CPEN 411: Computer Architecture

Slide Set #4: Pipelining

Instructor: Mieszko Lis

Original Slides: Professor Tor Aamodt

Slide background: Die photo of the Intel 486 (first pipelined x86 processor)

Intel 486

Intel began the i486 processor development program shortly after it introduced the 386 processor in 1985. From initial concept, the chip design team worked with the following CPU goals:

1) ensure binary compatibility with the 386 microprocessor and the 387 math coprocessor,

2) increase performance by two to three times over a 386/387 processor system at the same clock rate, and

3) extend the IBM PC standard architecture of the 386 CPU with features suitable for minicomputers.

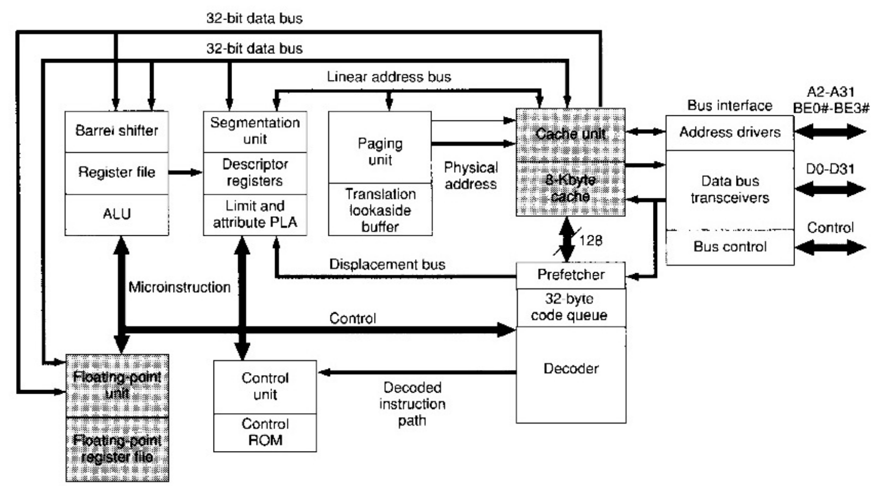


Figure 1. Block diagram of the i486 processor.

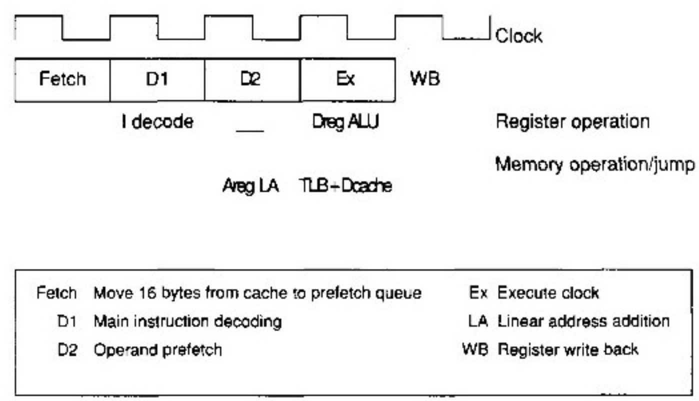


Figure 3. The 486 CPU pipeline.

John H. Crawford, “The i486 CPU: Executing Instructions in One Clock Cycle”, IEEE Micro, Feb. 1990

Introduction to Slide Set 4

In the last slide set we learned about how instruction set architectures are designed.

In this slide set, we start looking at how to implement hardware for the instruction set architecture (ISA).

We will design a processor at the "microarchitecture level". Every ISA can be implemented in many ways representing different tradeoffs in performance, cost, power, etc… (e.g., Intel Atom and Intel Core i7, both implement "x86")

We start by looking at how to translate an ISA specification into hardware. Then, we learn about a very important microarchitecture optimization called pipelining.

Learning Objectives

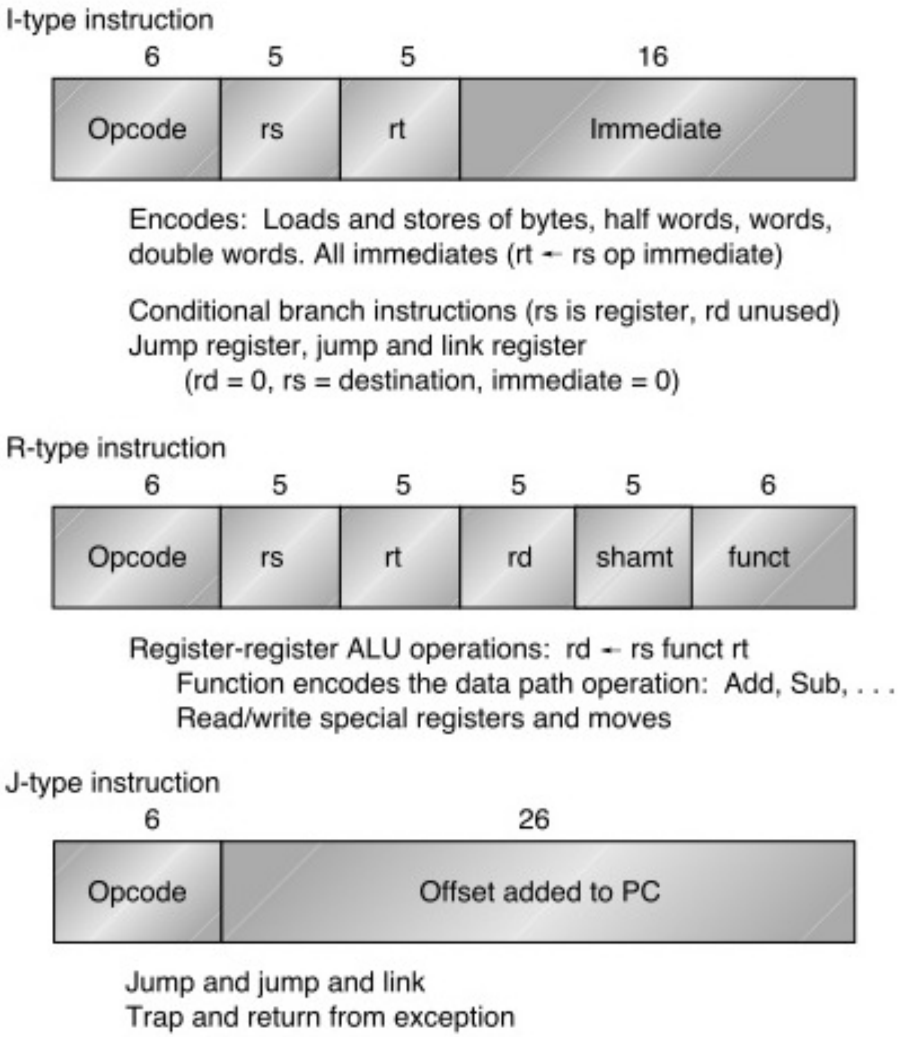
- By the time we finish discussing these slides, you should be able to:
  - Describe the five basic steps of instruction processing
  - Describe the components of a simple single-cycle processor implementation and how these components interact and also analyze its performance
  - Describe a simple multicycle processor and analyze it's performance
  - Define pipelining and explain it using a simple analogy
  - Describe arithmetic pipelines and the limitations of pipelining
  - Explain the simple five stage pipeline
  - Describe pipelined control signals and explain their purpose
  - Define the term hazard in the context of computer architecture; list and explain three important types of hazards in instruction pipelines
  - Describe the fundamental approaches used to overcome hazards

Implementing MIPS64

After defining an instruction set architecture we want to design an implementation of that processor.

Rather than start by designing a super-duper optimized microprocessor that implements MIPS64, let's first consider how to define a very simple processor. This processor is not very fast, but it will run MIPS64 programs correctly.

Recall: MIPS64 Encoding



- These instruction encodings help simplify the hardware implementation.
- One reason is that the opcode is always in the same place regardless of the instruction format.
- Another reason is that the width of each instruction is the same.
- The "register specifiers" (rs,rt,rd) are at a known position regardless of which instruction the opcode specifies.

A Simple RISC Implementation

1. Instruction fetch cycle (IF)
   - Send PC to memory and fetch the current instruction from memory. Update the PC to next sequential PC by adding 4 (NOTE: SimpleScalar adds 8—important in Assignment #3)

2. Instruction Decode/Register fetch cycle (ID)
   - decode instruction (what does it do?)
   - read source registers
   - compute branch target and condition

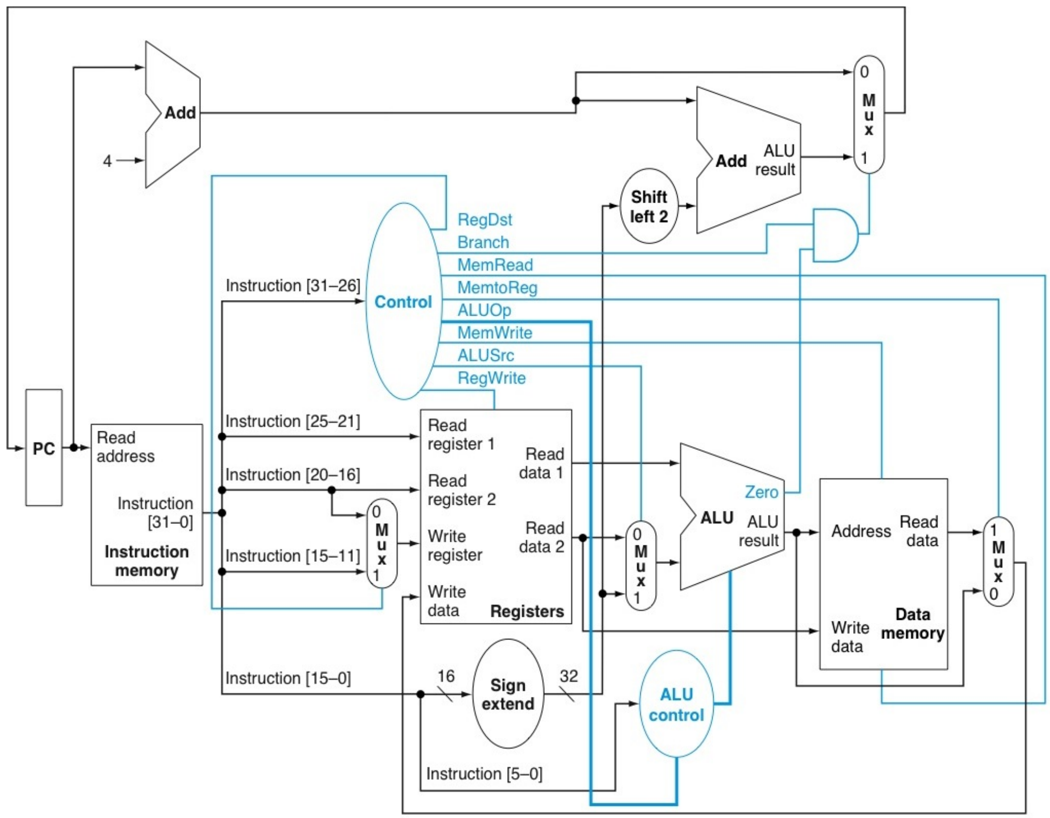
3. Execute/effective address cycle (EX)
   - memory reference: ALU computes effective address from base register and offset.
   - register-register ALU instruction: ALU performs operation
   - register-immediate ALU instruction: ALU performs operation

A Simple RISC Implementation

4. Memory access (MEM)
   - load: read memory @ effective address
   - store: write value from register to effective address

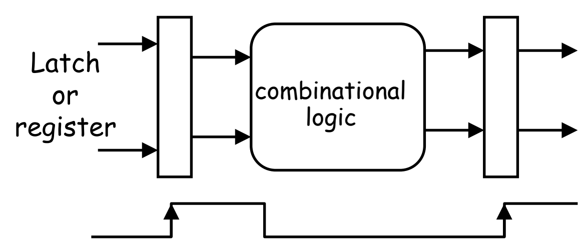
5. Write-back cycle (WB)
   - Register-Register/Register Immediate/Load: Write the result into the register file.

Simple Single Cycle MIPS64



- Start by looking at each component one at a time.
- This is just <u>**ONE**</u> way to build a processor that implements (most of) MIPS64.
- “How was this design created?” Next few slides we look at each component and see how it relates to the ISA (MIPS64) we want to implement.

Current Logical State of the Machine ⊢ **Next Logical State of the Machine**

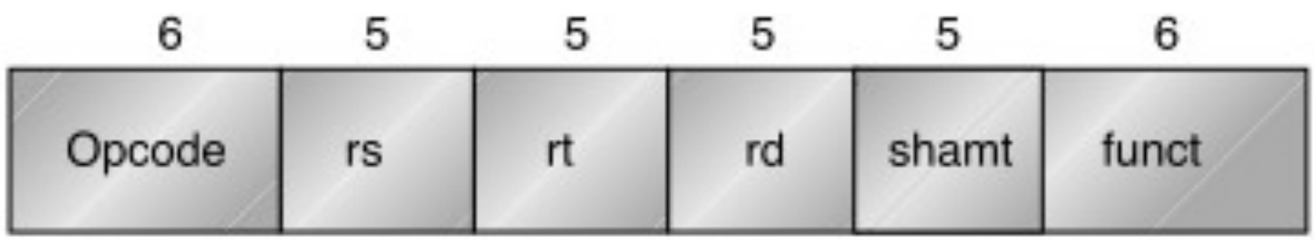


Instructions are 1’s and 0’s…

Example: The following “assembly code”

DADD R1,R2,R3

Is translated into “machine language” (1’s and 0’s) as:



000000 0010 0011 0001 00000 101100

Details of encoding is not important here…

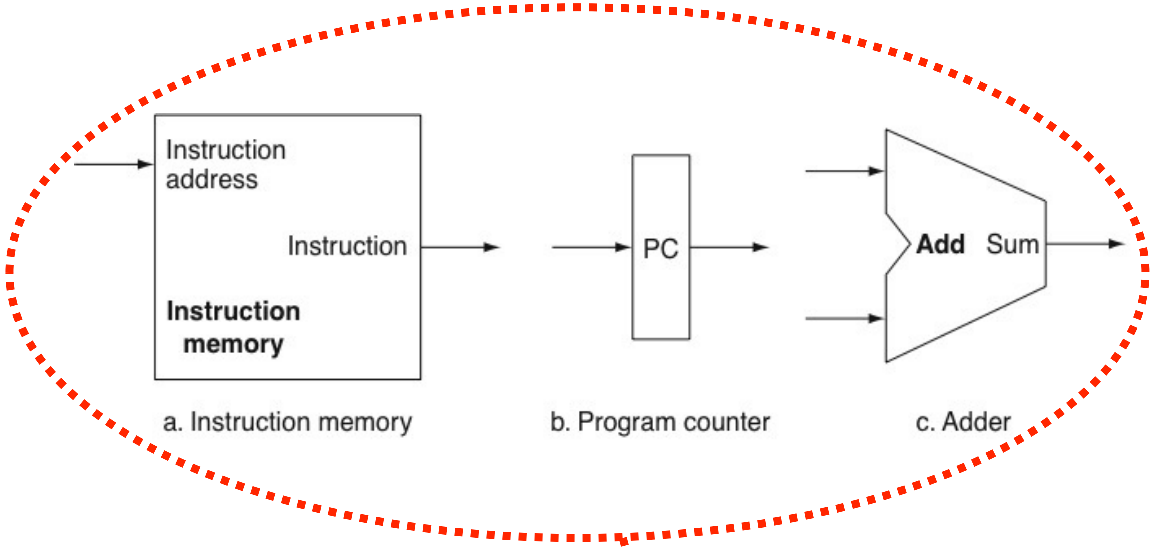
More important: Where are these 0’s and 1’s stored?

