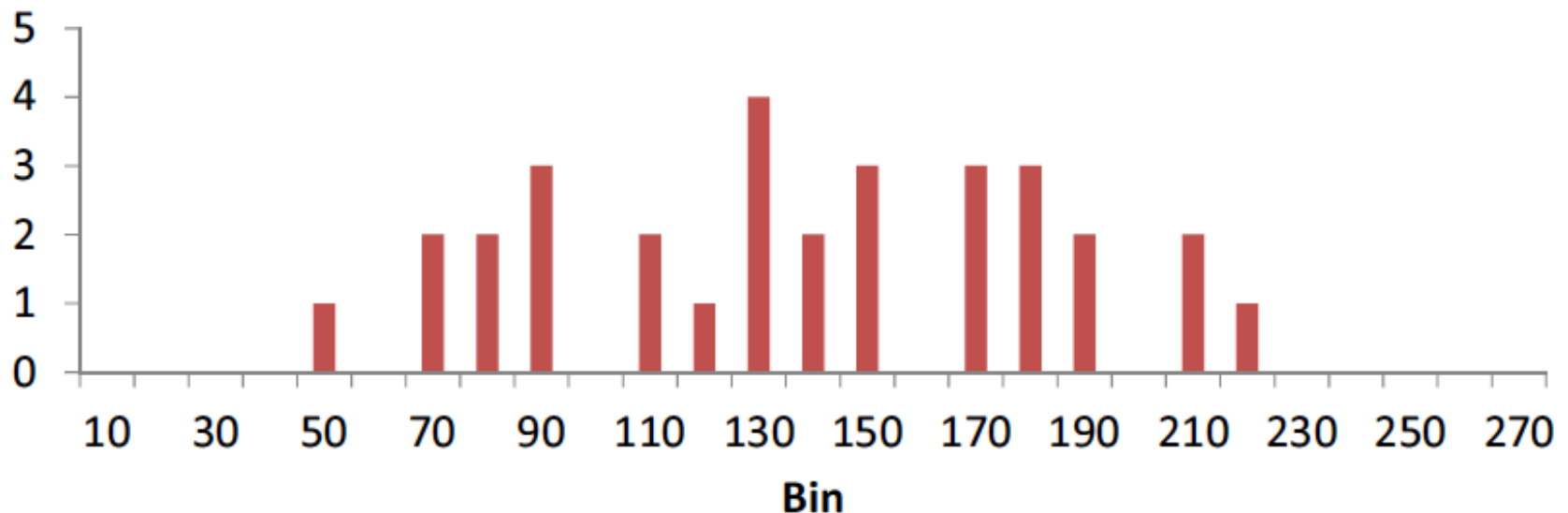   - And, it will be fun to optimize…

Midterm I Solutions and Grades

- Midterm I Solutions posted
  - Please check them and use them for your learning

- Midterm I Grades posted
  - Distributions are also online

Midterm I Grade Distribution

- Total = 270 + 45 (Bonus)
- Average = 134
- Median = 133
- Maximum = 215
- Minimum = 45
- Standard Deviation: 46

Midterm I Post Mortem

- We will distribute the midterms at the end of lecture today

- Remember this is only 15% of your grade
  - We will take into account your progress throughout the course in determining your final grade

- Recitation this Friday will cover Midterm I solutions
  - As well as anything else you will ask

Readings for Today

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

- Fatahalian and Houston, "A Closer Look at GPUs," CACM 2008.

- See slides today for more readings (optional but recommended)

Readings for Next Week

- Memory Hierarchy and Caches

- Cache chapters from P&H: 5.1-5.3
- Memory/cache chapters from Hamacher+: 8.1-8.7
- An early cache paper by Maurice Wilkes
  - Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.

Last Lectures

- Memory Dependence Handling in OoO Processors
- Data Flow
- SIMD Processing
  - Array vs. vector processors

- Virtual Memory (Justin and Yoongu)

Today

- SIMD Processing
- GPU Fundamentals
- VLIW

Approaches to (Instruction-Level) Concurrency

- Pipelined execution
- Out-of-order execution
- Dataflow (at the ISA level)
- SIMD Processing
- VLIW

- Systolic Arrays
- Decoupled Access Execute

Review: 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

Review: SIMD Array Processing vs. VLIW

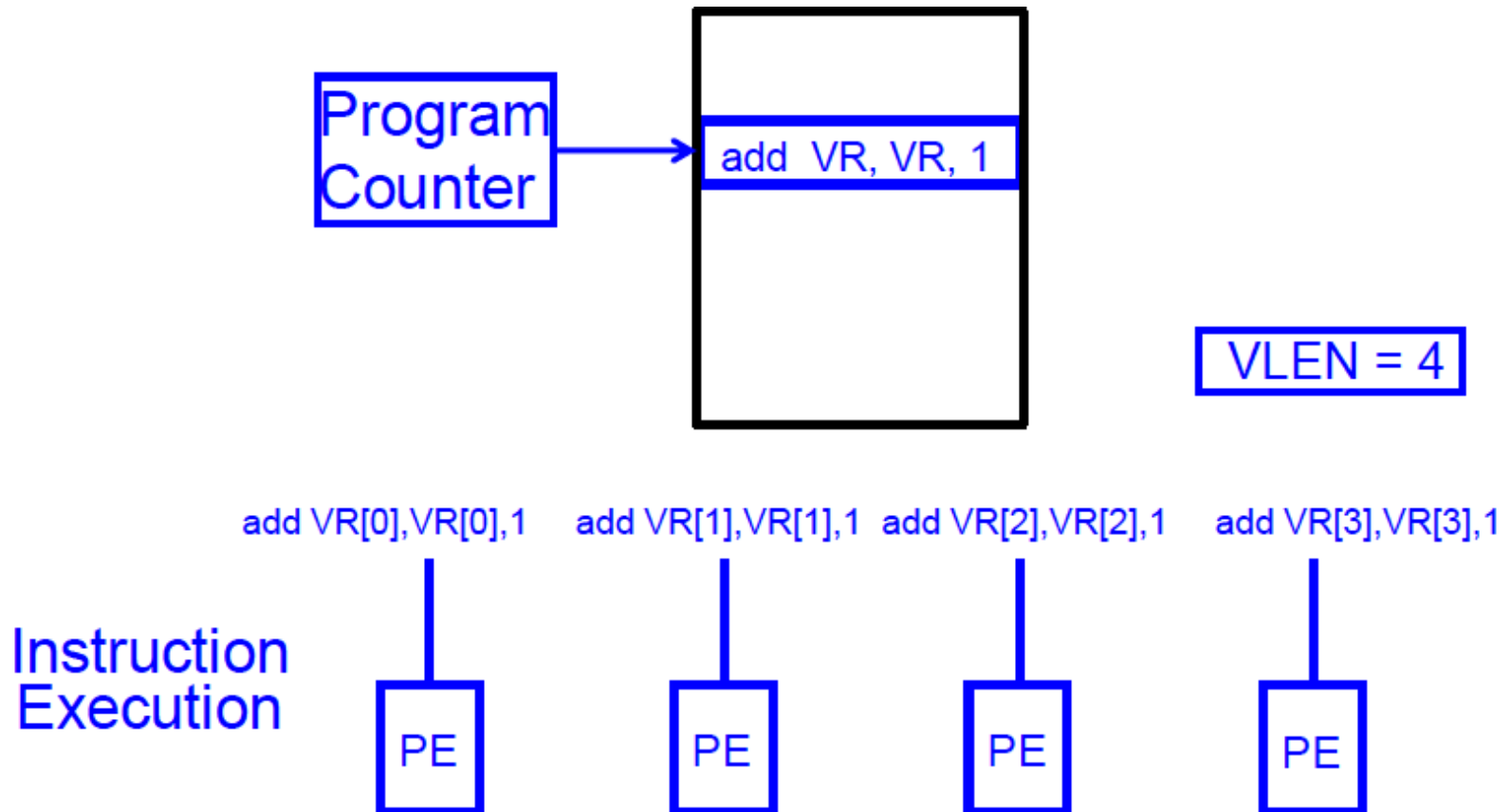- VLIW
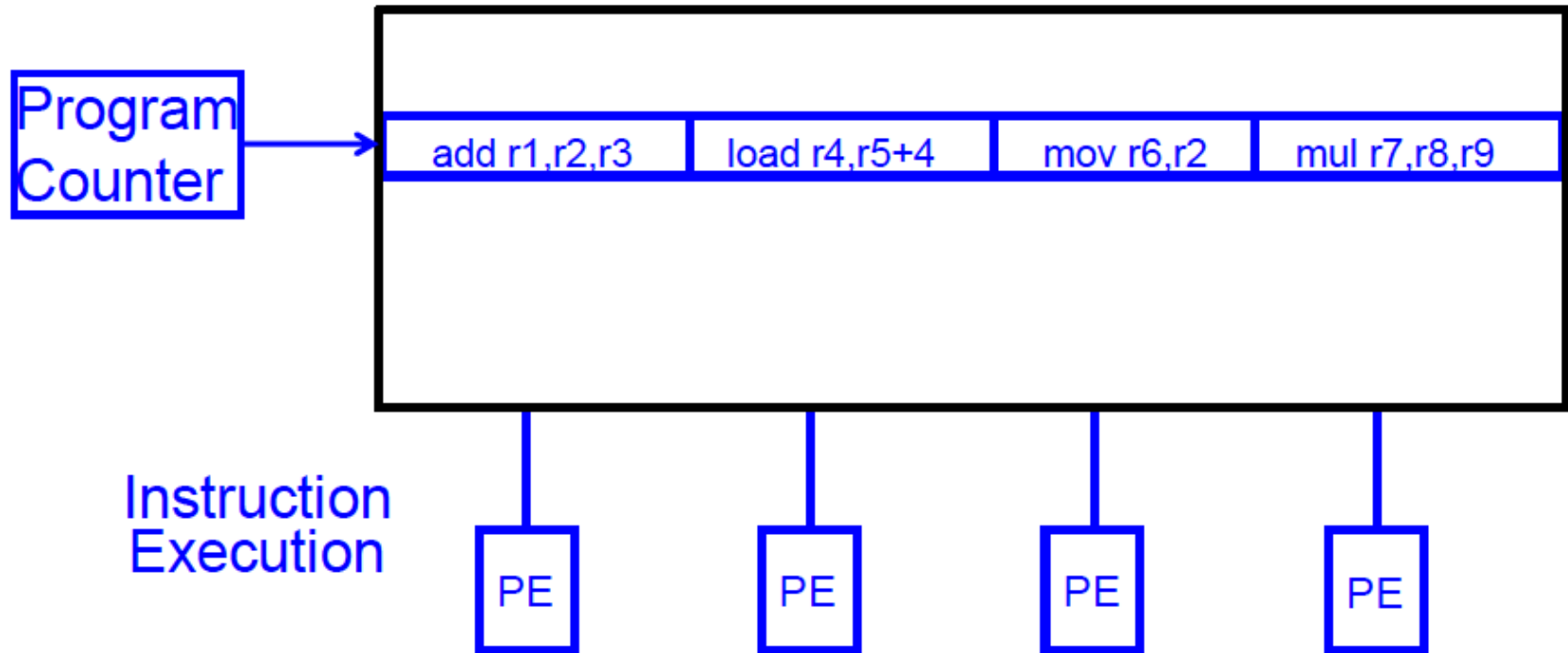
Review: SIMD Array Processing vs. VLIW

- Array processor

Review: 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

Review: 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

Review: 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

Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.

