

CS/ECE 5381/7381  
Computer Architecture  
Spring 2023

Dr. Manikas  
Computer Science  
Lecture 18: Apr. 6, 2023

# Assignments

- Project 6 (7381 only)
  - Due TODAY Thur., Apr. 6 (11:59 pm)
- Quiz 8 – due Sat., Apr. 8 (11:59 pm)
  - Covers concepts from Modules 9 and 10

# Quiz 8 Details

- The quiz is open book and open notes.
- You are allowed 90 minutes to take this quiz.
- You are allowed 2 attempts to take this quiz - your highest score will be kept.
  - Note that some questions (e.g., fill in the blank) will need to be graded manually
- Quiz answers will be made available 24 hours after the quiz due date.

# Data-Level Parallelism

(Chapter 4, Hennessy and Patterson)

Note: some course slides adopted  
from publisher-provided material

# Data-Level Parallelism

(Chapter 4, Hennessy and Patterson)

Note: some course slides adopted  
from publisher-provided material

# Outline

- 4.1 Introduction
- 4.2 Vector Architecture
- 4.4 Graphics Processing Units (GPU's)

# Flynn's Taxonomy (Ch. 1)

- Single instruction stream, single data stream (SISD)
  - Instruction-Level Parallelism (Ch. 3)
- Single instruction stream, multiple data streams (SIMD) (Ch. 4)
  - Vector architectures
  - Graphics processor units
- Multiple instruction streams, multiple data streams (MIMD)
  - Thread-Level Parallelism (Ch. 5)

# Outline

- 4.1 Introduction
- 4.2 Vector Architecture
- 4.4 Graphics Processing Units (GPU's)

# Vector Architectures

- Basic idea:
  - Read sets of data elements into “vector registers”
  - Operate on those registers
  - Disperse the results back into memory
- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth

返数

# Review: Instruction Level Parallelism

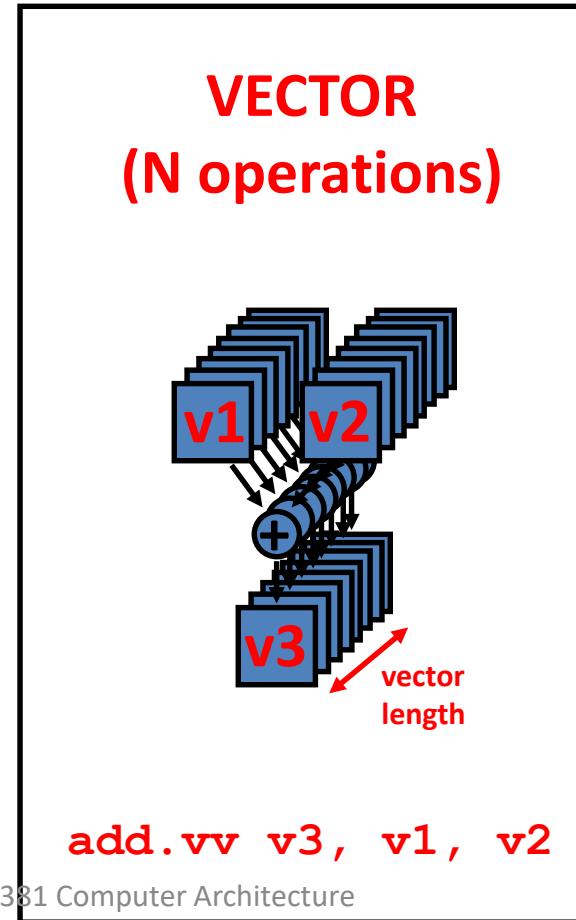
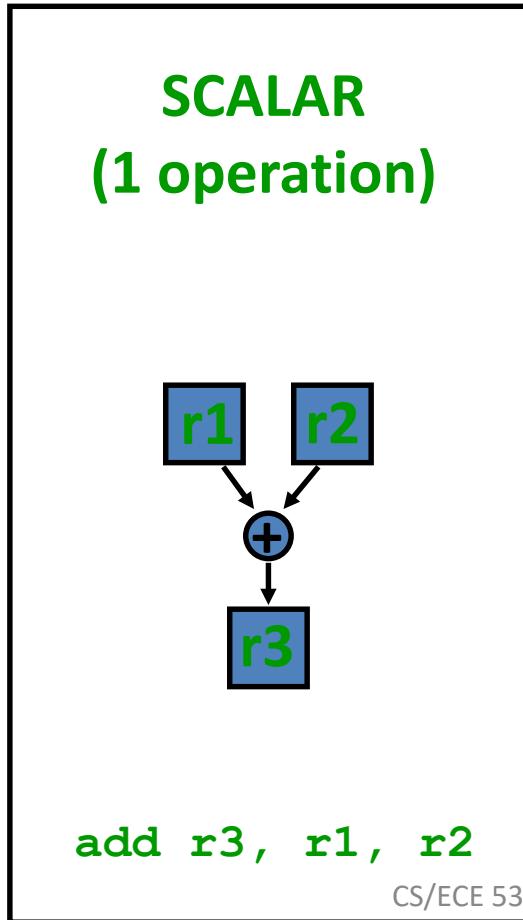
- High speed execution based on *instruction level parallelism* (ILP): potential of short instruction sequences to execute in parallel
- High-speed microprocessors exploit ILP by:
  - Pipelined execution: overlap instruction
  - Out-of-order execution (commit in-order)
    - E.g., Tomasulo's algorithm

