18-447
Computer Architecture
Lecture 14: SIMD Processing
(Vector and Array Processors)

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2015, 2/18/2015

Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures

- Multi-cycle and Microprogrammed Microarchitectures

- Pipelining

- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, …

- Out-of-Order Execution

- Issues in OoO Execution: Load-Store Handling, …

- Alternative Approaches to Instruction Level Parallelism

Approaches to (Instruction-Level) Concurrency

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- SIMD Processing (Vector and array processors, GPUs)
- VLIW
- Decoupled Access Execute
- Systolic Arrays

Reminder: Announcements

- Lab 3 due this Friday (Feb 20)
  - Pipelined MIPS
  - Competition for high performance
    - You can optimize both cycle time and CPI
    - Document and clearly describe what you do during check-off

- Homework 3 due Feb 25
  - A lot of questions that enable you to learn the concepts via hands-on exercise
  - Remember this is all for your benefit (to learn and prepare for exams)
    - HWs have very little contribution to overall grade
    - Solutions to almost all questions are online anyway
    - But I would still like you to do them (for your benefit)!

Homework 3.1: Feedback Form

- Due Monday Feb 23

- I would like your feedback on the course
- Easy to fill in
- Can submit anonymously, if you wish
- Worth 0.25% of your grade
- Need to get checked off after submitting to get your grade points
  - Can email
  - If anonymous, show that you are turning in and have a TA check you off

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.

Recap of Last Lecture

- OoO Execution as Restricted Data Flow
- Memory Disambiguation or Unknown Address Problem
- Memory Dependence Handling
  - Conservative, Aggressive, Intelligent Approaches
- Load Store Queues
- Design Choices in an OoO Processor
- Combining OoO+Superscalar+Branch Prediction
- Example OoO Processor Designs

- Data Flow (at the ISA level) Approach to Concurrency
  - Characteristics
  - Supporting dynamic instances of a node: Tagging, Context IDs, Frames
  - Example Operation
  - Advantages and Disadvantages
  - Combining Data Flow and Control Flow: Getting the Best of Both Worlds

Reminder: Intel Pentium 4 Simplified

Mutlu+, "Runahead Execution," HPCA 2003.
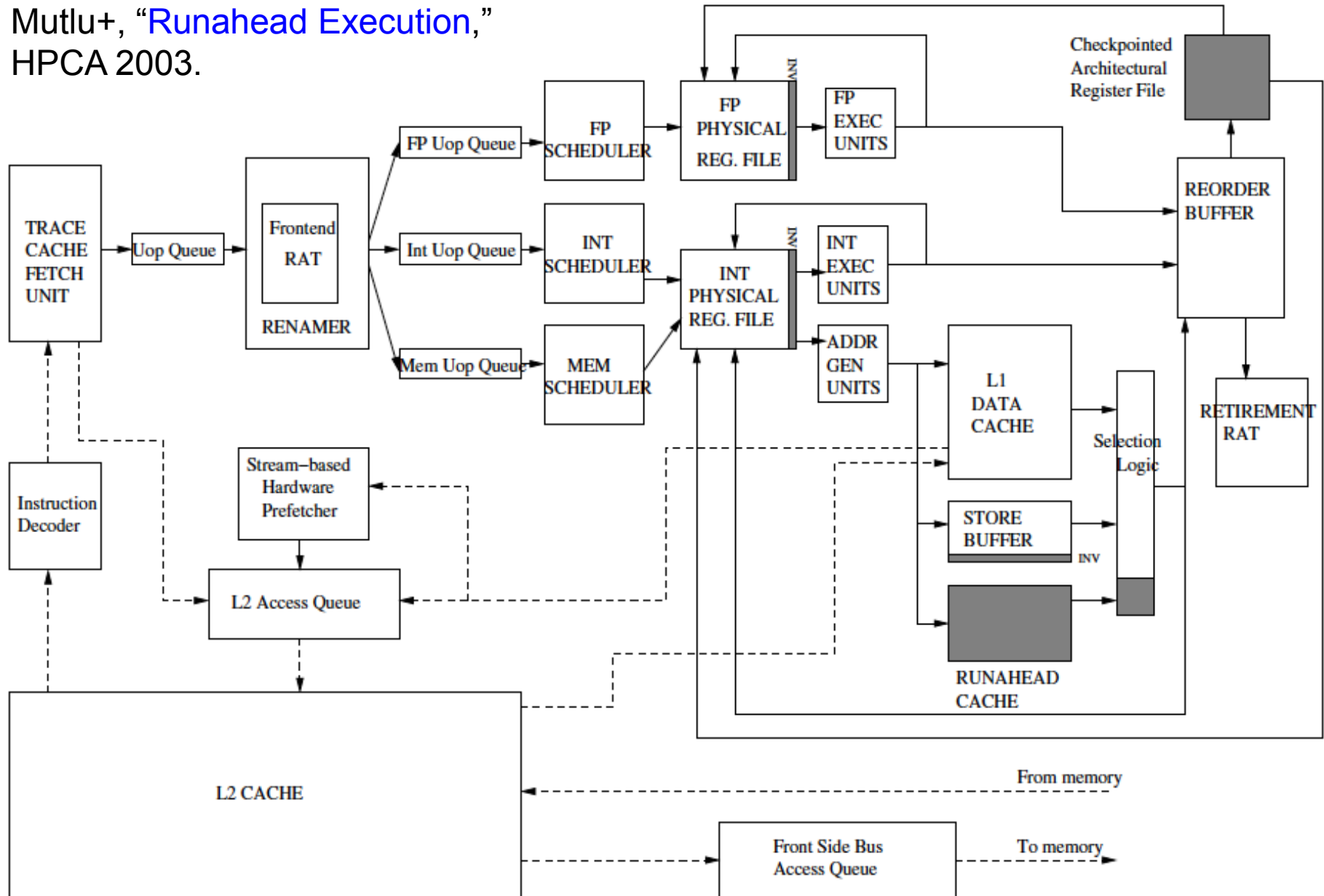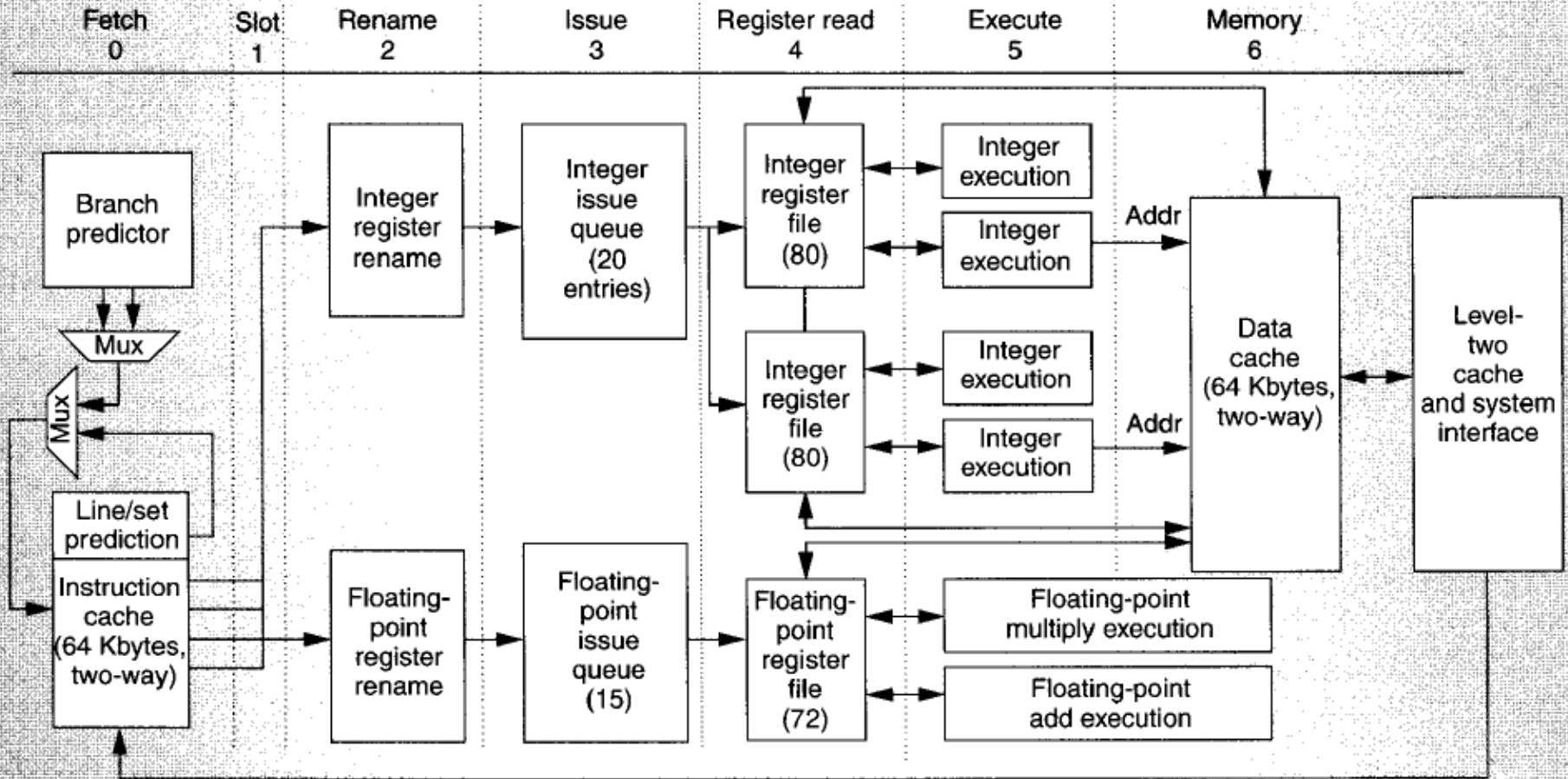
Reminder: Alpha 21264

Figure 2. Stages of the Alpha 21264 instruction pipeline.

Kessler, "The Alpha 21264 Microprocessor," IEEE Micro, March-April 1999.