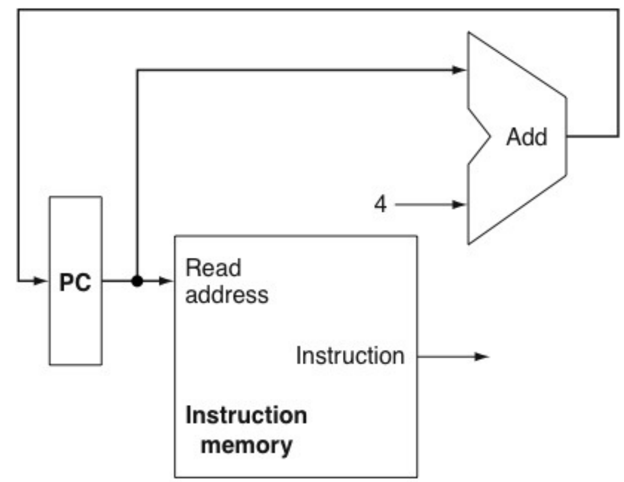
Instruction Fetch

- Before we can execute an instruction, we need to get the 1’s and 0’s that tell us what the instruction should do.
- The “machine language” instructions are stored in an instruction memory.
- The next instruction we should “execute” is pointed to by a program counter (PC).
- After executing an instruction we go to the next sequential instruction unless the instruction is a branch or jump.

Building a Datapath...

Instruction Fetch





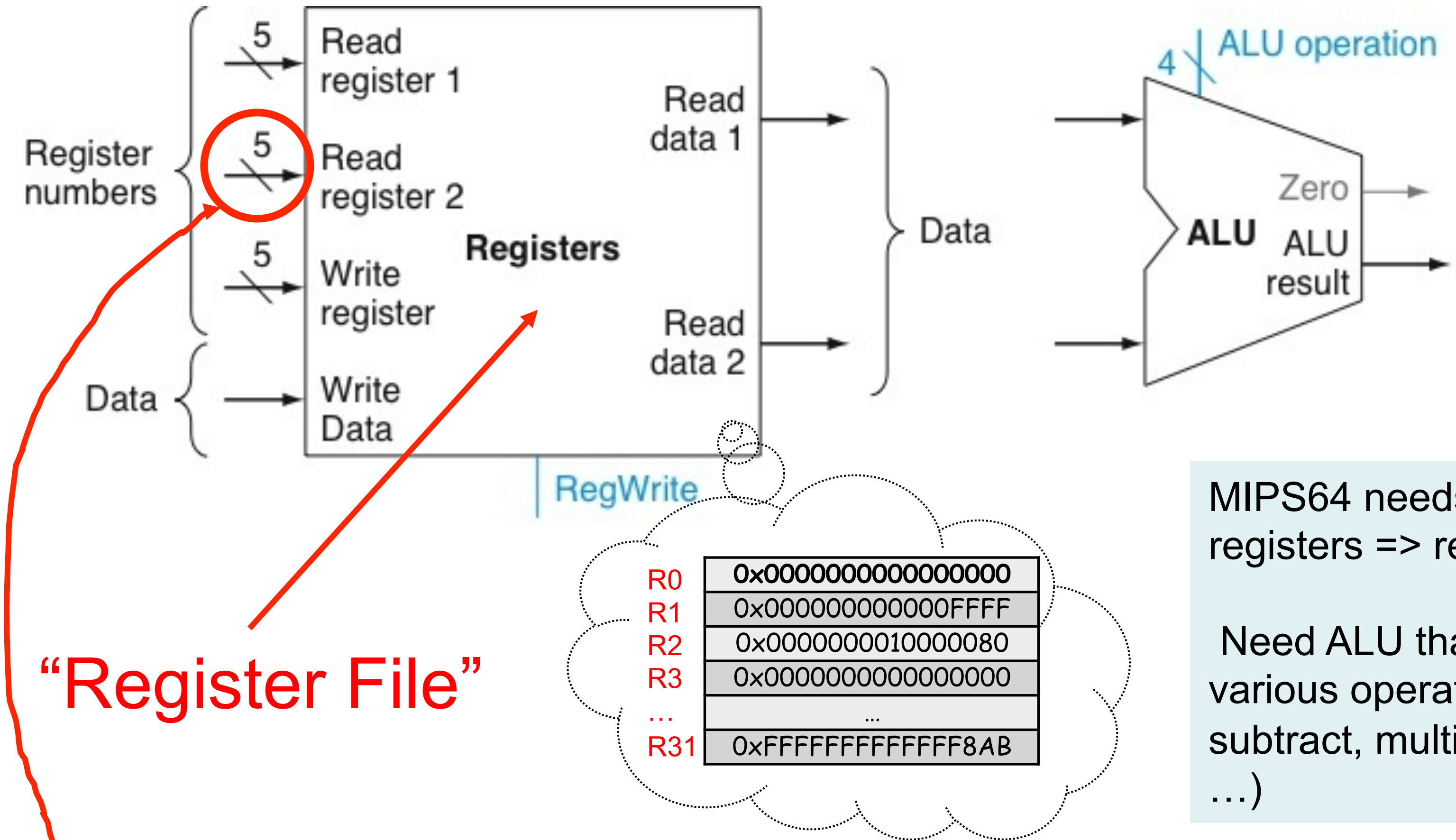
Send PC to memory and fetch the current instruction from memory. Update the PC to next sequential PC by adding 4.

Arithmetic Instructions

- Arithmetic instructions (e.g., DADD R1,R2,R3) read two register operands (R2, R3) and write to a third register operand (R1).
- We need somewhere to store the registers. We will place them in a “register file”.
- We also need some hardware to do the arithmetic. We can place hardware for various operations (addition, subtraction, multiplication, shift, etc…) inside one “block” which we will call an “arithmetic logic unit” (ALU)

Building a Datapath...

R-type ALU instructions



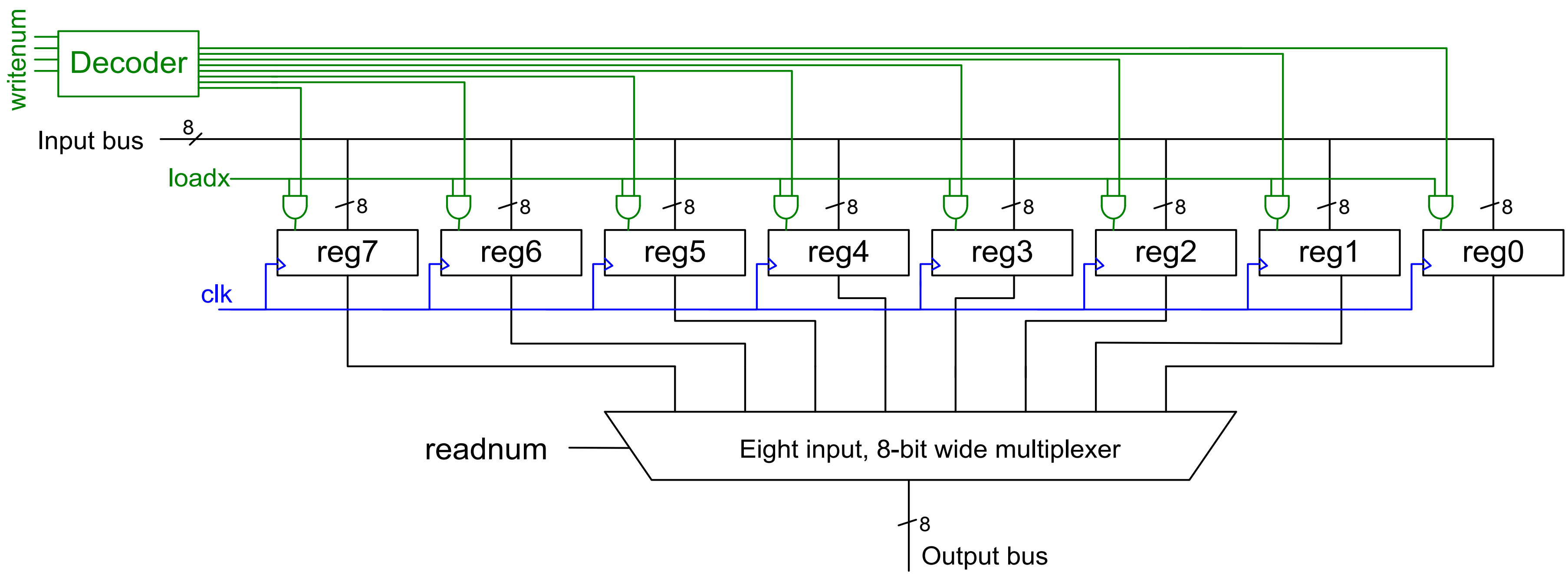
"Register File"

MIPS64 needs registers => register file.

Need ALU that supports various operations (add, subtract, multiple, and, or, xor, …)

5-bits encodes 2^5=32 register numbers ("specifiers"), each register contains 64-bits. Total number of registers in register file is 32 (total storage = 32x64 = 2048 bits).

Internal organization of Register File

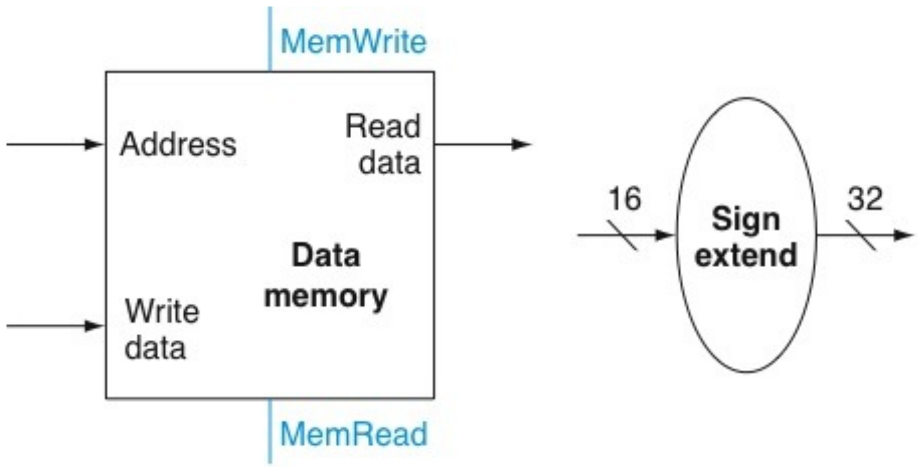


- In EECE 353, you designed this register file in VHDL.

Accessing Data in Memory

- In addition to registers, we need a larger memory space to hold data. We need to be able to both read values from this memory using load instructions, e.g., LD R1, 0(R2), and write values to it using store instructions, e.g., SD R1, 0(R2).
- For load and store instructions, MIPS64 supports displacement addressing with a 16-bit displacement. However, memory addresses are 64-bits.
- Also, the displacement value can be negative.
- Thus, we need to be able to "sign extend" the displacement value before the ALU operates on them.

Hardware for Loads and Stores



a. Data memory unit

b. Sign-extension unit

Example of sign extension:

-128 = 10000000 (8-bits)
-128 = 1111111110000000 (16-bits)

What about Branches/Jumps?

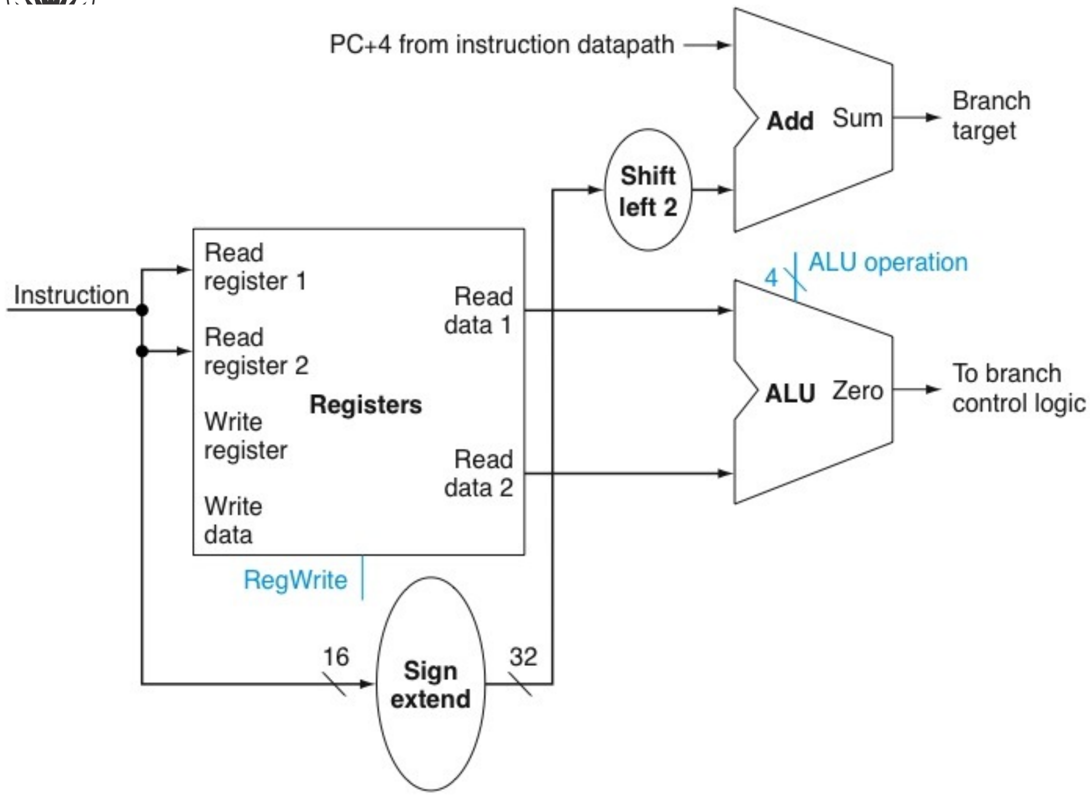
- Real programs have “if” statements, “switch” statements, function calls and returns, etc… How does hardware support them?

