

---

# **C335**

## **Computer Structures**

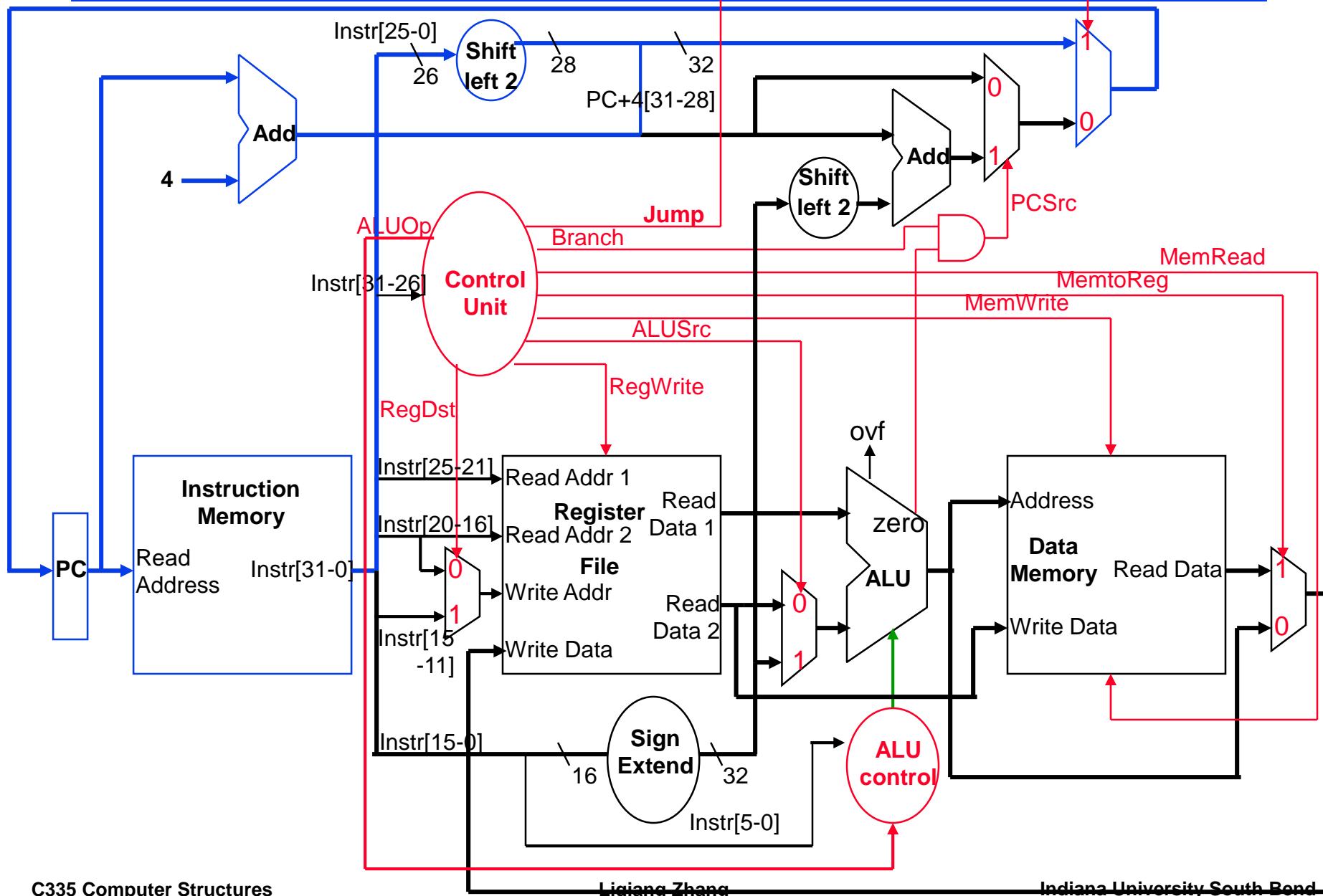
### **Pipelined Datapath Design**

Dr. Liqiang Zhang

Department of Computer and Information Sciences

Adapted from Morgan Kaufmann and others

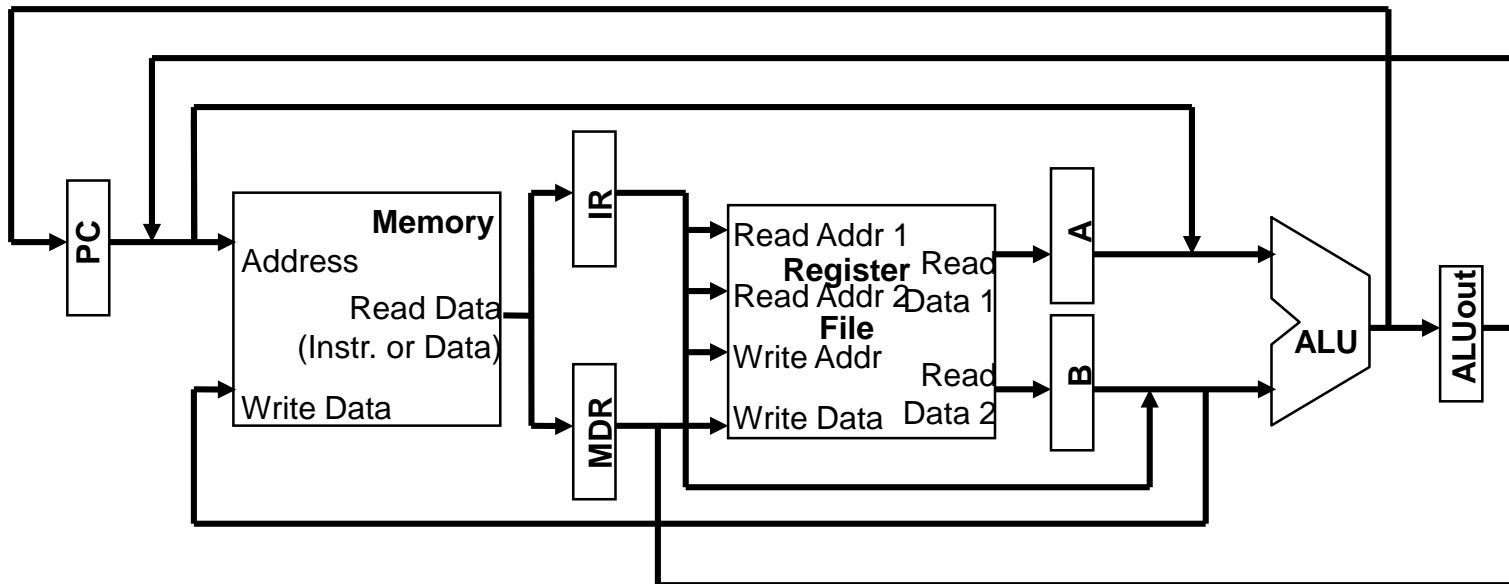