# Problems with conventional approach

- Limits to conventional exploitation of ILP:
  - 1) *pipelined clock rate*: at some point, each increase in clock rate has corresponding CPI increase (branches, other hazards)
  - 2) *instruction fetch and decode*: at some point, it's hard to fetch and decode more instructions per clock cycle
  - 3) *cache hit rate*: some long-running (scientific) programs have very large data sets accessed with poor locality; others have continuous data streams (multimedia) and hence poor locality

# Alternative Model: Vector Processing

- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"



# EXAMPLE : VECTOR ADDITION

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix}$$

$$\Rightarrow c_i = a_i + b_i \quad i = \{0, n-1\}$$

USE "VECTOR PROCESSING"

TO DO THESE ADDITIONS IN PARALLEL

VECTORS A, B, C (Each of size n)

PROGRAM?

$$C = A + B$$

WILL TAKE

1 ITERATIONS

NO LOOP : NO CONTROL HAZARDS

PROGRAM? C/C++

for ( $i \geq 0$ ;  $i < N$ ;  $i++$ )

$$c[i] = a[i] + b[i];$$

WILL TAKE N ITERATIONS

NOTE: EACH ADDITION IS INDEPENDENT

$$c_0 = a_0 + b_0$$

$$c_1 = a_1 + b_1$$

$$c_2 = a_2 + b_2$$

:

$$c_{N-1} = a_{N-1} + b_{N-1}$$

NO DATA  
DEPENDENCIES



WE CAN DO  
ADDITION IN  
PARALLEL

# Vector Processing Definitions

- Vector - a set of scalar data items, all of the same type, stored in memory
- Vector processor - an ensemble of hardware resources, including vector registers, functional pipelines, processing elements, and register counters for performing vector operations
- Vector processing occurs when arithmetic or logical operations are applied to vectors

# Properties of Vector Processors

- 1) Single vector instruction specifies lots of work
  - equivalent to executing an entire loop
  - fewer instructions to fetch and decode
- 2) Computation of each result in the vector is independent of the computation of other results in the same vector
  - deep pipeline without data hazards; high clock rate
- 3) Hardware checks for data hazards only between vector instructions (once per vector, not per vector element)

# Properties of Vector Processors

- 4) Access memory with known pattern
  - elements are all adjacent in memory => highly interleaved memory banks provides high bandwidth.
  - access is initiated for entire vector => high memory latency is amortised (no data caches are needed)
- 5) Control hazards from the loop branches are reduced
  - nonexistent for one vector instruction

# EXAMPLE : VECTOR ADDITION

$$\begin{bmatrix} c \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix}$$

SPATIAL  
LOCALITY

$$\Rightarrow c_i = a_i + b_i \quad i = \{0, n-1\}$$

PROGRAM? C/C++

for ( $i \geq 0$ ;  $i < N$ ;  $i++$ )

$$c[i] = a[i] + b[i];$$

CONTROL HAZARDS

WILL TAKE N ITERATIONS

NOTE: EACH ADDITION IS INDEPENDENT

$$c_0 = a_0 + b_0$$

$$c_1 = a_1 + b_1$$

$$c_2 = a_2 + b_2$$

:

$$c_{N-1} = a_{N-1} + b_{N-1}$$

NO DATA

DEPENDENCIES



WE CAN DO

ADDITION IN

PARALLEL

# More Properties of Vector Processors

- Vector operations: arithmetic (add, sub, mul, div), memory accesses, effective address calculations
- Multiple vector instructions can be in progress at the same time => more parallelism
- Applications to benefit
  - Large scientific and engineering applications
  - Multimedia applications

