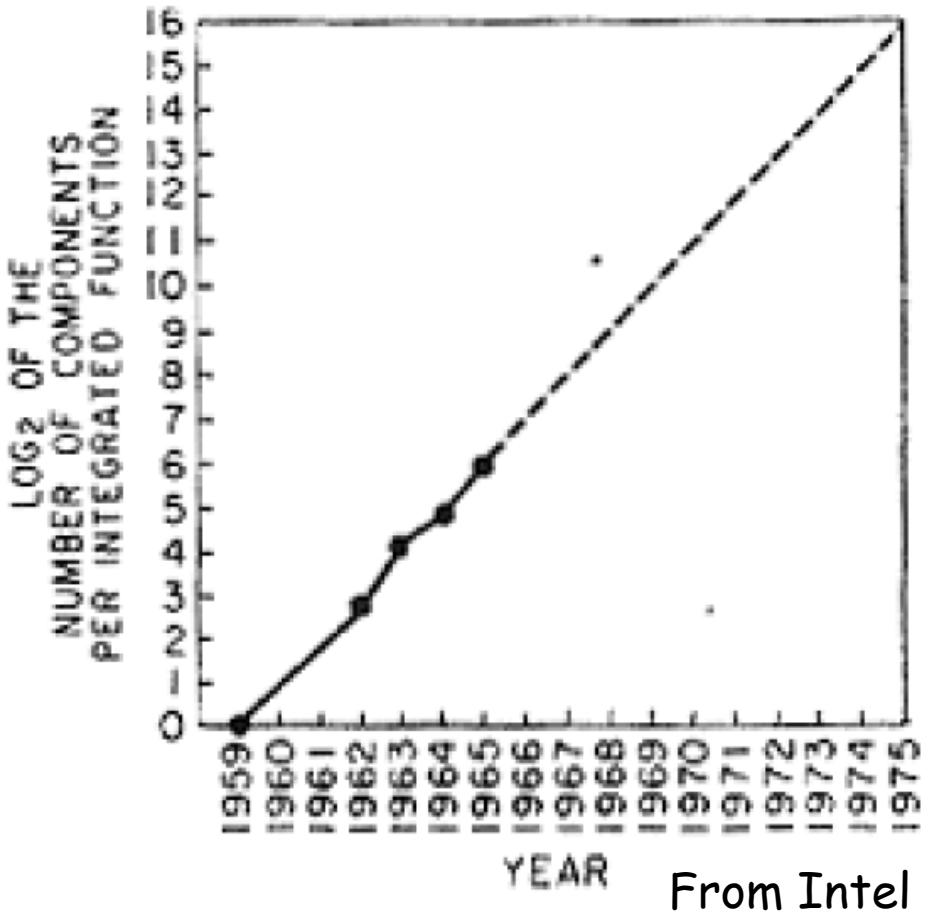
CS572 Micro Architecture Midterm Review

Dr. Tao Xie

Fall, 2017

These slides are adapted from notes by Dr. David Patterson (UCB) and Dr. Xiao Qin (Auburn)

Amazing Underlying Technology Change



- **In 1965, Gordon Moore sketched out his prediction of the pace of silicon technology.**

- **Moore's Law: The number of transistors incorporated in a chip will approximately double every 24 months.**

- **Decades later, Moore's Law remains true.**

Why Study Computer Architecture

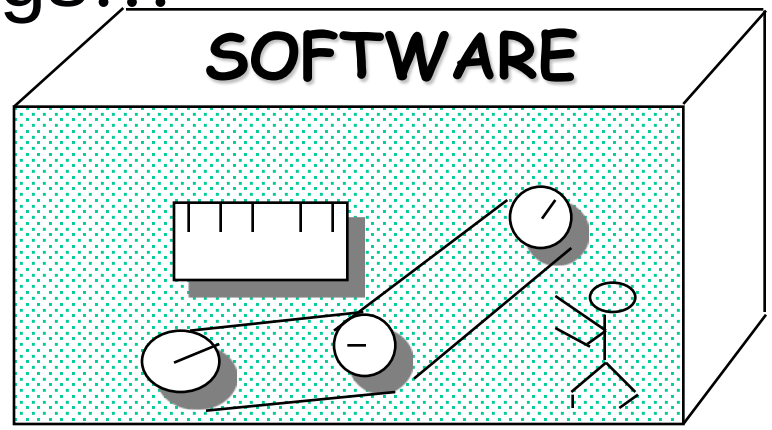
Based on SPEED, the CPU has increased dramatically, but memory and disk have increased only a little. This has led to dramatic changed in architecture, Operating Systems, and programming practices.

Answer: Technology playing field is always changing

Understand hardware for software tuning

What is Computer Architecture ?

- The science and art of selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals.
- An analogy to architecture of buildings…



Two notions of performance

Plane	DC to Paris	Speed	Passengers	Throughput (pmph)
Boeing 747	6.5 hours	610 mph	470	286,700
BAD/Sud Concodre	3 hours	1350 mph	132	178,200

- Which has higher performance?
  1. Time to deliver 1 passenger?
  2. Time to deliver 400 passengers?

How to Measure Time?

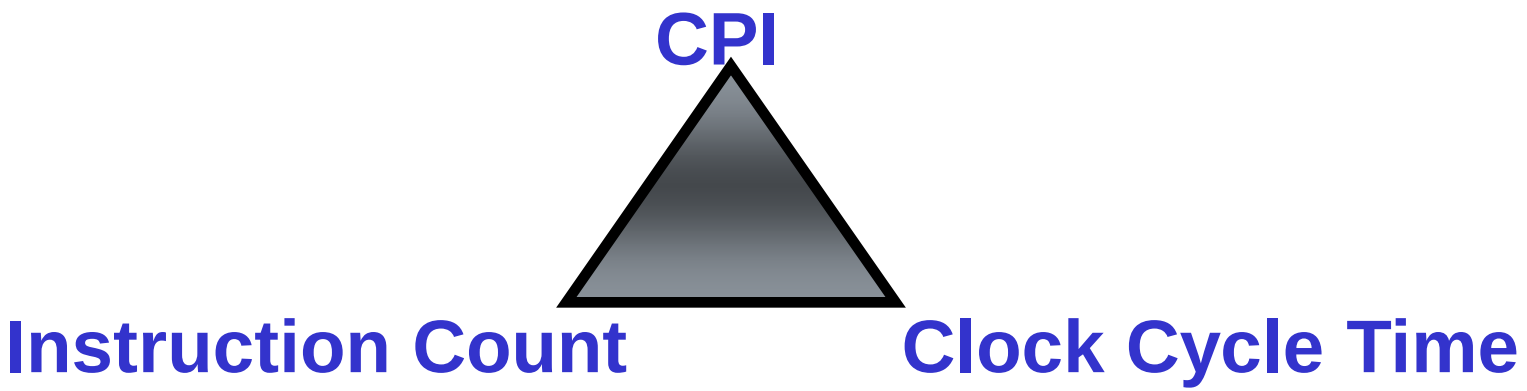
- User ⇒ actual elapsed time to complete particular task is only true basis for comparison
  - sum of I/O time, User + System CPU, time spent on other tasks, boot time, etc.
  - alternatives may mislead!
- CPU designer ⇒ want measure relating to how fast processor hardware can perform basic functions (CPU execution time)

“Iron Triangle” of CPU Performance

– CPU execution time for program
  = Clock Cycles for program x Clock Cycle Time

– Substituting for clock cycles:

CPU execution time for program
= (Instruction Count x CPI)
x Clock Cycle Time
= Instruction Count x CPI x Clock Cycle Time



Final thoughts: Performance Equation

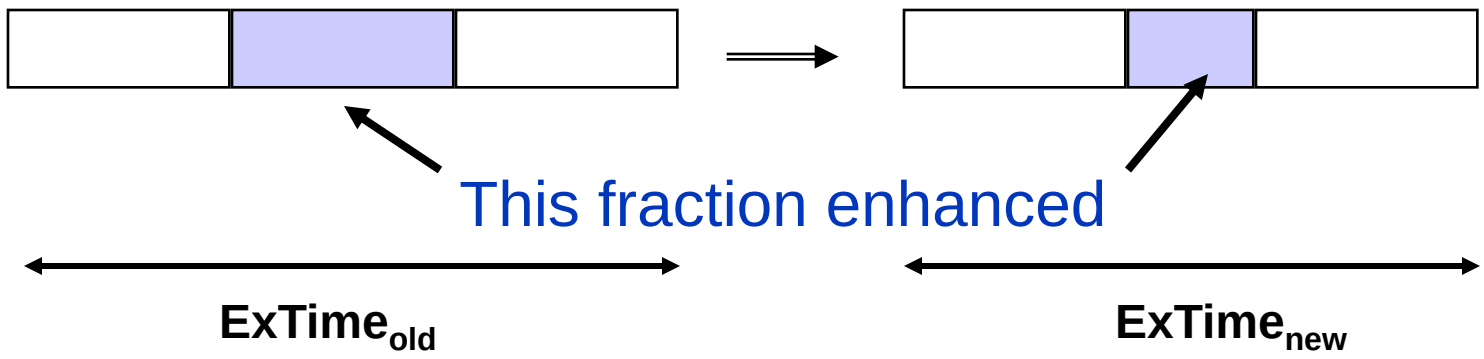
$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	Inst Count	CPI	Clock Rate
Program	X		
Compiler	X	(X)	
Inst. Set.	X	X	
Organization		X	X
Technology			X

Quantitative Design: Amdahl's Law

$$ExTime_{new} = ExTime_{old} \times \left[(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}} \right]$$

$$Speedup_{overall} = \frac{ExTime_{old}}{ExTime_{new}} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$



Quantitative Design: Amdahl's Law

- Floating point (FP) instructions improved to run 2X; but only 10% of actual instructions are FP. Suppose the old execution time is $ExTime_{old}$, What are the current execution time and speedup?