# Review: Single-cycle Datapath



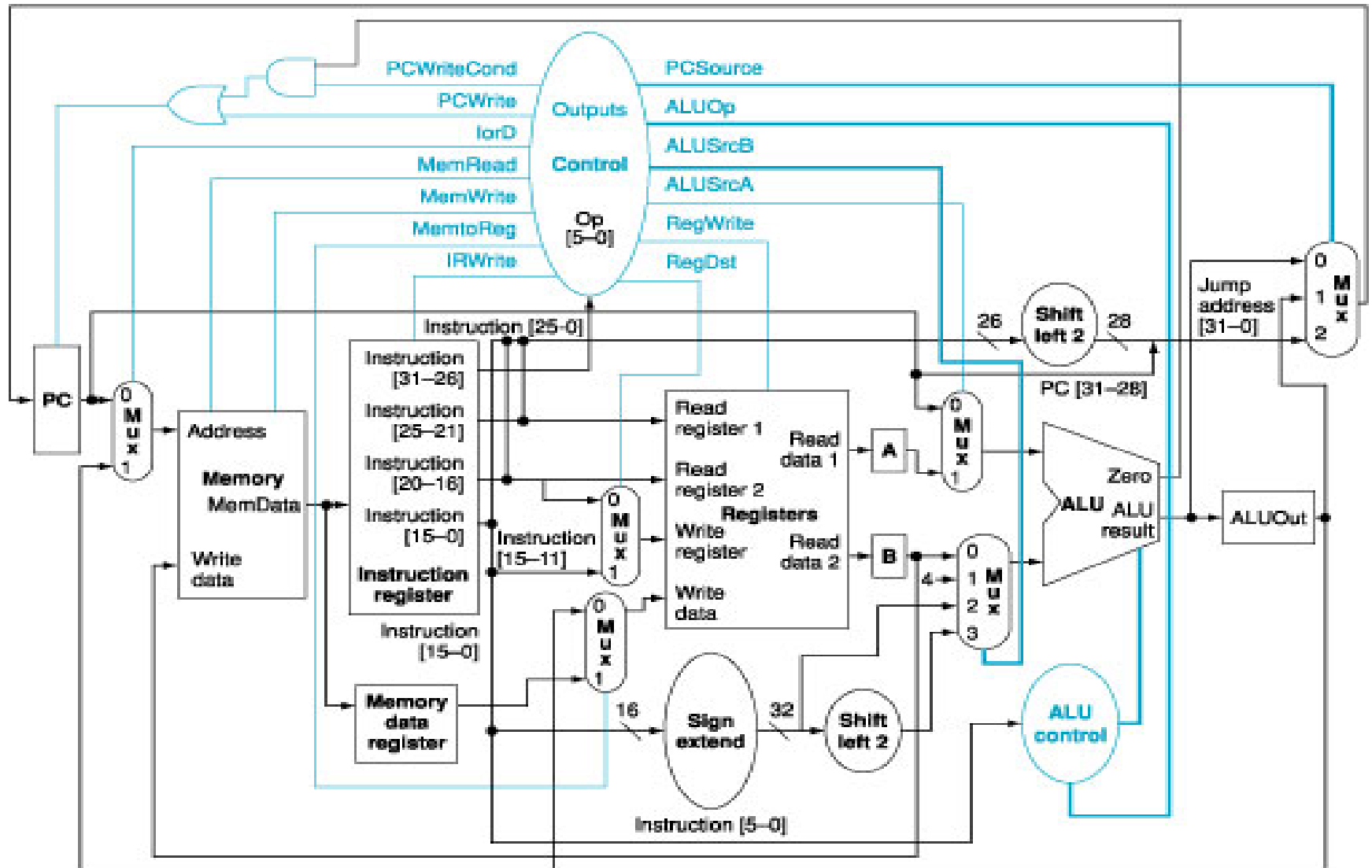
# Review: Multi-cycle Datapath

## ❑ Another approach

- use a “smaller” cycle time
- have different instructions take different numbers of cycles
- a “multicycle” datapath



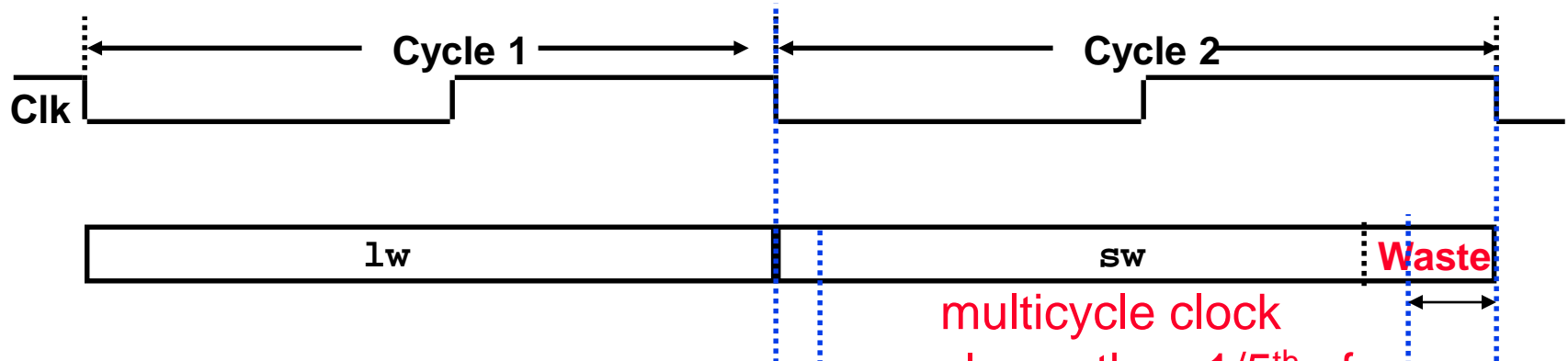