# Basic Vector Architectures

- Vector processor: ordinary pipelined scalar unit + vector unit
- Example architecture: VMIPS
  - “Vector-Register” version of MIPS

# Components of a vector-register processor

- Vector Registers: each vector register is a fixed length bank holding a single vector
  - has at least 2 read and 1 write ports
  - typically 8-32 vector registers, each holding 64-128 64 bit elements
  - VMIPS: 8 vector registers, each holding 64 elements (16 Rd ports, 8 Wr ports)

## VECTOR REGISTER

HOLDS 64-BIT ELEMENTS

WHY? RECALL FROM PREVIOUS

LECTURE: IEEE DOUBLE-PRECISION

⇒ 64 BITS PER ELEMENT

VECTOR PROCESSOR: MATH + SCIENCE

PROBLEMS ⇒ NEED DOUBLE-PRECISION

# Components of a vector-register processor

- Vector Functional Units (FUs): fully pipelined, start new operation every clock
  - typically 4 to 8 FUs: FP add, FP mult, FP reciprocal ( $1/X$ ), integer add, logical, shift;
  - may have multiple of same unit
  - VMIPS: 5 FUs (FP add/sub, FP mul, FP div, FP integer, FP logical)

WHY DO WE NEED INTEGER

FUNCTIONS FOR VECTOR PROCESSING?

$$\begin{bmatrix} X \\ Z_0 \\ Z_1 \\ Z_2 \\ \vdots \\ Z_{N-1} \end{bmatrix} = \text{Scalar } K \begin{bmatrix} Y \\ Y_0 \\ Y_1 \\ Y_2 \\ \vdots \\ Y_{N-1} \end{bmatrix} + \begin{bmatrix} X \\ x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{bmatrix}$$

X, Y, Z  
ARE  
VECTORS

K IS A

SCALAR

$$z_0 = k y_0 + x_0$$

$$z_1 = k y_1 + x_1$$

$$z_2 = k y_2 + x_2$$

$\vdots$  ↗ multiply scalar  
By each element in  
vector ↴

$\Rightarrow$  "INTEGER function"

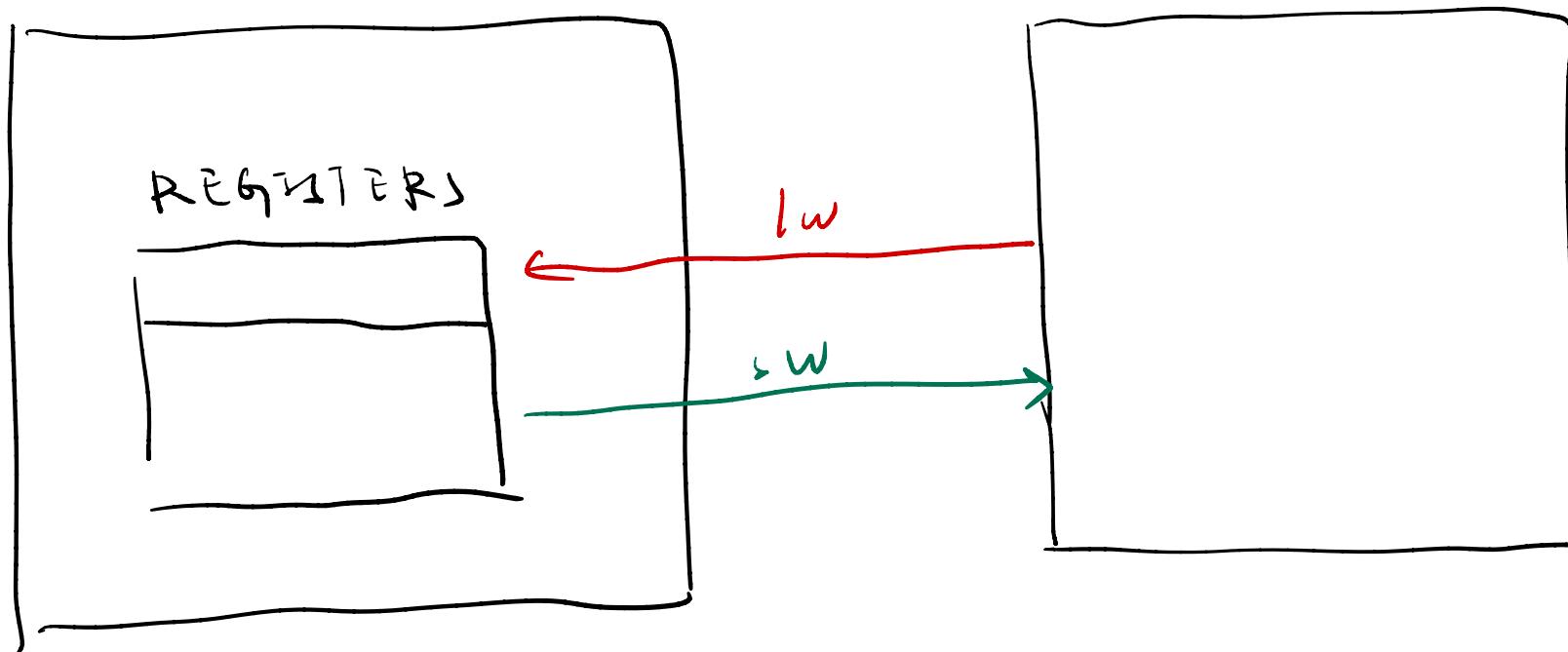
# Components of a vector-register processor