Review: Data Flow:
Exploiting Irregular Parallelism

Review: Pure Data Flow Pros and Cons

- Advantages
  - Very good at exploiting irregular parallelism
  - Only real dependencies constrain processing

- Disadvantages
  - Debugging difficult (no precise state)
    - Interrupt/exception handling is difficult (what is precise state semantics?)
  - Implementing dynamic data structures difficult in pure data flow models
  - Too much parallelism? (Parallelism control needed)
  - High bookkeeping overhead (tag matching, data storage)
  - Instruction cycle is inefficient (delay between dependent instructions), memory locality is not exploited

Review: Combining Data Flow and Control Flow

- Can we get the best of both worlds?

- Two possibilities

  - Model 1: Keep control flow at the ISA level, do dataflow underneath, preserving sequential semantics

  - Model 2: Keep dataflow model, but incorporate some control flow at the ISA level to improve efficiency, exploit locality, and ease resource management
    - Incorporate threads into dataflow: statically ordered instructions; when the first instruction is fired, the remaining instructions execute without interruption in control flow order (e.g., one can pipeline them)

Review: Data Flow Summary

- Data Flow at the ISA level has not been (as) successful

- Data Flow implementations under the hood (while preserving sequential ISA semantics) have been very successful
  - Out of order execution

Approaches to (Instruction-Level) Concurrency

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- SIMD Processing (Vector and array processors, GPUs)
- VLIW
- Decoupled Access Execute
- Systolic Arrays

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
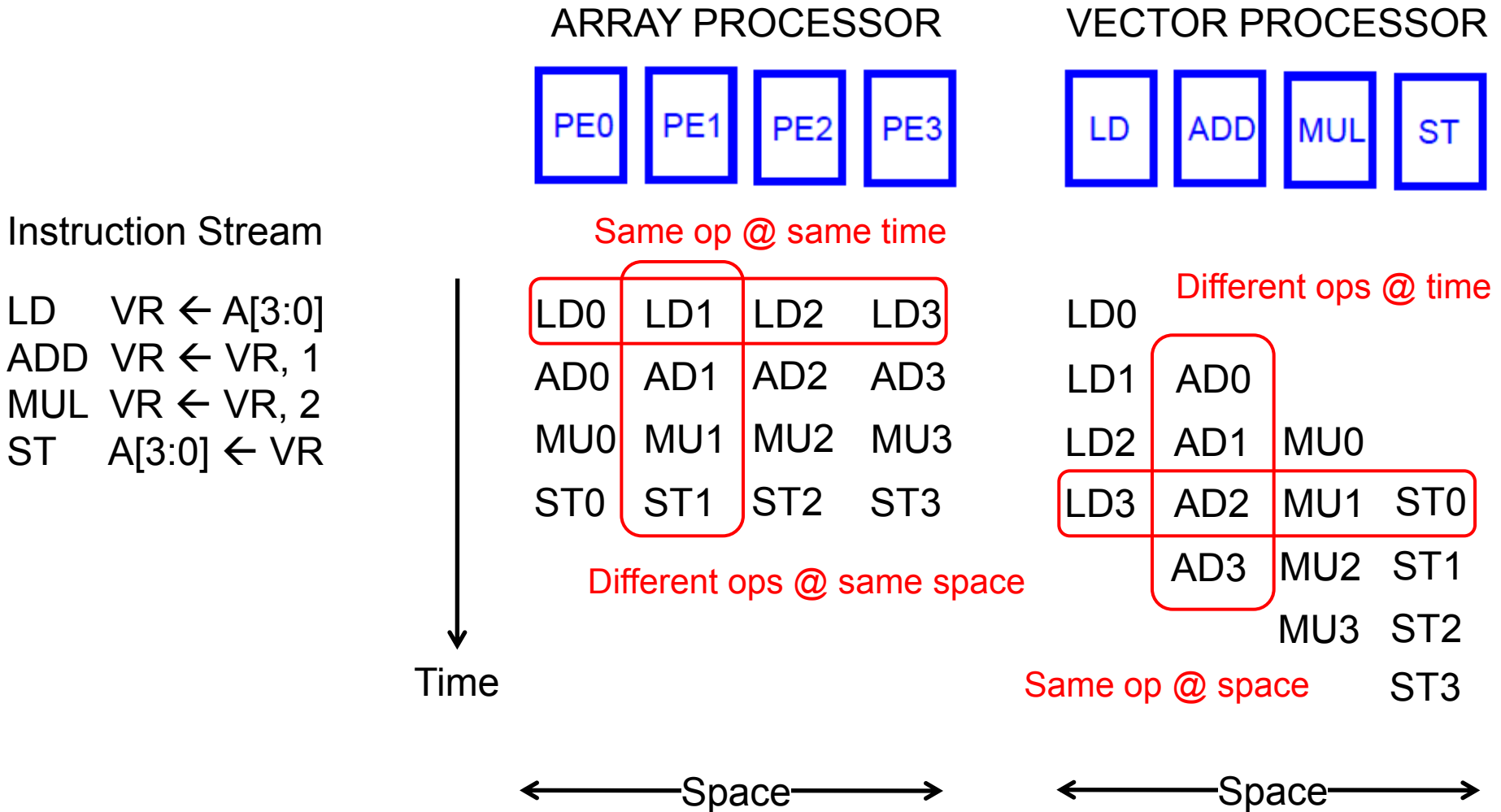 - Multithreaded processor

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 using different spaces

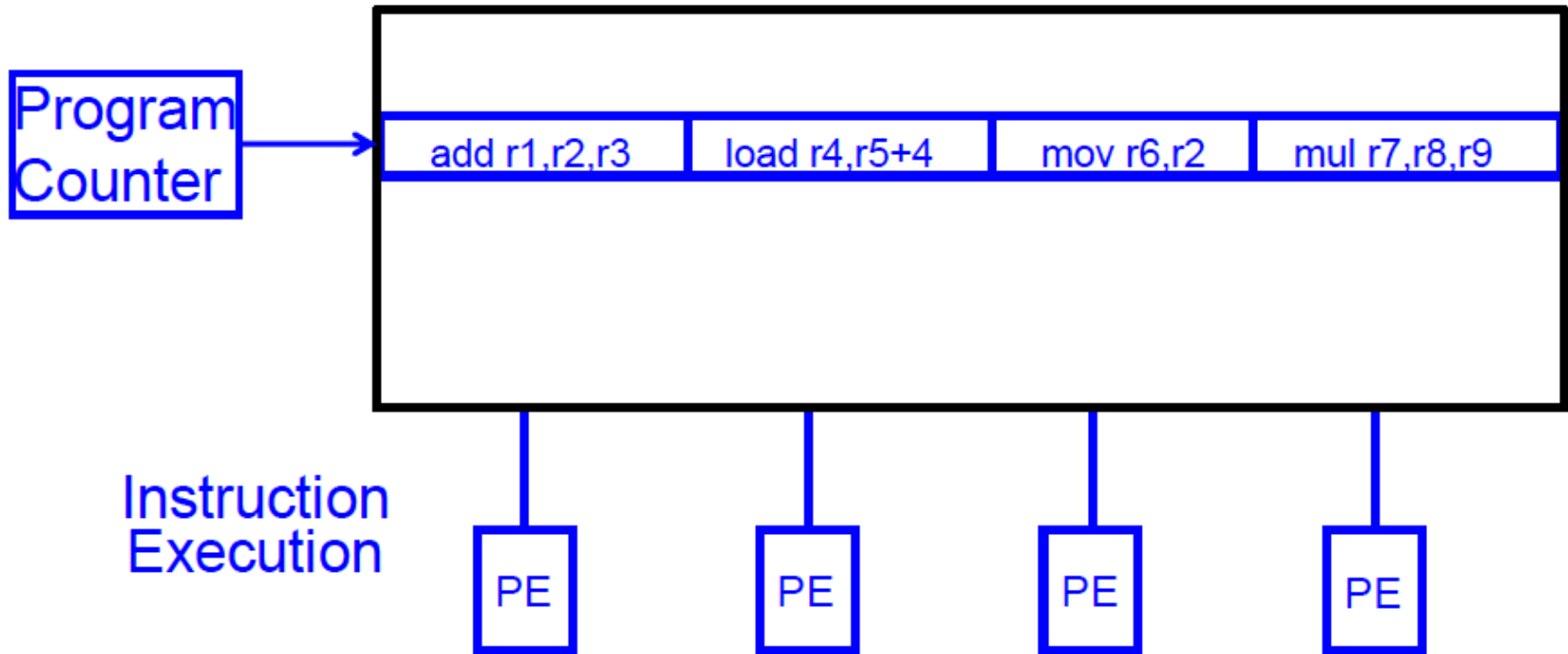 - Vector processor: Instruction operates on multiple data elements in consecutive time steps using the same space