$$\mathbf{ExTime_{new} = ExTime_{old} \times (0.9 + 0.1/2) = ExTime_{old} \times 0.95}$$

$$\mathbf{Speedup_{overall} = \frac{1}{0.95} = 1.053}$$

$$\mathbf{Speedup = \frac{ExTime_{old}}{ExTime_{new}} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}}$$

$$\mathbf{Speedup = \frac{1}{(1 - 0.1) + 0.1/2} = 1.053}$$

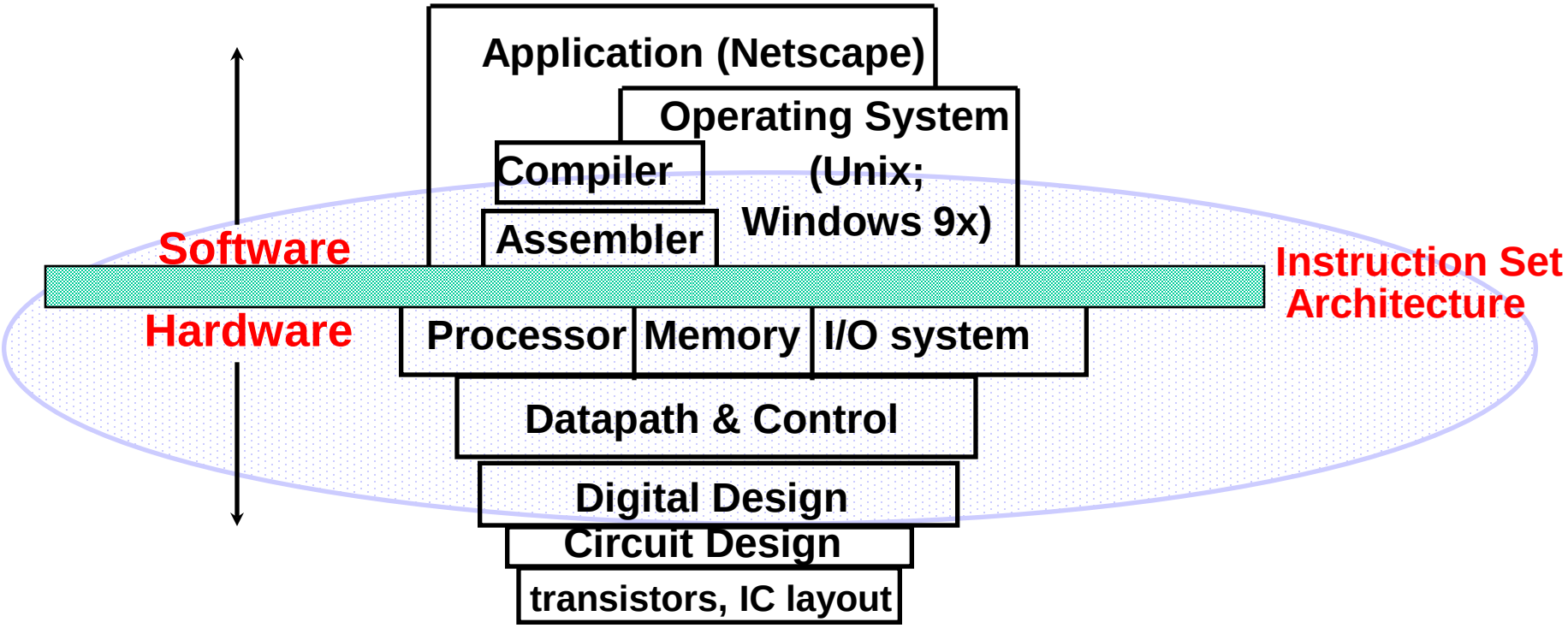
Multiple Enhancements

Amdahl's Law can be generalized to handle multiple enhancements. If only one enhancement can be used at a time during program execution, then

$$\text{Speedup} = \left[1 - \sum_i \text{FE}_i + \sum_i \frac{\text{FE}_i}{\text{SE}_i} \right]^{-1}$$

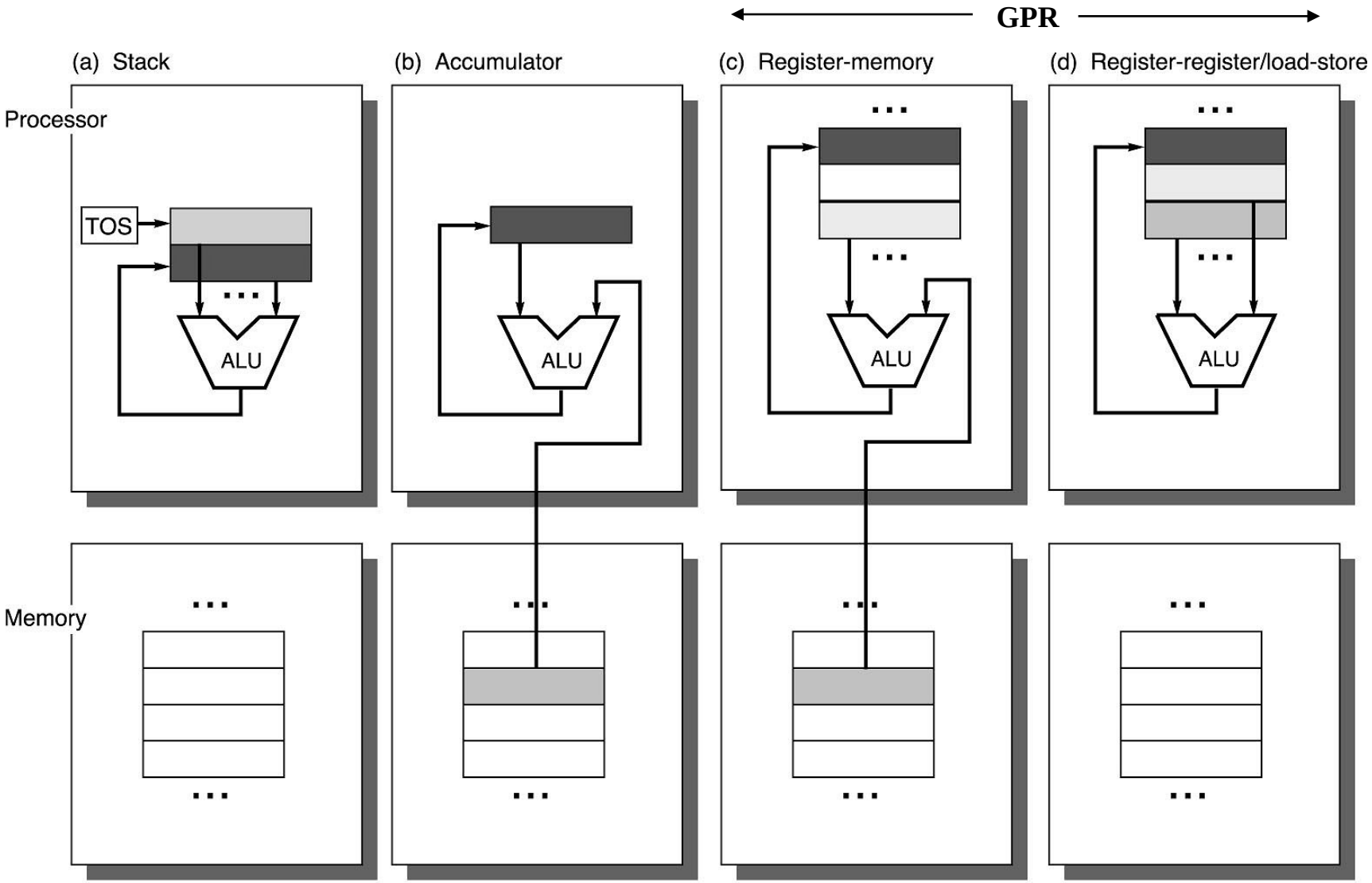
where FE_i is the fraction of time that enhancement i can be used and SE_i is the speedup of enhancement i. For a single enhancement the equation reduces to the familiar form of Amdahl's Law.

Instruction Set Architecture (ISA)



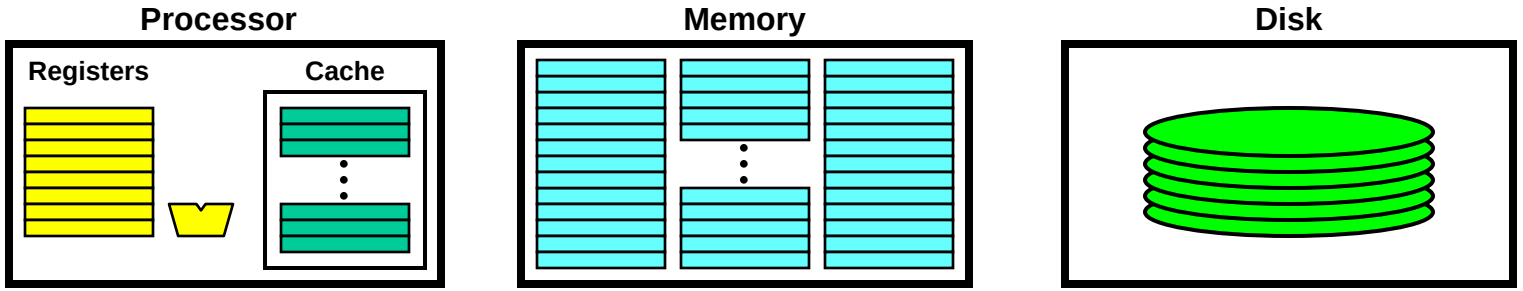
- Serve as an interface between software and hardware.
- Provides a mechanism by which the software tells the hardware what should be done.