- Vector Load-Store Units (LSUs)
  - fully pipelined unit to load or store a vector;  
may have multiple LSUs
  - VMIPS: 1 LSU,  
bandwidth is 1 word per cycle after initial delay

RECALL MIPS

PROCESSOR

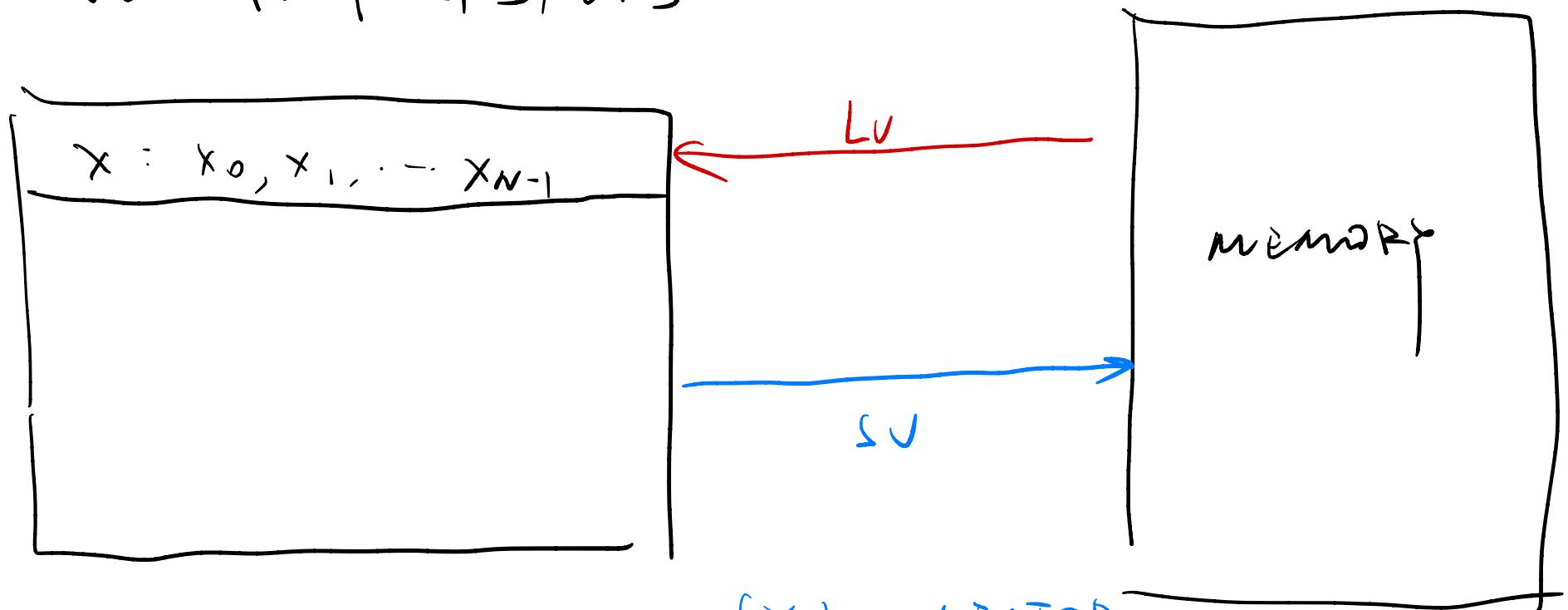
MEMORY



"SCRATCH" REGISTERS (ONE VALUE)