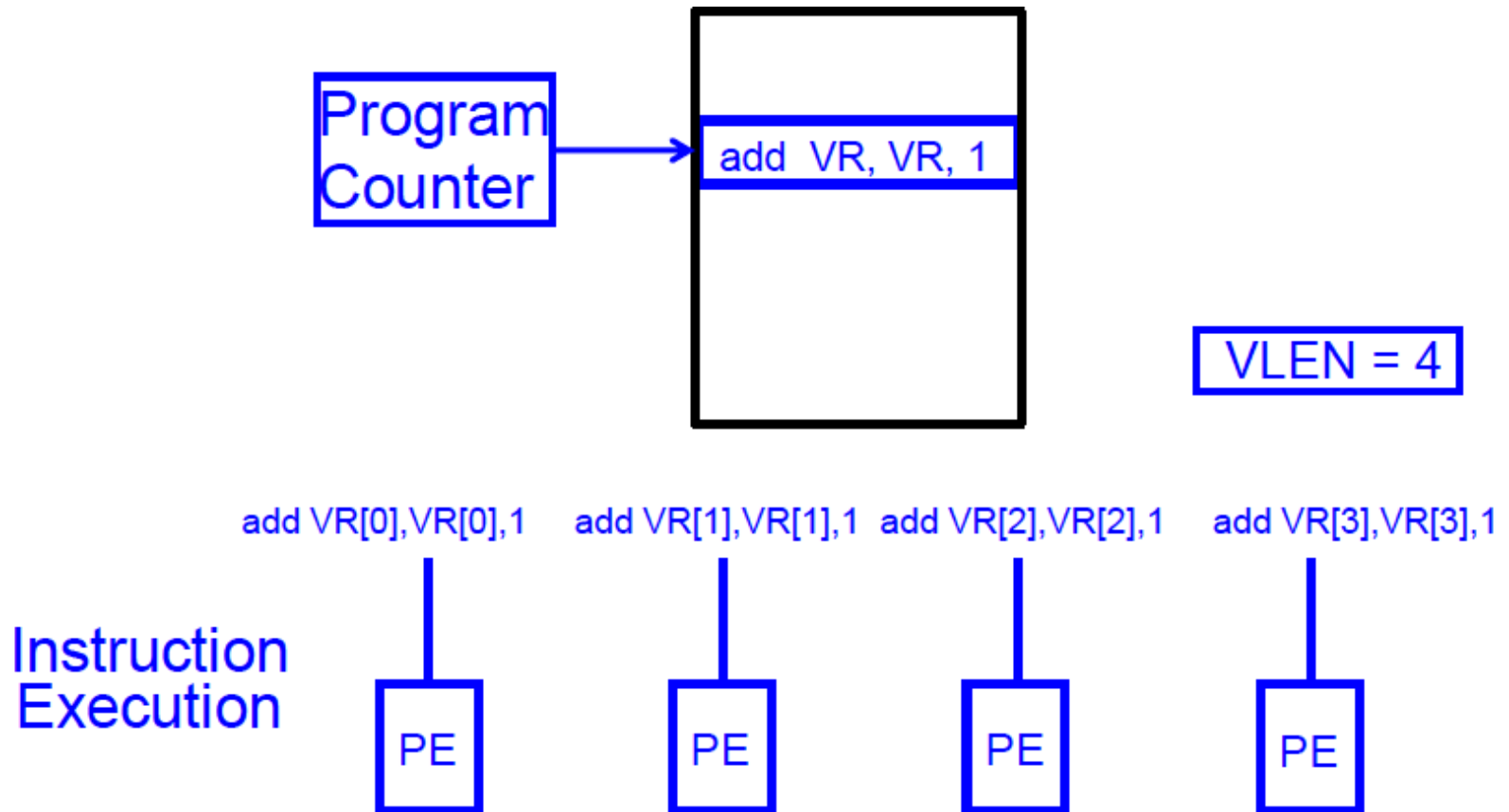# Array vs. Vector Processors

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

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

Different ops @ same space

Time

Space

Different ops @ time

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

Same op @ space

Space

SIMD Array Processing vs. VLIW

- VLIW: Multiple independent operations packed together by the compiler

Program Counter

| add r1,r2,r3 | load r4,r5+4 | mov r6,r2 | mul r7,r8,r9 |

Instruction Execution

PE   PE   PE   PE

SIMD Array Processing vs. VLIW

- Array processor: Single operation on multiple (different) data elements

Program Counter → add VR, VR, 1

VLEN = 4

add VR[0],VR[0],1   add VR[1],VR[1],1   add VR[2],VR[2],1   add VR[3],VR[3],1

Instruction Execution

PE   PE   PE   PE

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 registers/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 requirements

+ Highly regular memory access pattern
- Can interleave vector data elements across multiple memory banks for higher memory bandwidth (to tolerate memory bank access latency)
- Prefetching a vector is relatively easy

+ 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

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

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 Processing in More Depth

Vector Registers

- Each vector data register holds N M-bit values
- Vector control registers: VLEN, VSTR, VMASK
- Maximum VLEN can be N
  - Maximum number of elements stored in a vector register
- 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)

M-bit wide   M-bit wide

V0,0   V1,0
V0,1   V1,1

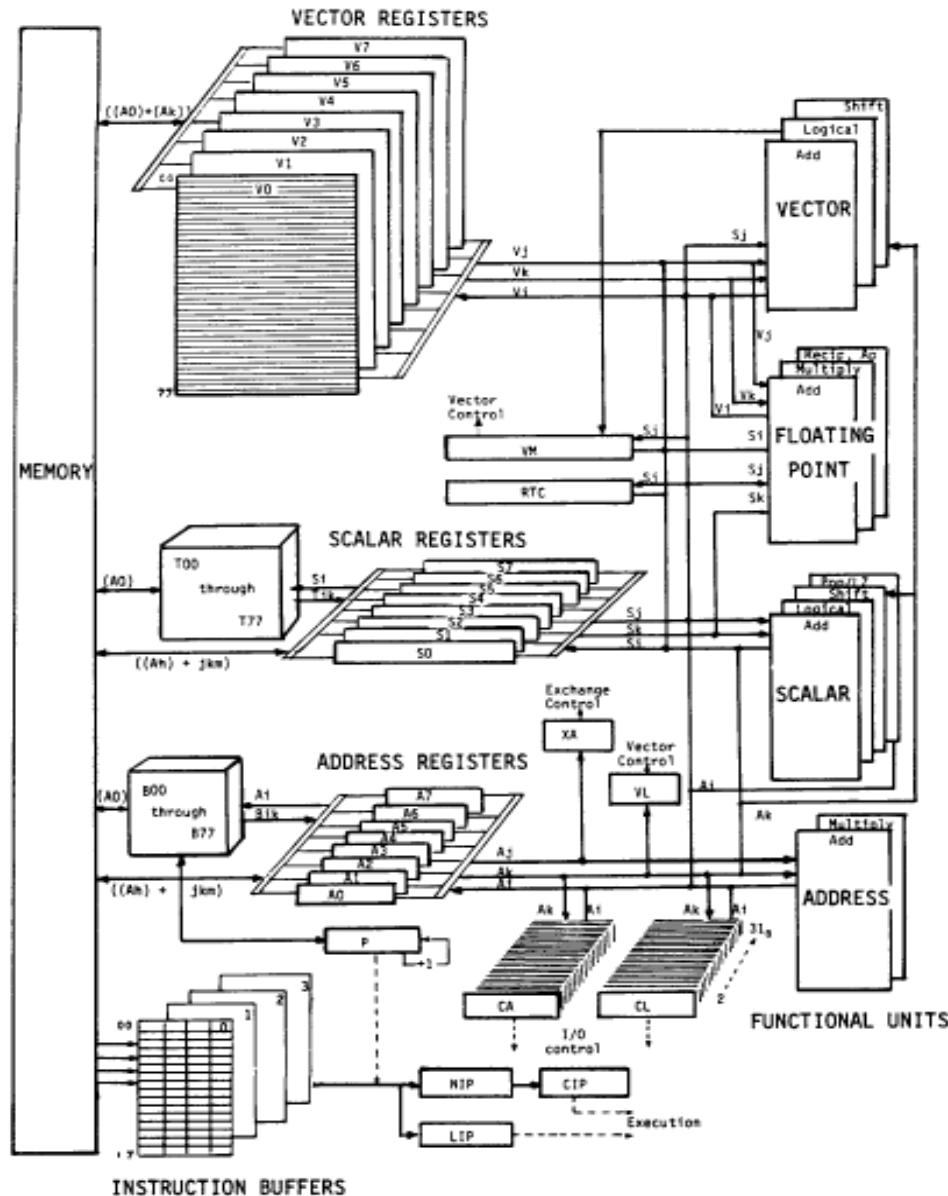V0,N-1   V1,N-1
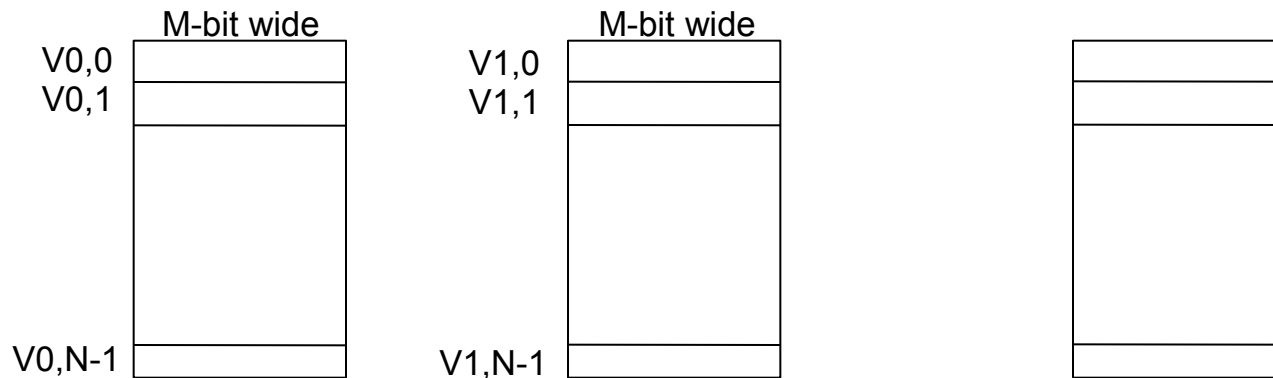
Vector Functional Units