Operand Locations in Four ISA Classes



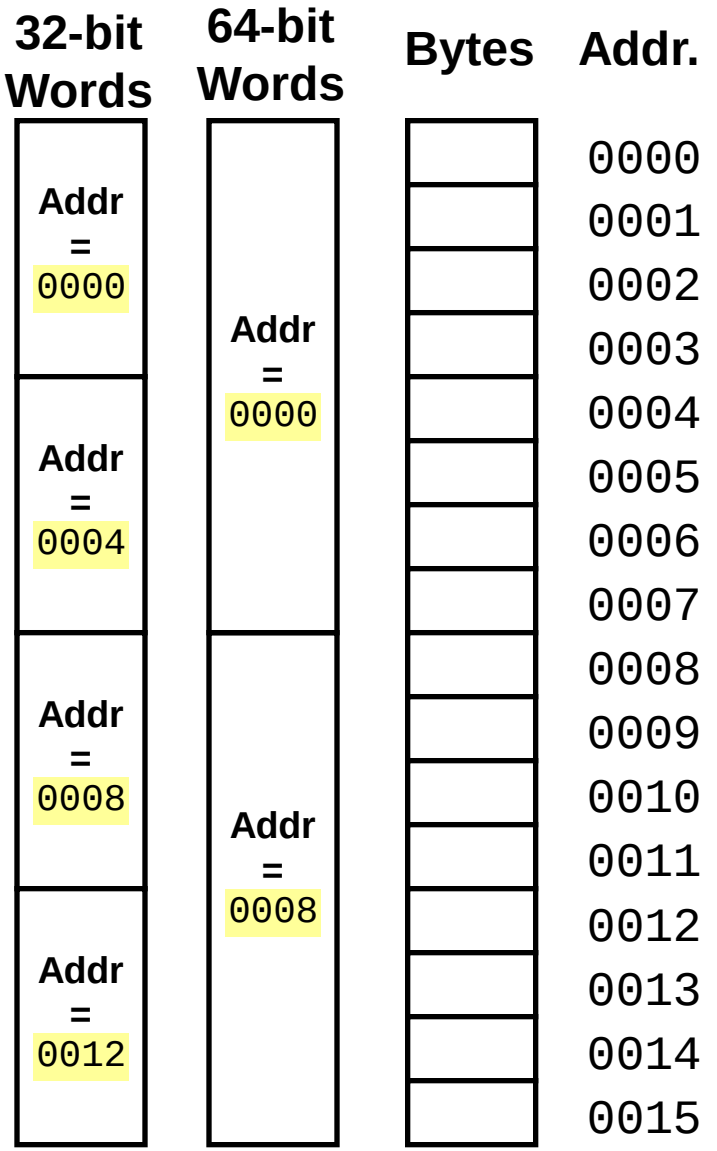
General Purpose Registers (GPR)

- Why GPRs Dominate?
  - Registers are much faster than memory (even cache)
    - Register values are available immediately
    - When memory isn't ready, processor must wait ("stall")
  - Registers are convenient for variable storage
    - Compiler assigns some variables just to registers
    - More compact code since small fields specify registers (compared to memory addresses)



Memory Addressing

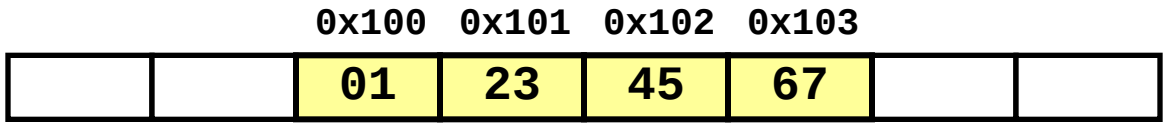
- Memory is byte addressed and provides access for bytes (8 bits), half words (16 bits), words (32 bits), and double words(64 bits).
- Addresses Specify Byte Locations
  - Address of the first byte in word
  - Successive word addresses differ by 4 (32-bit)



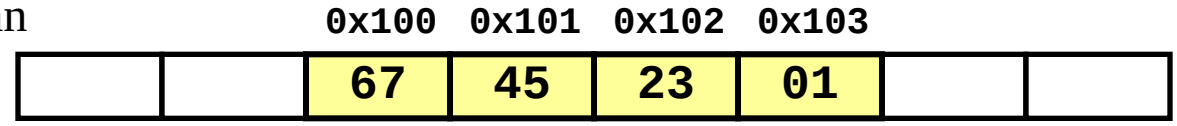
Addressing Objects: Endianess and Alignment

- Big Endian: address of most significant byte = word address (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: address of least significant byte = word address (xx00 = Little End of word)
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

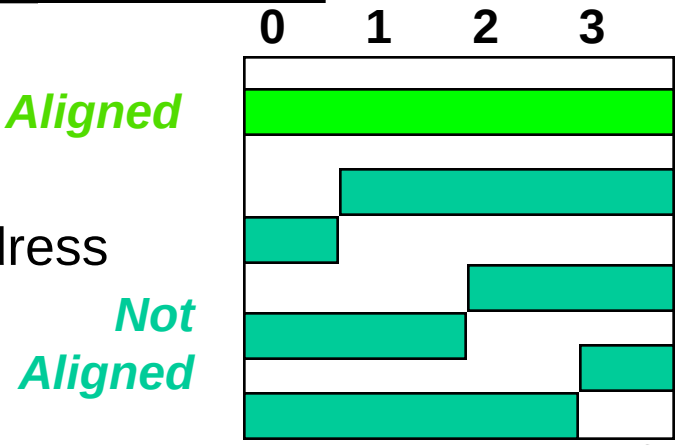
Big Endian



Little Endian



Alignment: require that objects fall on address that is multiple of their size.

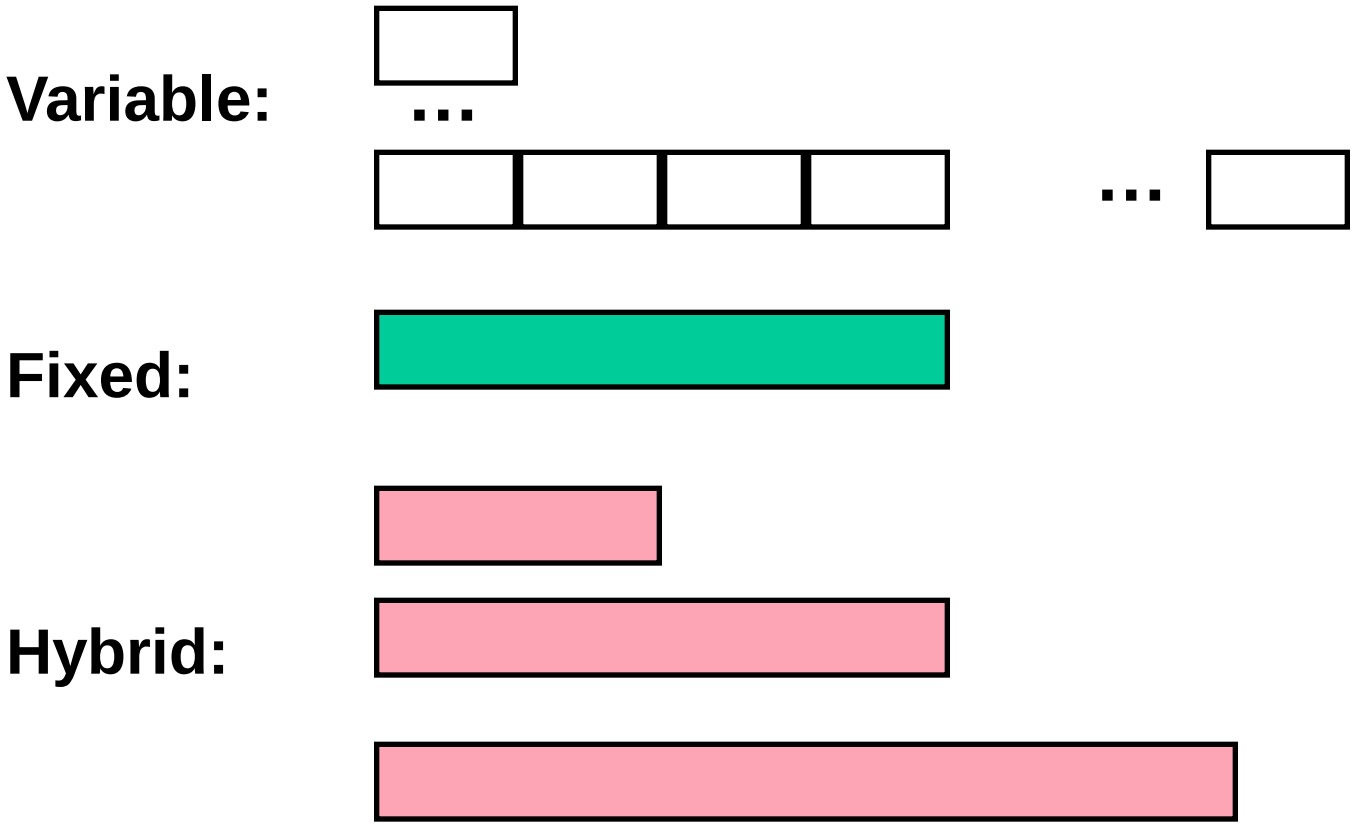


Types of Addressing Modes (VAX)

	Addressing Mode	Example	Action
1.	Register direct	Add R4, R3	R4 <- R4 + R3
2.	Immediate	Add R4, #3	R4 <- R4 + 3
3.	Displacement	Add R4, 100(R1)	R4 <- R4 + M[100 + R1]
4.	Register indirect	Add R4, (R1)	R4 <- R4 + M[R1]
5.	Indexed	Add R4, (R1 + R2)	R4 <- R4 + M[R1 + R2]
6.	Direct	Add R4, (1000)	R4 <- R4 + M[1000]
7.	Memory Indirect	Add R4, @(R3)	R4 <- R4 + M[M[R3]]
8.	Autoincrement	Add R4, (R2)+	R4 <- R4 + M[R2] R2 <- R2 + d
9.	Autodecrement	Add R4, (R2)-	R4 <- R4 + M[R2] R2 <- R2 - d
10.	Scaled	Add R4, 100(R2)[R3]	R4 <- R4 + M[100 + R2 + R3*d]

- Studies by [Clark and Emer] indicate that modes 1-4 account for 93% of all operands on the VAX.