# VECTOR PROCESSING

VECTOR REGISTERS



VECTOR

STORE VECTOR

LOAD-STORE UNIT

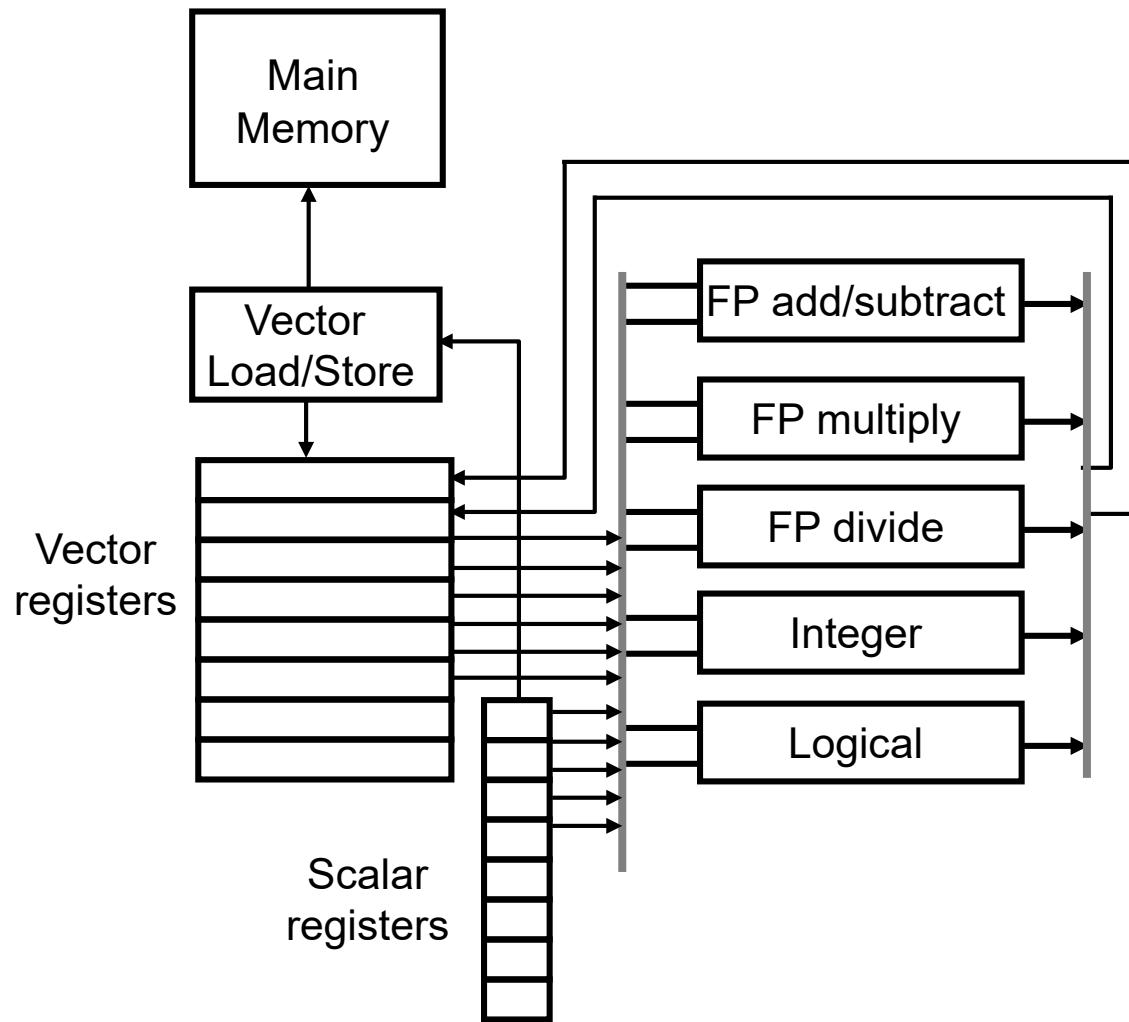
DOES THIS OPERATION

(LSU)