- Use deep pipeline to execute element operations
  → fast clock cycle

- Control of deep pipeline is simple because elements in vector are independent

V
1

V
2

V
3

Six stage multiply pipeline

V1 * V2 → V3
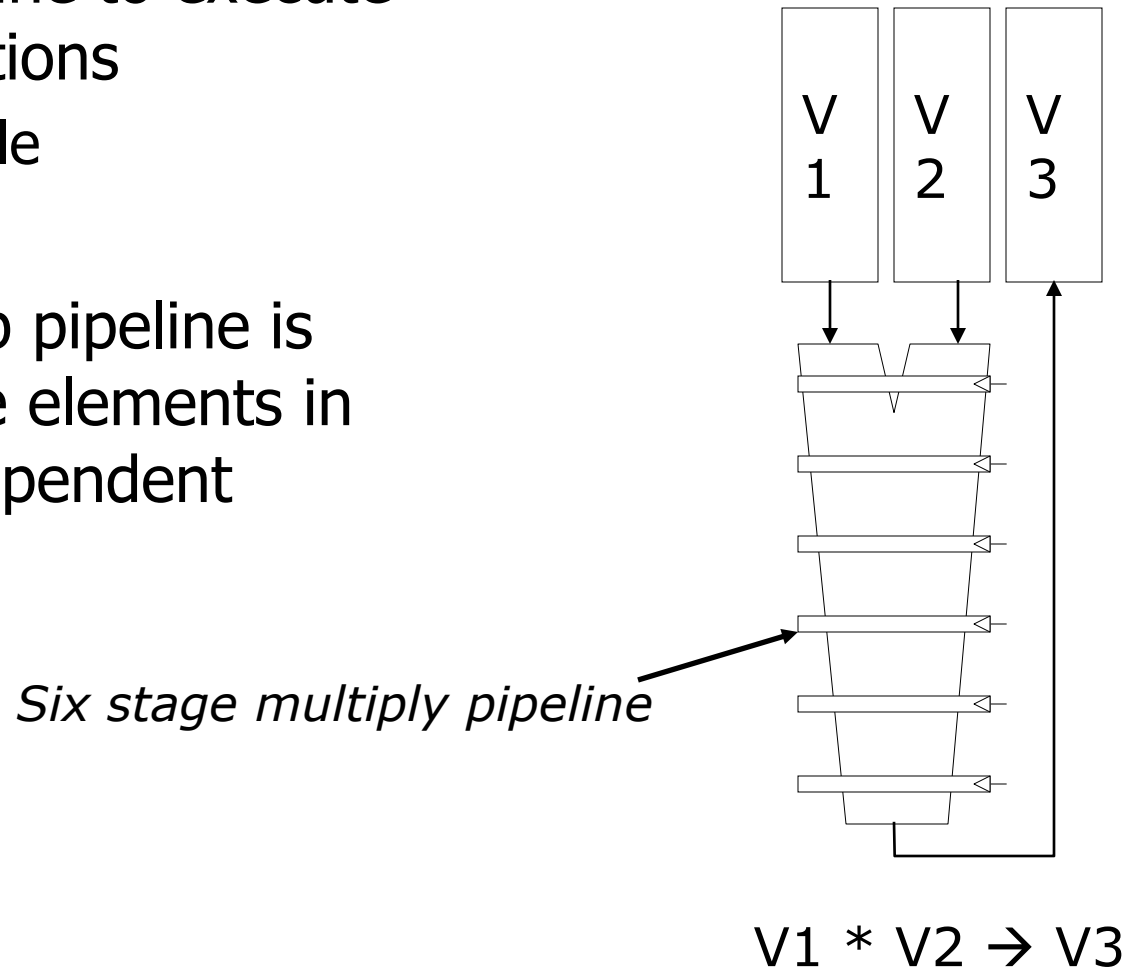
Vector Machine Organization (CRAY-1)

- 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
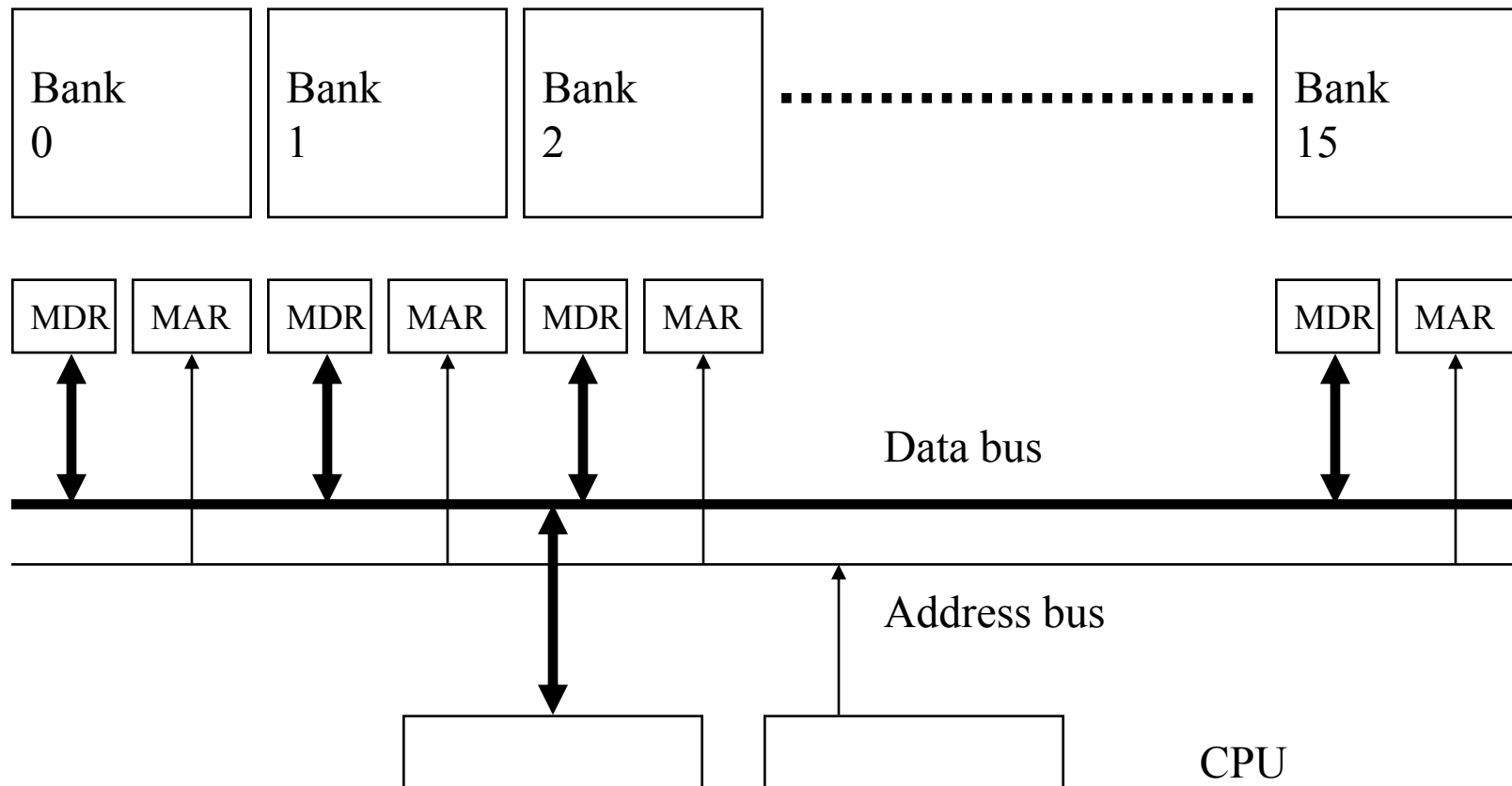- 8 64-bit scalar registers
- 8 24-bit address registers

Loading/Storing Vectors from/to Memory

- Requires loading/storing multiple elements

- Elements separated from each other by a constant distance (stride)
  - Assume stride = 1 for now

- Elements can be loaded in consecutive cycles if we can start the load of one element per cycle
  - Can sustain a throughput of one element per cycle

- Question: How do we achieve this with a memory that takes more than 1 cycle to access?

- Answer: Bank the memory; interleave the elements across banks

Memory Banking

- Memory is divided into banks that can be accessed independently; banks share address and data buses (to minimize pin cost)
- Can start and complete one bank access per cycle
- Can sustain N parallel accesses if all N go to different banks

| Bank 0 | Bank 1 | Bank 2 | .... | Bank 15 |

MDR MAR   MDR MAR   MDR MAR   MDR MAR

Data bus

Address bus

CPU

Vector Memory System

- Next address = Previous address + Stride
- If stride = 1 & consecutive elements interleaved across banks & number of banks >= bank latency, then can sustain 1 element/cycle throughput