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

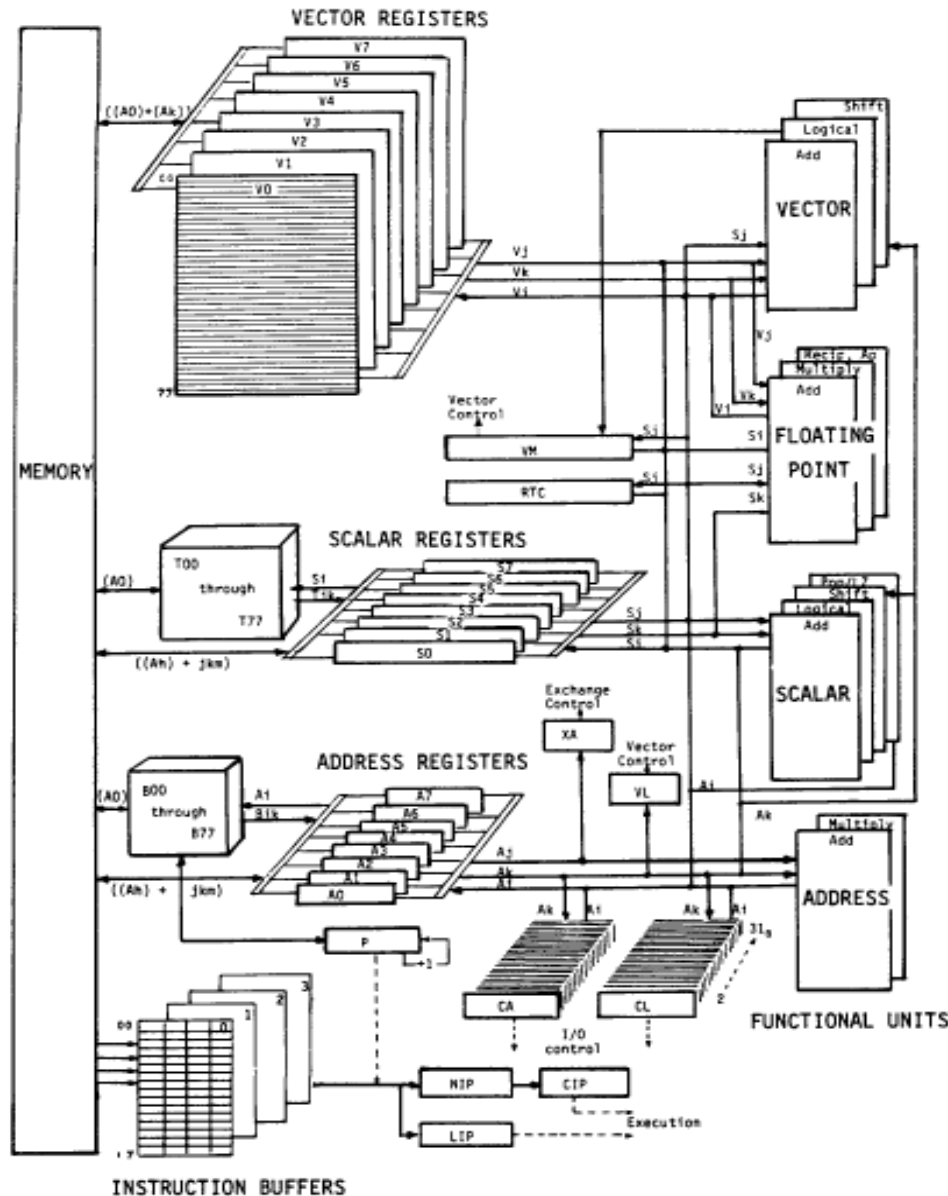| M-bit wide		M-bit wide		
V0,0		V1,0		
V0,1		V1,1		
V0,N-1		V1,N-1		
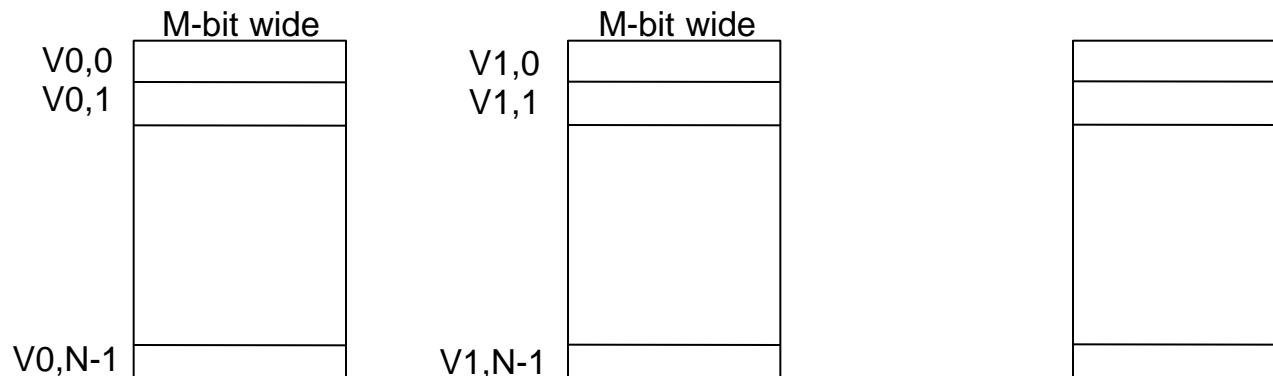
Vector Functional Units

- Use deep pipeline (=> fast clock) to execute element operations

- Simplifies control of deep pipeline because elements in vector are independent

V1   V2   V3

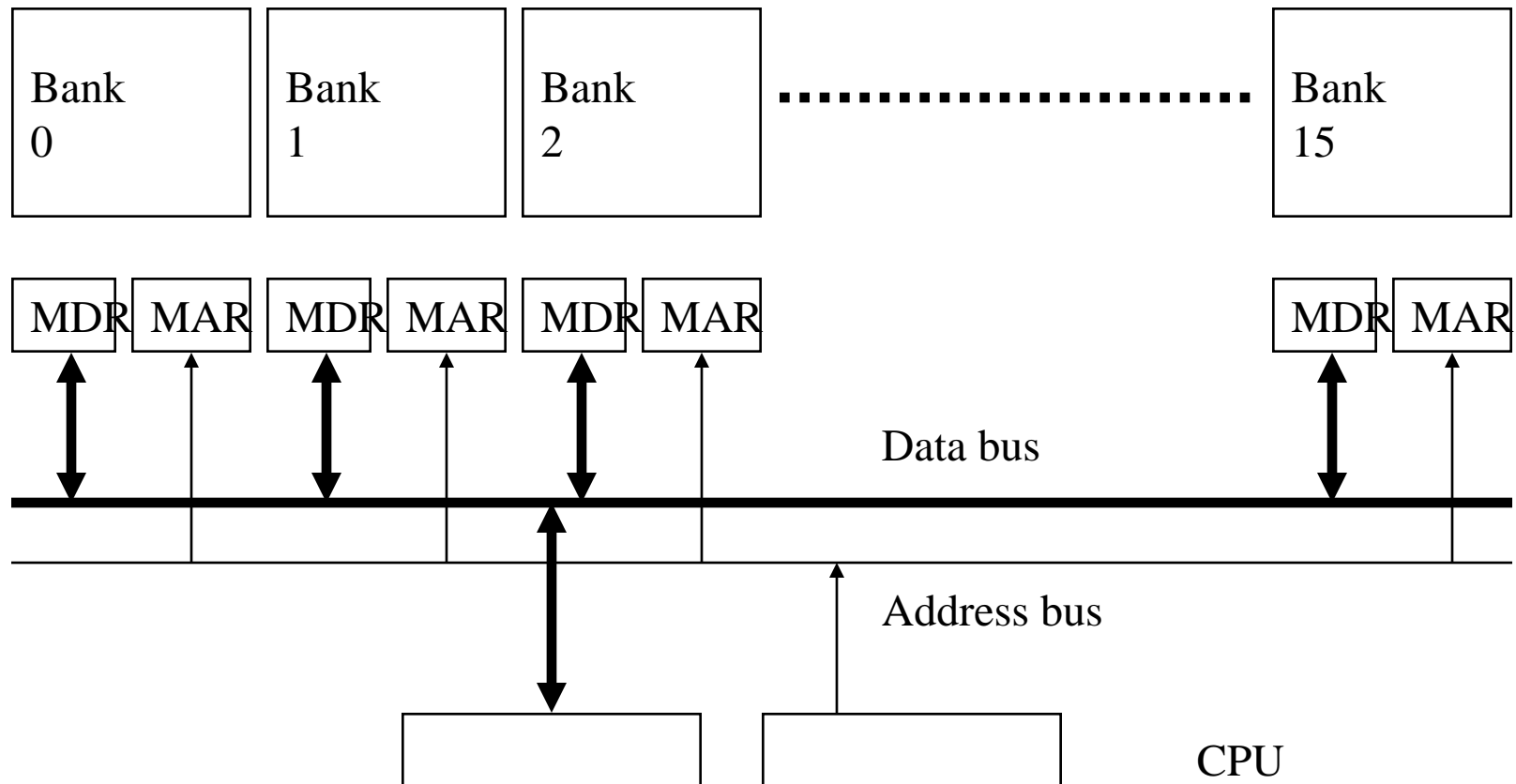*Six stage multiply pipeline*
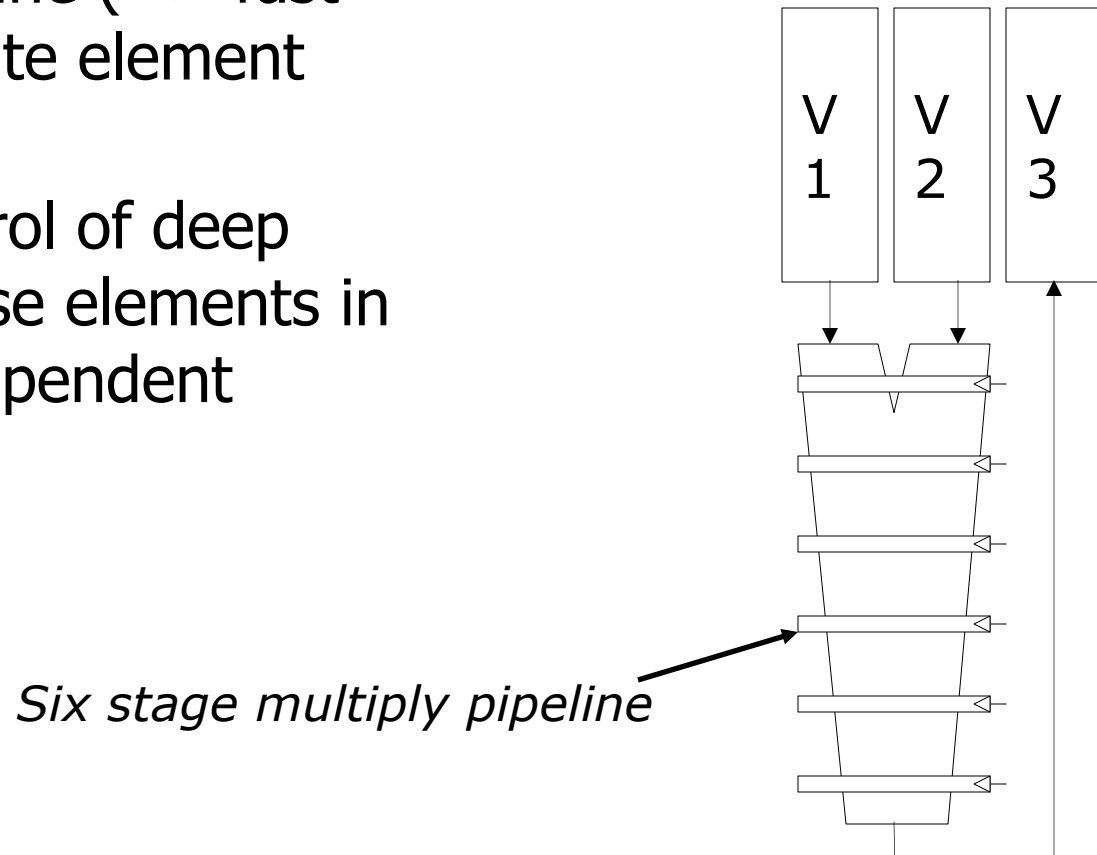
V3 <- v1 * v2