# Components of a vector-register processor

- Scalar registers
  - single element for FP scalar or address
  - VMIPS: 32 GPRs (General-Purpose Register), 32 FPRs (Floating-Point Register)
    - they are read out and latched at one input of the FUs
- Cross-bar to connect FUs, LSUs, registers
  - cross-bar to connect Rd/Wr ports and FUs

# VMIPS: Basic Structure



- 8 64-element vector registers
- 5 FUs; each unit is fully pipelined, can start a new operation on every clock cycle
- Load/store unit - fully pipelined
- Scalar registers

# VMIPS Instructions

- ADDVV.D: add two vectors
- ADDVS.D: add vector to a scalar
- LV/SV: vector load and vector store from address

# VMIPS Program Example

- Solve  $Y = aX + Y$ 
  - $a$  is a scalar,  $X, Y$  are vectors (64 element registers)

L.D	F0,a	; load scalar $a$
LV	V1,Rx	; load vector $X$
MULVS.D	V2,V1,F0	; vector-scalar multiply
LV	V3,Ry	; load vector $Y$
ADDVV.D	V4,V2,V3	; add
SV	Ry,V4	; store the result

- Requires 6 instructions vs. almost 600 for MIPS

# Vector Execution Time

- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies
- VMIPS functional units consume one element per clock cycle
  - Execution time is approximately the vector length
- *convoy*
  - Set of vector instructions that could potentially execute together