Vector Registers

Base

Stride

Address Generator

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

Memory Banks
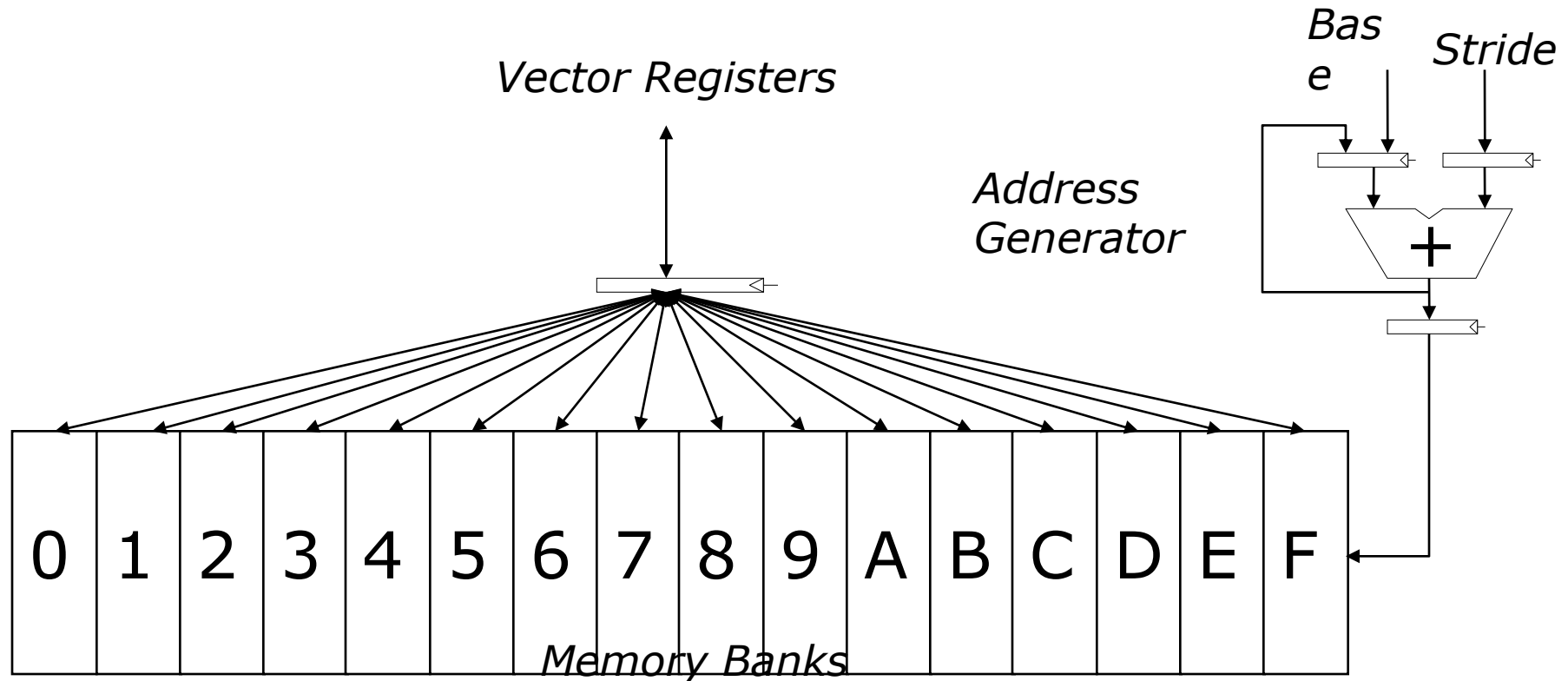
Scalar Code Example

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

- Scalar code (instruction and its latency)

```
      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 (In Order)

- 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: consecutive words are stored in consecutive banks)
  - 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
- Vectorized loop (each instruction and its latency):

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

7 dynamic instructions
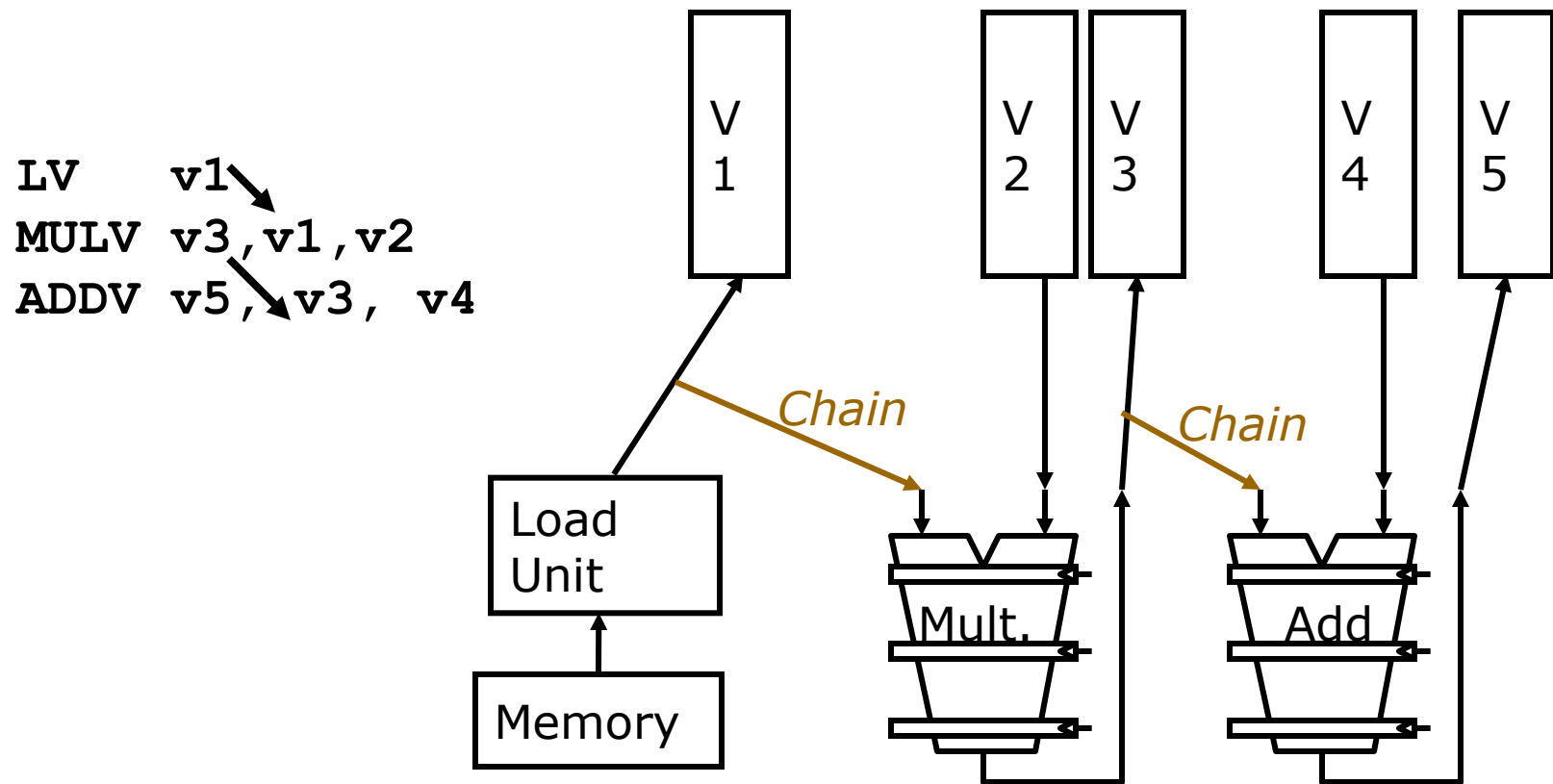
Basic Vector Code Performance

- Assume no chaining (no vector data forwarding)
  - i.e., output of a vector functional unit cannot be used as the direct input of another
  - The entire vector register needs to be ready before any element of it can be used as part of another operation
- One memory port (one address generator)
- 16 memory banks (word-interleaved)

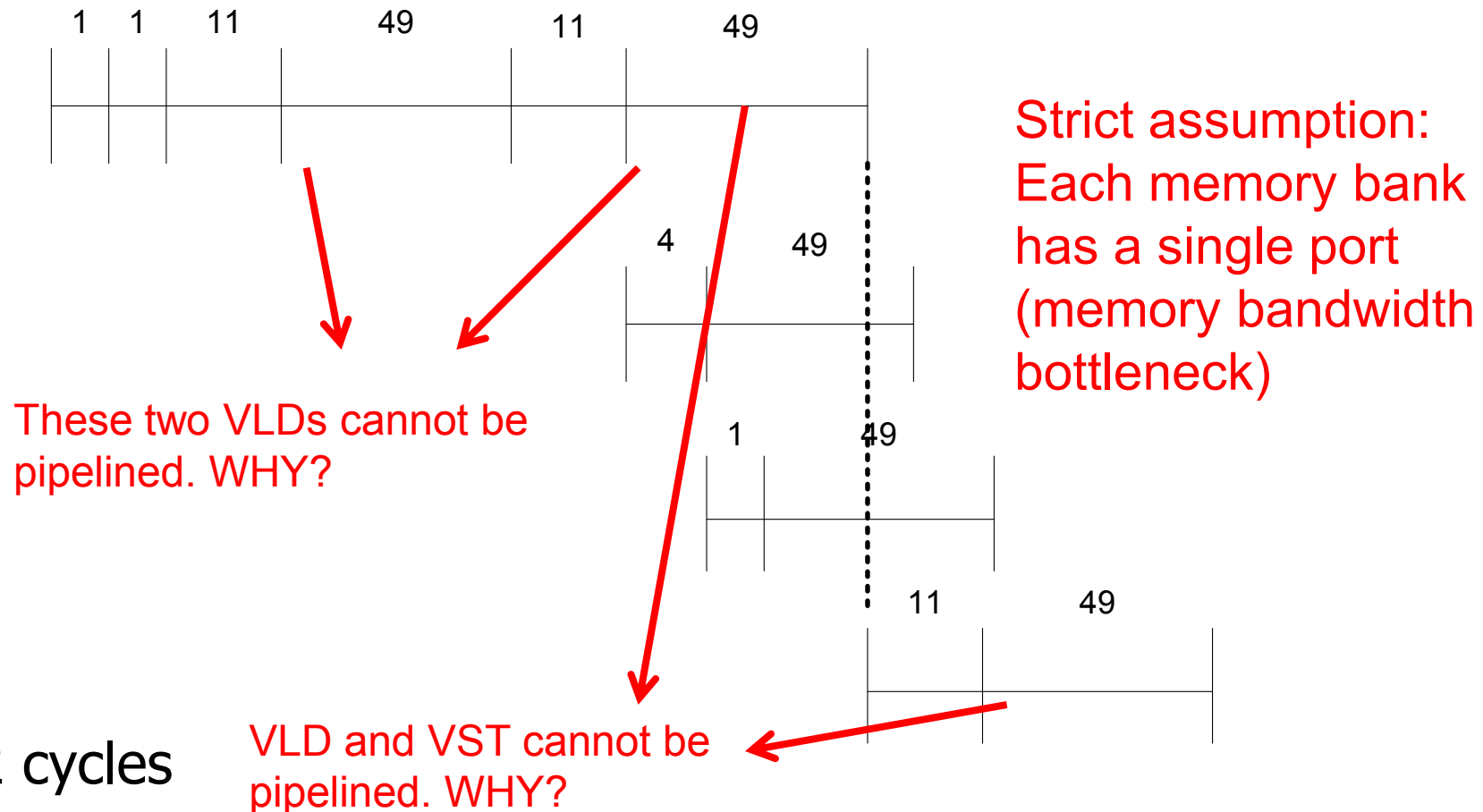| 1 | 1 | 11 | 49 | 11 | 49 | 4 | 49 | 1 | 49 | 11 | 49 |

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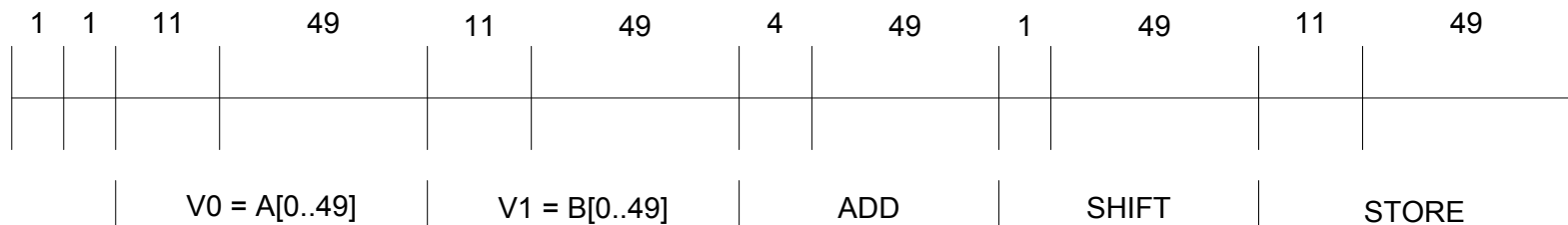
- 285 cycles

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

- 182 cycles

VLD and VST cannot be pipelined. WHY?

Vector Code Performance – Multiple Memory Ports