# A Multi-cycle Implementation



# Single Cycle vs. Multiple Cycle Timing

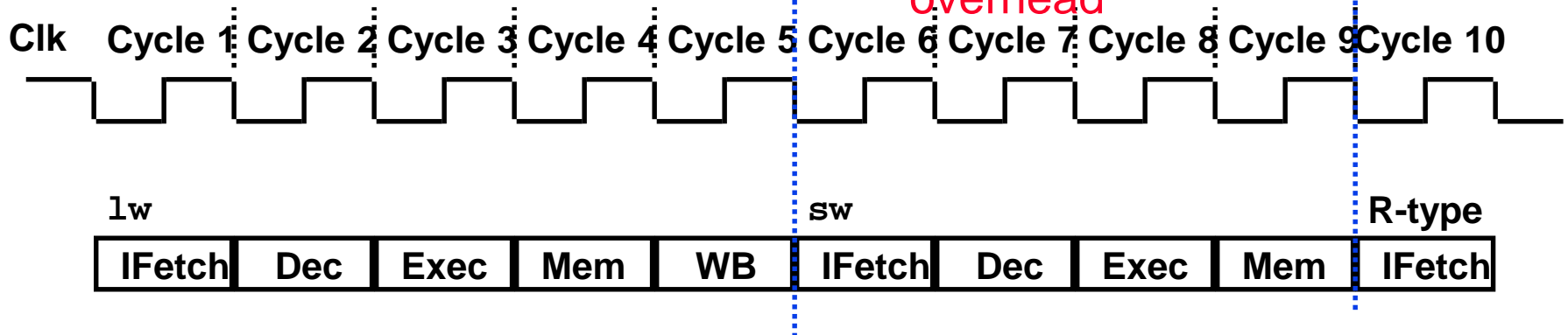


## Single Cycle Implementation:



multicycle clock  
slower than  $1/5^{\text{th}}$  of  
single cycle clock  
due to stage register  
overhead