- For (conditional) branches, hardware must do two things:
  1. Determine <u>target</u> PC of branch. This is the instruction address to go to if the branch is “taken”. Target is encoded using PC-relative addressing. “Target PC” = “Branch PC” + 4 + Offset.

  2. Determine the <u>outcome</u> of branch--is the branch “taken”? A branch is “taken” if the condition was satisfied. Jump instruction are always “taken”.

- Outcome of branch is determined by reading values from register file and comparing them.

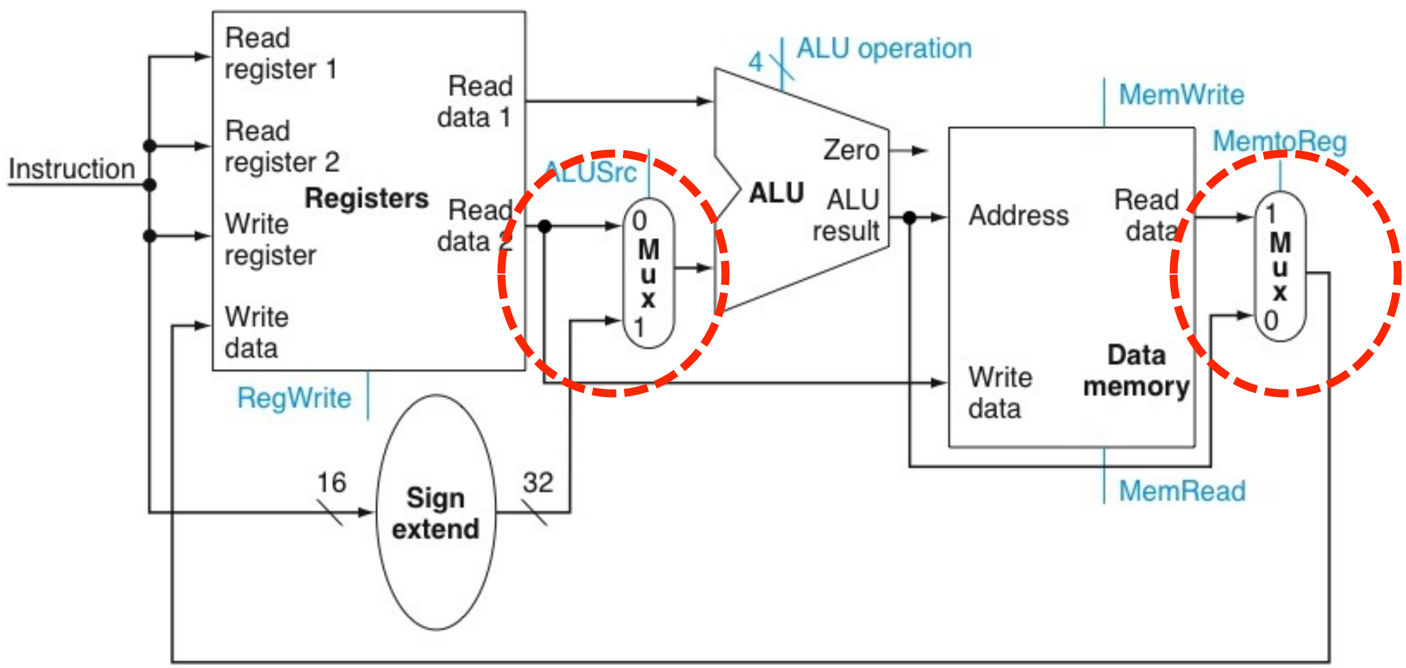
Datapath for Conditional Branches



Instructions are <u>aligned</u> in memory (start at an address which is a multiple of 4 bytes).

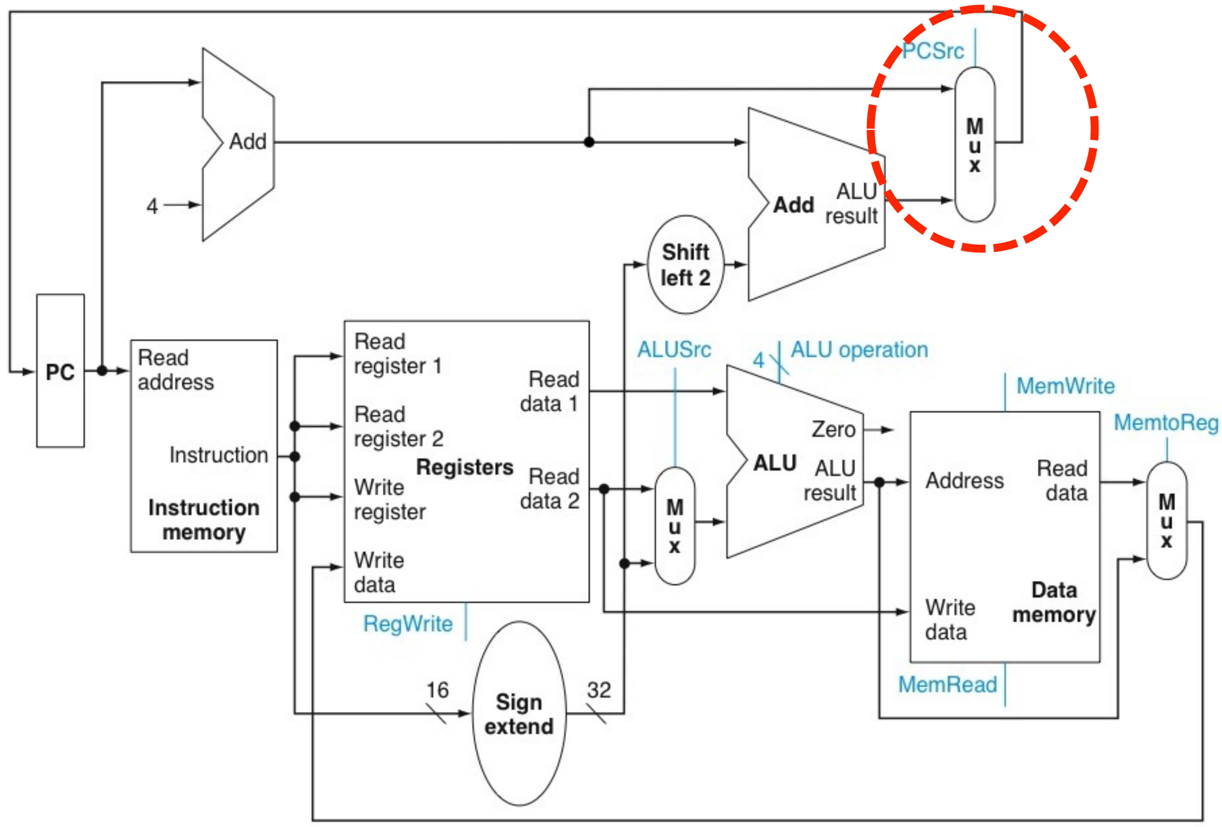
The lower two bits of instruction address are always zero, so to save some space we do not store them in the instruction. Thus, the hardware must shift the sign extended PC-offset by 2-bits (filling in the least significant bits with zero).

Combining Support for Loads/Stores and Arithmetic Instructions



- Above, we combine the hardware introduced in Slides 14 and 17.
- To help us do this, we add two multiplexers:
  - First one before the ALU: Arithmetic instructions read two source registers, but load and store instructions instead combine one register with sign extended displacement field.
  - Second one after data memory: Value written to register file from ALU for R-type, but from data memory for load instructions.

Combine with Branch/Jump



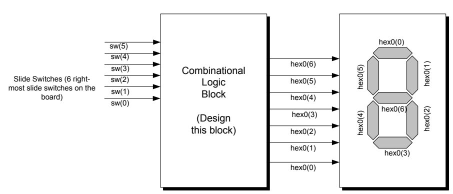
- Above, we combine logic for instruction access (slide 12) with logic for arithmetic and memory instructions (slide 20) with logic for conditional branch execution (slide 19)
- To help us do this, we add one more mux to select the "next PC" value.

Datapath Control

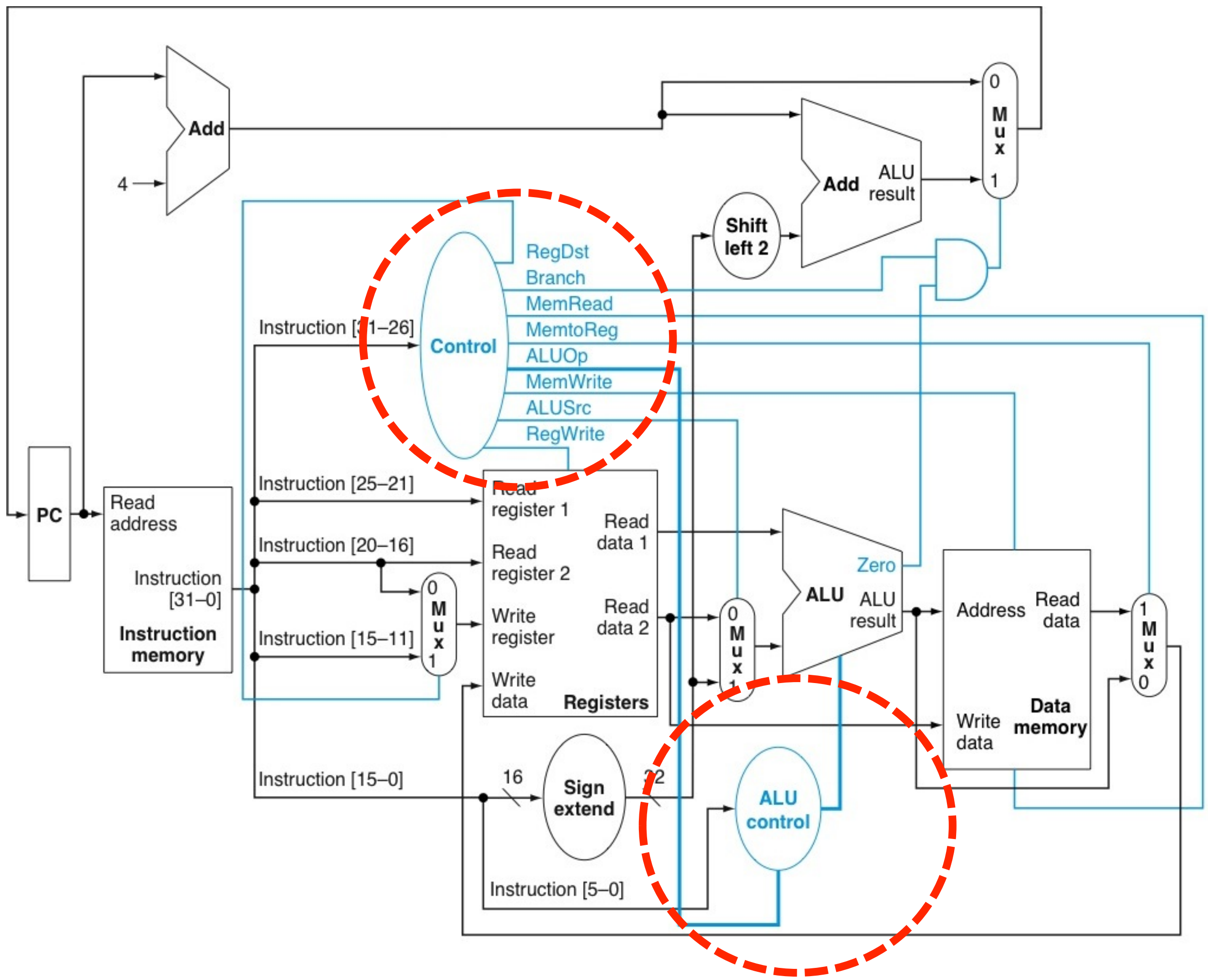
In the prior slide, we have combined the hardware for the different instructions. Each hardware block had a “control input” which we still need to connect to something. The thing missing is a control unit telling each blocks what to do.

After we know what the instruction is—i.e., the 1’s and 0’s, the control unit can set all the control signals.

The simplest control unit would be a purely combinational logic that reads the 1’s and 0’s of the instruction and then determines how to set the control signals for the data path hardware on the prior slide. This combinational logic is similar to the code converter used to drive the 7-segment displace in EECE 353 Lab #1.



Single Cycle MIPS64



The blocks circled are control units.

The top unit sets its outputs only based on the opcode field

The bottom unit decodes the “funct” field for R-type instructions.

Example: Performance Comparison

- Consider a processor with a “magical” clock that adjusts itself each cycle to be only as long as the current instruction needs it to be. So, one clock period could be 100 ps, and the next 200 ps, and the following one 50 ps (i.e., whatever we want it to be).

  How much faster is this “magical” processor than a more realistic processor that uses a clock that operates at a fixed frequency?

- For both processors assume the following:
- The hardware takes the following time for different operations
  - Memory Units: 200 picoseconds (ps)
  - ALU and adders: 100 ps
  - Register file (read or write): 50 ps
  - multiplexers, control unit, PC accesses, sign extension unit, and wires have no delay...
- Software:
  - 25% Loads, 10% stores, 45% ALU, 15% branches, 5% jumps...