Generic Examples of Instruction Formats



Instruction Formats

- If code size is most important, use variable length instructions:
  (1)Difficult control design to compute next address
  (2) complex operations, so use microprogramming
  (3) Slow due to several memory accesses

- If performance is most important, use fixed length instructions
  (1) Simple to decode, so use hardware
  (2) Works well with pipelining
  (3) Wastes code space because of simple operations

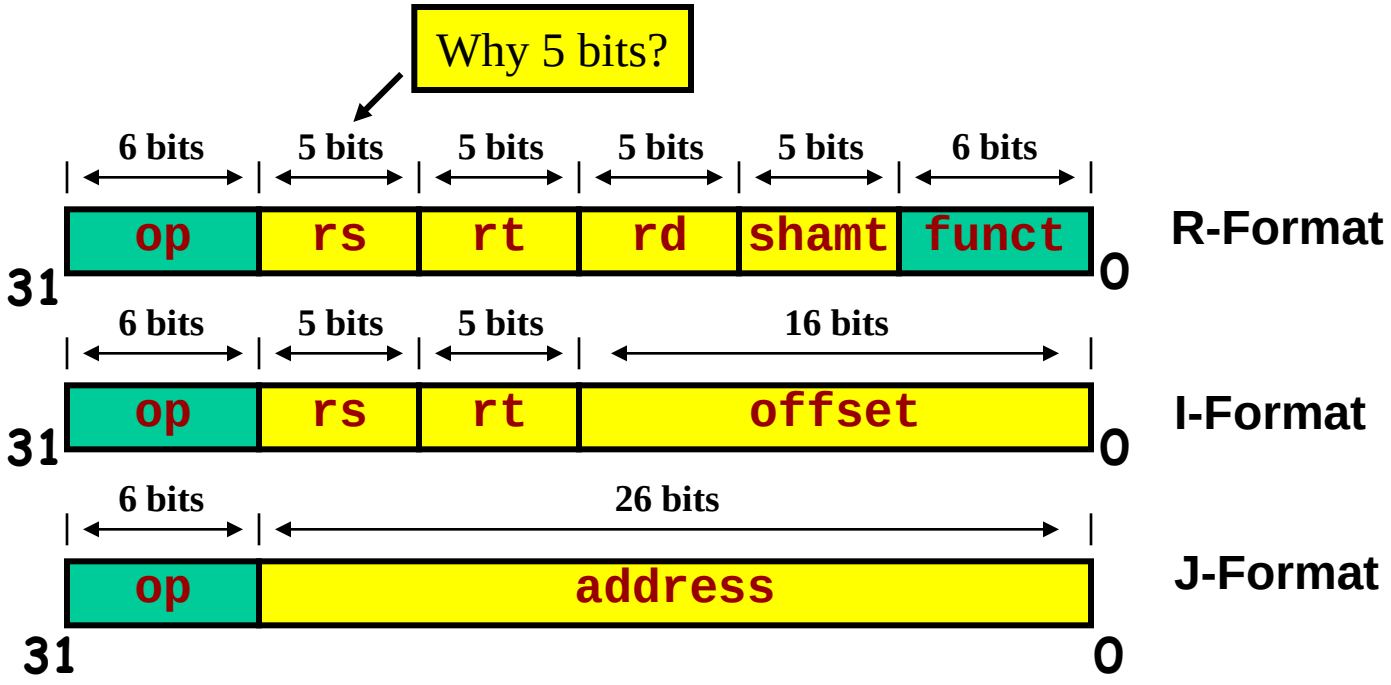
- Hybrid: Recent embedded machines (ARM 32-bit, MIPS) added an optional mode to execute subset of 16-bit wide instructions (Thumb, MIPS16); per procedure decide performance or density

MIPS Design Principles

1. Simplicity Favors Regularity
   - Keep all instructions a single size
   - Always require three register operands in arithmetic instructions
2. Smaller is Faster
   - Has only 32 registers rater than many more
3. Good Design Makes Good Compromises
   - Comprise between providing larger addresses and constants instruction and keeping instruction the same length
4. Make the Common Case Fast
   - PC-relative addressing for conditional branches
   - Immediate addressing for constant operands

MIPS Instructions

- All instructions exactly 32 bits wide
- Different formats for different purposes
- Similarities in formats ease implementation



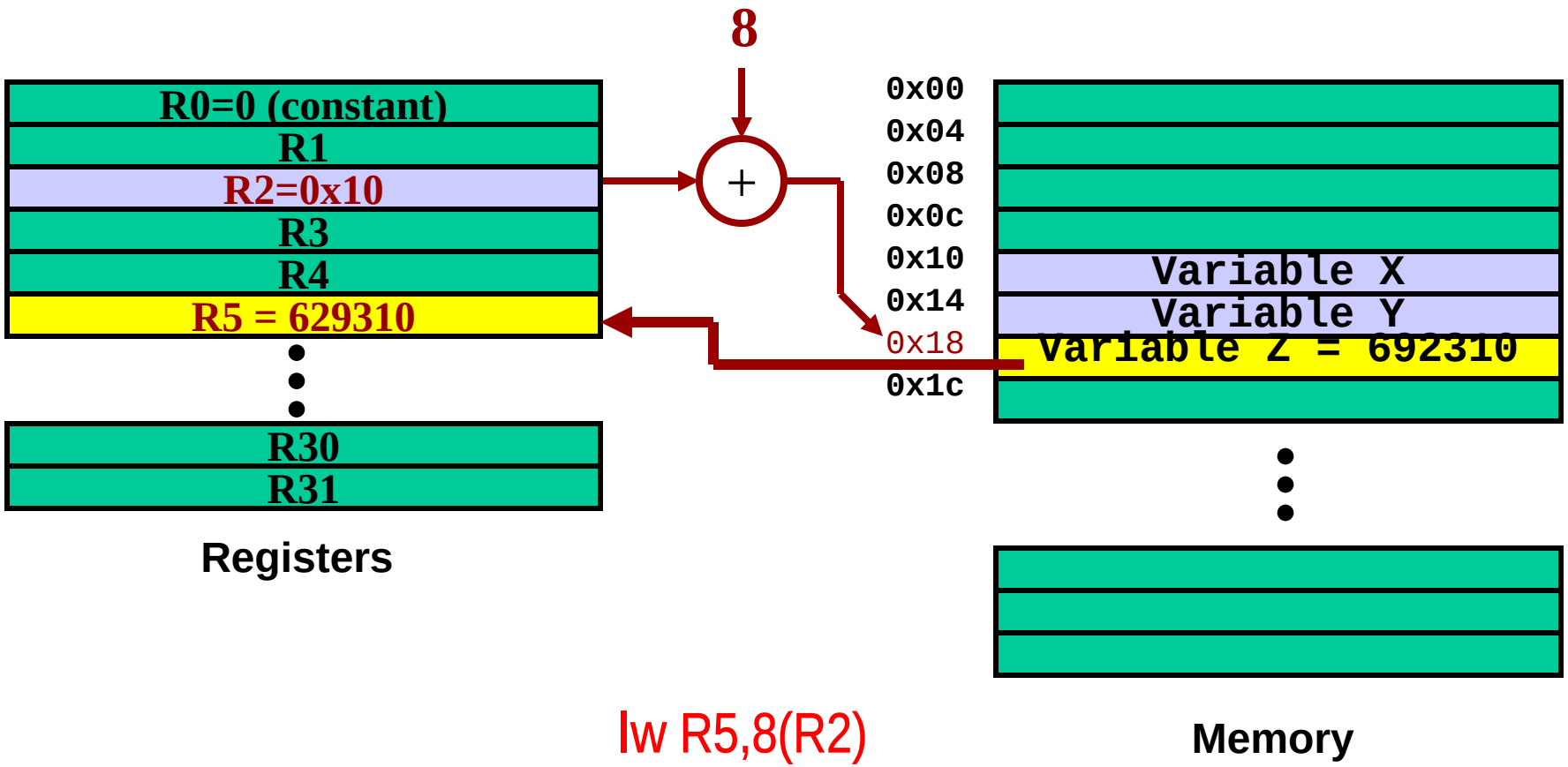
MIPS Data Transfer Instructions

- Transfer data between registers and memory
- Instruction format (assembly)