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

Vector Registers

Base

Stride

Address Generator

+

0 1 2 3 4 5 6 7 8 9 A B C D E F

Memory Banks
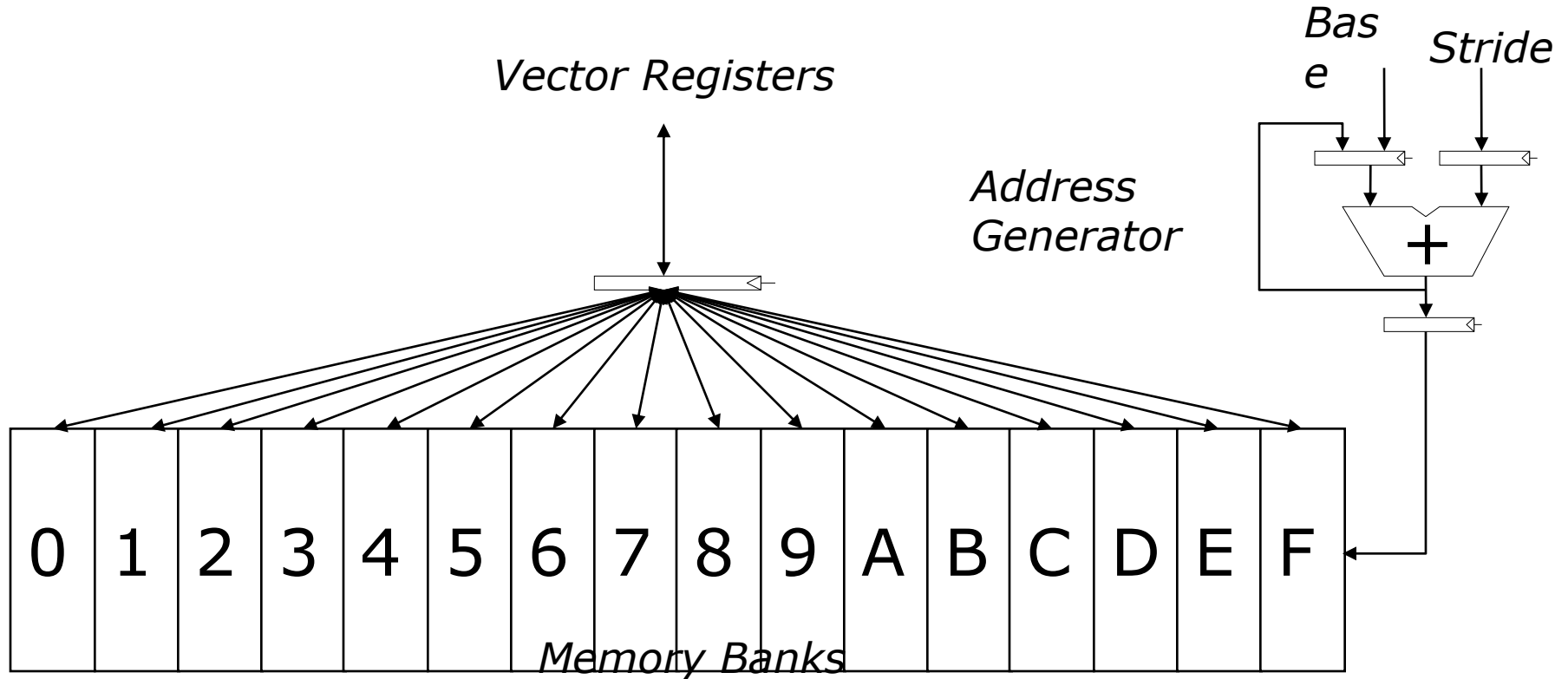
Scalar Code Example

- 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  ;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*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*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)**

```
 1   1    11        49        11        49        4        49        1        49        11        49
 |   |    |         |         |         |         |        |         |        |         |         |
 |   |    |         |         |         |         |        |         |        |         |         |

         V0 = A[0..49]        V1 = B[0..49]           ADD            SHIFT              STORE
```
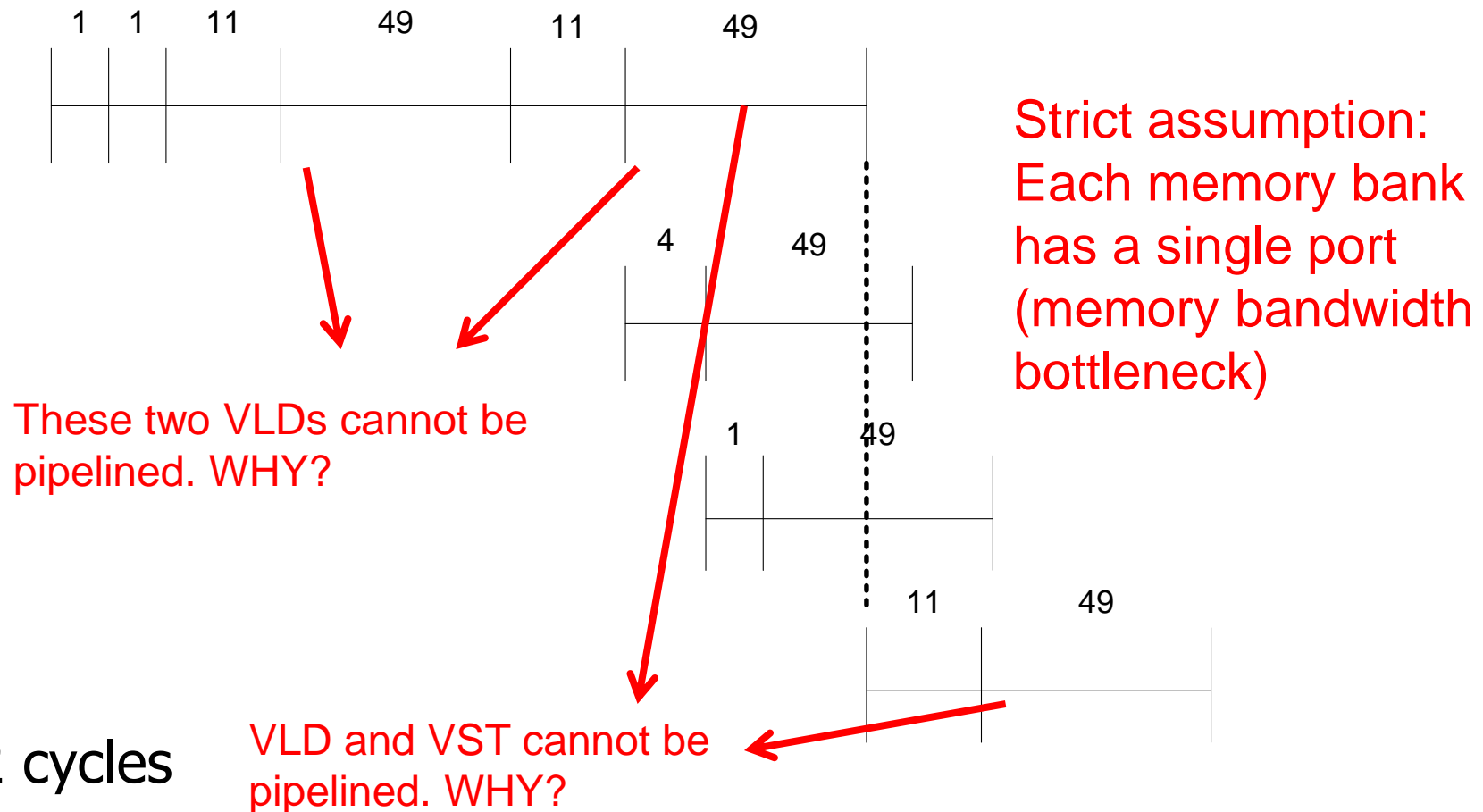
- **285 cycles**

Vector Chaining

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


```
LV    v1
MULV v3,v1,v2
ADDV v5, v3, v4
```
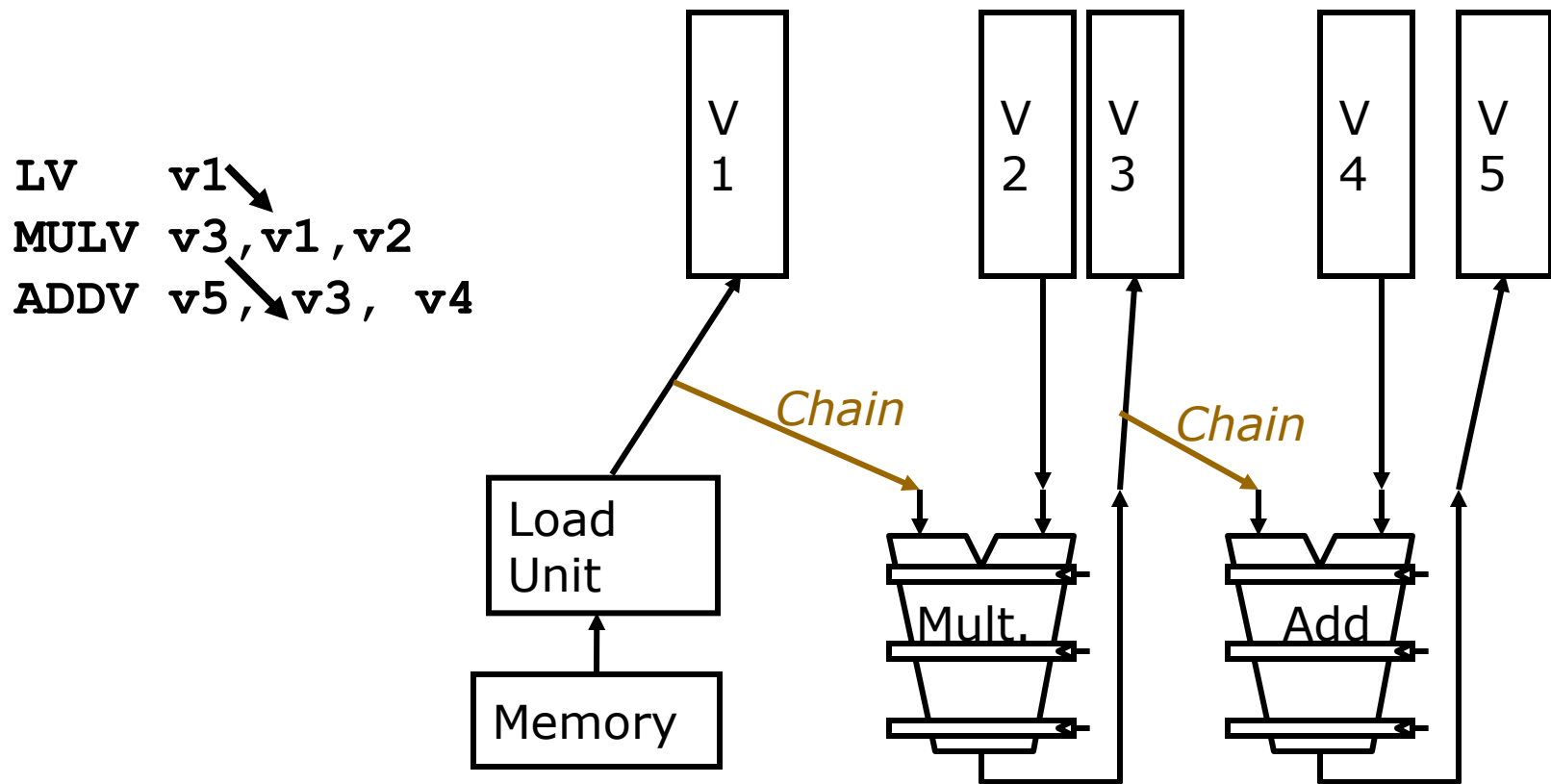
Vector Code Performance - Chaining

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

| 1 | 1 | 11 | 49 | 11 | 49 |

Strict assumption: Each memory bank has a single port (memory bandwidth bottleneck)

These two VLDs cannot be pipelined. WHY?

| 4 | 49 |

| 1 | 49 |

| 11 | 49 |

VLD and VST cannot be pipelined. WHY?

- 182 cycles

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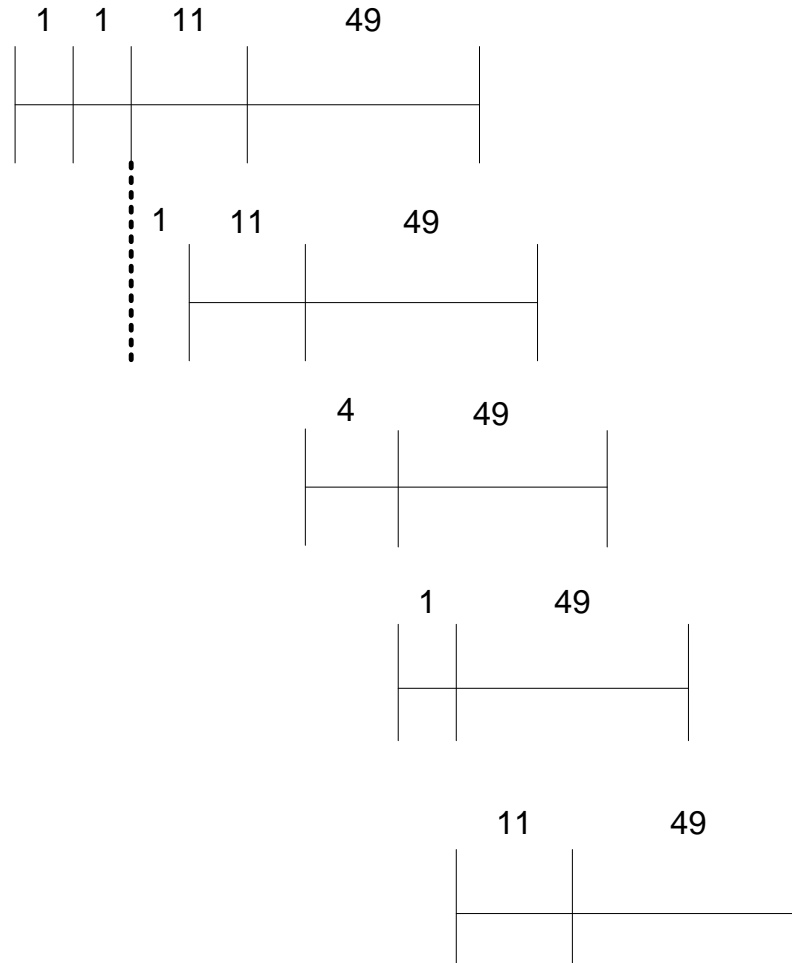
- 79 cycles

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
		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

```
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   C[2]

M[1]=1   C[1]

M[0]=0   C[0]
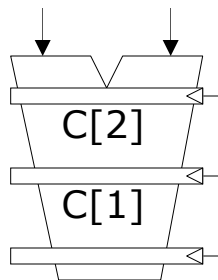
Write Enable   *Write data port*

Density-Time Implementation

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