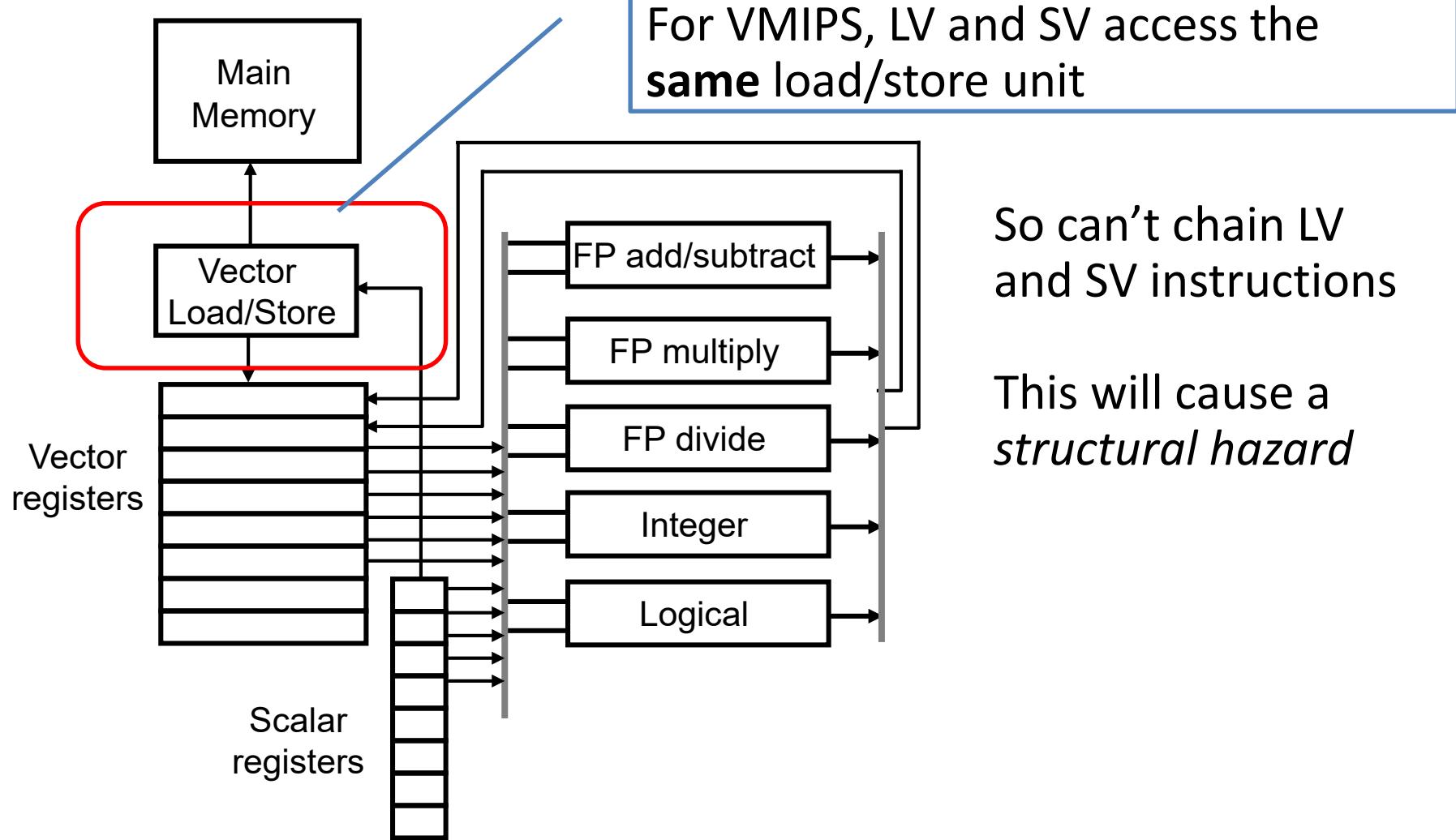
# Chimes

- Sequences with read-after-write dependency hazards can be in the same convoy via *chaining*
- *Chaining*
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available
- *Chime*
  - Unit of time to execute one *convoy*
  - $m$  convoys executes in  $m$  chimes
  - For vector length of  $n$ , requires  $m \times n$  clock cycles

# Guidelines for Chaining Instructions

- How to determine which instructions we can chain?
- Item to consider: structural hazards – are two instructions competing for the same resource?
- For VMIPS, LV and SV access the **same** load/store unit
- So can't chain LV and SV instructions
  - This will cause a *structural hazard*

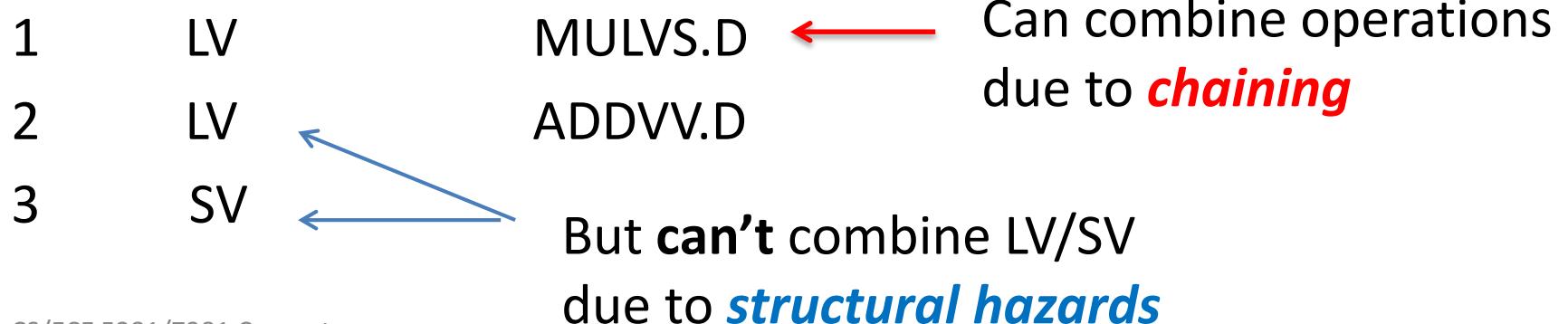
# VMIPS: Basic Structure



# Example

LV	V1,Rx	;load vector X
MULVS.D	V2,V1,F0	;vector-scalar multiply
LV	V3,Ry	;load vector Y
ADDVV.D	V4,V2,V3	;add two vectors
SV	Ry,V4	;store the sum

Convoys:



# Example

Convoys:

1	LV	MULVS.D
2	LV	ADDVV.D
3	SV	

3 **chimes**, 2 FP ops per result, cycles per FLOP = 1.5

For 64 element vectors, requires  $64 \times 3 = 192$  **clock cycles**

# Example 4.2-1

- Assume we have the following VMIPS code (next slide), which computes the equation  $Z = aX + Y$ 
    - X and Y are vectors, while value  $a$  is a scalar
    - Assume that value  $a$  has already been loaded into register F0
- 
- a) Arrange this code sequence into **convoys**.
  - b) How many **chimes** will this sequence take?
  - c) Given that the vector size is 64 elements, how many **clock cycles** are required to execute the chimes?