  lw $dest, offset($addr) load word
  sw $src, offset($addr) store word

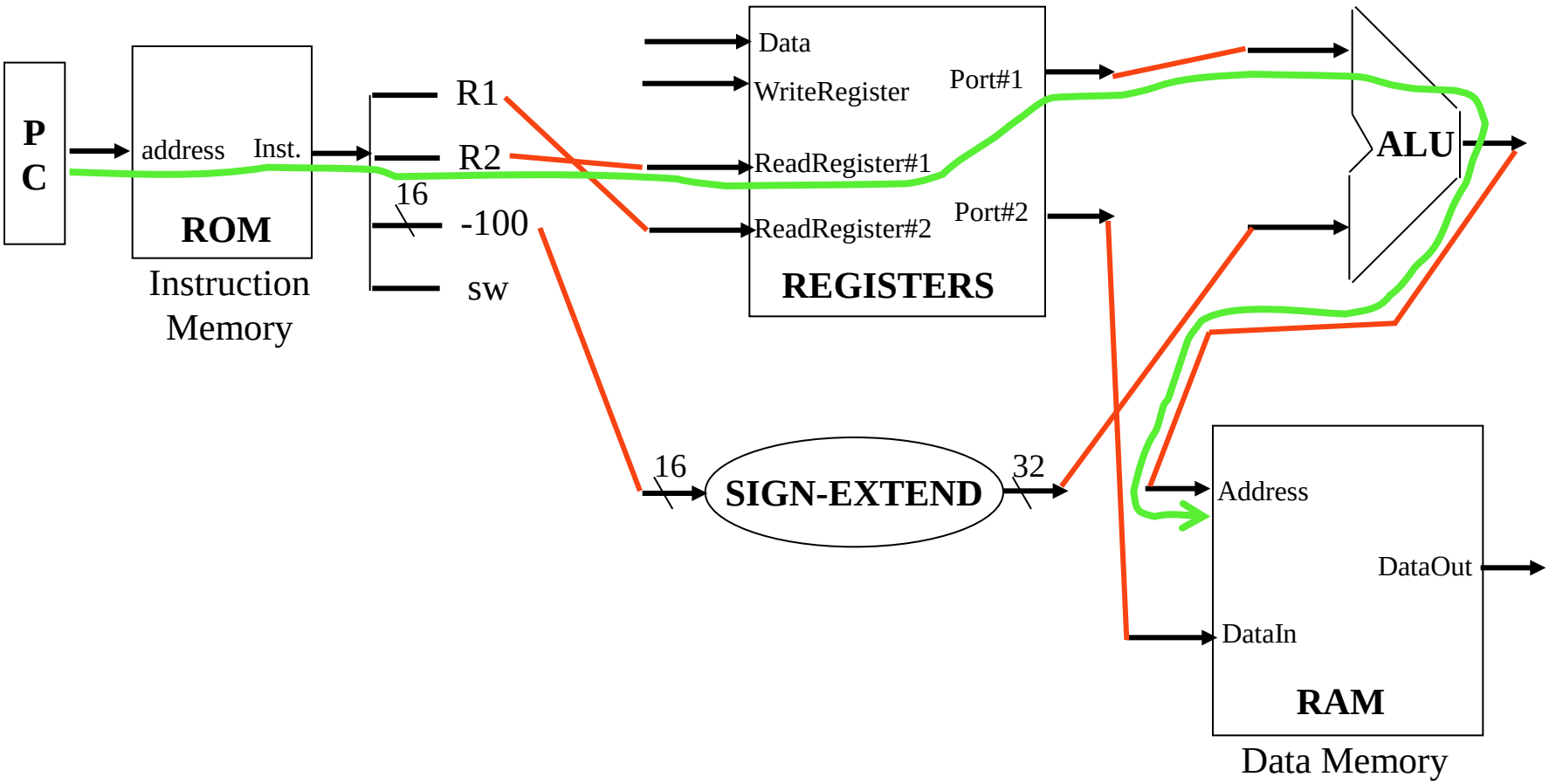
- Uses:
  - Accessing a variable in main memory
  - Accessing an array element

Example - Loading a Simple Variable

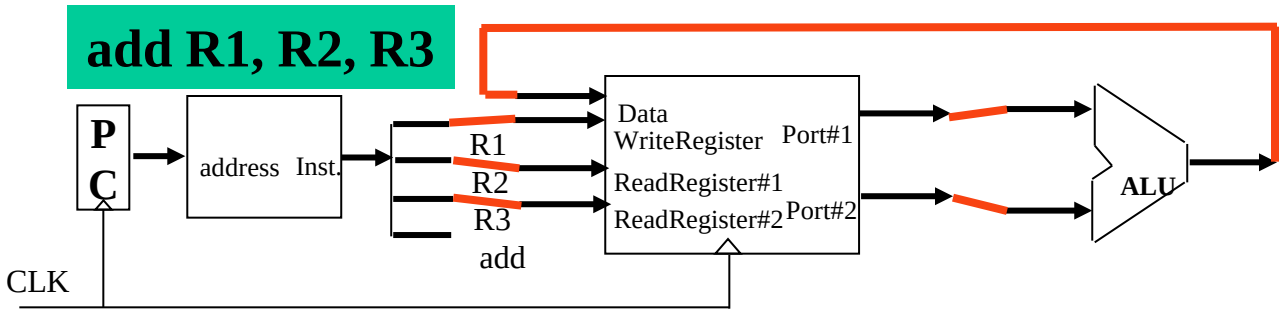


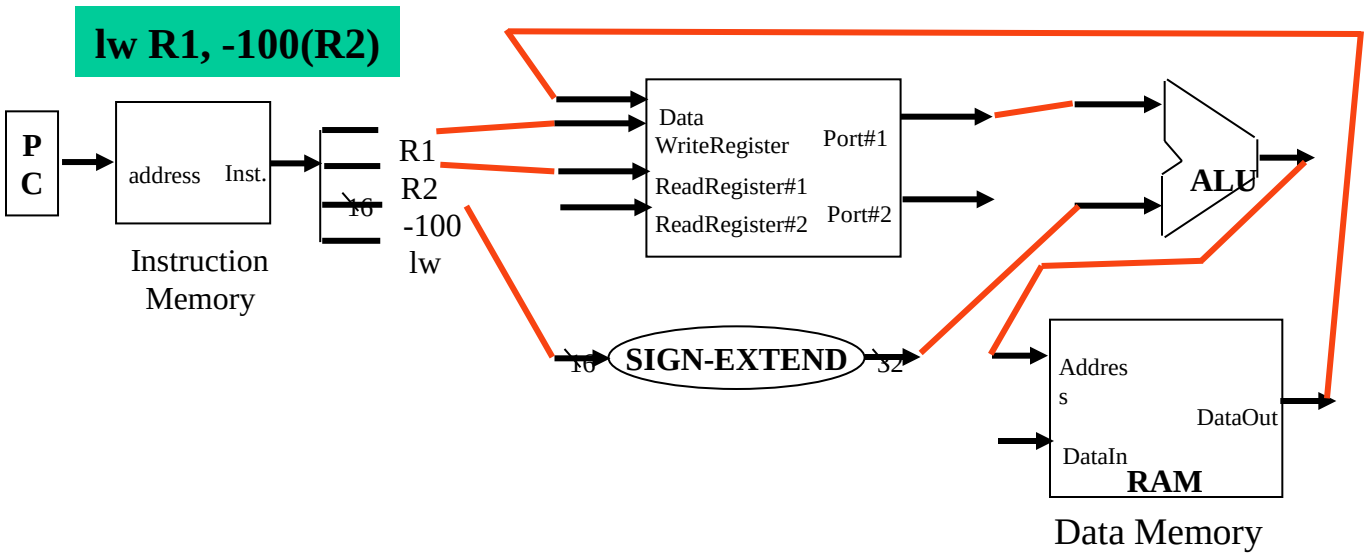
Critical Path for sw

sw R1, -100(R2)