Solution

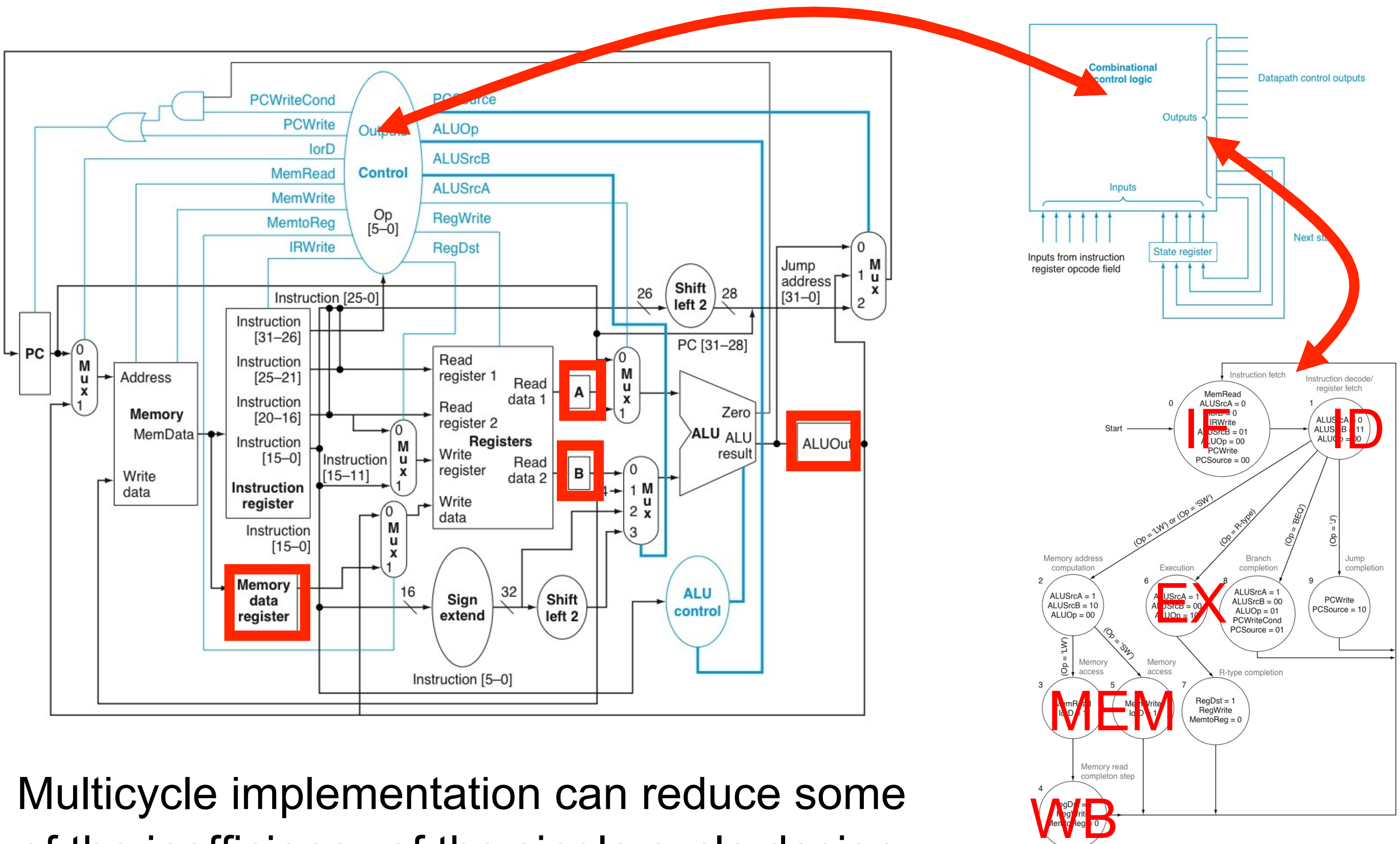
- Execution Time = IC * CPI * Clock Time
- For both designs, we know CPI = 1.0

Inst. Class	Functional units used by the instruction class				
R-Type	Inst. Fetch	Reg. Read	ALU	Reg. Write	
Load	Inst. Fetch	Reg. Read	ALU	Mem. Read	Reg. Write
Store					
Branch					
Jump					

Inst. Class	Inst. Mem	Reg. Read	ALU	Data Mem	Reg. Wr	Total
R-type	200	50	100	0	50	400 ps
Load						
Store						
Branch						
Jump						

Solution, cont'd (we will take this up in class)

A Multi Cycle MIPS64 Implementation



- Multicycle implementation can reduce some of the inefficiency of the single cycle design.
- Still has poor performance (small subset of HW in use at any time)

Example Multi Cycle MIPS64 Impl.

- Loads: 4 cycles (IF/Reg Read+Ex/Mem/WB)
- Stores: 3 cycles (IF/Reg Read+Ex/Mem)
- ALU: 2 cycles (IF/Reg Read+Ex+WB)
- Branches/Jumps: 2 cycles (IF/Reg Read+Ex)

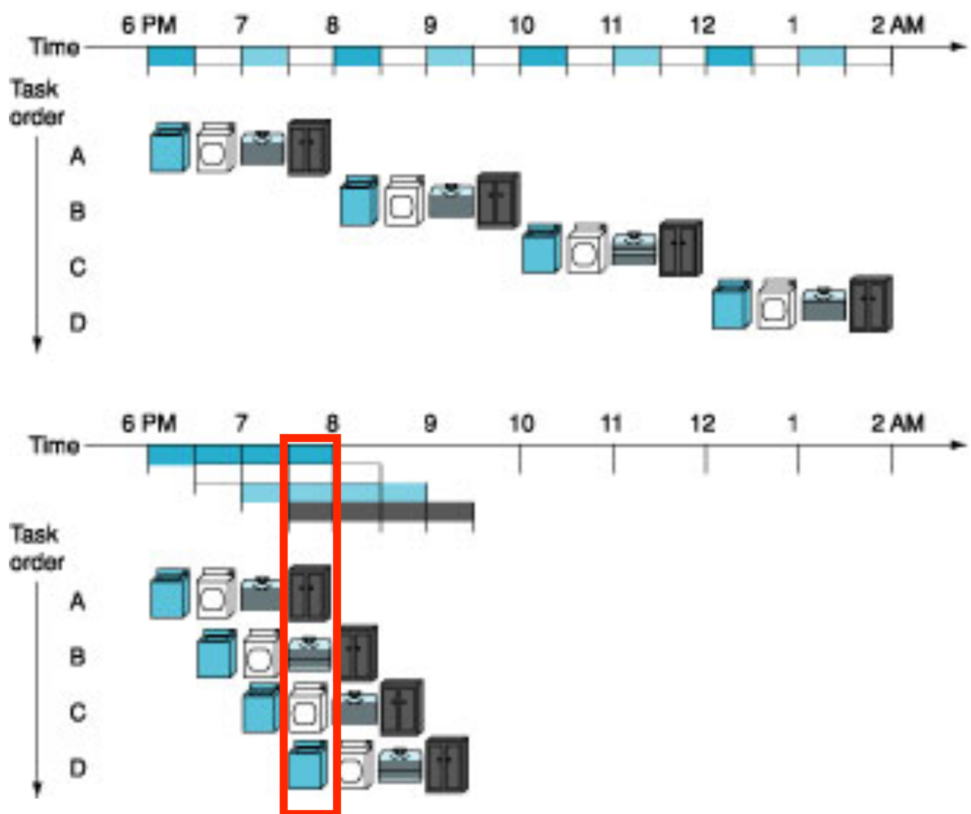
Pipelining: Analogy

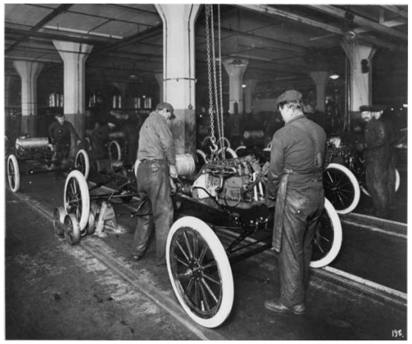
Early in Time

Later in Time

1st task

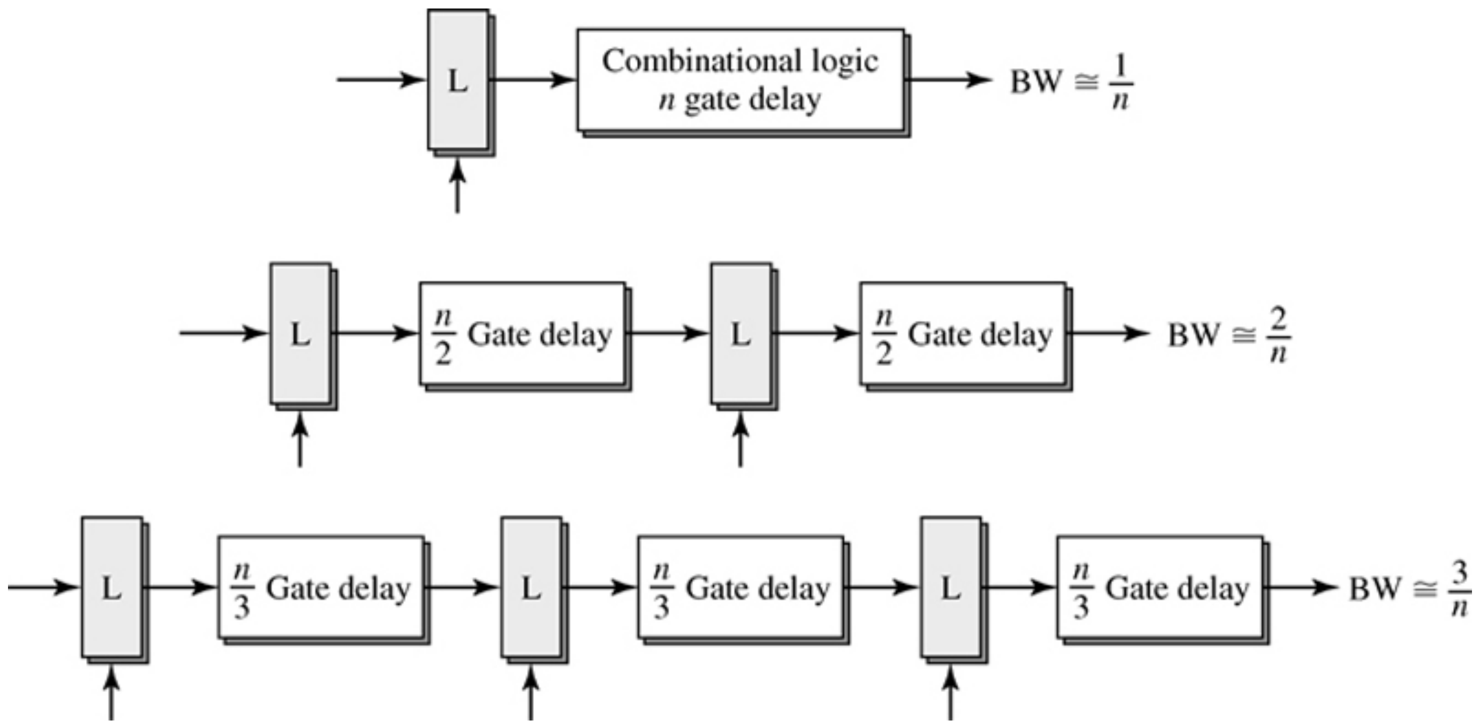
Last task



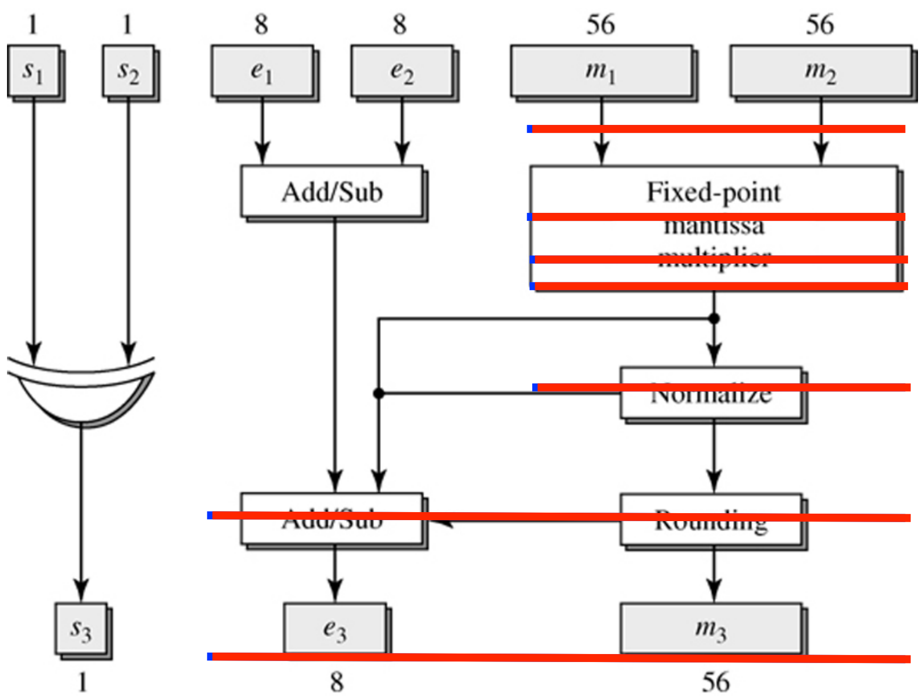


Simultaneous activity
From 7:30 to 8:00 pm

Pipelining Combination Logic



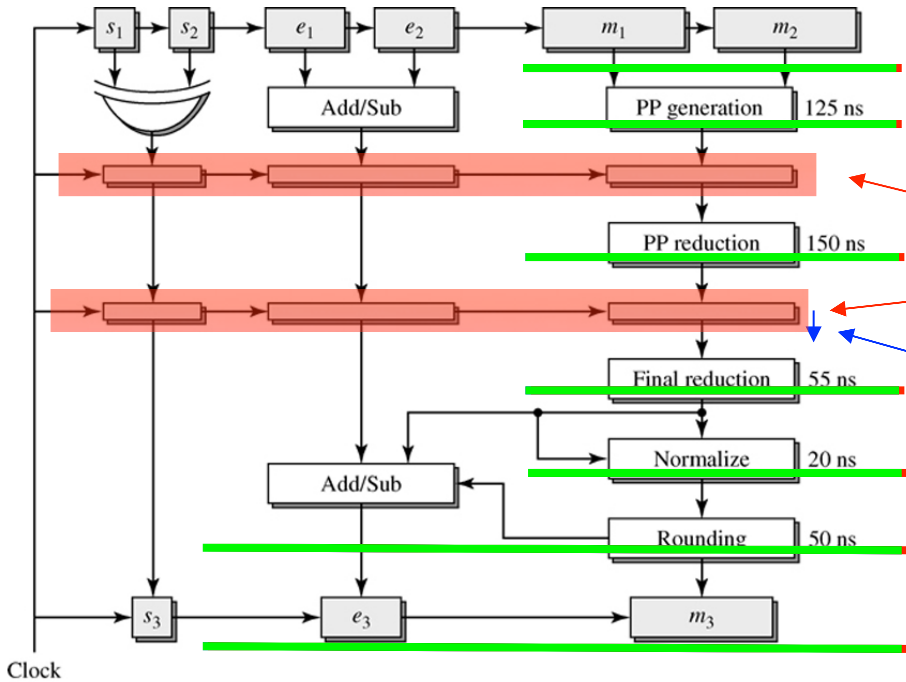
Example: Floating-Point Multiplier



Module	Chip Count	Delay, ns
Partial Product generation	34	125
Partial Product reduction	72	150
Final reduction	21	55
Normalization	2	20
Rounding	15	50
Exponent section	4	
Input latches	17	
Output latches	10	
Total	175	400

Pipelined Floating-Point Multiplier

[Waser and Flynn, 1982]



82 ICs

setup time = 5 ns
clock to output = 17 ns

Clock Period = 150+5+17 = 172 ns

Cost = 175+82=257 => 45% increase

$$\frac{\text{Throughput}_{\text{pipelined}}}{\text{Throughput}_{\text{un-pipelined}}} = \frac{\text{Freq}_{\text{pipelined}}}{\text{Freq}_{\text{un-pipelined}}} = \frac{400\text{ns}}{172\text{ns}} = 2.3 \Rightarrow 130\% \text{ increase}$$