```
M[7]=1
M[6]=0        A[7]    B[7]
M[5]=1
M[4]=1              C[5]
M[3]=0
M[2]=0              C[4]
M[1]=1
M[0]=0              C[1]
```

Write data port
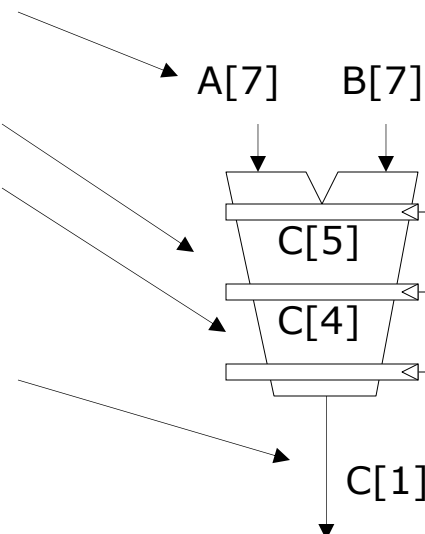
Some Issues

- 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

$A_{4\times6}$ $B_{6\times10}$ $\rightarrow$ $C_{4\times10}$ (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
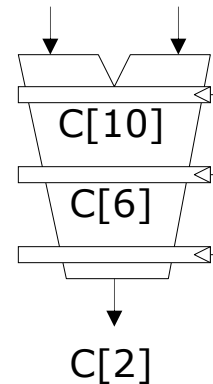
Remember: Array vs. Vector Processors

ARRAY PROCESSOR   VECTOR PROCESSOR

| PE0 | PE1 | PE2 | PE3 | | LD | ADD | MUL | ST |

Instruction Stream

LD   VR ← A[3:0]
ADD VR ← VR, 1
MUL VR ← VR, 2
ST   A[3:0] ← VR

Same op @ same time

Different ops @ time

LD0	LD1	LD2	LD3		LD0			
AD0	AD1	AD2	AD3		LD1	AD0		
MU0	MU1	MU2	MU3		LD2	AD1	MU0	
ST0	ST1	ST2	ST3		LD3	AD2	MU1	ST0
						AD3	MU2	ST1
							MU3	ST2
								ST3

Different ops @ same space

Same op @ space

Time

Space   Space

40
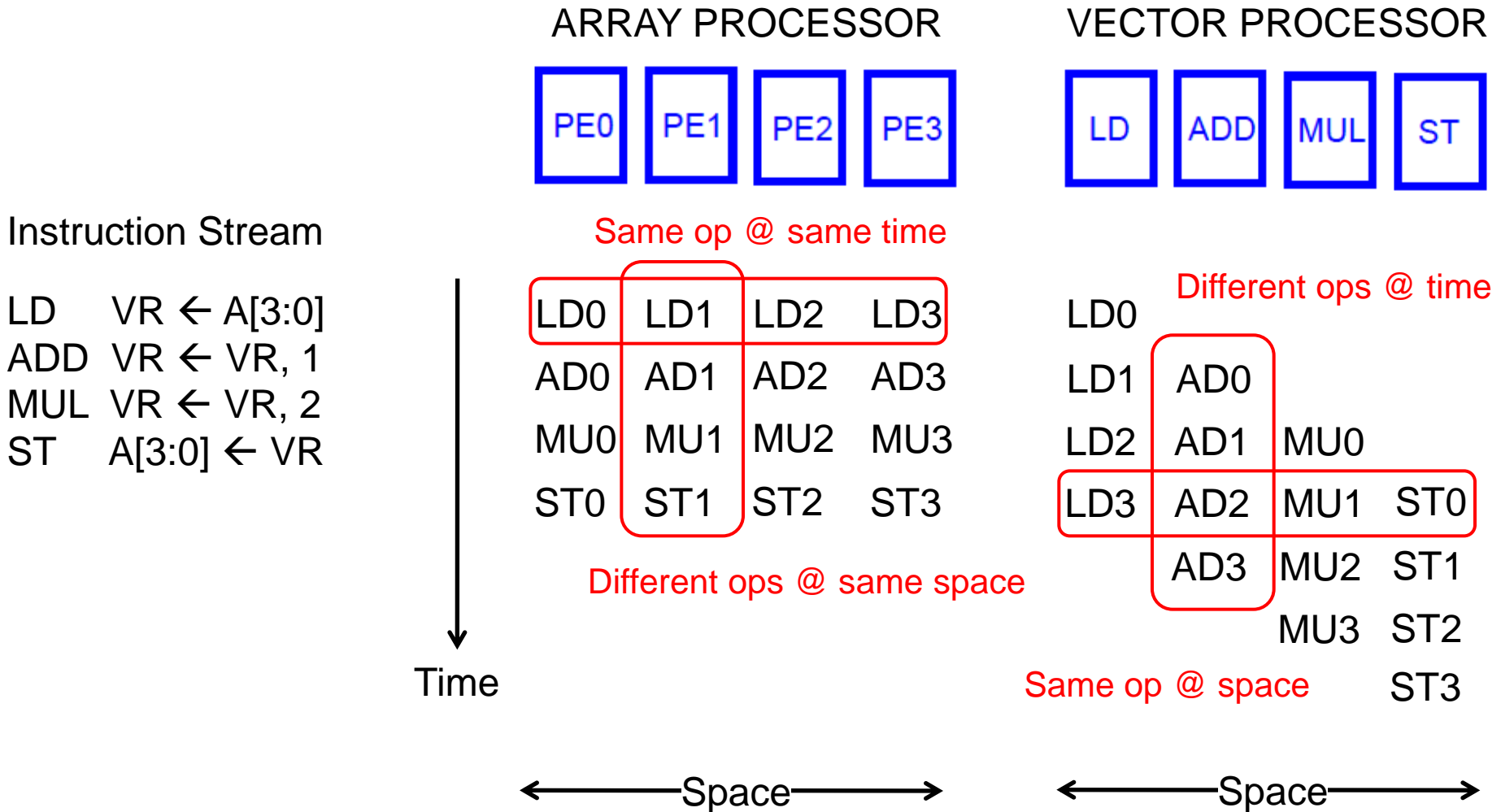
Vector Instruction Execution

ADDV C,A,B

Execution using one pipelined functional unit
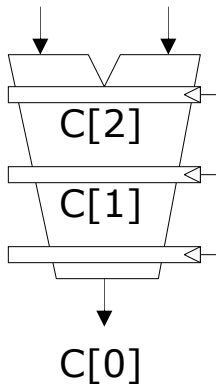
Execution using four pipelined functional units

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

C[2]
C[1]

C[0]

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]

C[8]
C[4]

C[9]
C[5]

C[10]
C[6]

C[11]
C[7]

C[0]   C[1]   C[2]   C[3]

Vector Unit Structure

Functional Unit

Vector Registers

Elements 0, 4, 8, …

Elements 1, 5, 9, …

Elements 2, 6, 10, …

Elements 3, 7, 11, …

Lane

Memory Subsystem
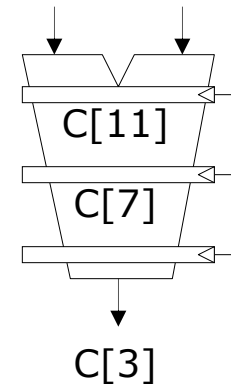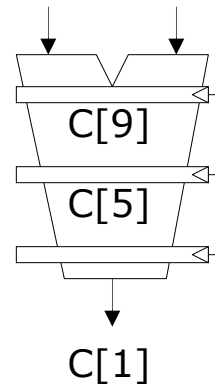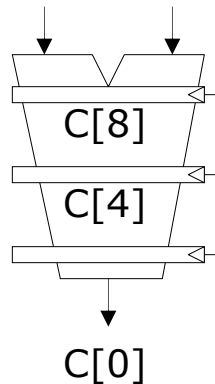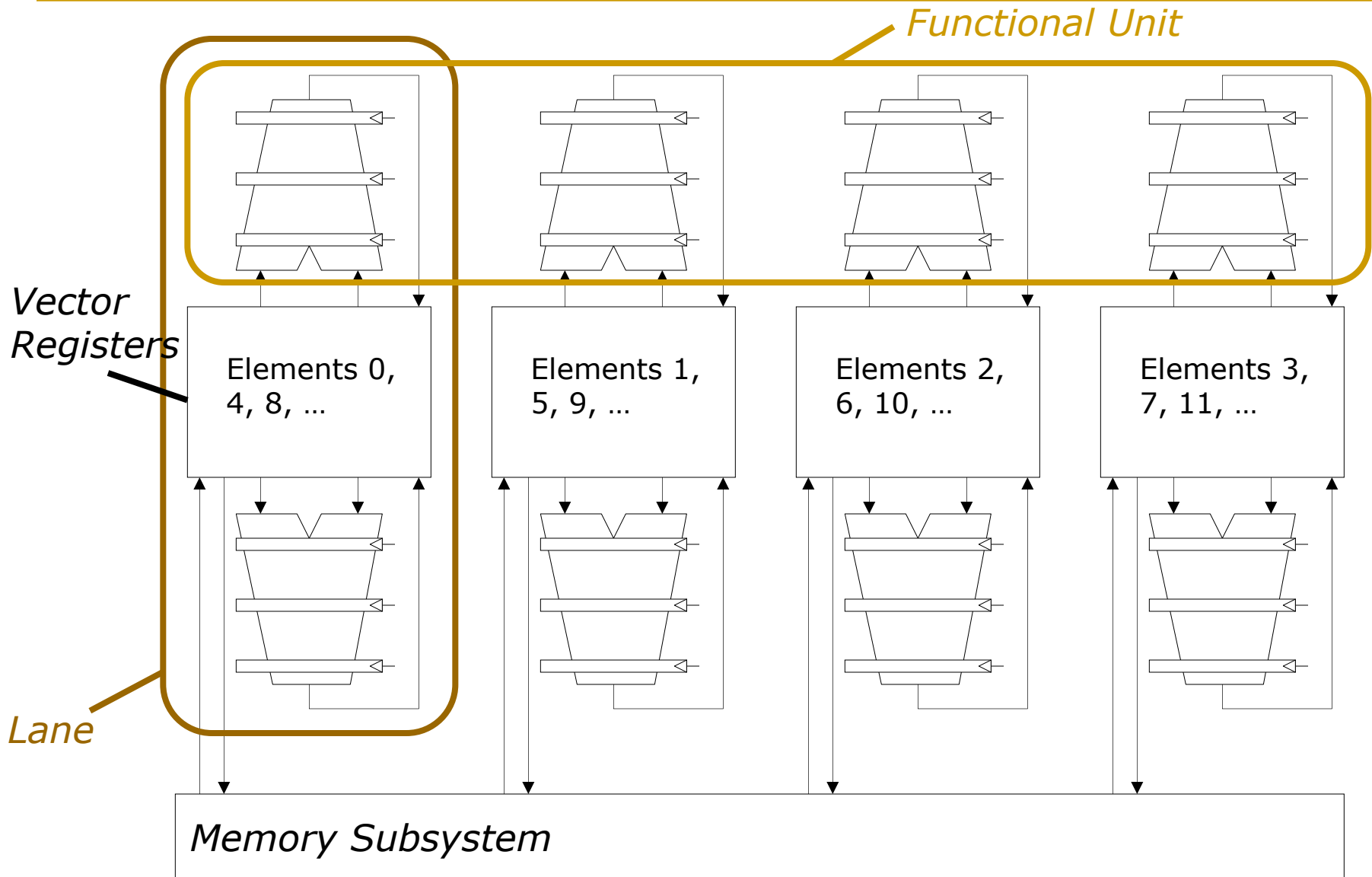
Vector Instruction Level Parallelism

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
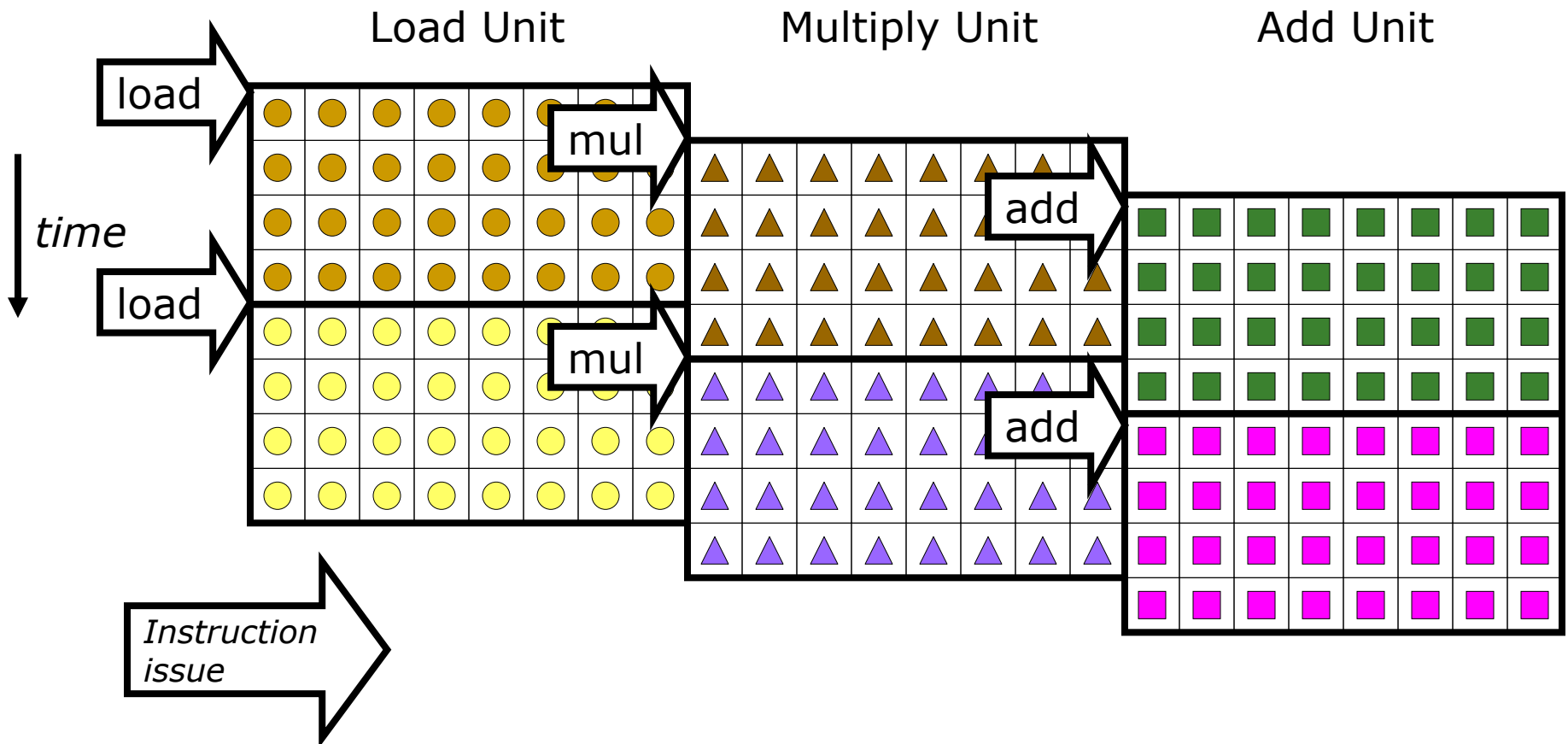
Automatic Code Vectorization

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

Scalar Sequential Code

Vectorized Code

Iter. 1

Iter. 2

Time

load
load
add
store

load
load
add
store

load   load
load   load
add   add
store   store

Iter.
1

Iter.
2

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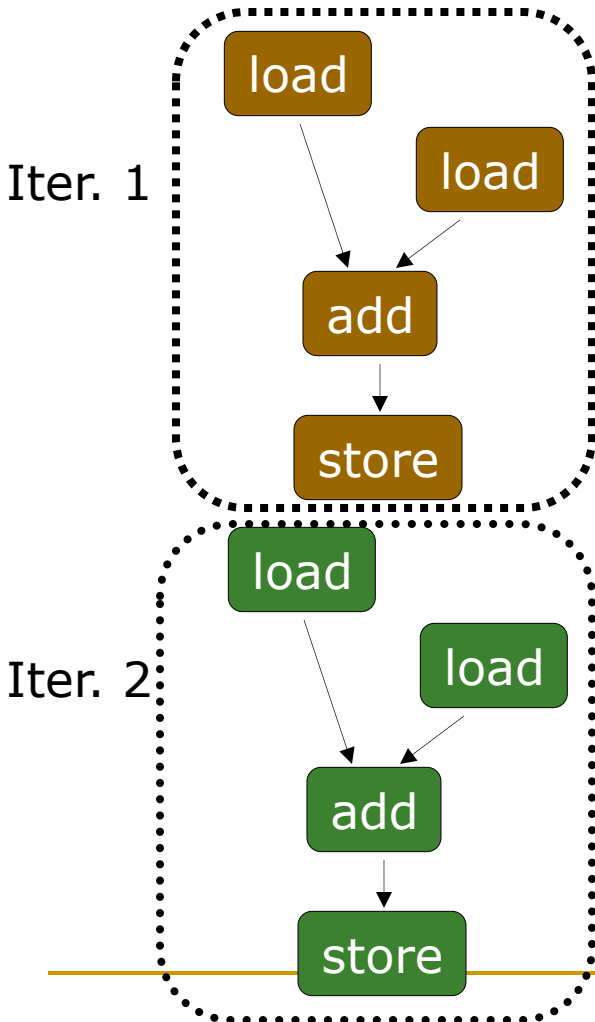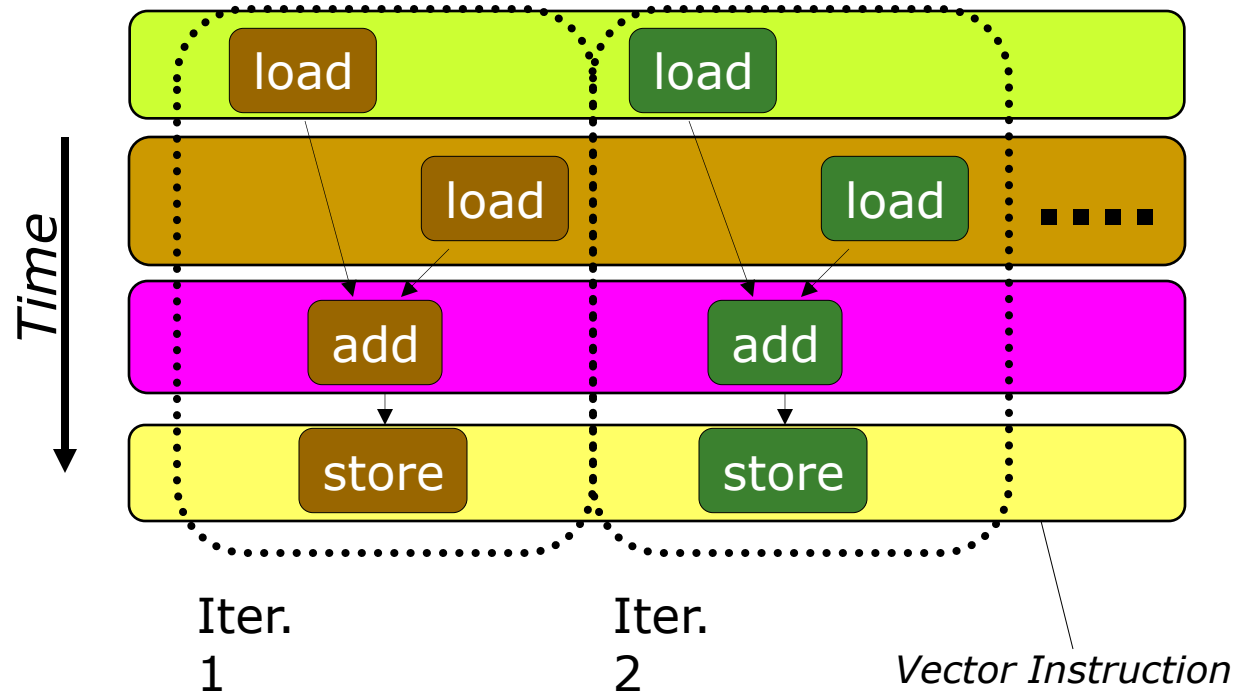
Vectorization is a compile-time reordering of operation sequencing
⇒ requires extensive loop dependence analysis

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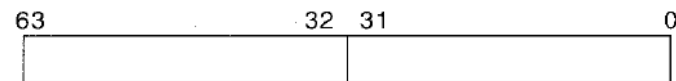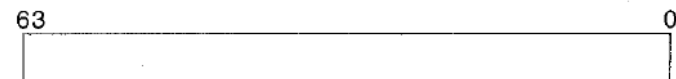
Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

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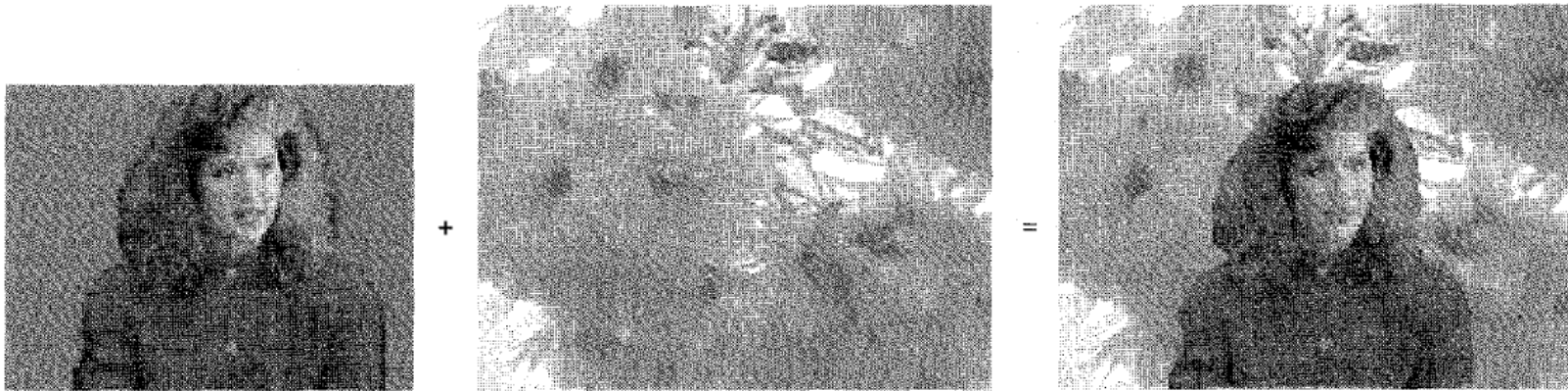
Figure 8. Chroma keying: image overlay using a background color.

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)

PAND MM4, MM1

MM4	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0

MM1	0×0000	0×0000	0×FFFF	0×FFFF	0×0000	0×0000	0×FFFF	0×FFFF

MM4	0×0000	0×0000	Y_5	Y_4	0×0000	0×0000	Y_1	Y_0

PANDN MM1, MM3

MM1	0×0000	0×0000	0×FFFF	0×FFFF	0×0000	0×0000	0×FFFF	0×FFFF

MM3	X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0

MM1	X_7	X_6	0×0000	0×0000	X_3	X_2	0×0000	0×0000

POR MM4, MM1

MM4	X_7	X_6	Y_5	Y_4	X_3	X_2	Y_1	Y_0
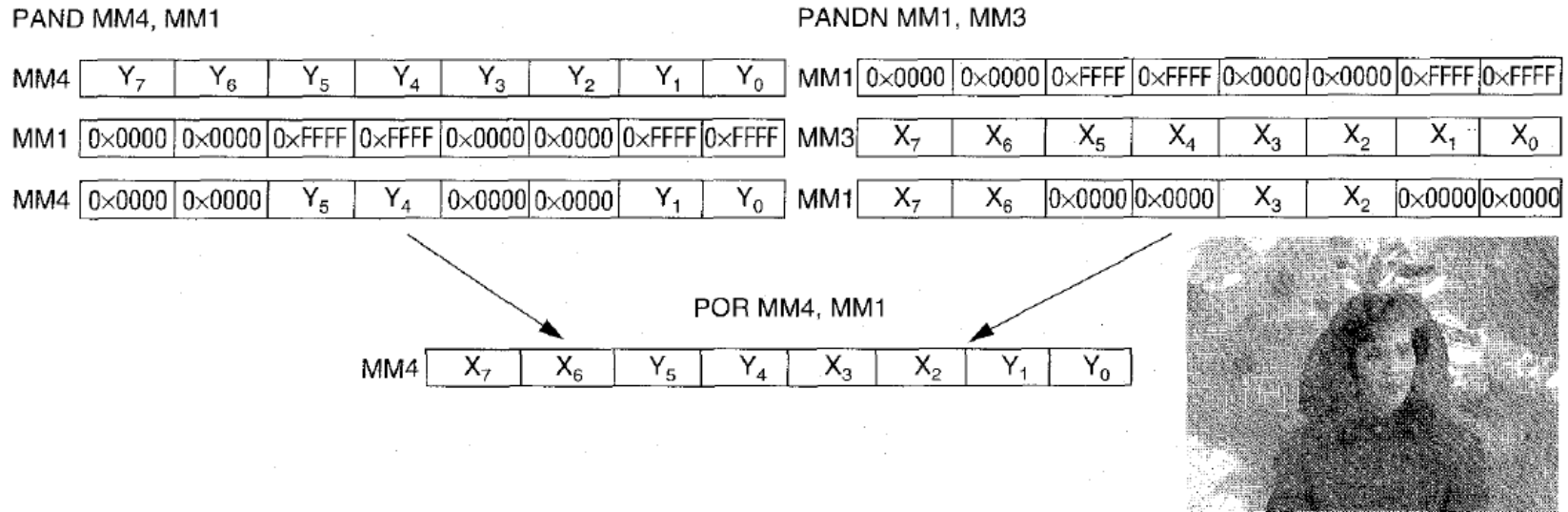
Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