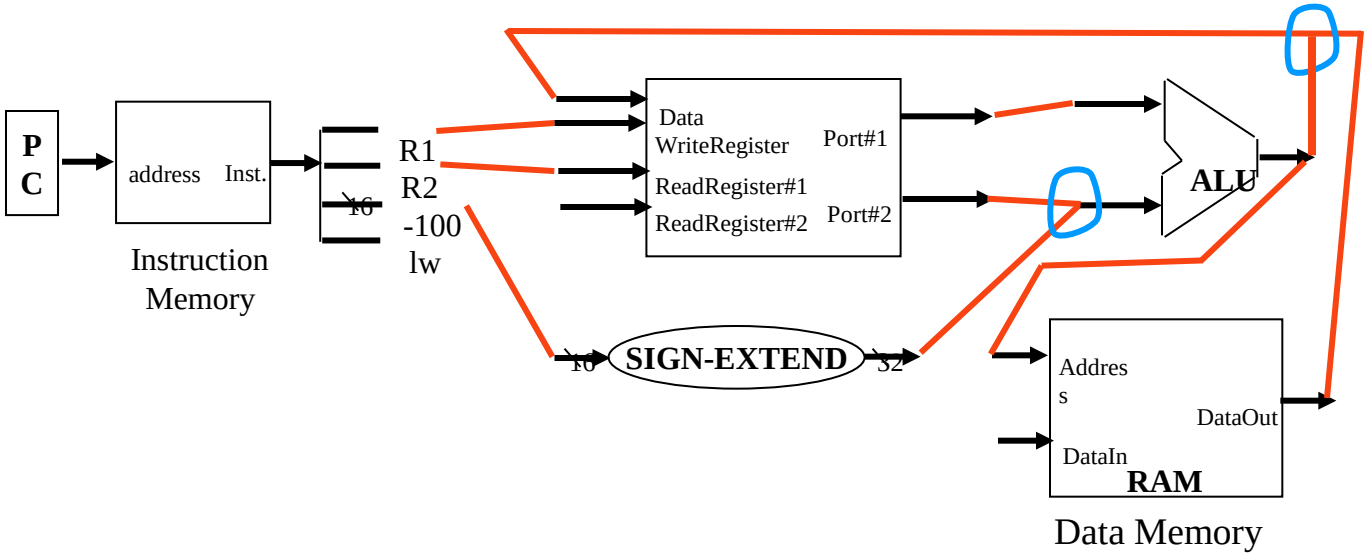


Datapath Connections for MIPS add and lw

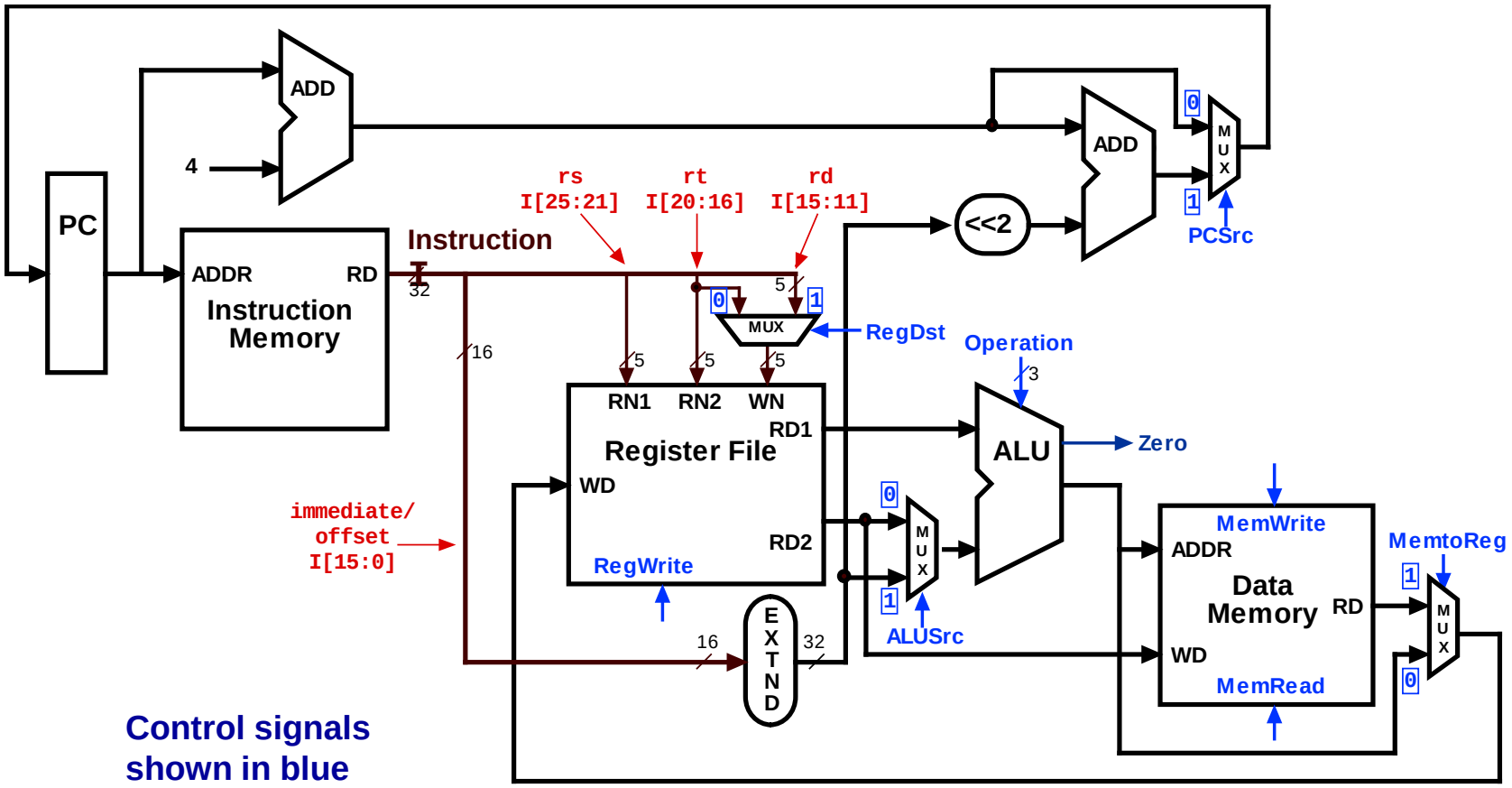




Datapath Connections for MIPS add and lw

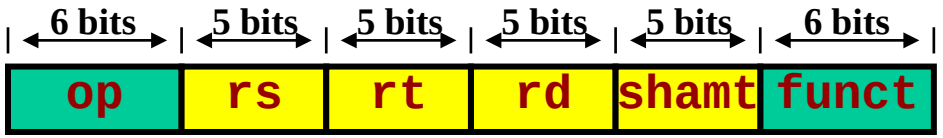


Complete Single-Cycle Datapath



Control Unit Design

- Desired function:
  - Given an instruction word….
  - Generate control signals needed to execute instruction
- Implemented as a *combinational* logic function:
  - Inputs



R-Format

    - Instruction word - op and funct fields
    - ALU status output - Zero
  - Outputs - processor control points
    - ALU control signals
    - Multiplexer control signals
    - Register File & memory control signal

Control Unit Structure

- Control unit as shown: one huge logic block
- Idea: decompose into smaller logic blocks
  - Smaller blocks can be faster
  - Smaller blocks are easier to work with
- Observation (rephrased):
  - The only control signal that depends on the `funct` field is the ALU Operation signal
  - Idea?: separate logic for ALU control

ALU Control: Truth Table

- Use don't care values to minimize length
  - Ignore F5, F4 (they are always "10")
  - Assume ALUOp never equals "11"

ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Operation
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Instr. type	Operation	funct	Desired Action	ALU Ctl.	ALUOp
data transfer	lw	XXXXXX	add	010	00
data transfer	sw	XXXXXX	add	010	00
branch	beq	XXXXXX	subtract	110	01
r-type	add	100000	add	010	10
r-type	sub	100010	subtract	110	10
r-type	and	100100	and	000	10
r-type	or	100101	or	001	10
r-type	slt	101010	set on less than	111	10

Alternatives to Single-Cycle

- Multicycle Processor Implementation
  - Shorter clock cycle
  - Multiple clock cycles per instruction
  - Some instructions take more cycles then others
  - Less hardware required
- Pipelined Implementation
  - Overlap execution of instructions
  - Try to get short cycle times and low CPI
  - More hardware required … but also more performance!

Multicycle Approach

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Our control signals will not be determined directly by instruction
  - e.g., what should the ALU do for a “subtract” instruction?
- We’ll use a finite state machine for control

Idea behind multicycle approach

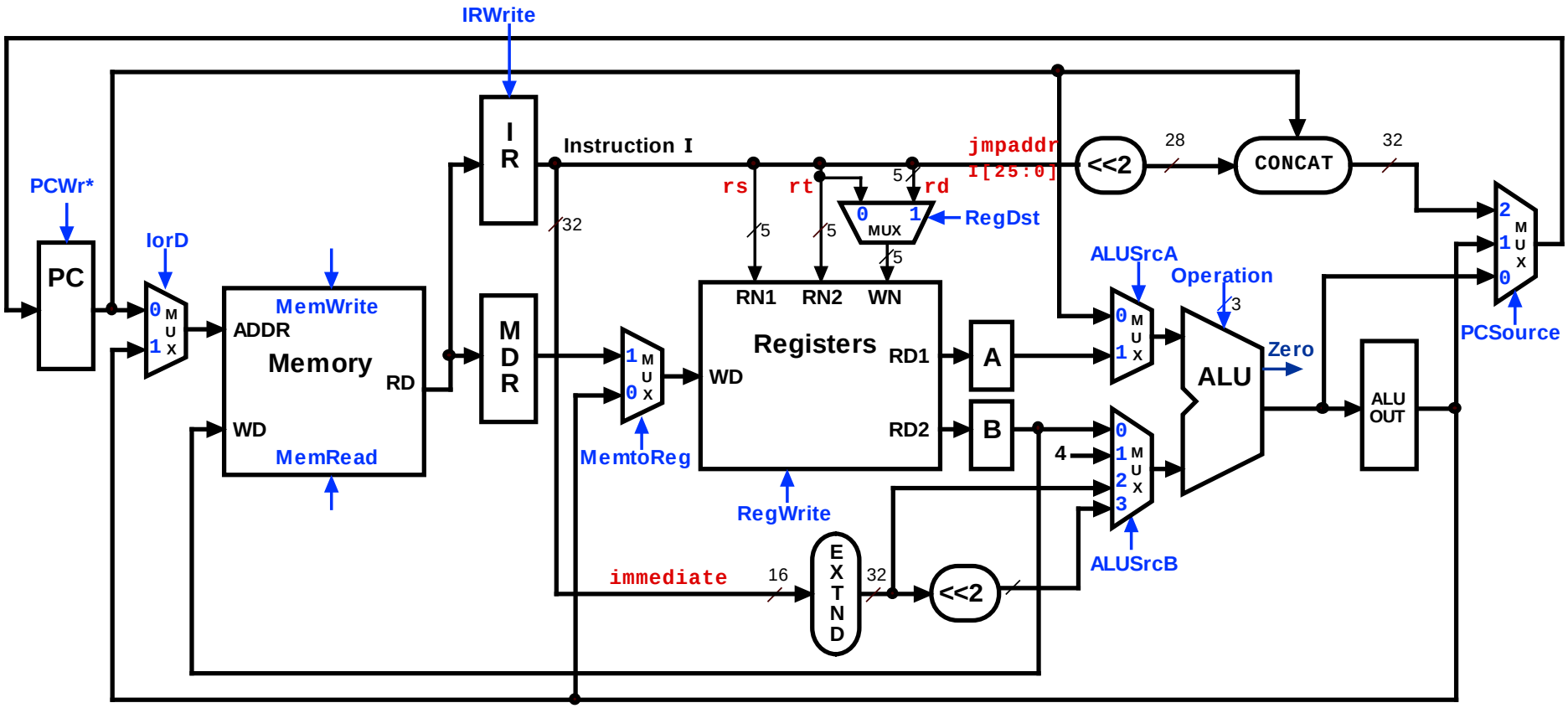
- We define each instruction from the ISA perspective (do this!)
- Break it down into steps following our rule that data flows through at most one major functional unit (e.g., balance work across steps)
- Introduce new registers as needed (e.g, A, B, ALUOut, MDR, etc.)
- Finally try and pack as much work into each step
  (avoid unnecessary cycles)
  while also trying to share steps where possible
  (minimizes control, helps to simplify solution)

Summary:

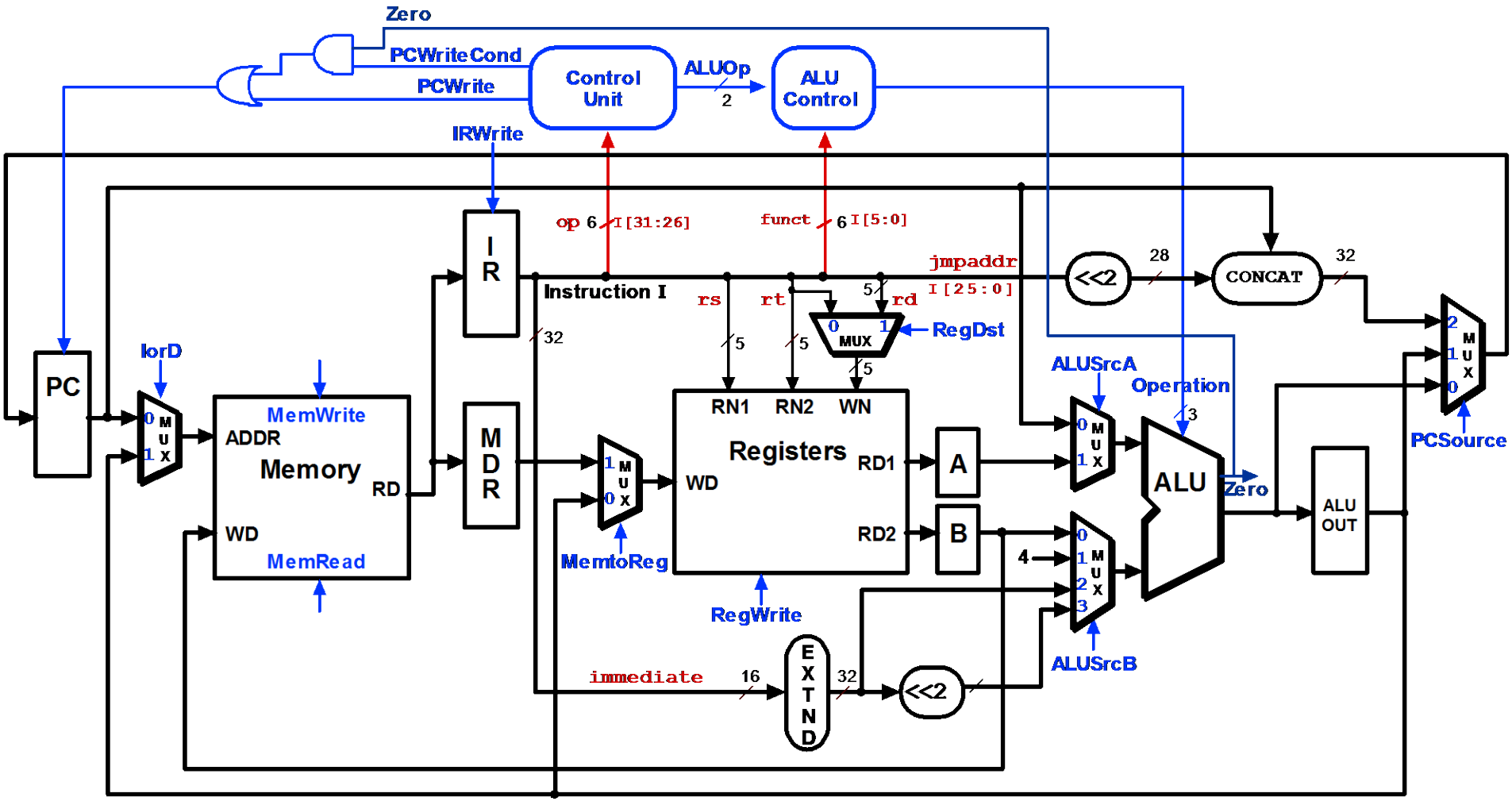
Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	If (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

Full Multicycle Datapath



Full Multicycle Implementation



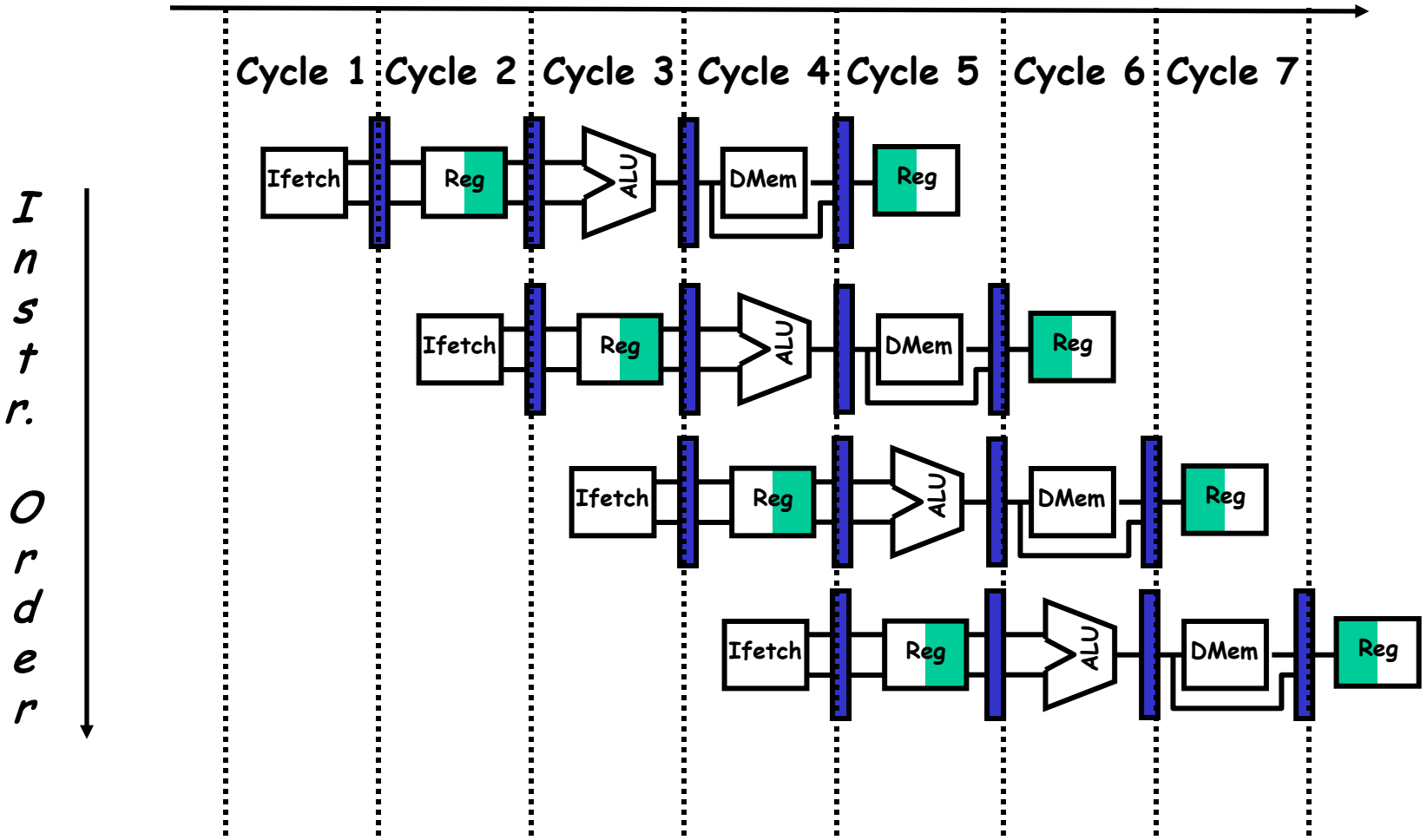
What is Pipelining?

- A way of speeding up execution of instructions

- *Key idea*:

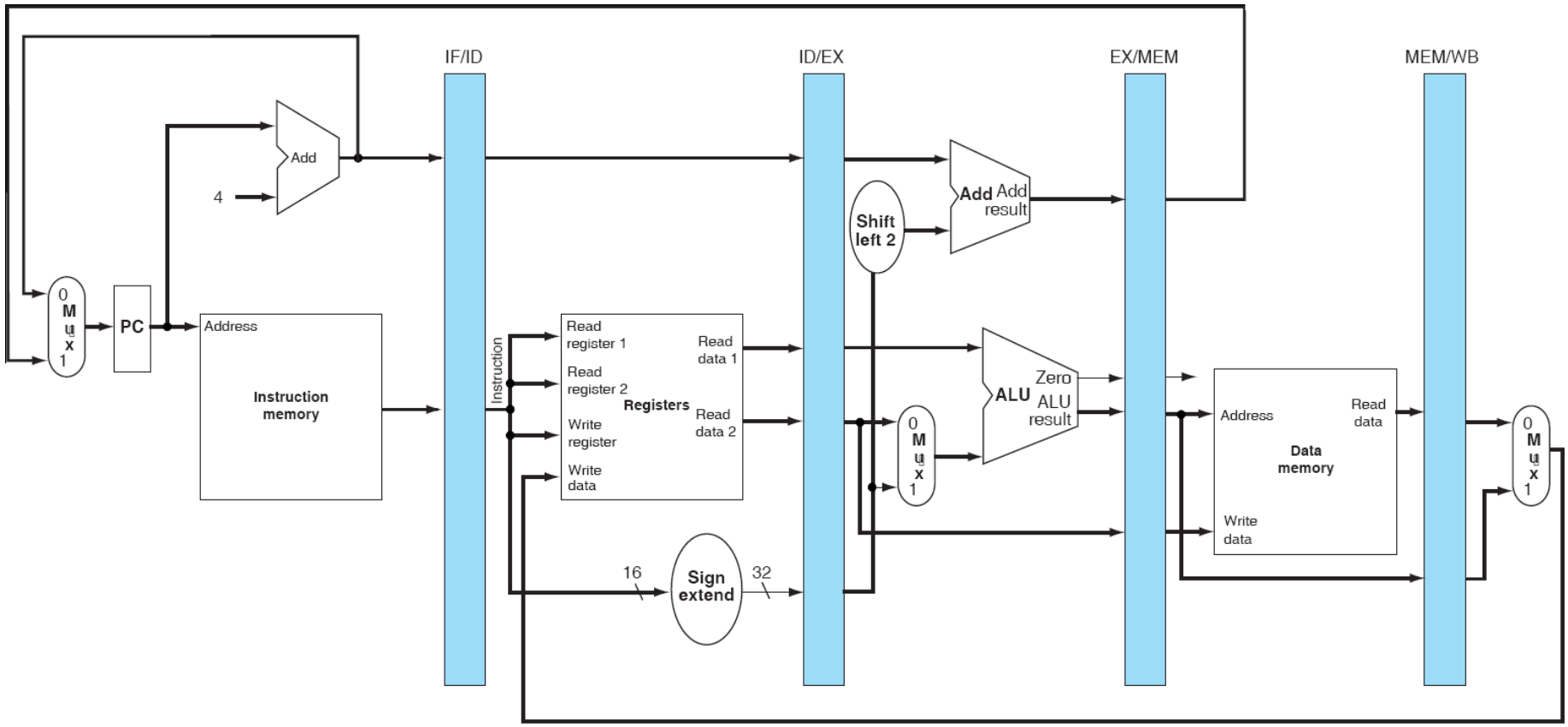
overlap execution of multiple instructions