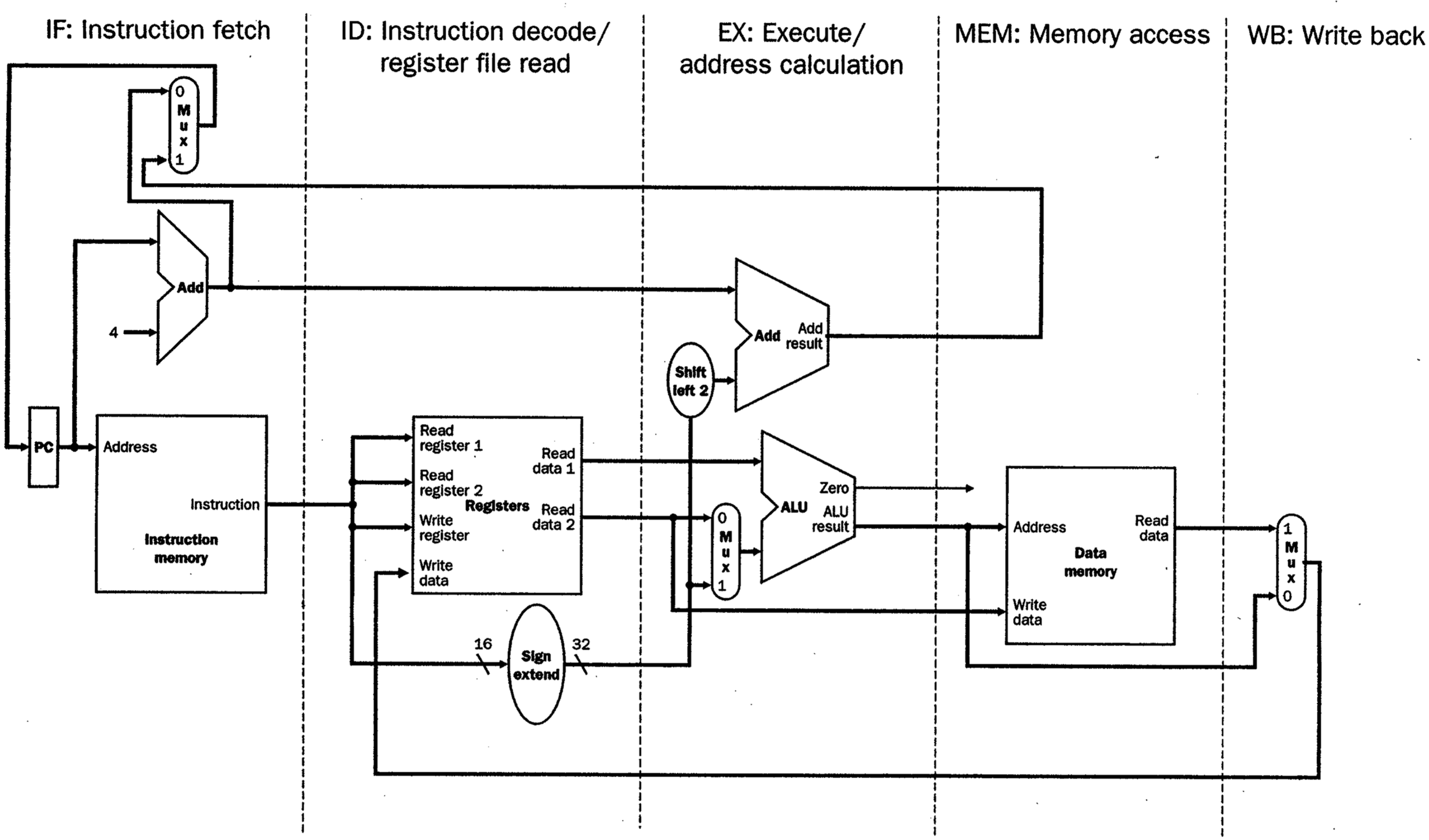
Pipelining Idealism/Challenges

- Uniform Subcomputations
  - Goal: Each stage has same delay
  - Achieve by balancing pipeline stages

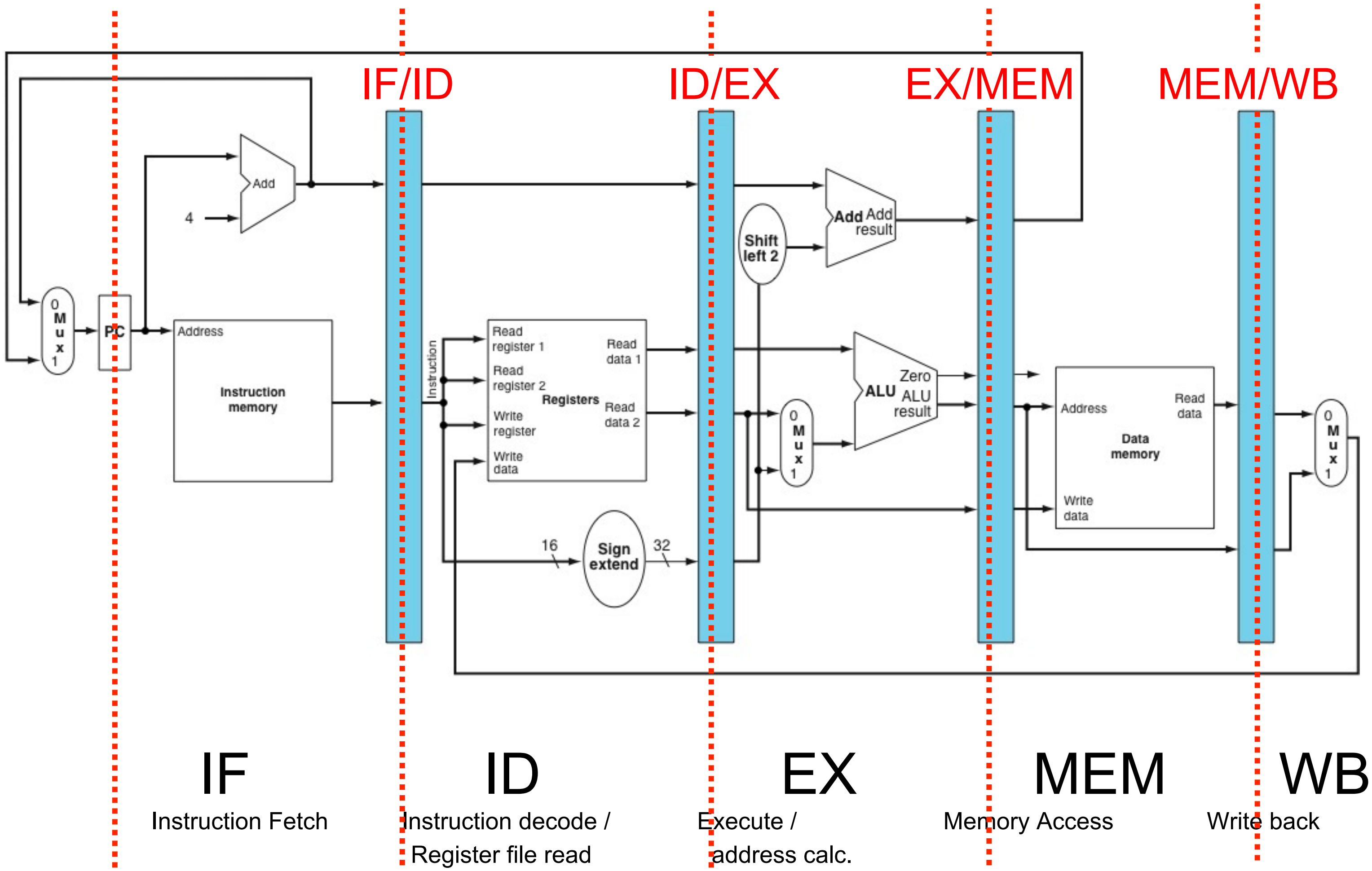
- Identical Computations
  - Goal: Each computation uses same number of stages

- Independent Computations
  - Goal: start new computation each cycle

Instruction Pipelining…
Start with Single Cycle MIPS64 Processor



Add registers between stages of instruction pipeline



- "Pipeline Registers" keep value after rising edge of clock. Identify pipeline register by stage before and after it.
- NOT visible to programmer.

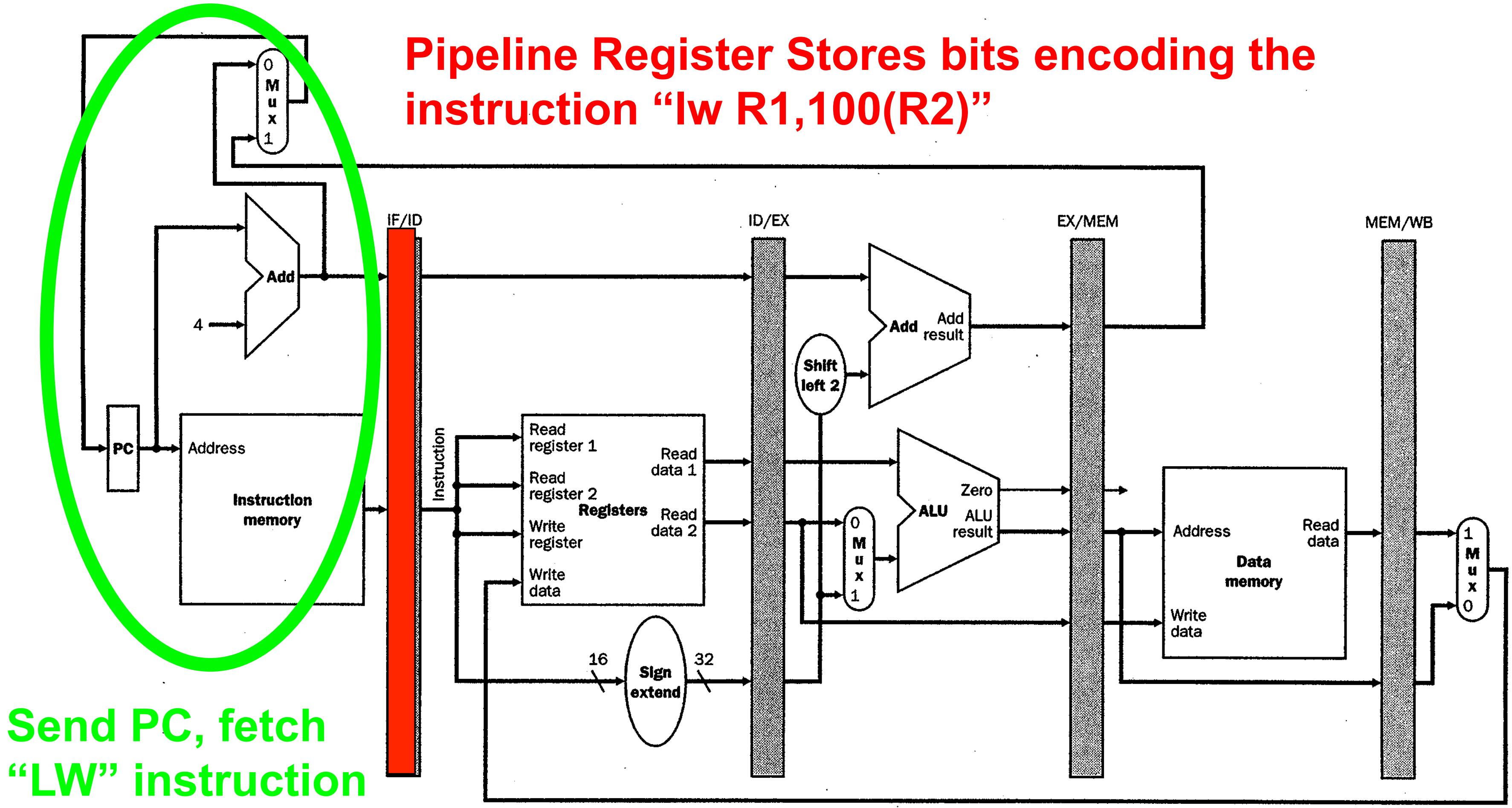
Instruction Pipelining Example

Let's look at how a single instruction flows through a 5-stage pipeline:

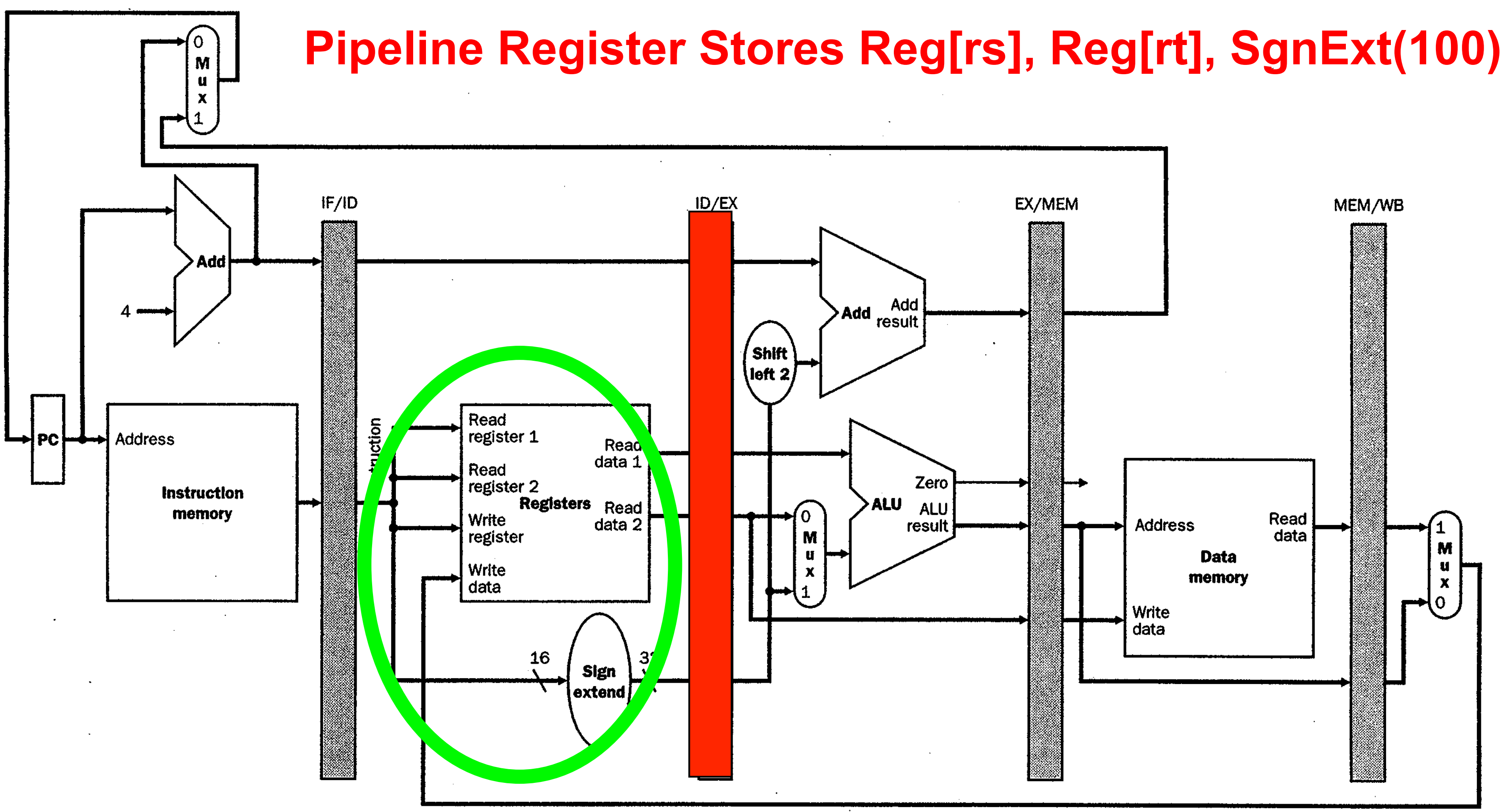
lw R1, 100(R2)