## Multiple Cycle Implementation:



# How Can We Make It Even Faster?

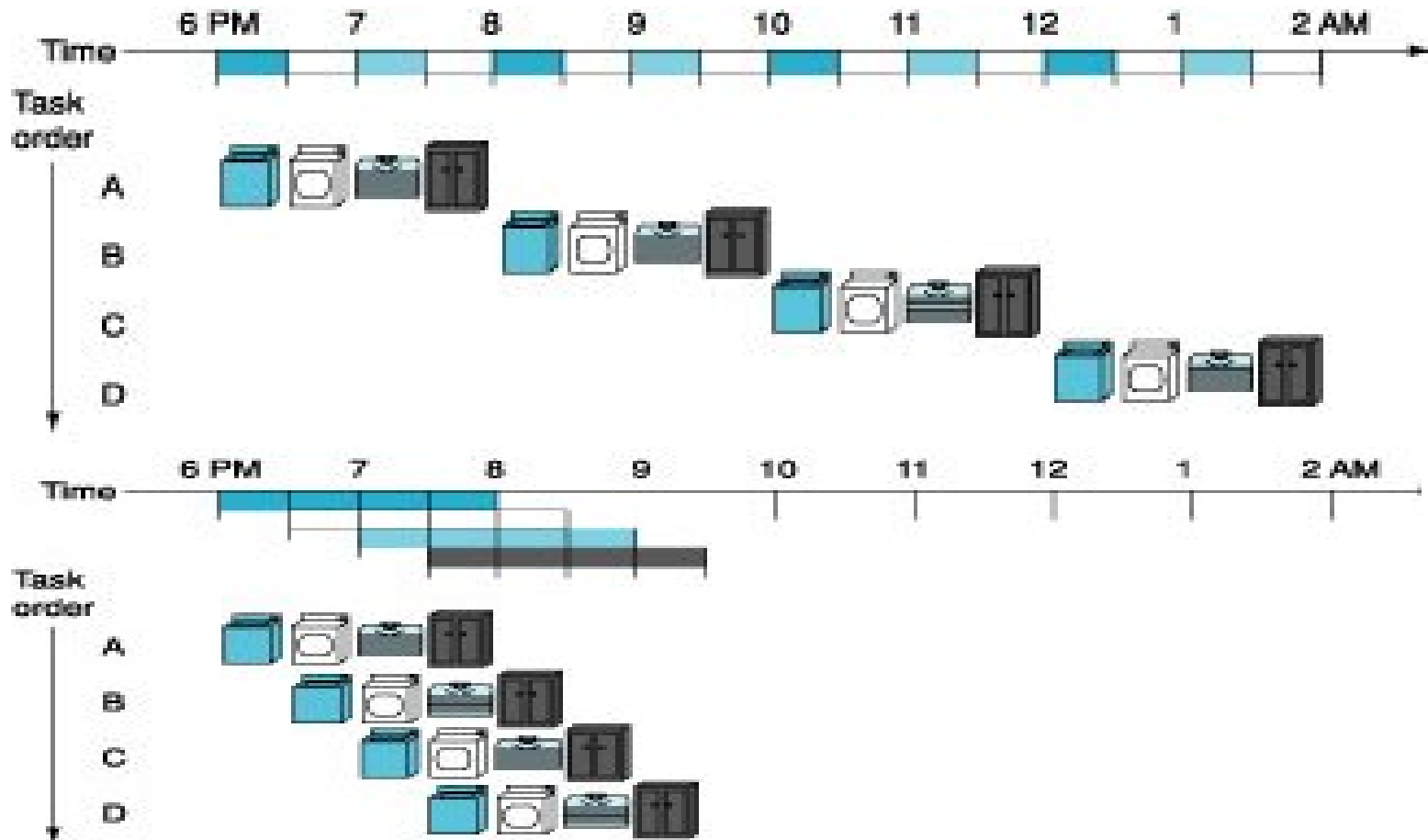


- ❑ Split the multiple instruction cycle into smaller and smaller steps
  - There is a point of diminishing returns where as much time is spent loading the state registers as doing the work
- ❑ Start fetching and executing the next instruction before the current one has completed
  - **Pipelining** – (all?) modern processors are pipelined for performance
- ❑ Fetch (and execute) more than one instruction at a time (out-of-order superscalar and VLIW/EPIC)
- ❑ Fetch (and execute) instructions from more than one instruction stream (hyperthreading)