The Basic Pipeline For MIPS

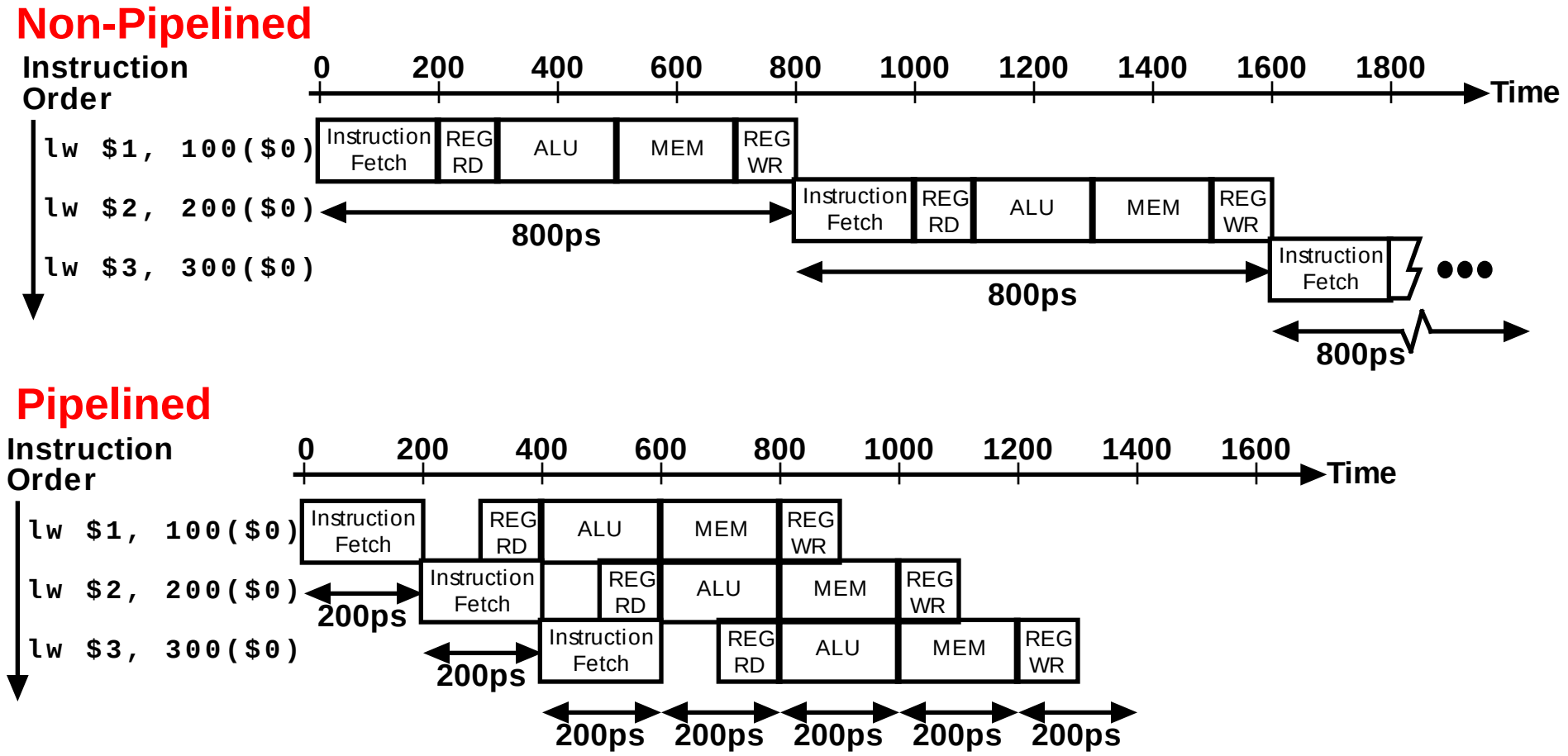


What do we need to add to actually split the datapath into stages?

Basic Pipelined Processor



Single-Cycle vs. Pipelined Execution



Pipeline Hazards

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards**:** two different instructions use same h/w in same cycle
  - Data hazards**:** Instruction depends on result of prior instruction still in the pipeline
  - Control hazards**:** Pipelining of branches & other instructions that change the PC

Example

Consider the following MIPS code fragments, each containing two instructions. For each code fragment identify the type of hazard that exists between the two instructions and the **registers involved**.

LW R4, 0(R2)

ADD R3, R4, R2

RAW: add requires the value of R4 returned by lw

Structural Hazards

- Attempt to use same resource twice at same time
- Example: Single Memory for instructions, data
  - Accessed by IF stage
  - Accessed at same time by MEM stage
- Solutions ?
  - Delay second access by one clock cycle
  - Provide separate memories for instructions, data
    - This is what the book does
    - This is called a "Harvard Architecture"
    - Real pipelined processors have separate caches

Dealing with Structural Hazards

Stall

- low cost, simple
- Increases CPI
- use for rare case since stalling has performance effect

Pipeline hardware resource

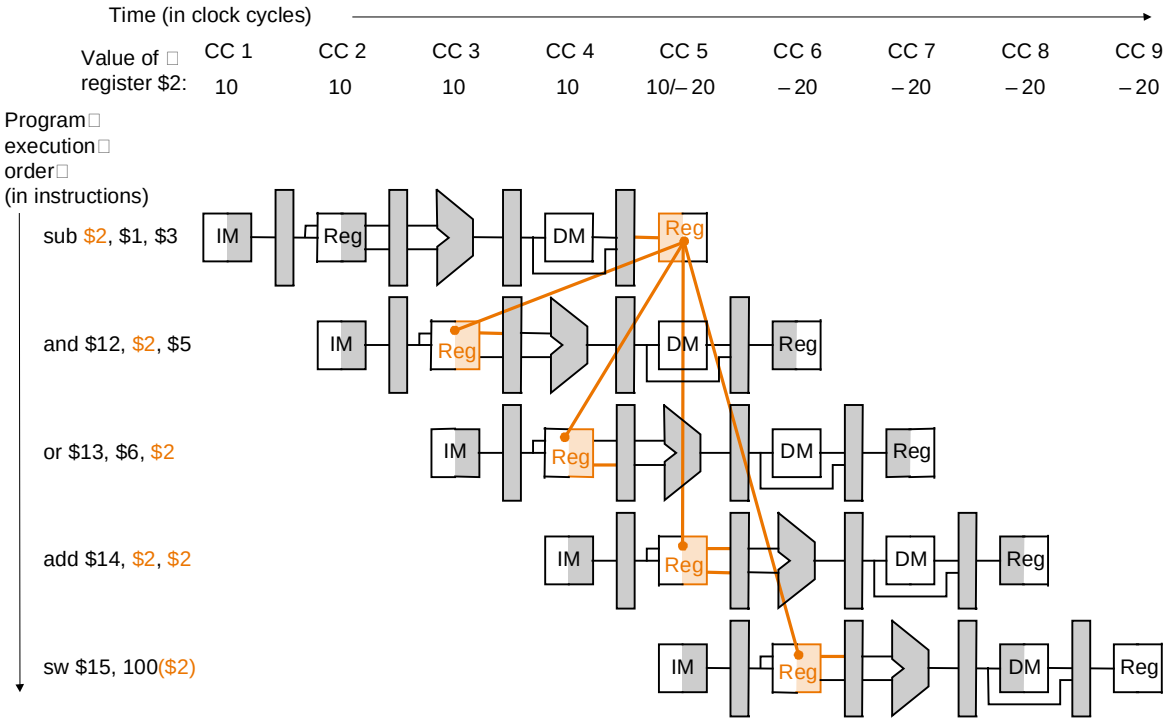
- useful for multi-cycle resources
- good performance
- sometimes complex e.g., RAM

Replicate resource

- good performance
- increases cost (+ maybe interconnect delay)
- useful for cheap or divisible resources

Data Hazards

- Data hazards occur when data is used before it is stored



The use of the result of the SUB instruction in the next three instructions causes a data hazard, since the register is not written until after those instructions read it.

Data Hazards

- Solutions for Data Hazards
  - Stalling
  - Forwarding:
    - connect new value directly to next stage
  - Reordering

Sample Question

For each code fragment identify the type of hazard that exists between the two instructions and the **registers involved**.

LW R1, 0(R2)

ADD R3, R1, R2

RAW: add requires the value of R1 returned by lw

Control Hazards

A *control hazard* is when we need to find the destination of a branch, and can't fetch any new instructions until we know that destination.

A branch is either

- Taken: PC <= PC + 4 + Imm
- Not Taken: PC <= PC + 4

Control Hazard Solutions

- Stall
  - stop loading instructions until result is available
- Predict
  - assume an outcome and continue fetching (undo if prediction is wrong)
  - lose cycles only on *mis-predict*
- Delayed branch
  - specify in architecture that following instruction is always executed

Example

Instruction Number	Clock Number									
	1	2	3	4	5	6	7	8	9	10
Load Instruction	IF	ID	EX	MEM	WB					
Instruction i+1		IF	ID	EX	MEM	WB				
Instruction i+2			IF	ID	EX	MEM	WB			
Instruction i+3				Stall	IF	ID	EX	MEM	WB	
Instruction i+4						IF	ID	EX	MEM	WB
Instruction i+5							IF	ID	EX	MEM
Instruction i+6								IF	ID	EX

Sample Question

- True or False

1. Moore's Law gives a quick way to find the speedup from some enhancement.
2. Hazards prevent next instruction from executing during its designated clock cycle.

Multiple Choices

- Which statement about pipelining is false?
  - Multiple instructions are being processed at same time
  - Best case speedup of N, which is the number of stages
  - Instructions never interfere with each other
  - Stages are isolated by registers

C

Amdahl's Law

- Suppose a program runs in 100 seconds on a computer, with multiply operations responsible for 80 seconds of this time. How much do you have to improve the speed of multiplication if you want your program to run 2 times faster?

N = 8/3