Pipelined “lw R1, 100(R2)”

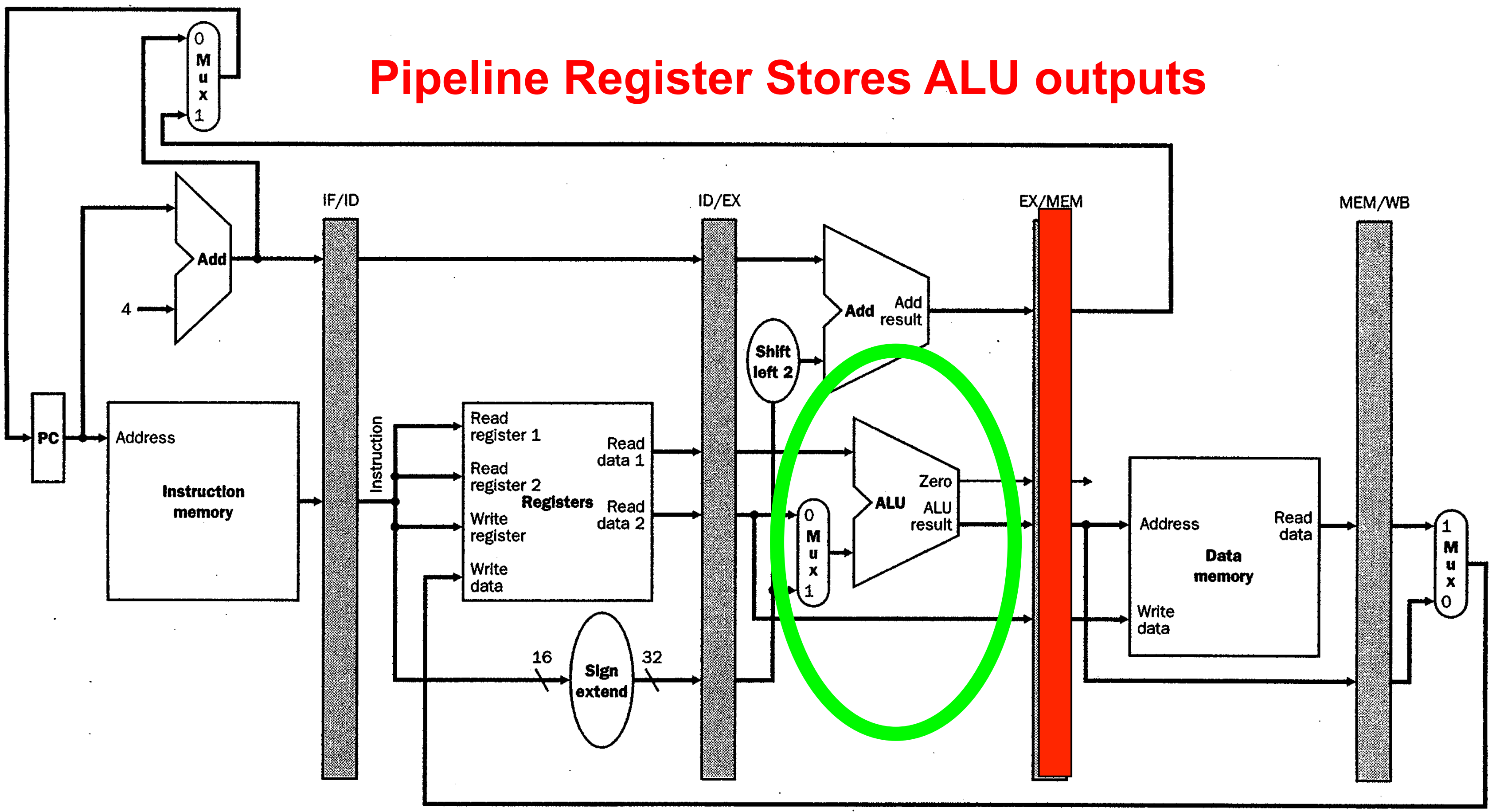
- Add registers between stages of execution



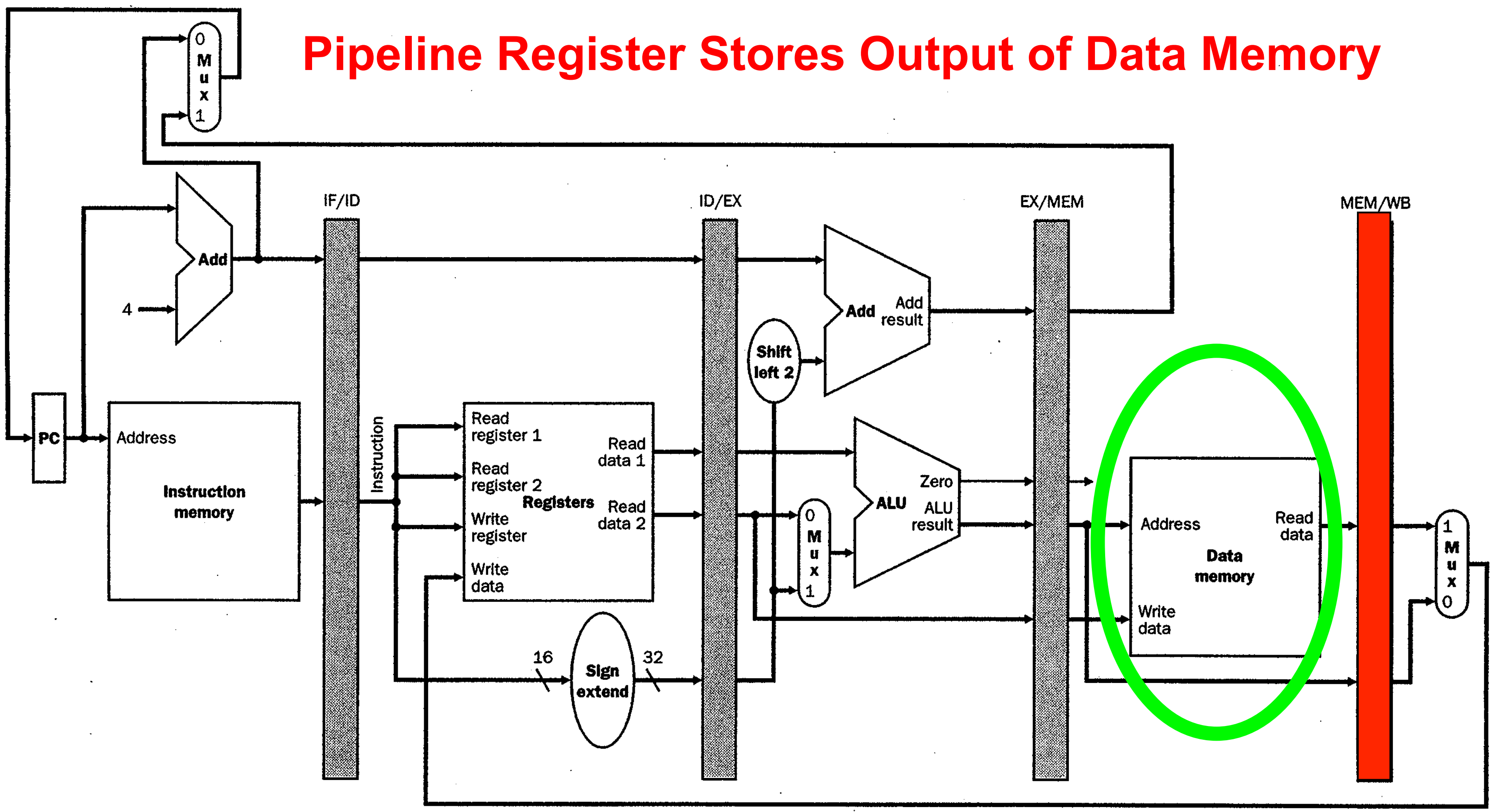
Pipelined “lw R1, 100(R2)”



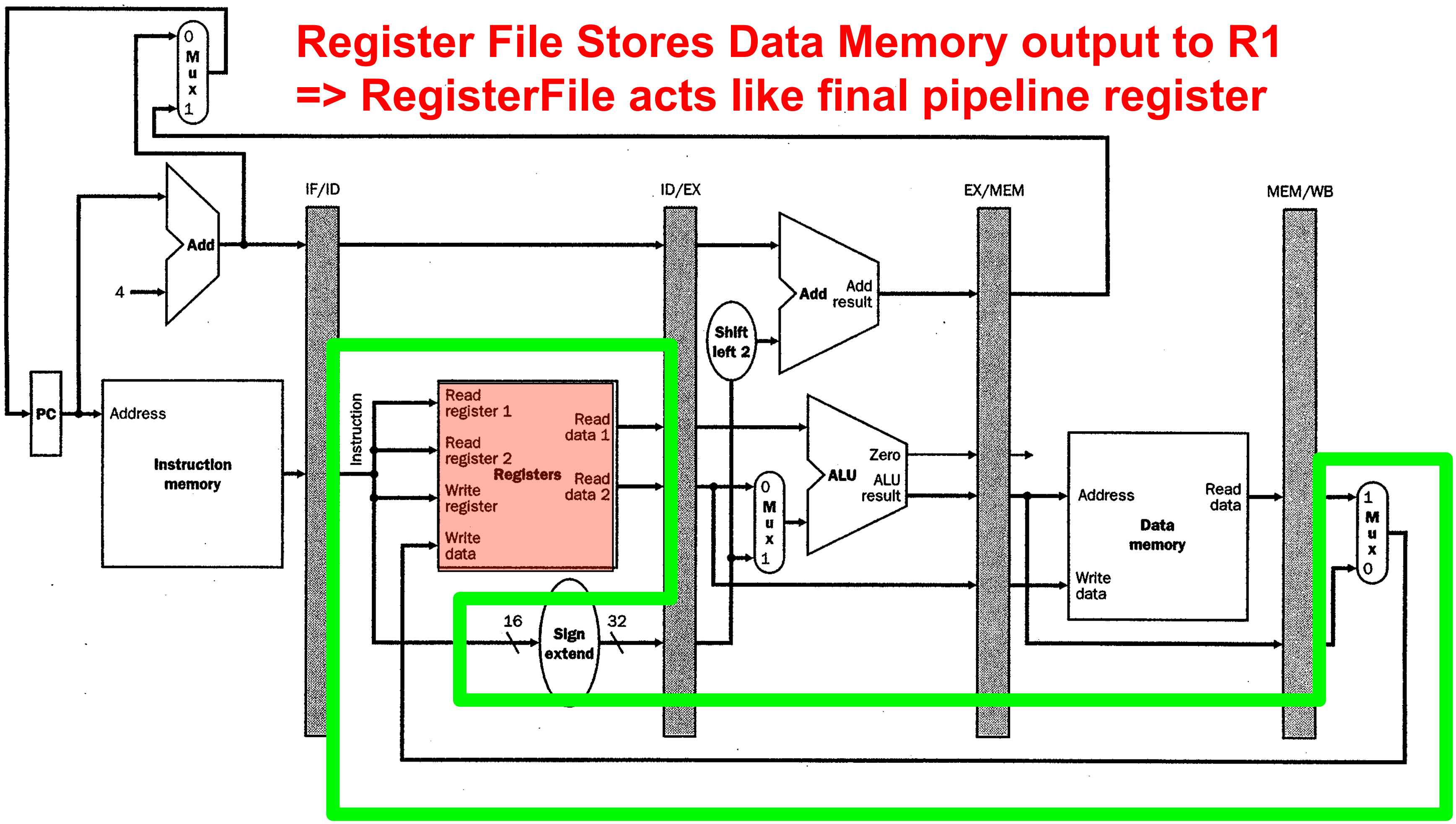
Pipelined “lw R1, 100(R2)”



Pipelined "lw R1, 100(R2)"



Pipelined “lw R1, 100(R2)”



Question

Mystery Slide…

Pipelining Idealism/Challenges

- Uniform Subcomputations
  - Goal: Each stage has same delay
  - Achieve by balancing pipeline stages
- Identical Computations
  - Goal: Each computation uses same number of stages
  - Achieve by unifying instruction types
- Independent Computations
  - Goal: Avoid hazards
  - Look for ways to minimize pipeline stalls

ISA Impact

- Uniform Subcomputations
  - Memory addressing modes <=> Disparity of speed between processor and memory.
- Identical Computations
  - RISC: reducing complexity makes each instruction use roughly same number of stages.
- Independent Computations
  - Reg-Reg (“Load-Store”) : Makes it easier to identify dependencies (versus Reg-Mem).