```
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
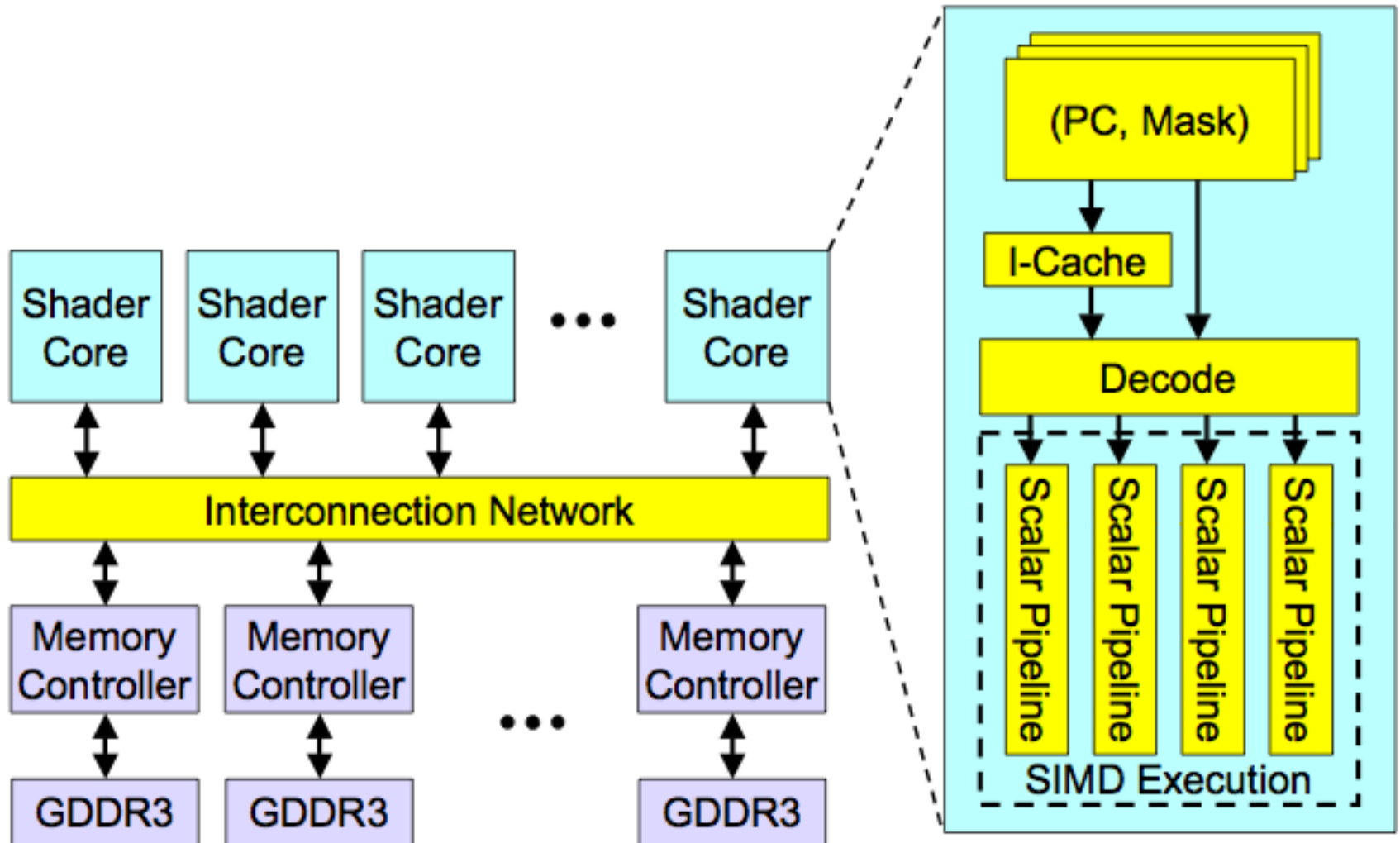
Concept of "Thread Warps" and SIMT

- **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 …
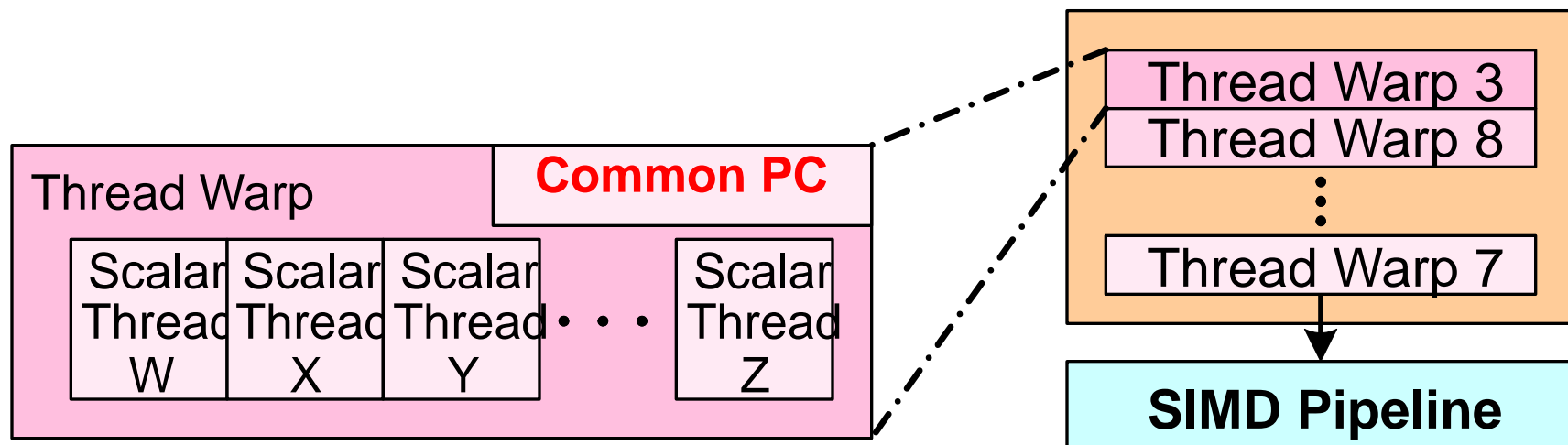
Loop Iterations as Threads