# The Laundry Analogy for Pipelining

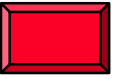


- Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away

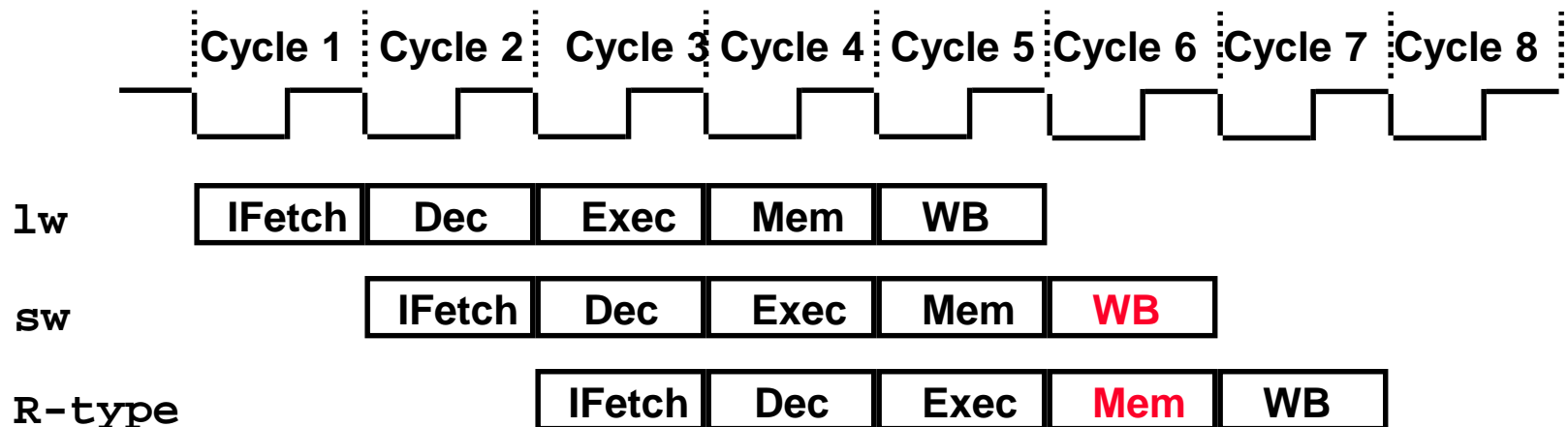


- If there is enough work to do, the speedup due to  
c pipelining is equal to the number of stages in the pipeline. nd

# A Pipelined MIPS Processor



- ❑ Start the **next** instruction before the current one has completed
  - improves **throughput** - total amount of work done in a given time
  - instruction **latency** (time from the start of an instruction to its completion) is **not** reduced



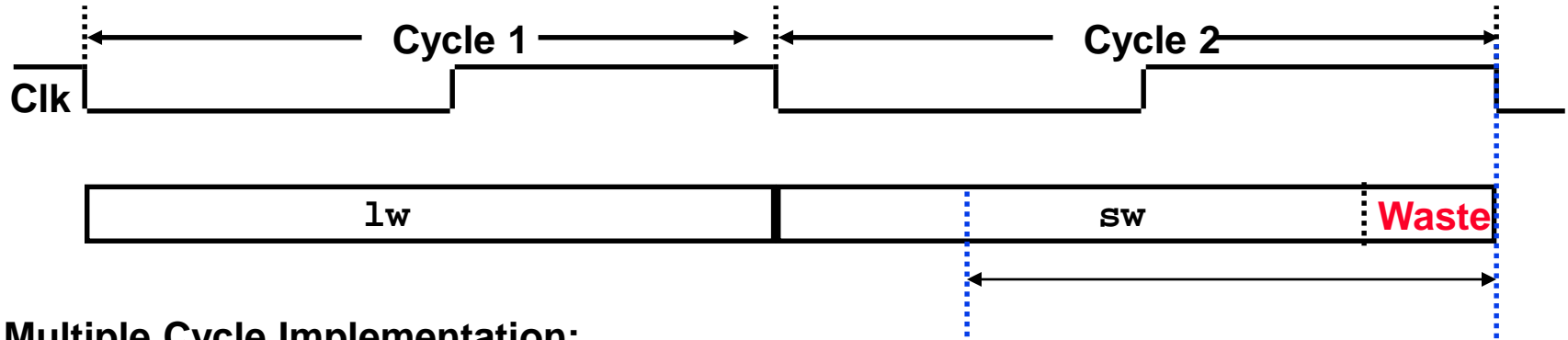
- clock cycle (pipeline stage time) is limited by the slowest stage
- for some instructions, some stages are **wasted** cycles