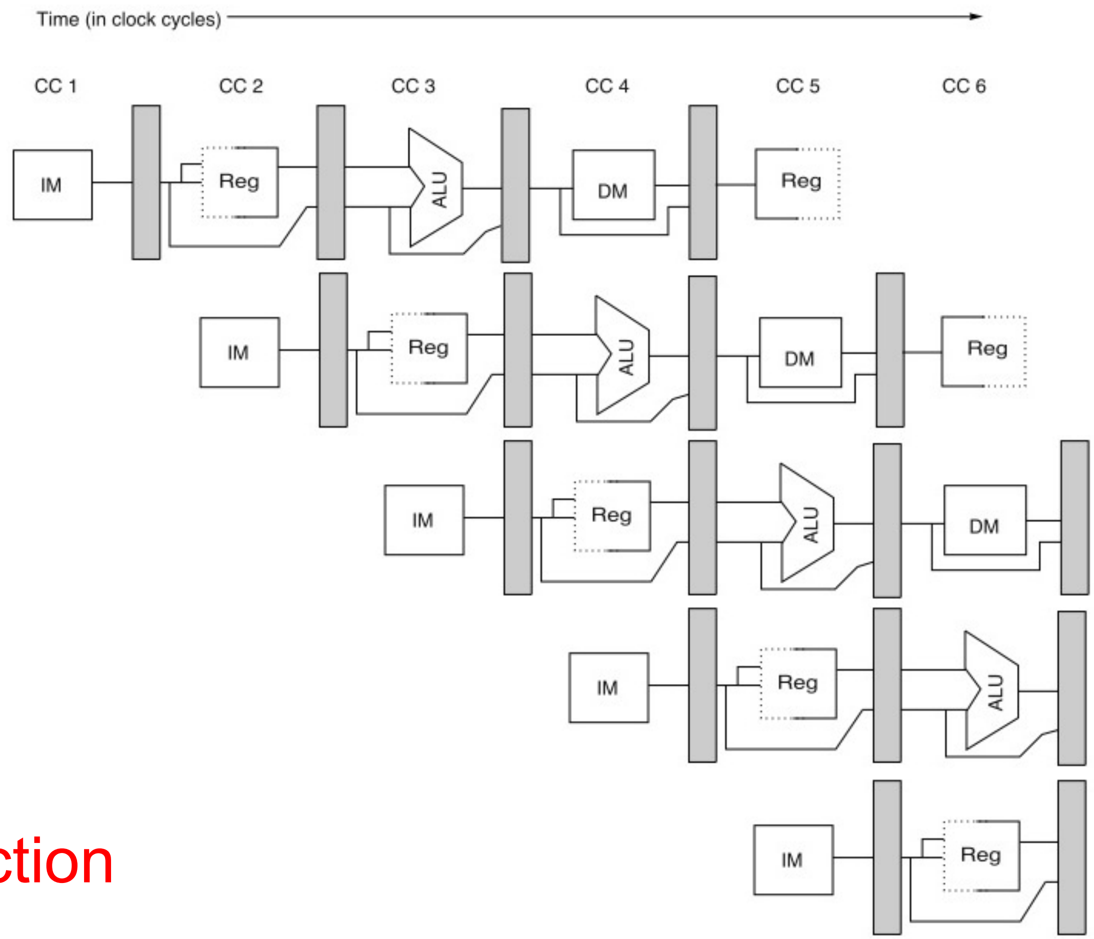
Analysis of Instruction Pipelining

Step 1: Imagine hardware is replicated for each instruction…

Early in Time

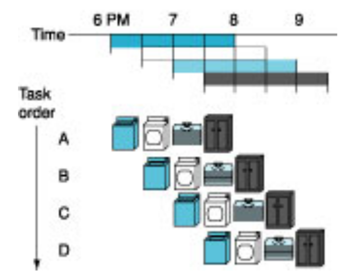
Later in Time

1st Inst.



Last Instruction

Compare to slide 21 for washer/ dryer/folding in laundry example.



Step 2: Abstraction

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

- ID,…,WB = “instruction decode,” …, “write back” (slide 6-7)
- Above example:
  - CPI = 1.0 (ideal, can’t be faster)
  - In reality, instructions may have “dependencies”, hardware may have “hazards” that increase CPI above 1.0

Pipelining introduces potential “hazards”

	Clock Number								
	1	2	3	4	5	6	7	8	9
DADD R1,R2,R3	IF	ID	EX	MEM	WB				
DSUB R4,R1,R5		IF	ID	EX	MEM	WB			
AND R6,R1,R7			IF	ID	EX	MEM	WB		
OR R8,R1,R9				IF	ID	EX	MEM	WB	
XOR R10,R1,R11					IF	ID	EX	MEM	WB

- How do instructions after DADD get the correct value for R1?

Pipeline Hazards

1. Structural Hazards
2. Data Hazards
3. Control Hazards

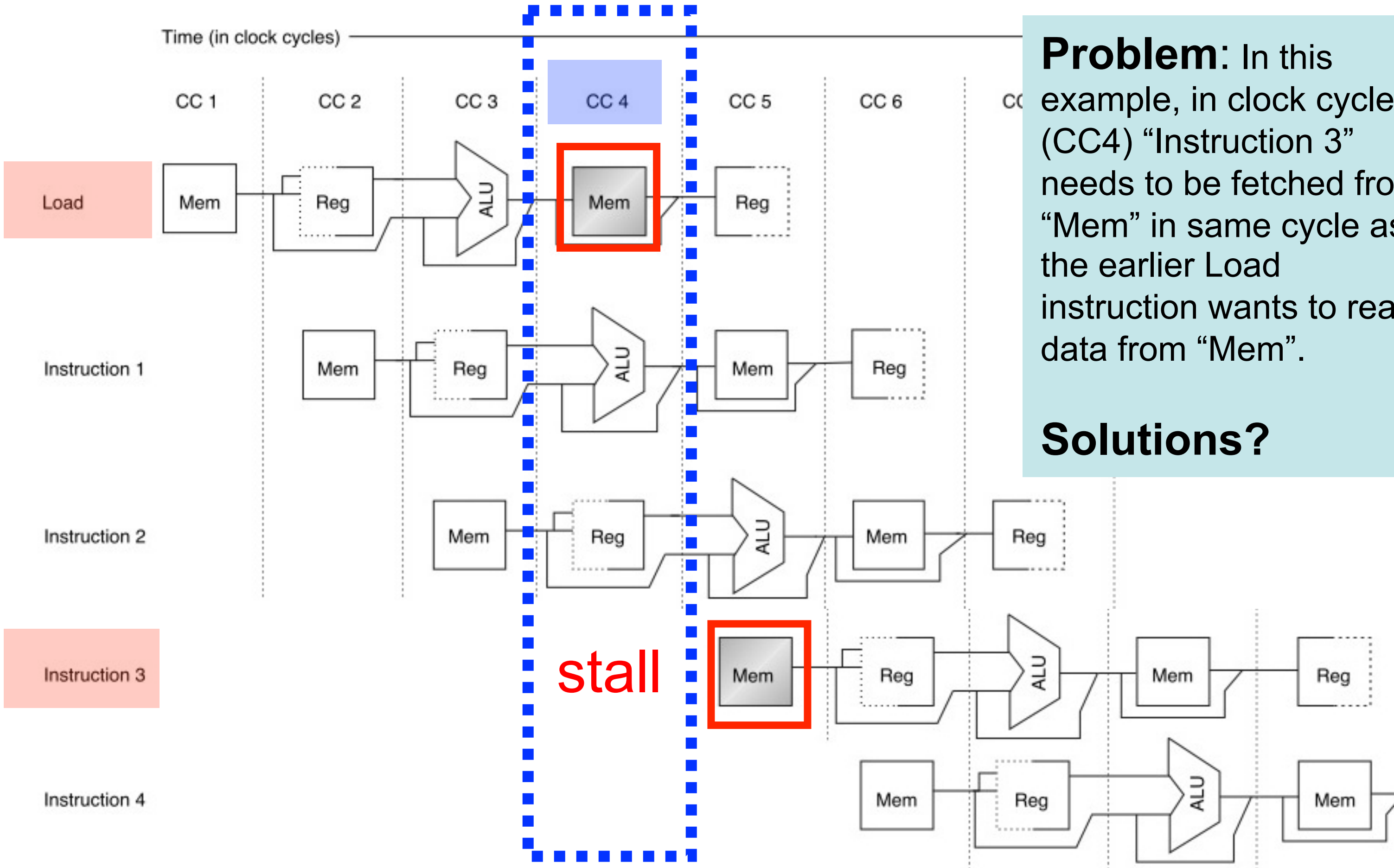
- We will discuss each type in turn. For each one consider:
  1. How is the hazard caused?
  2. What are the solutions to “resolve” the hazard?
  3. How do different solutions differ in their impact on performance?

Structural Hazards

- Caused by resource conflicts
  - Two or more instructions want to use the same hardware at the same time.
- Two ways to resolve structural hazards:
  - Solution 1: Stall
    - Later instruction waits until earlier instruction is finished using resource… BUT this can lead to reduction in performance
  - Solution 2: Replicate hardware
    - Often we can eliminate need for stalling by replicating hardware or building hardware capable of servicing multiple instructions at the same time.

Structural Hazard Example

What if data and instructions are placed in same memory (at different locations)?



Problem: In this example, in clock cycle 4 (CC4) "Instruction 3" needs to be fetched from "Mem" in same cycle as the earlier Load instruction wants to read data from "Mem".

Solutions?

Question...

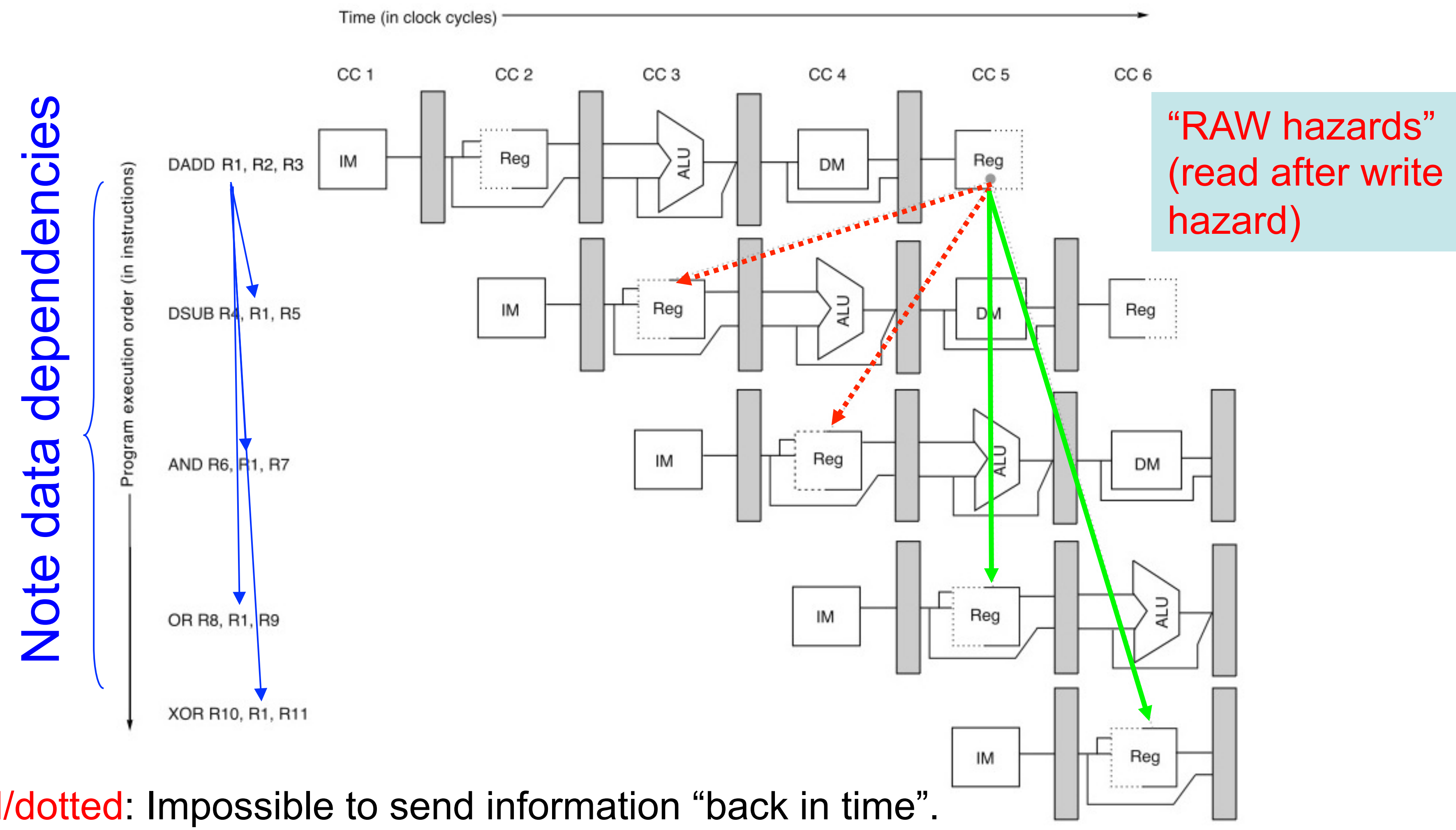
Which processor is faster (& by how much?)

Assume:

- 40% of instructions are data memory references.
- Ignoring structural hazard CPI = 1.0
- Processor with structural hazard has clock rate that is 1.05 times faster than processor w/o.

Solution...

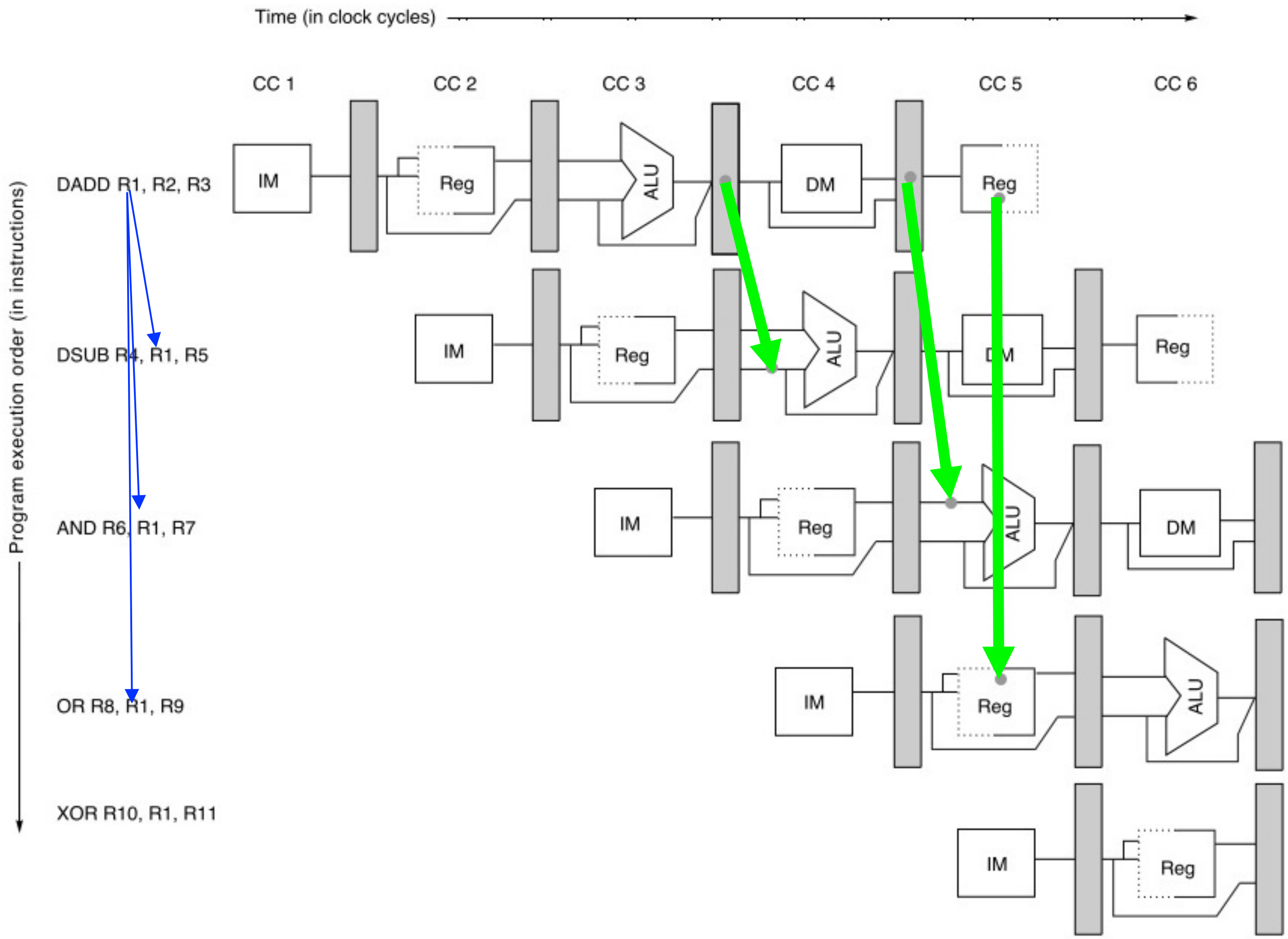
Problem: Data Hazards



Red/dotted: Impossible to send information "back in time".