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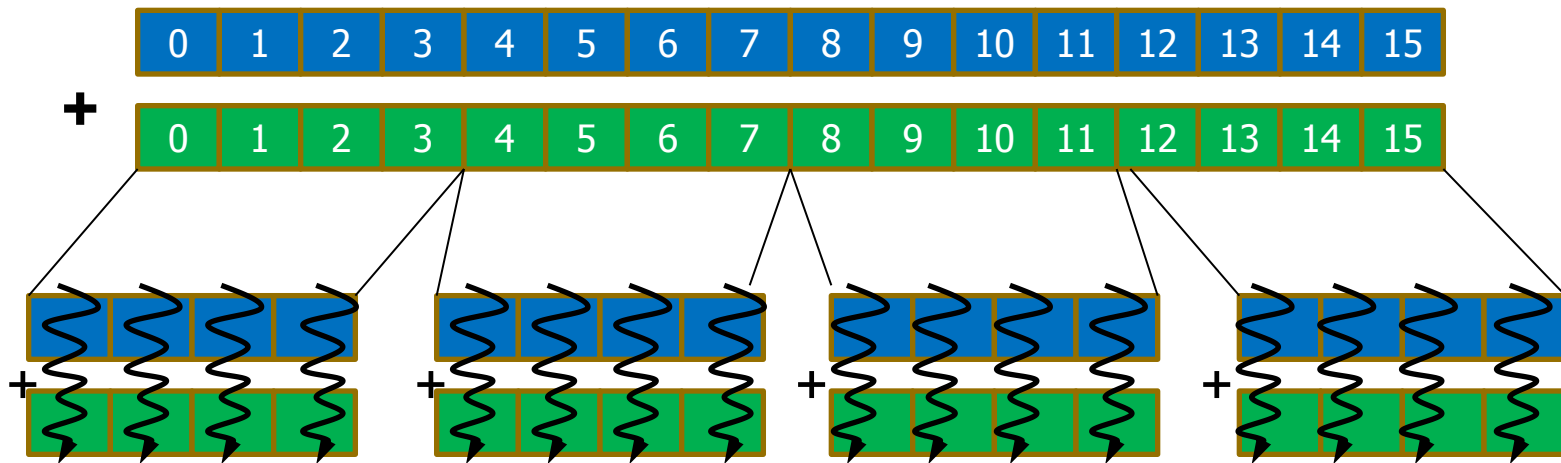
Scalar Sequential Code

Vectorized Code

Time

Iter. 1

Iter. 2

Iter. 1

Iter. 2
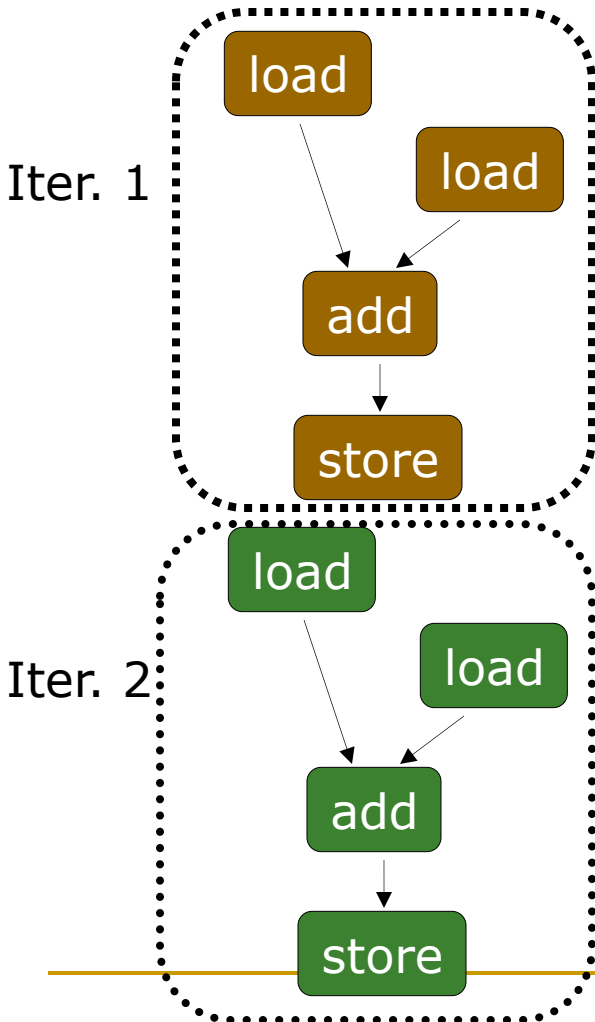
Vector Instruction
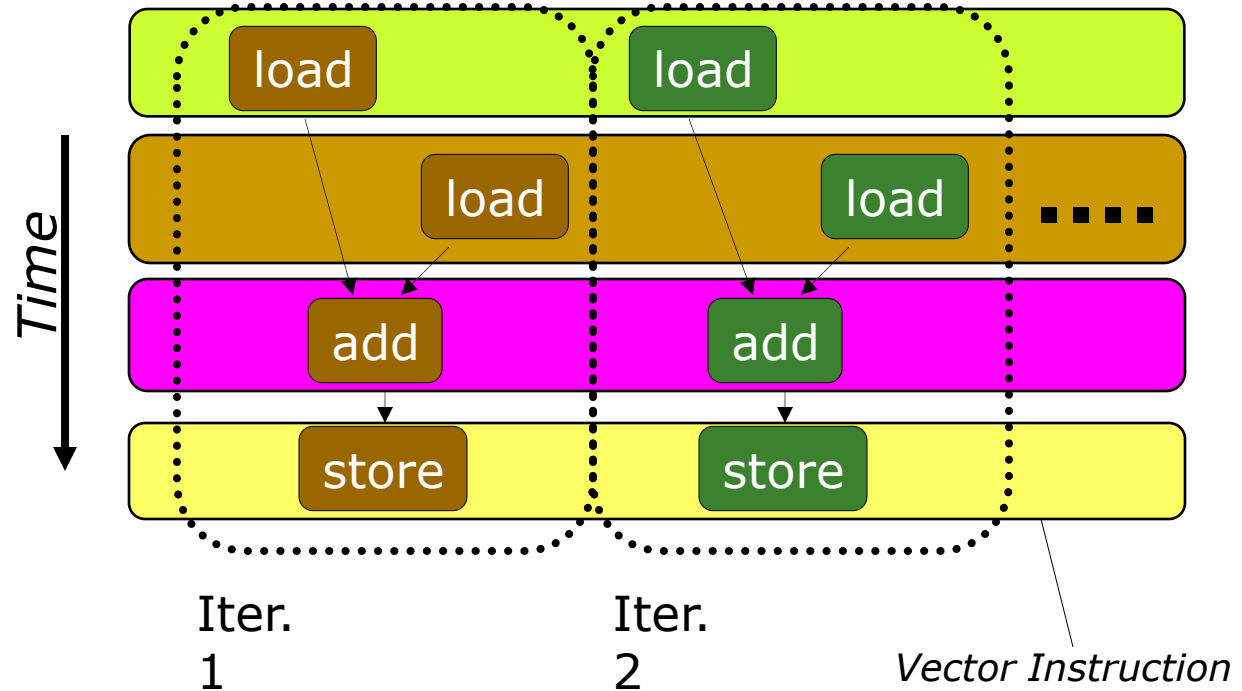
SIMT Memory Access

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

Let's assume N=16, blockDim=4 → 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

Warps available for scheduling

- Thread Warp 3
- Thread Warp 8
- Thread Warp 7

SIMD Pipeline

- I-Fetch
- Decode
- RF · · · RF
- ALU · · · ALU
- D-Cache
- All Hit? Data
- Writeback

Warps accessing memory hierarchy

Miss?

- Thread Warp 1
- Thread Warp 2
- Thread Warp 6
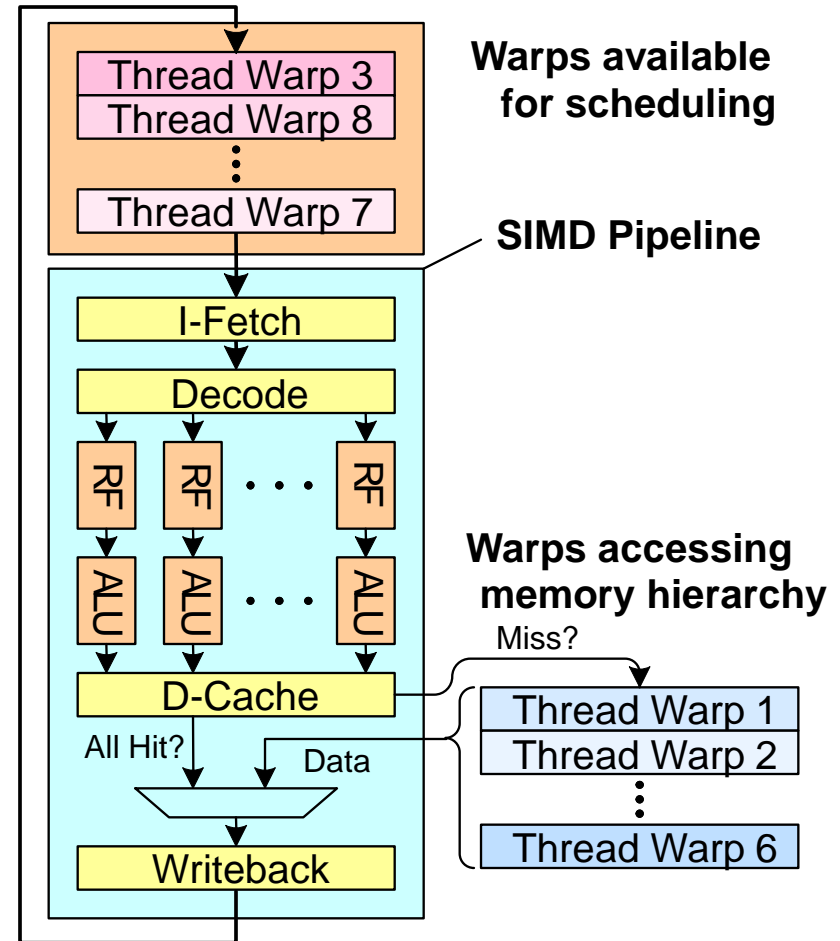
Warp-based SIMD vs. Traditional SIMD

- 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

We did not cover the following slides in lecture. These are for your preparation for the next lecture.

Branch Divergence Problem in Warp-based SIMD

- **SPMD Execution on SIMD Hardware**
  - NVIDIA calls this "Single Instruction, Multiple Thread" ("SIMT") execution
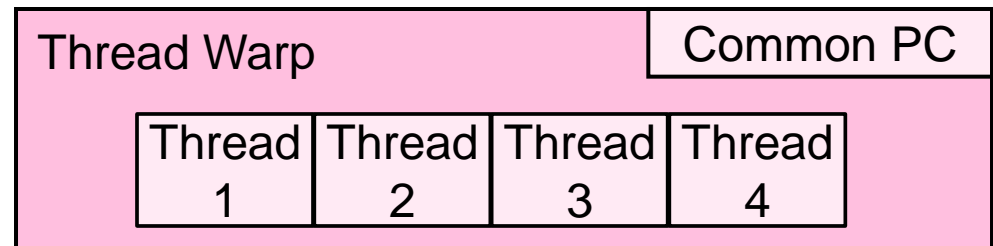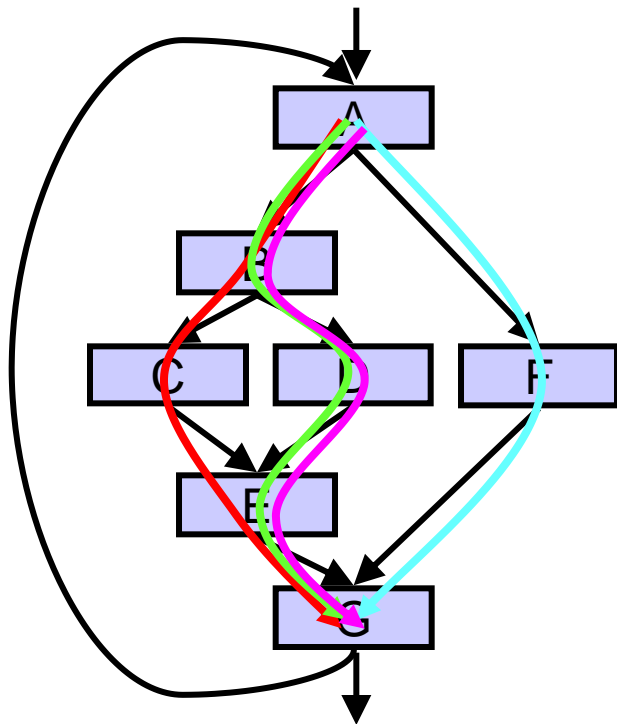
Control Flow Problem in GPUs/SIMD