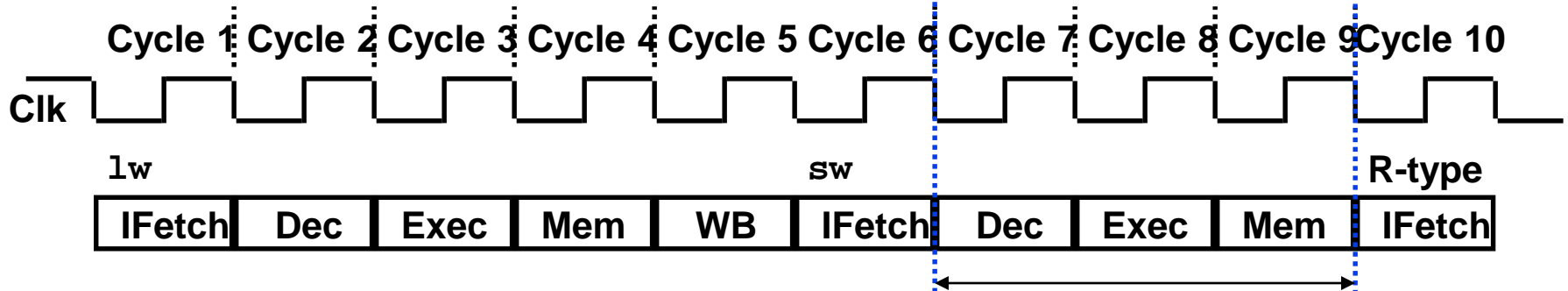


# Single Cycle, Multiple Cycle, vs. Pipeline

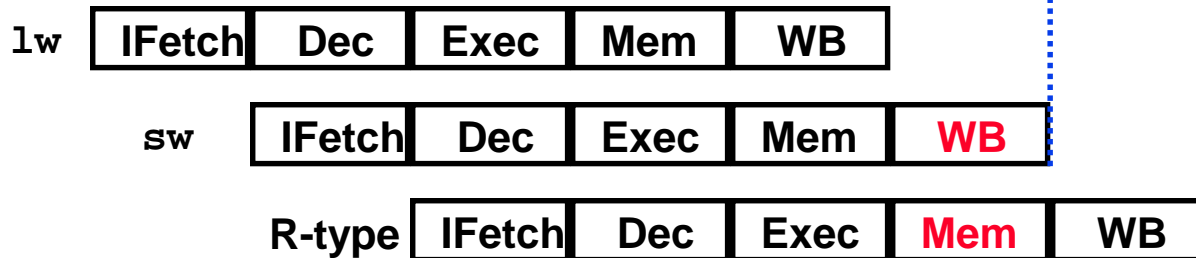
## Single Cycle Implementation:



## Multiple Cycle Implementation:



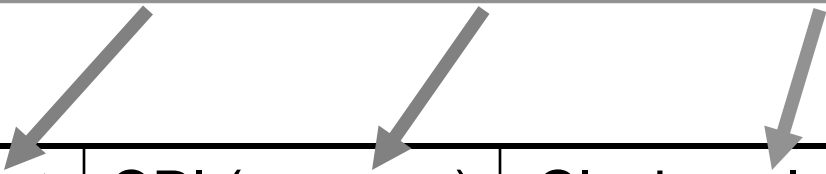
## Pipeline Implementation:



# About CPU Performance

$$\text{CPU Performance} = 1 / \text{CPUExTime}$$

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



	Instr. count	CPI (average)	Clock cycle time
Program Language	X	X	
Compiler	X	X	
ISA	X	X	
Organization		X	X
Technology			X

# Some Calculations...



- ❑ Assume a program has 1000 instructions.
  - 70% of them are R-type,
  - 10% are lw,
  - 10% are sw,
  - 10% are beq.
- ❑ Assume each of the five stages (datapath) costs 200 ps.
- ❑ What is the CPU EX time if we use single-cycle datapath?
- ❑ What if we use multi-cycle datapath w/o pipeline?
- ❑ What if we use multi-cycle datapath w/ pipeline?

# Some Calculations...

---

	Instr. count	CPI (ave.)	Clock cycle time	CPU Ex time
Single-cycle datapath				
Multi-cycle datapath w/o pipeline				
Multi-cycle datapath w/ pipeline				