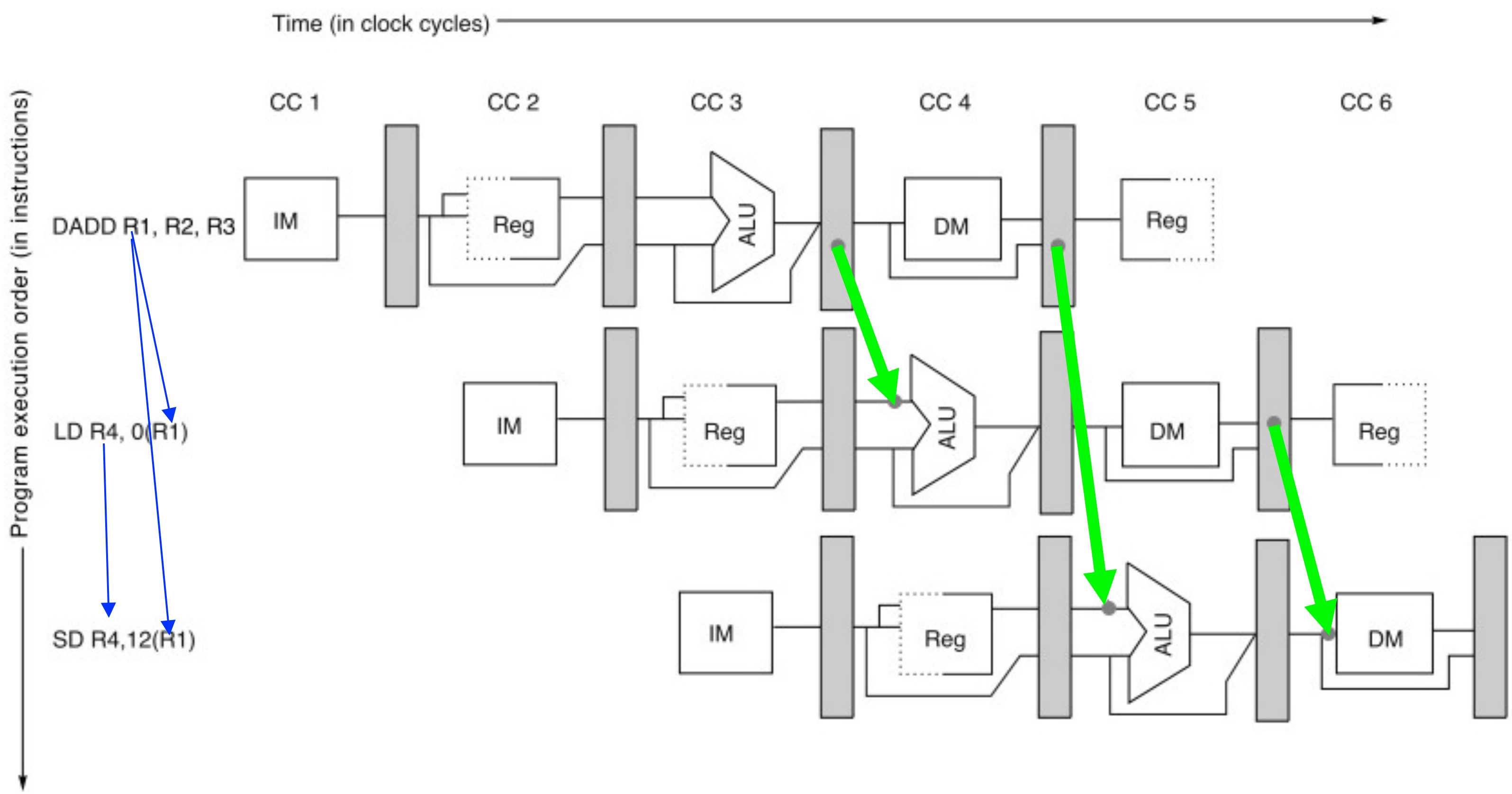
Green/solid: Can potentially send information to other part of pipeline in same cycle or to a later cycle… requires hardware support and consideration of what can happen in a single clock cycle.

Forwarding



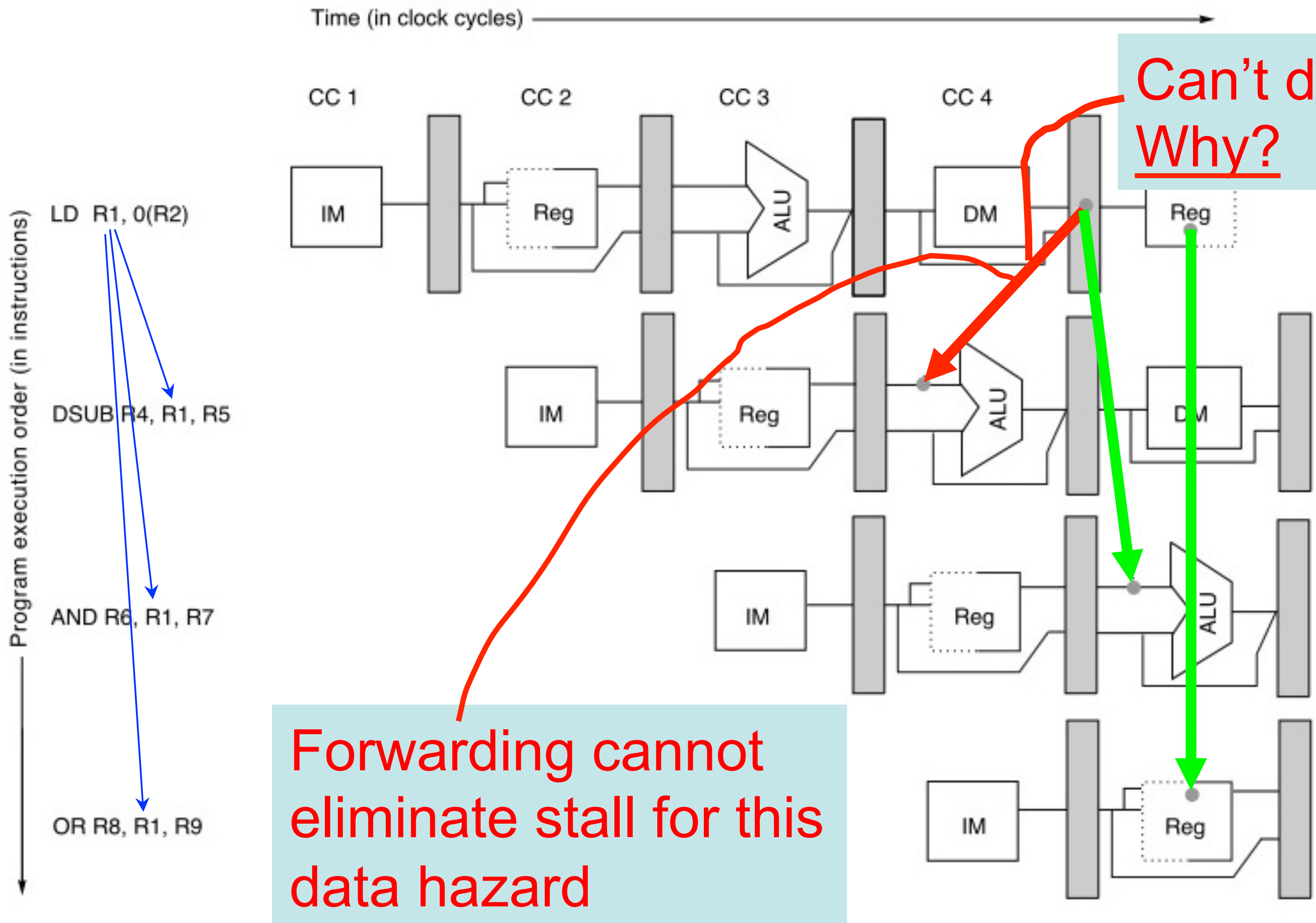
Note: forwarding always occurs <u>within a single clock cycle</u>.

Forwarding Example #2



Questions:
1. How do we know a forwarding path is helpful?
2. How do we control the forwarding path?

Unavoidable Stalls



Can't do this. <u>Why?</u>

Forwarding cannot eliminate stall for this data hazard

Control Hazards

	Clock Number							
	1	2	3	4	5	6	7	8
branch instruction	IF	ID	EX	MEM	WB			
branch successor		IF	IF	ID	EX	MEM	WB	
branch succ. + 1				IF	ID	EX	MEM	WB
branch succ. + 2					IF	ID	EX	MEM

- Branch may or may not change program counter (PC). Therefore, which instruction should follow a branch depends upon the result of <u>executing</u> the branch instruction.

- Result of branch typically known either at end of EX, or (in some optimized hardware implementations), by end of ID.

- <u>Option #1:</u> Stall after branch is detected, until branch target is known and branch condition is resolved.

Control Hazard Option #2: Predict "Not Taken"

	Clock Number							
	1	2	3	4	5	6	7	8
untaken branch	IF	ID	EX	MEM	WB			
branch instr. + 1		IF	ID	EX	MEM	WB		
branch instr. + 2			IF	ID	EX	MEM	WB	
branch instr. + 3				IF	ID	EX	MEM	WB

	Clock Number							
	1	2	3	4	5	6	7	8
TAKEN branch	IF	ID	EX	MEM	WB			
branch instr. + 1								
branch target								
branch target + 1								

We will fill in the table above in lecture

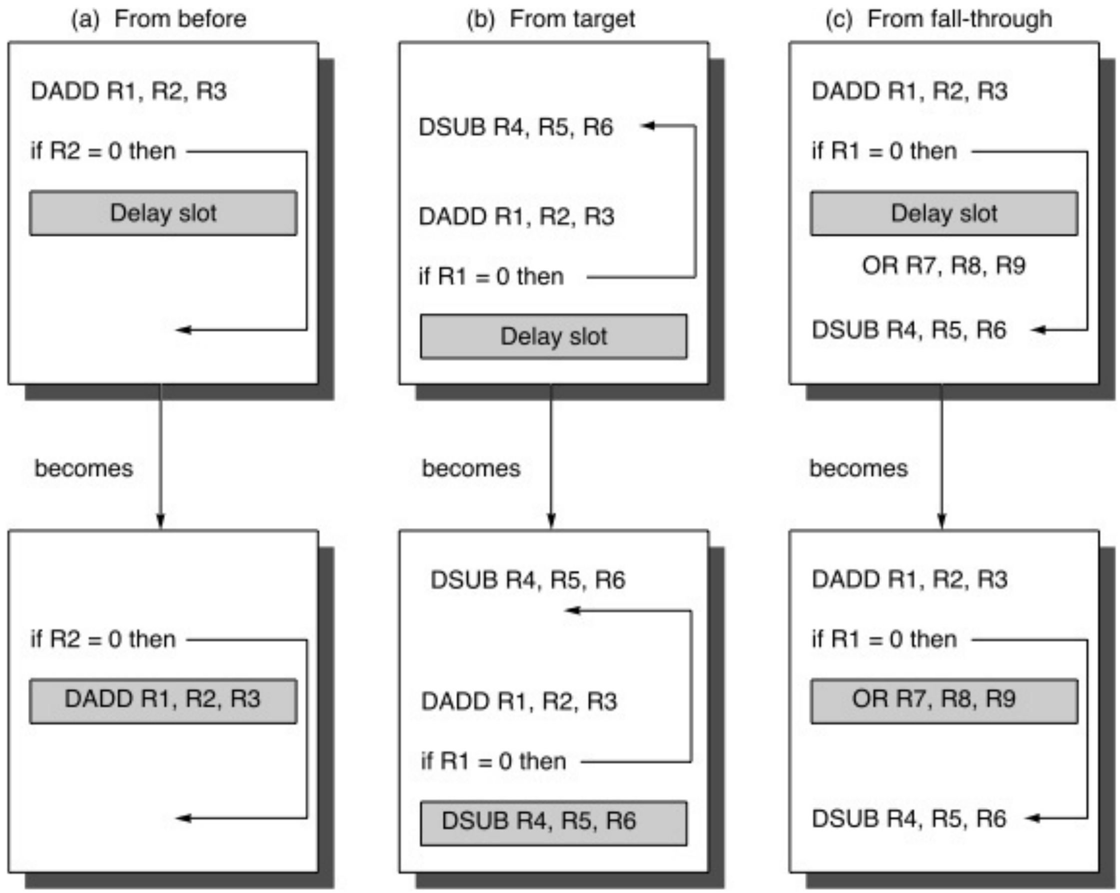
Option #3: Delayed Branches

- Delayed branch = branch that does not change control flow until after next N instructions (typically N=1).

	Clock Number							
	1	2	3	4	5	6	7	8
TAKEN branch	IF	ID	EX	MEM	WB			
branch delay (succ)		IF	ID	EX	MEM	WB		
branch TARGET			IF	ID	EX	MEM	WB	
branch TARGET + 1				IF	ID	EX	MEM	WB

(Delayed branches were motivated by 5-stage pipeline... “negative benefit” due to impl. complexity in modern microarchitectures)

Strategies for filling delay slot



Control Hazards

- Arise from the pipelining of branches (and other instructions, like jumps, that change the PC).
- Three mechanisms to deal with them:

1. Stop fetching until branch is resolved
   – flush instruction after branch and fetch correct one
   – Slide 53
2. Predict the outcome of the branch
   – only flush if we were wrong
   – Simplest approach “predict not taken” (Slide 54). We will look at better ways later.
3. Delayed branch
   – require the compiler (or assembly programmer) to “fill delay slot”
   – Slide 55-56

Summary of This Slide Set

In this slide set we learned about how to translate an instruction set architecture into a simple processor implementation.

Then, we saw how the performance of this implementation could be made faster by pipelining instruction execution.

We talked about the challenges of pipelining instruction execution, namely the three pipelining idealisms and hazards.

We also discussed ways to mitigate the effect of hazards.

In the next slide set, we are going to look at how pipelining is implemented in more detail. This will provide intuition that enables you to quickly and accurately analyze the performance characteristics of software executed on pipelined processors.