- **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.**
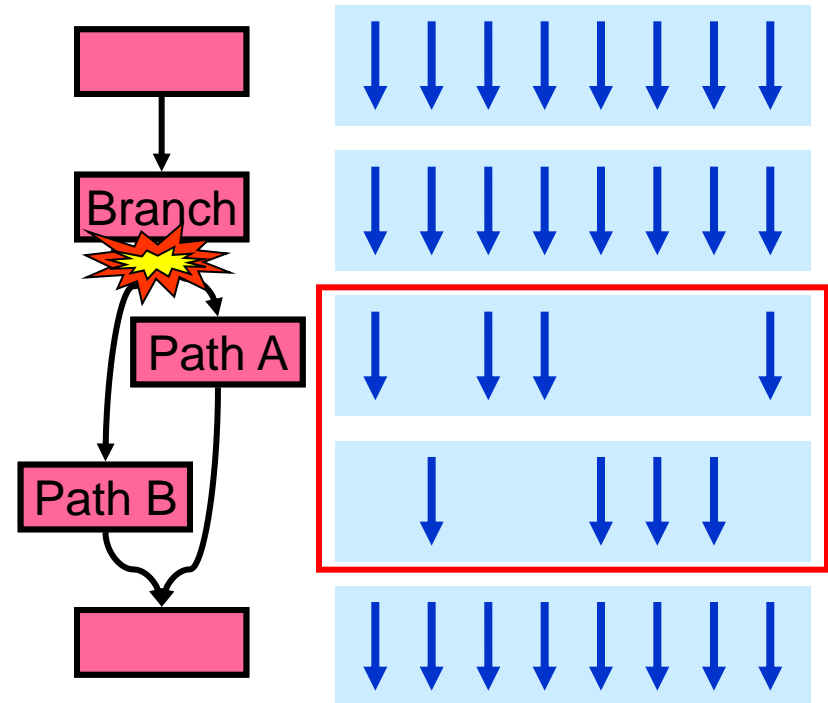
Branch Divergence Handling (I)
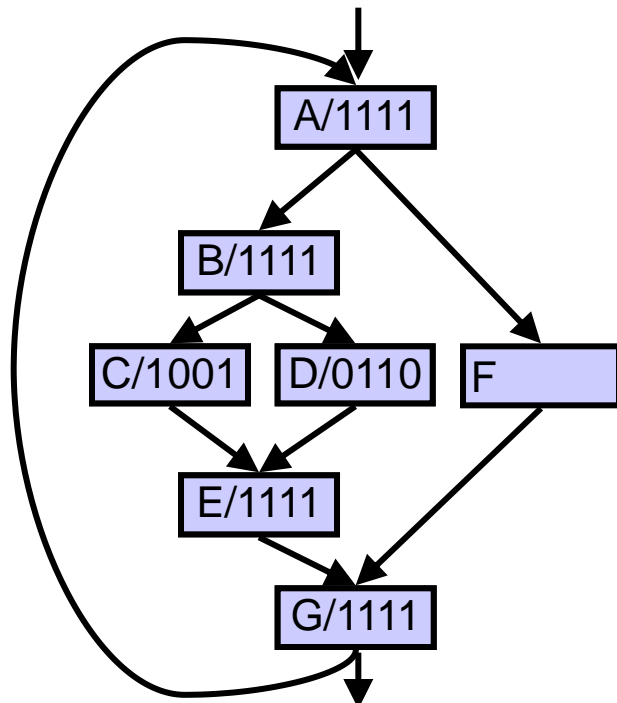
Stack

	Reconv. PC	Next PC	Active Mask
TOS →	-	E	1111
TOS →	E	D	0110
TOS →	E	E	1001
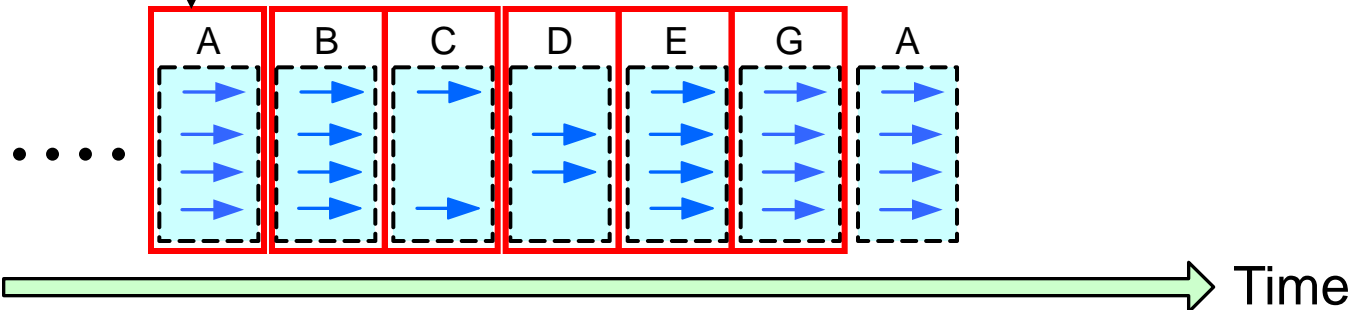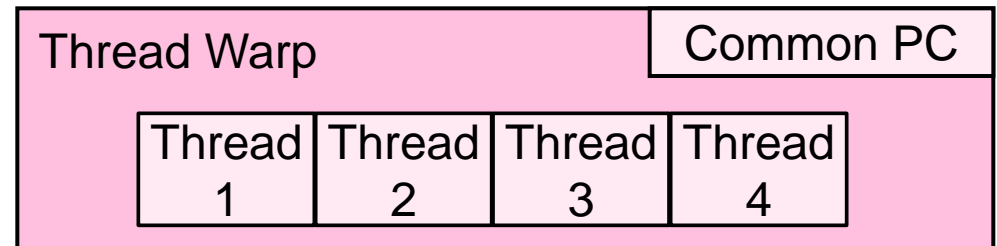
Branch Divergence Handling (II)

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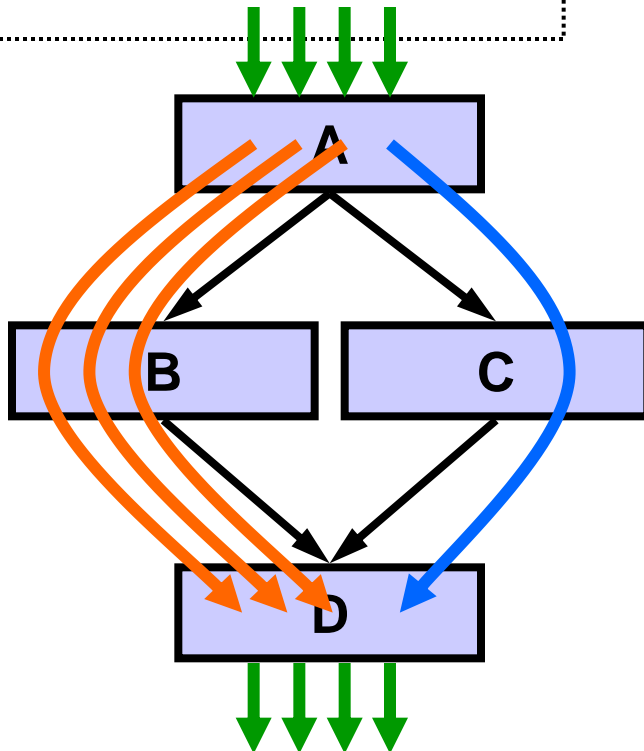

One per warp

Control Flow Stack

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

TOS →

Execution Sequence

A	C	B	D
1	0	1	1
1	0	1	1
1	0	1	1
1	1	0	1

Time →
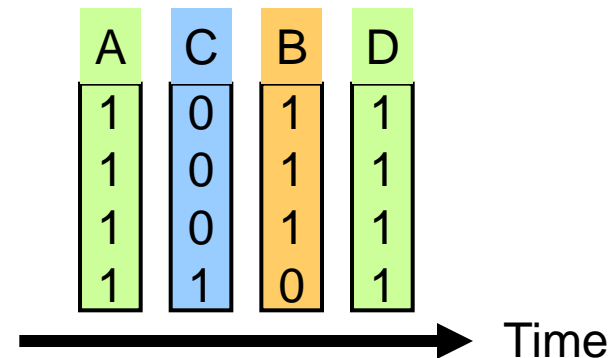
Dynamic Warp Formation

- 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
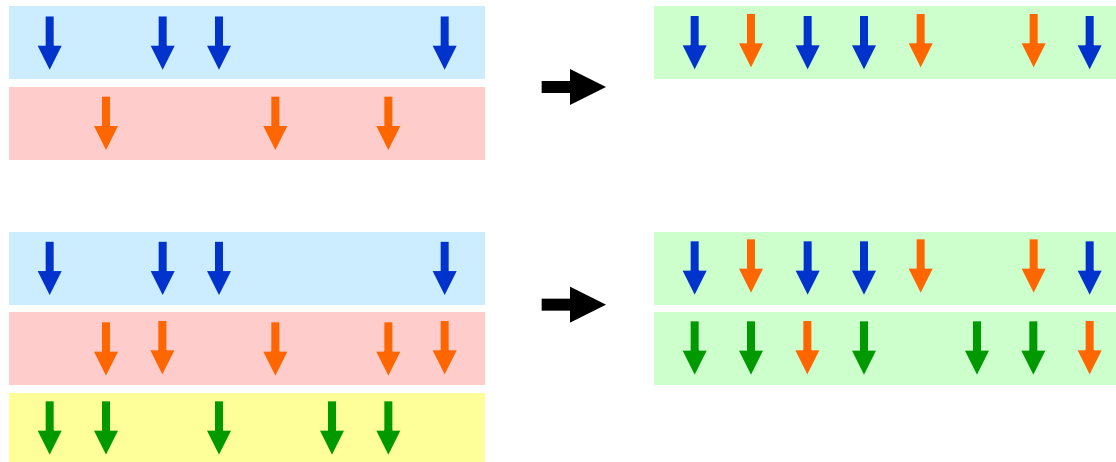
Dynamic Warp Formation/Merging

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)
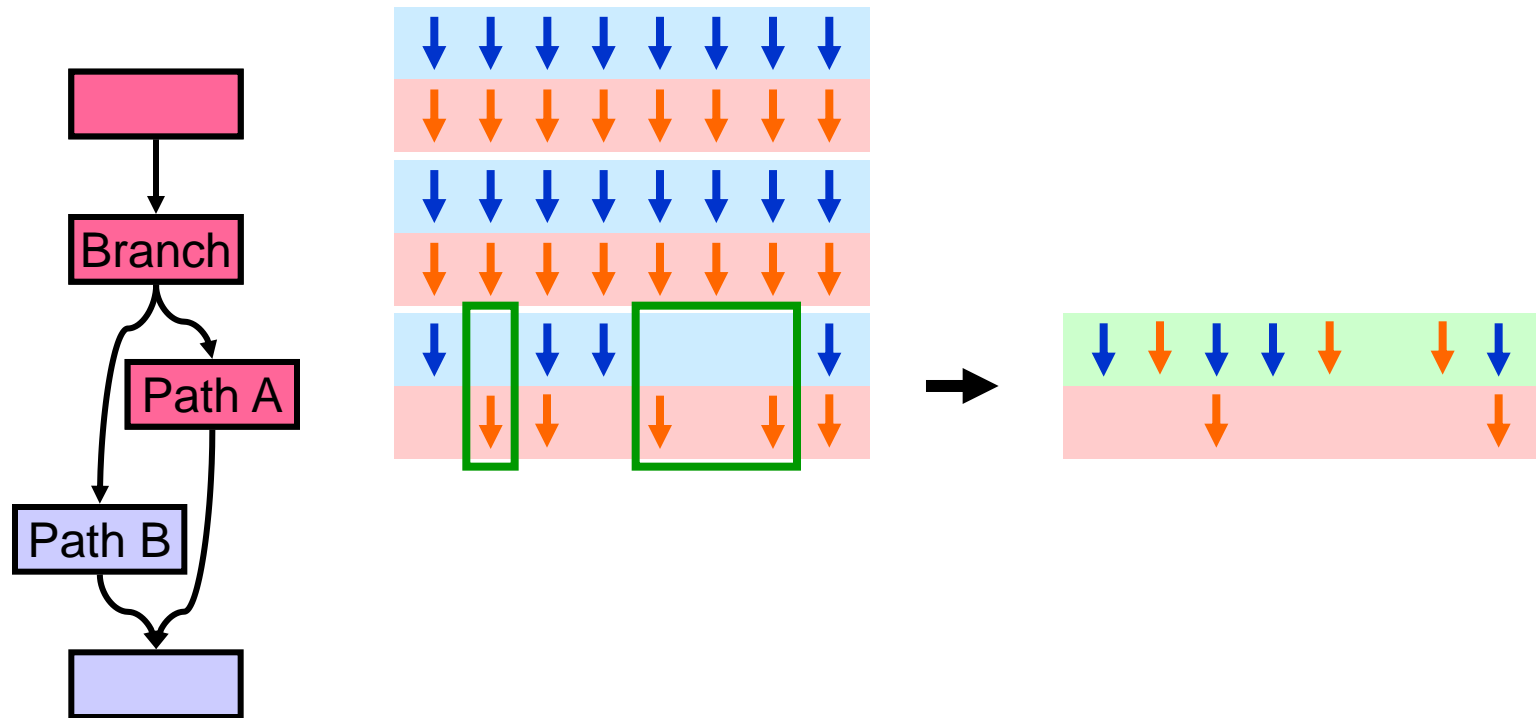
- Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO 2007.