# Some Calculations...

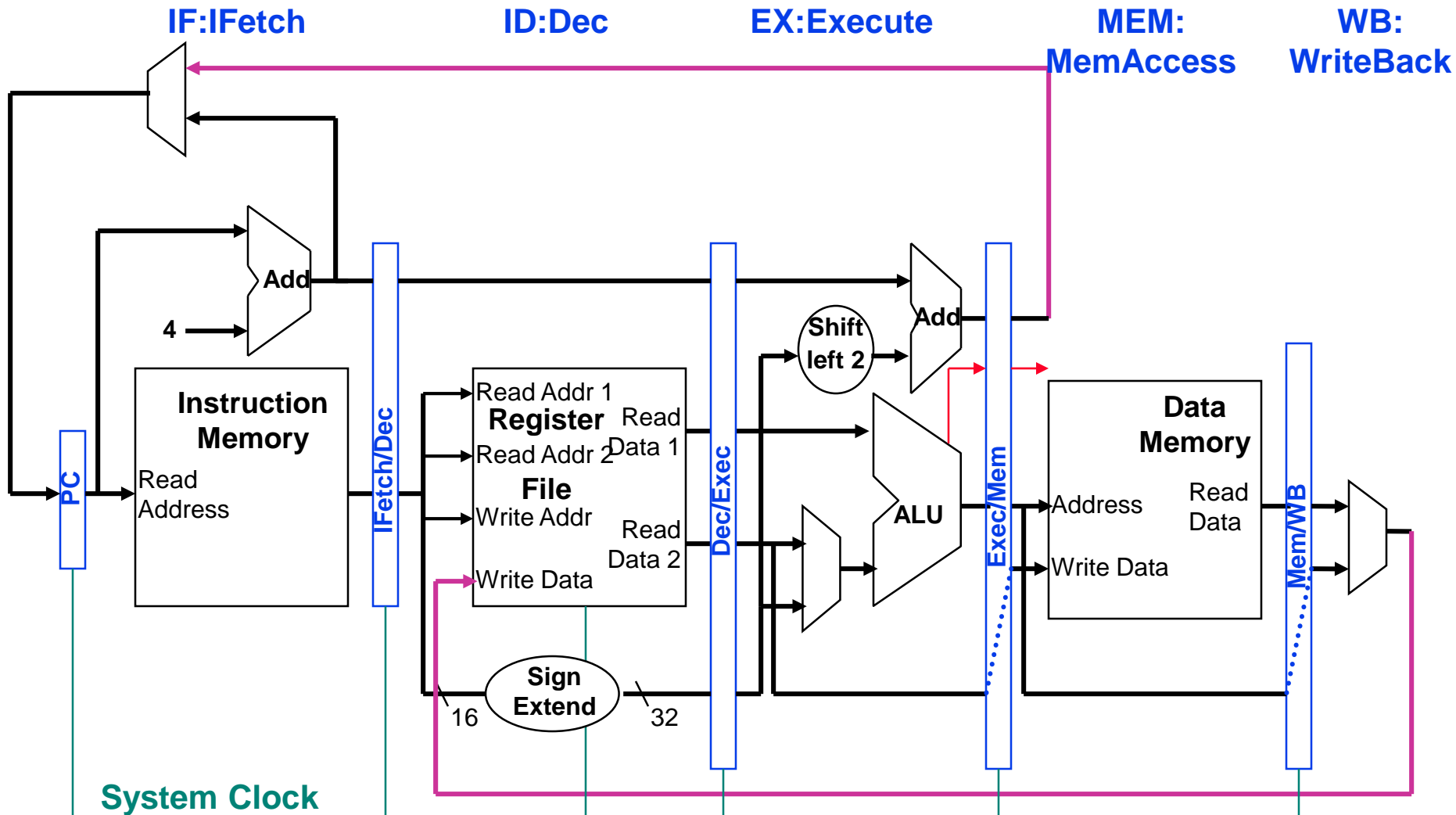


	Instr. count	CPI (ave.)	Clock cycle time	CPU Ex time
Single-cycle datapath	1000	1	200ps * 5	1000,000ps
Multi-cycle datapath w/o pipeline	1000	$(4*0.7+5*0.1+4*0.1+3*0.1) = 4$	200ps	800,000ps
Multi-cycle datapath w/ pipeline	1000	$\approx 1$	200ps	200,000ps

# MIPS Pipeline Datapath Modifications



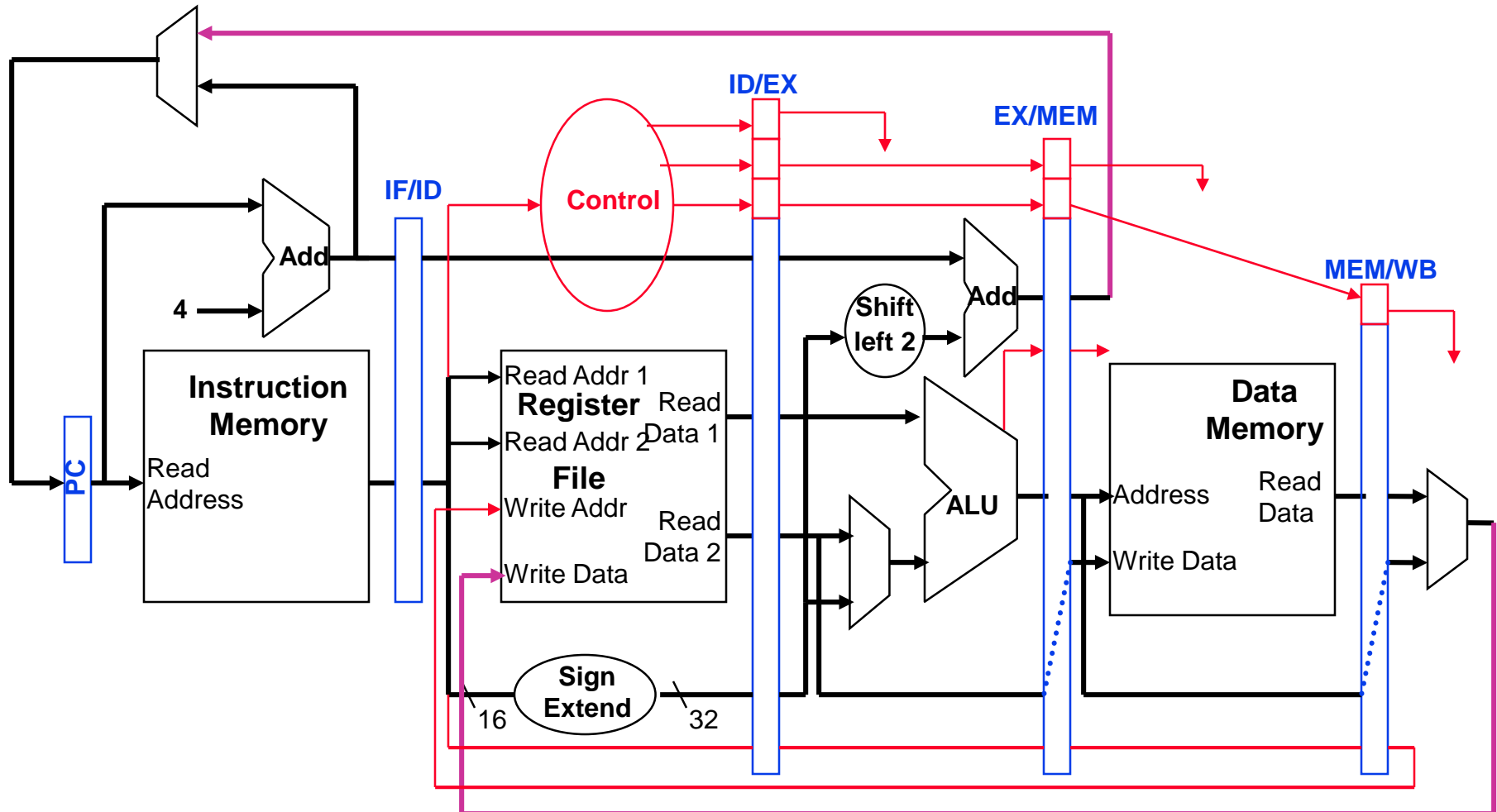
- ❑ What do we need to add/modify in our MIPS datapath?
  - pipeline registers between each pipeline stage to **isolate** them