- Chaining and 2 load ports, 1 store port in each bank
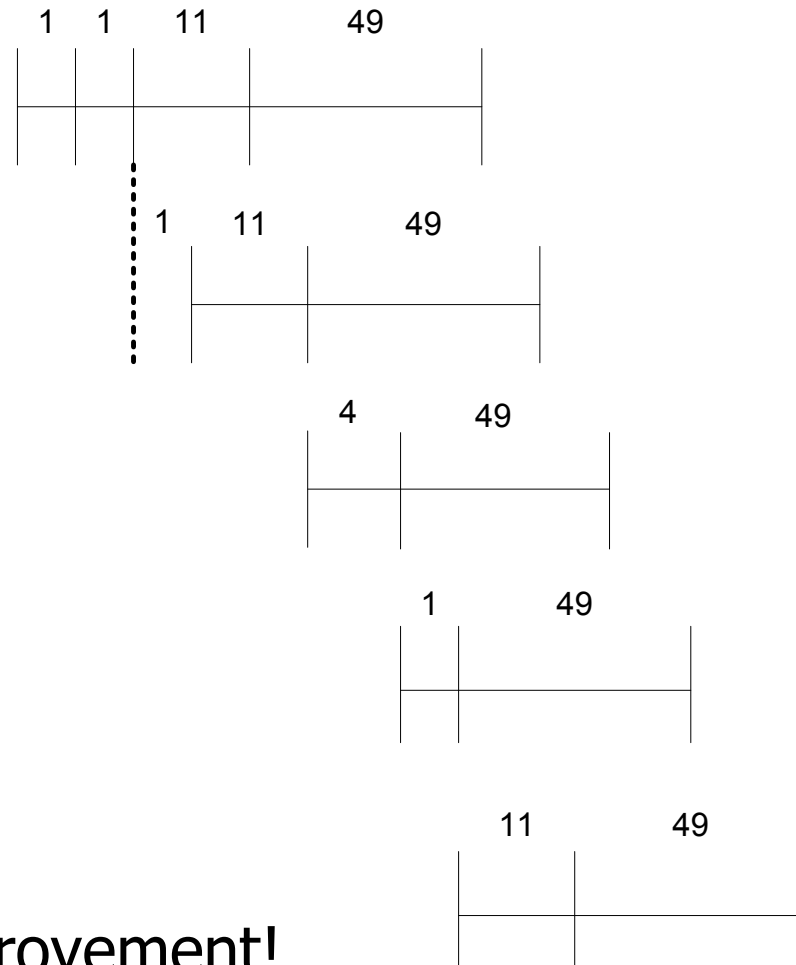
- 79 cycles
- 19X perf. improvement!

Questions (I)

- What if # data elements > # elements in a vector register?
  - Idea: 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)
  - Idea: Use indirection to combine/pack 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 (to Store)	Stored Vector (in Memory)	
0	3.14	Base+0	3.14
2	6.5	Base+1	X
6	71.2	Base+2	6.5
7	2.71	Base+3	X
		Base+4	X
		Base+5	X
		Base+6	71.2
		Base+7	2.71

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.

44

Another Example with Masking

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

Steps to execute the loop in SIMD code

1. Compare A, B to get
         VMASK

2. Masked store of A into C
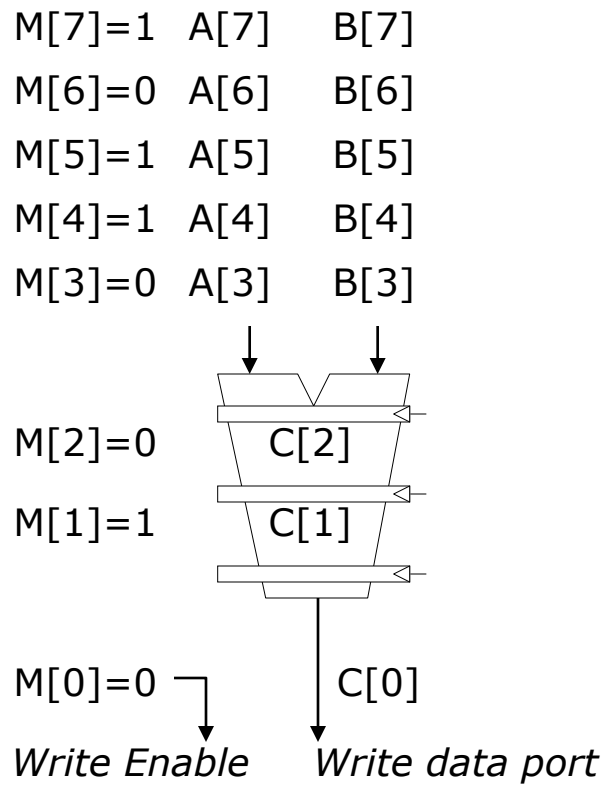
3. Complement VMASK

4. Masked store of B into C

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

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
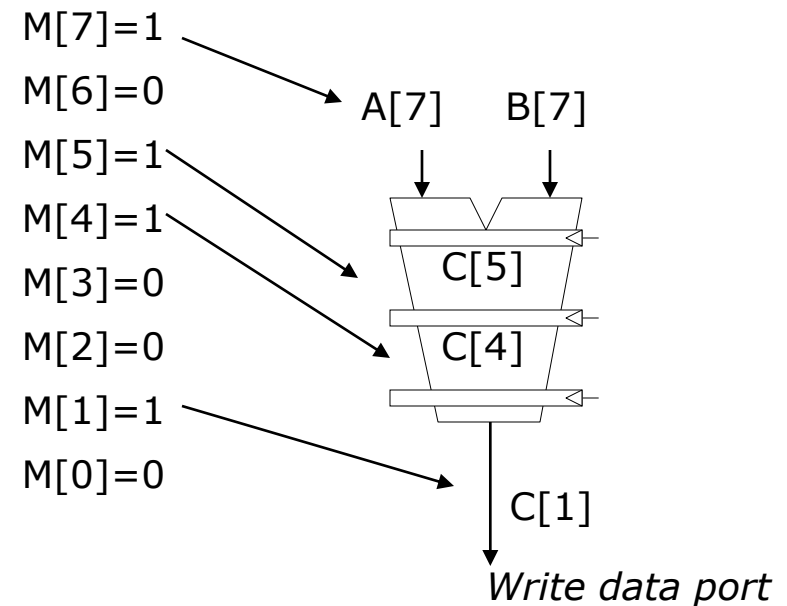
Which one is better?

Tradeoffs?

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, we can sustain 1 element/cycle throughput

- **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

A_0
0	1	2	3	4	5
6	7	8	9	10	11

B_0
0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20									
30									
40									
50									