# VMIPS code for Example 4.2-1

```
I0:  LV          V2,Rx    ;load vector X
I1:  MULVS.D    V3,V2,F0  ;multiply vector X by scalar a
I2:  LV          V4,Ry    ;load vector Y
I3:  ADDVV.D    V5,V3,V4  ;add vectors aX + Y
I4:  SV          V5,Rz    ;store vector result in Z
```

VMIPS CODE

ASSUME  $\alpha$

is already in  $F_0$

I0: LV  $V_2, R_x$

LOAD  $x$

I1: MULVS.D  $V_3, V_2, F_0$

MULT.  $x \cdot y \alpha$

I2: LV  $V_4, R_y$

LOAD  $y$

I3: MODVV.D  $V_5, V_3, V_4$

MOD  $\alpha x + y$

I4: SV  $V_5, R_z$

STORE IN  $z$

$$\Rightarrow z = \alpha x + y$$

a) ARRANGE CODE IN CONVOTS

WHAT CAN WE DO IN PARALLEL?

AVOID STRUCTURAL HAZARDS

⇒ Can't do MULTIPLE LV/SV

IN SAME CONVOY

1. LV MULT VS. D
  2. LV ADD VS. D
  3. SV
- } 3 CONVOTS

⑤ How many chimes for  
this sequence of convoy?

RECALL: one convoy takes  
one chime

We have 3 convoys

$\Rightarrow$  will take 3 chimes

③ GIVEN: VECTOR SIZE = 64 ELEMENTS

How many CLOCK CYCLES are required to execute the CHIMES?

RECALL: NUMBER OF CLOCK CYCLES

$$= (\text{NUMBER OF VECTOR ELEMENTS})$$

$$\cdot (\text{NUMBER OF CHIMES})$$

$$= (64) (3) = 192 \text{ CLOCK CYCLES}$$

# Outline

- 4.1 Introduction
- 4.2 Vector Architecture
- 4.4 Graphics Processing Units (GPU's)

# Graphical Processing Units

- Basic idea:
  - Heterogeneous execution model
    - CPU is the *host*, GPU is the *device*
  - Develop a C-like programming language for GPU
  - Unify all forms of GPU parallelism as *CUDA thread*
  - Programming model is “Single Instruction Multiple Thread”

GPU: GRAPHICS    PROCESSING    UNIT

VECTOR PROCESSING

RECALL: CPU: CENTRAL    PROCESSING  
UNIT

- THIS IS THE PROCESSOR IN

OUR MIPS FROM EARLIER

SCALAR PROCESSING

(C4.3)

# Threads and Blocks

- A thread is associated with each data element
- Threads are organized into blocks
- Blocks are organized into a grid
- GPU hardware handles thread management, not applications or OS

# NVIDIA GPU Architecture

- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files
- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

# Terminology

- Each thread is limited to 64 registers
- Groups of 32 threads combined into a SIMD thread or “warp”
- Up to 32 warps are scheduled on a single SIMD processor
  - Each warp has its own PC (program counter)
  - Thread scheduler uses scoreboard to dispatch warps
  - By definition, no data dependencies between warps
  - Dispatch warps into pipeline, hide memory latency

INSIDE YOUR PC.

CONTROL UNIT

PROCESSOR

CPU

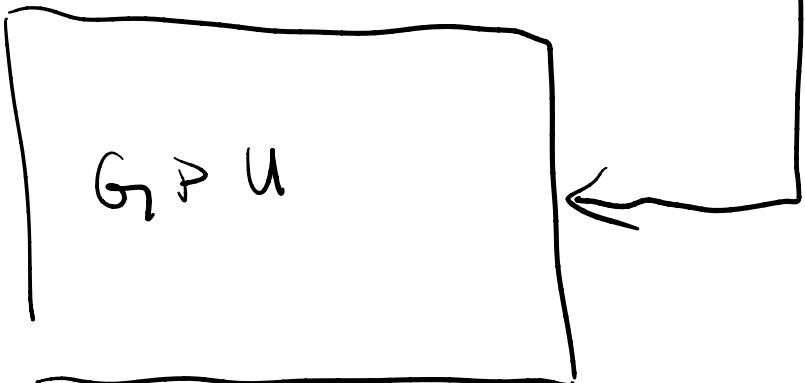
MAIN UNIT

OPERATING  
SYSTEM

Memory

GENERAL PURPOSE

PROCESSOR



"HELPER" PROCESSOR

FOCUS ON APPLICATIONS

THAT REQUIRE

VECTOR PROCESSOR

# Terminology (cont.)

- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor:
  - 32 SIMD lanes
  - Wide and shallow compared to vector processors