Dynamic Warp Formation Example
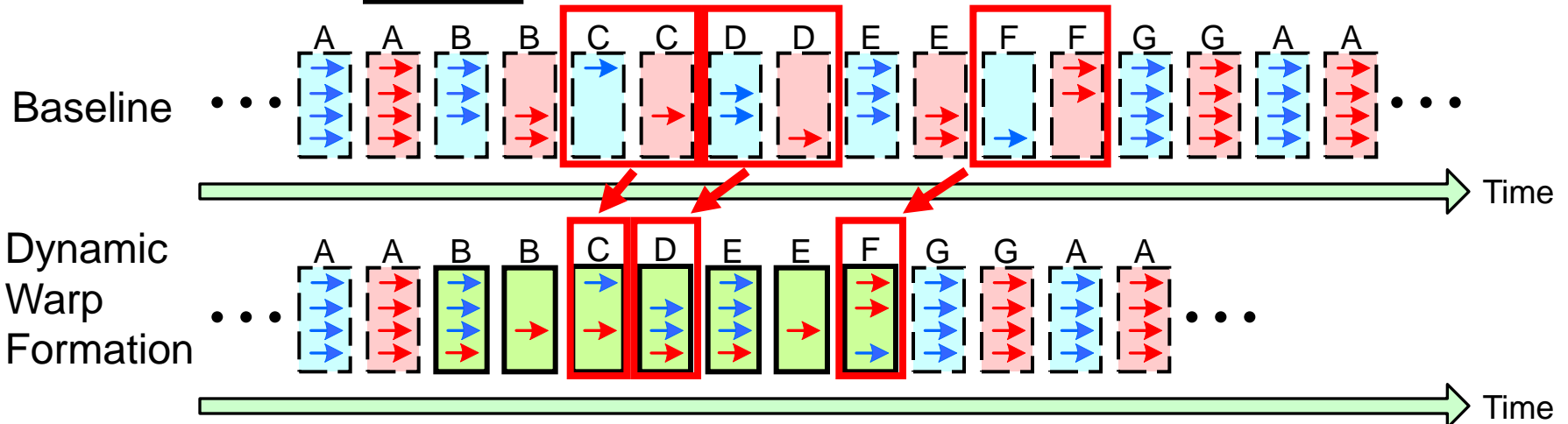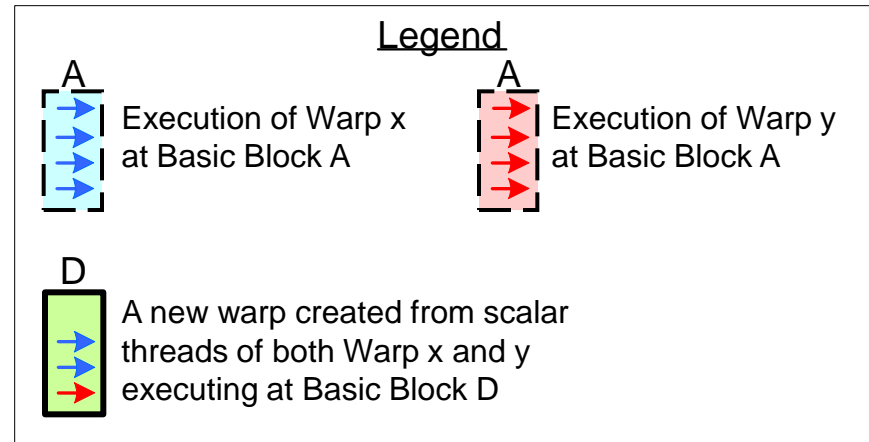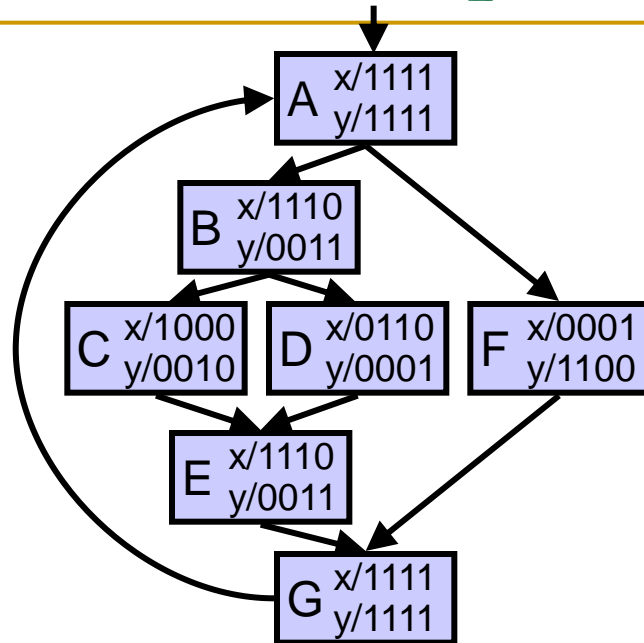
What About Memory Divergence?

- 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

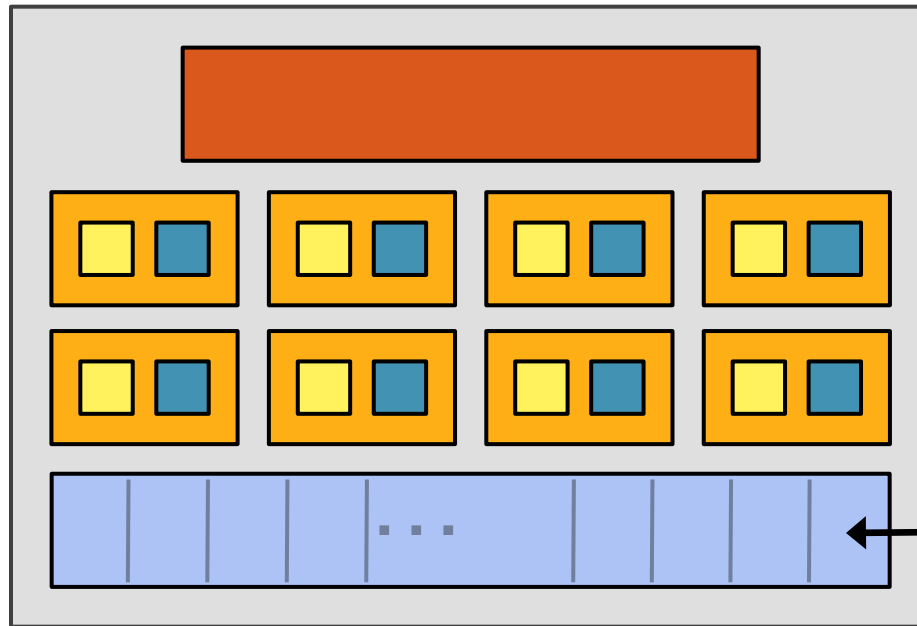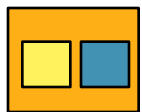 - Integrate solutions to branch and memory divergence

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
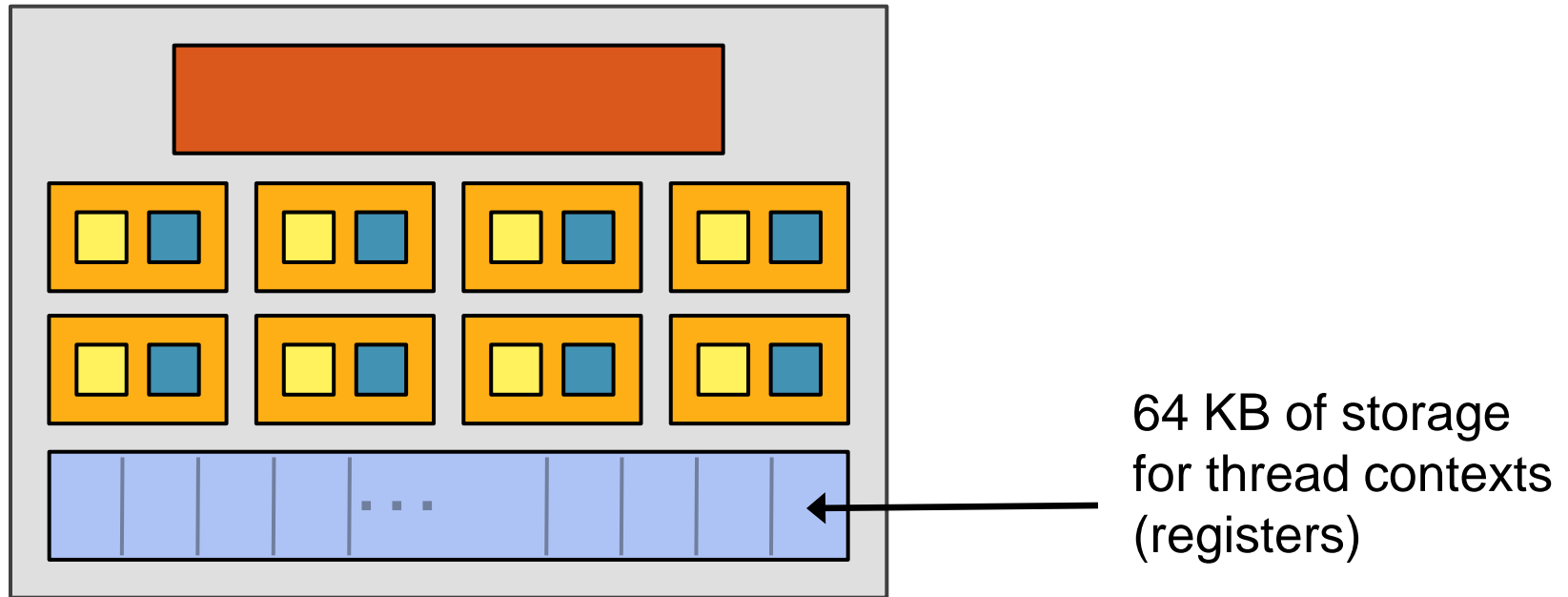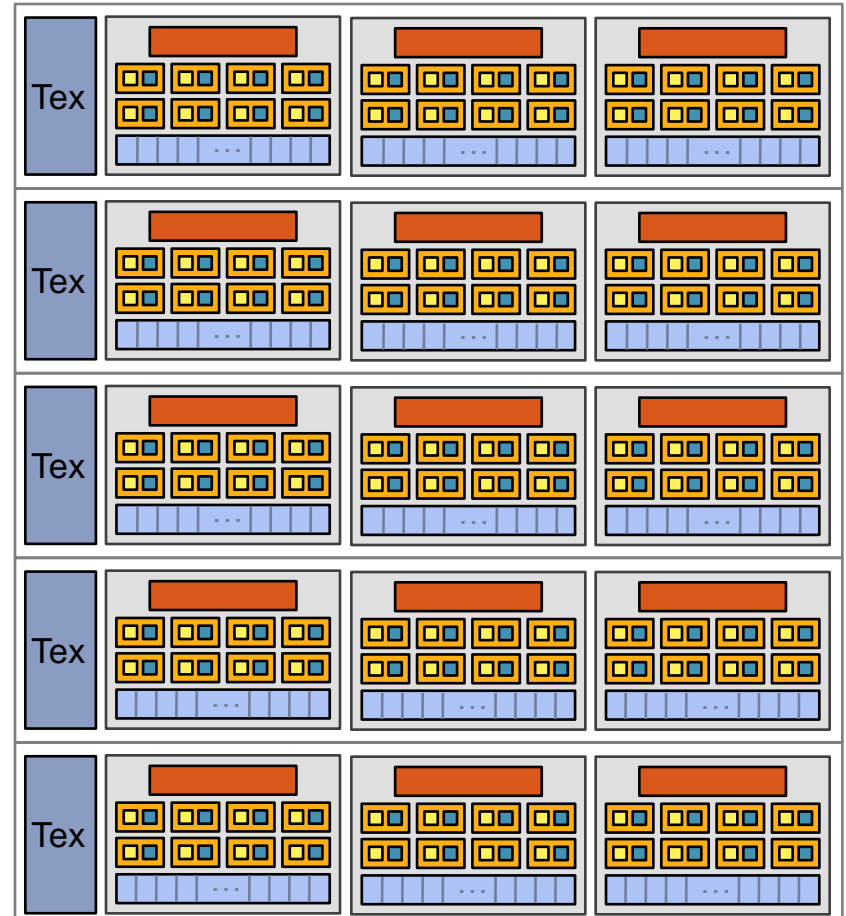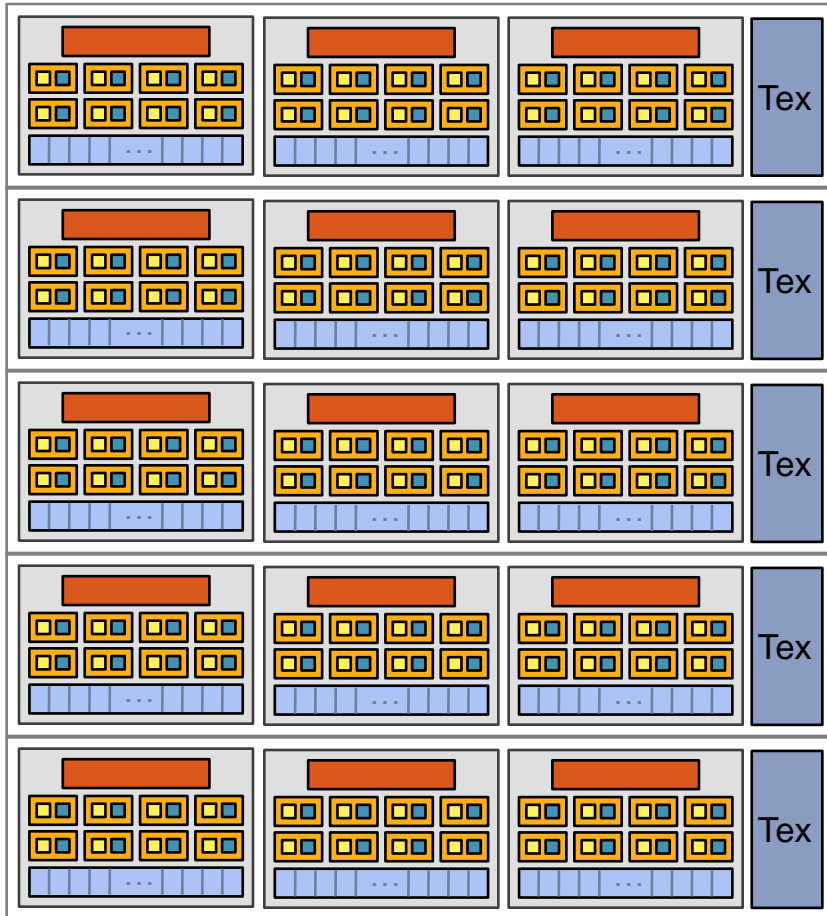
= execution context storage

NVIDIA GeForce GTX 285 "core"

64 KB of storage for thread contexts (registers)

- 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

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

Slide credit: Kayvon Fatahalian