$A_{4\times6} \cdot B_{8\times10} \rightarrow C_{4\times10}$ (dot products of rows & columns of A & B)

A: Load A_0 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 B_0 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?
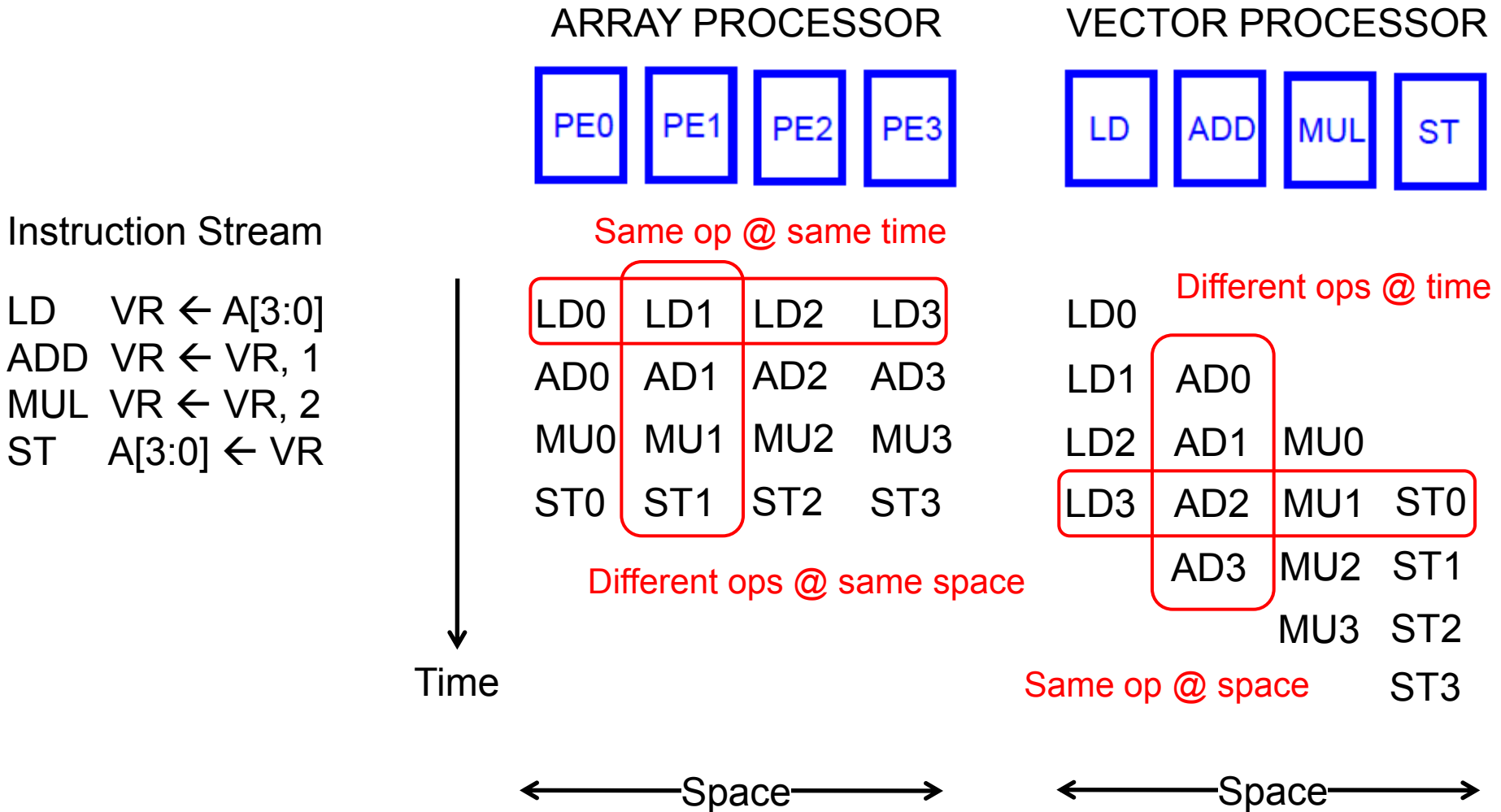
Minimizing Bank Conflicts

- More banks

- Better data layout to match the access pattern
  - Is this always possible?

- Better mapping of address to bank
  - E.g., randomized mapping
  - Rau, "Pseudo-randomly interleaved memory," ISCA 1991.

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
  - GPUs are a prime example we will cover in a bit more detail

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
AD0	AD1	AD2	AD3
MU0	MU1	MU2	MU3
ST0	ST1	ST2	ST3

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

Different ops @ same space

Same op @ space

Time ↓

← Space →   ← Space →

Vector Instruction Execution

VADD A,B → C

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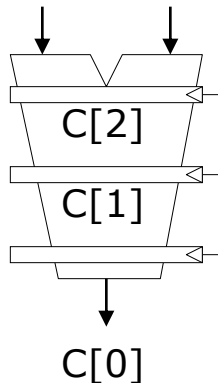
Execution using four pipelined functional units

| 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[9]   C[10]   C[11]
C[4]   C[5]   C[6]   C[7]

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

Vector Unit Structure

Functional Unit

Partitioned Vector Registers

Lane

Elements 0, 4, 8, …

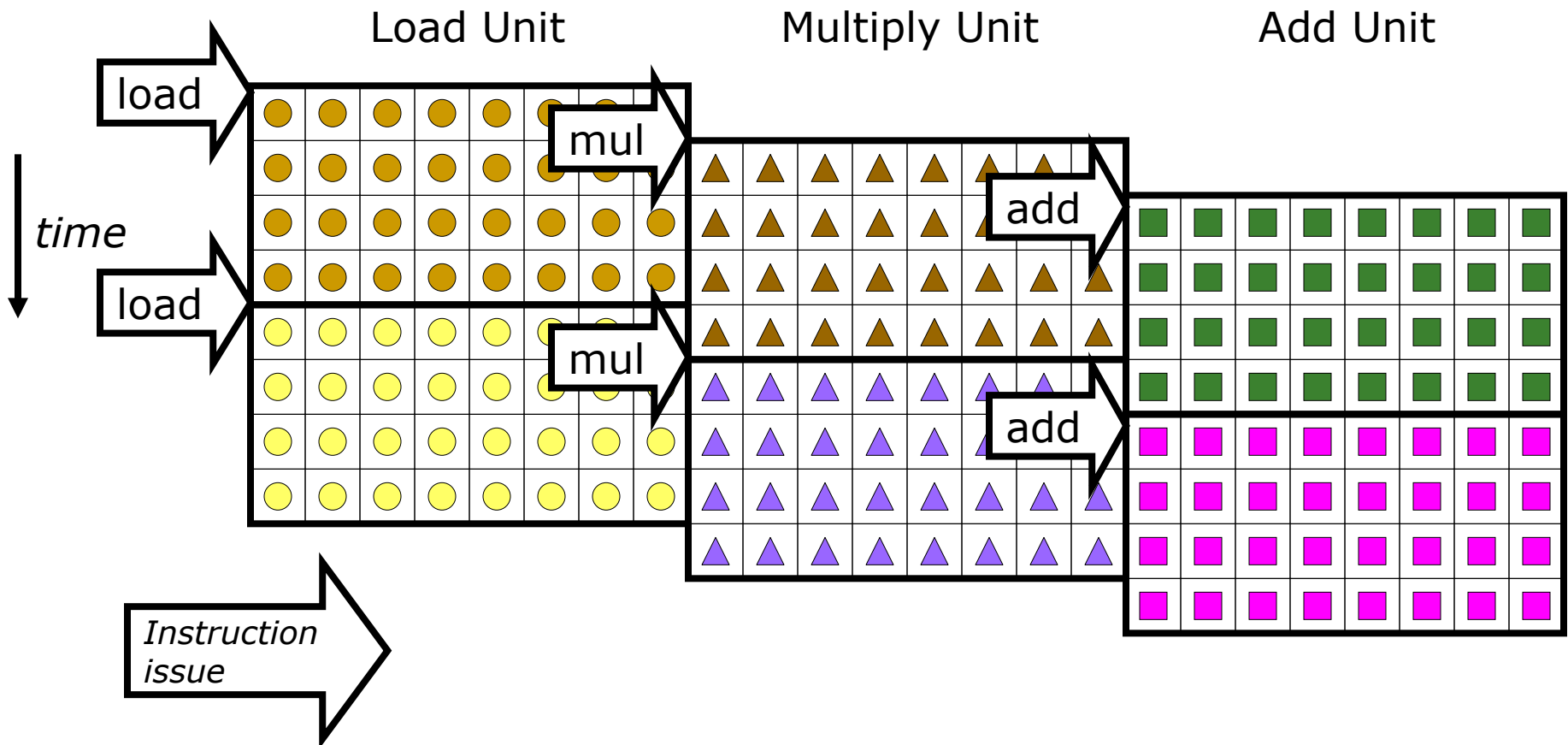Elements 1, 5, 9, …

Elements 2, 6, 10, …

Elements 3, 7, 11, …

Memory Subsystem
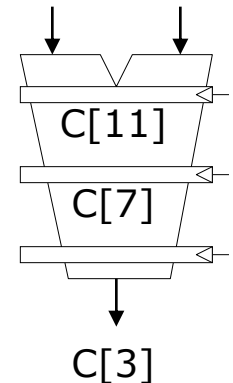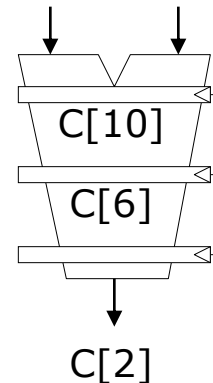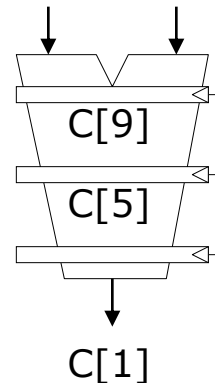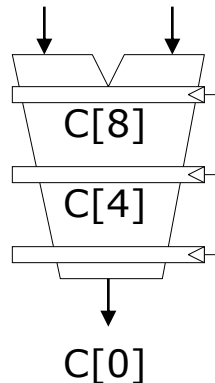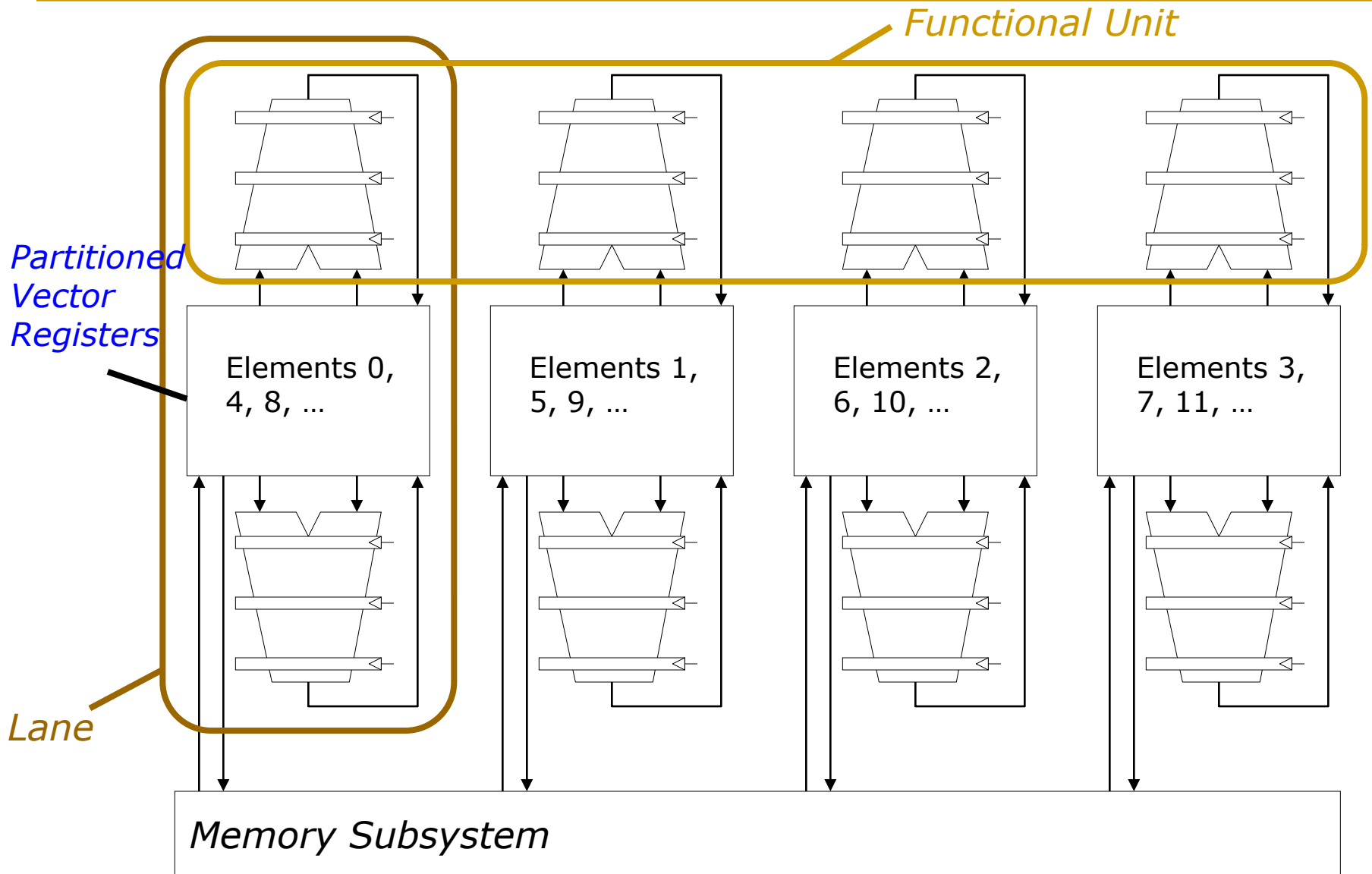
Vector Instruction Level Parallelism
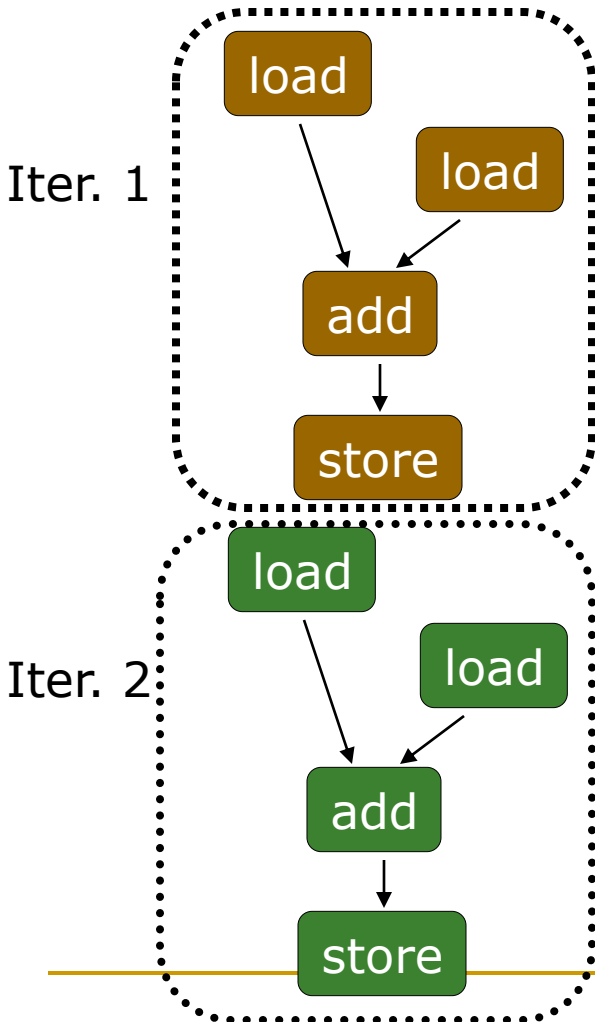
Can overlap execution of multiple vector instructions

❑ Example machine has 32 elements per vector register and 8 lanes

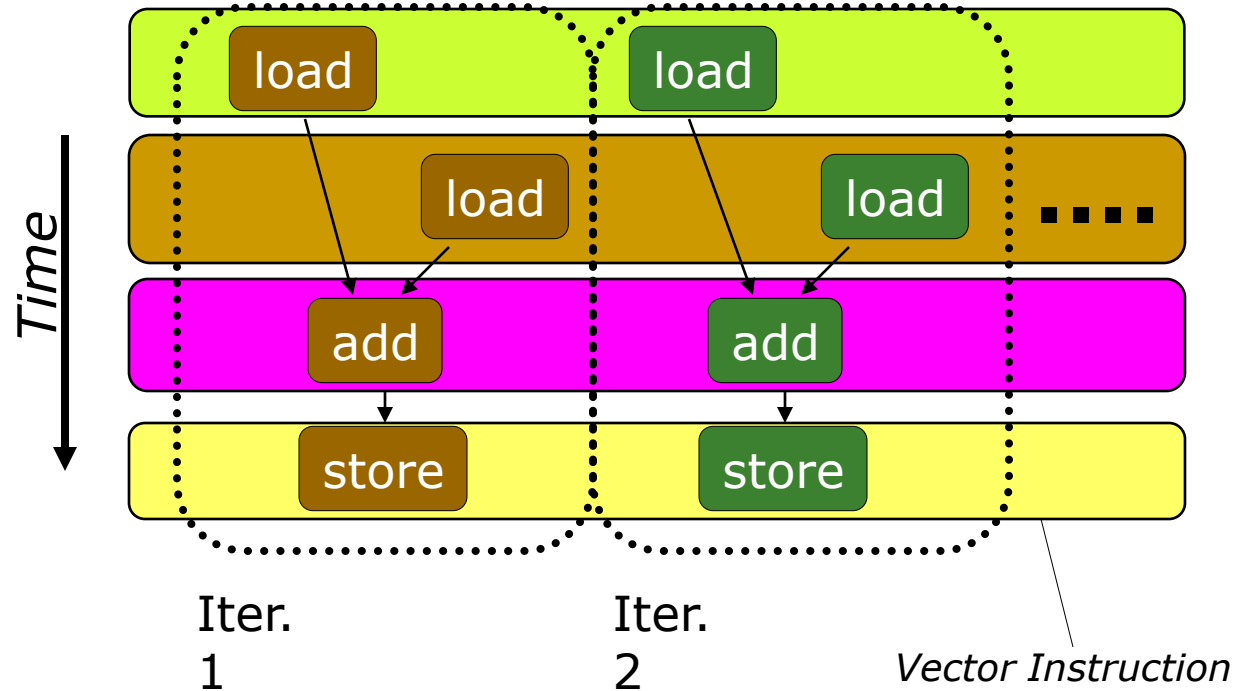❑ Completes 24 operations/cycle while issuing 1 vector instruction/cycle

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 are 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
  - Remember 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
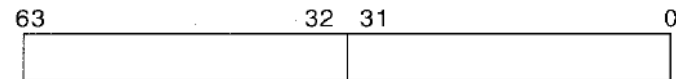
Intel Pentium MMX Operations

- Idea: One instruction operates on multiple data elements simultaneously
  - Ala array processing (yet much more limited)
  - Designed with multimedia (graphics) operations in mind
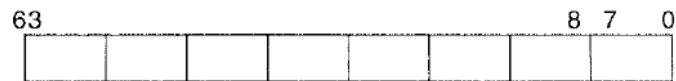
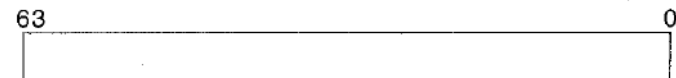Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

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)

- Goal: Overlay the human in image 1 on top of the background in image 2

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

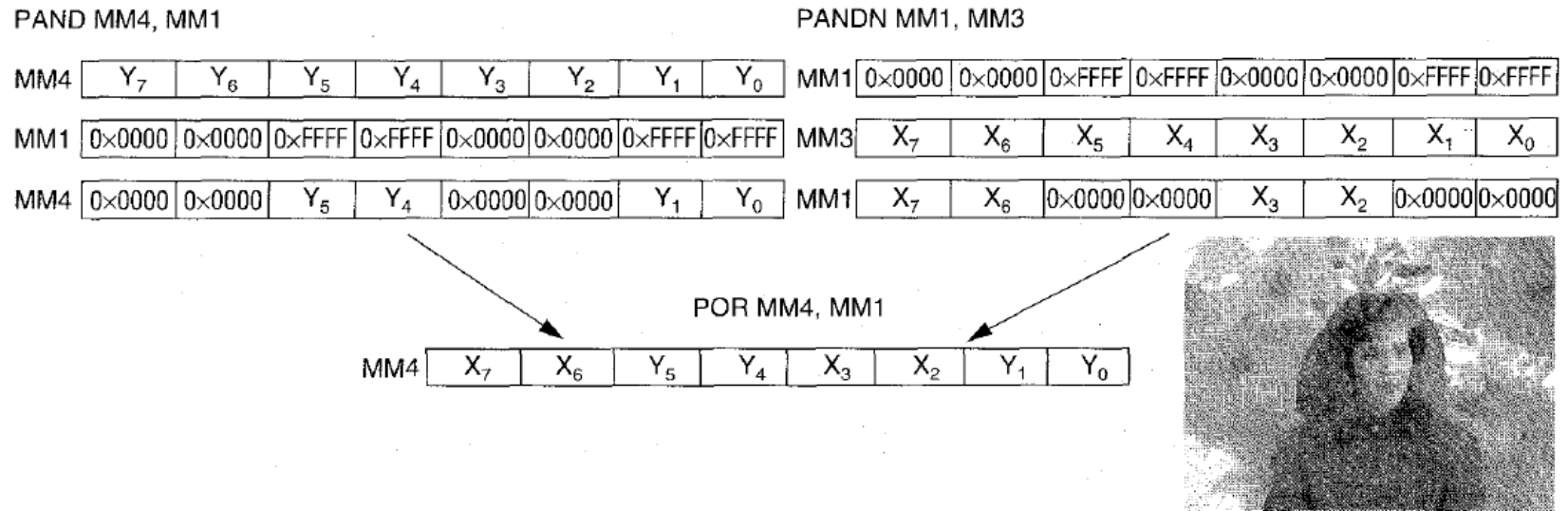| MM4 | X_7 | X_6 | Y_5 | Y_4 | X_3 | X_2 | Y_1 | Y_0 |

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.