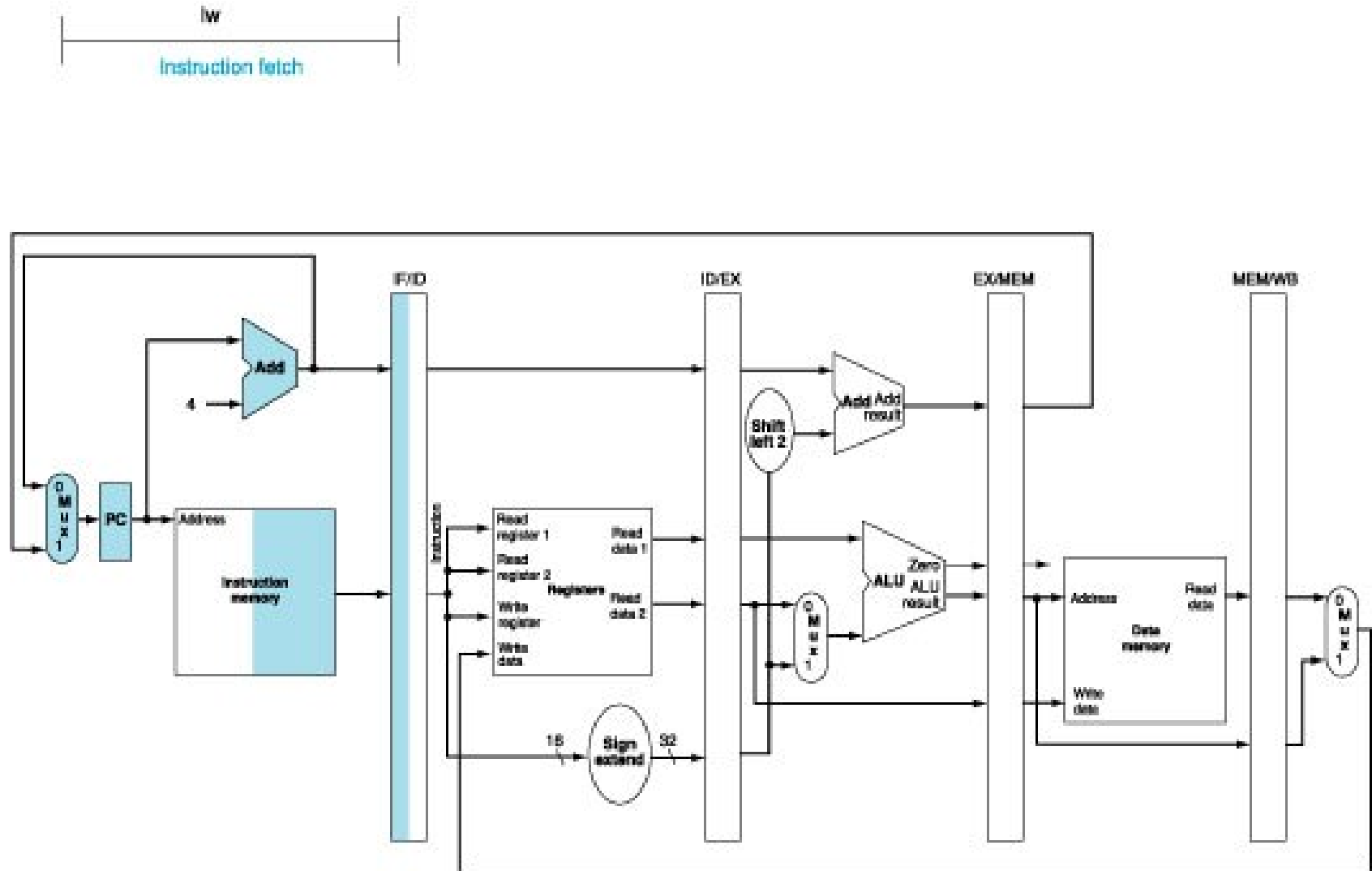
# MIPS Pipeline Control Path Modifications



- ❑ All control signals can be determined during Decode
  - and held in the **pipeline registers** between pipeline stages

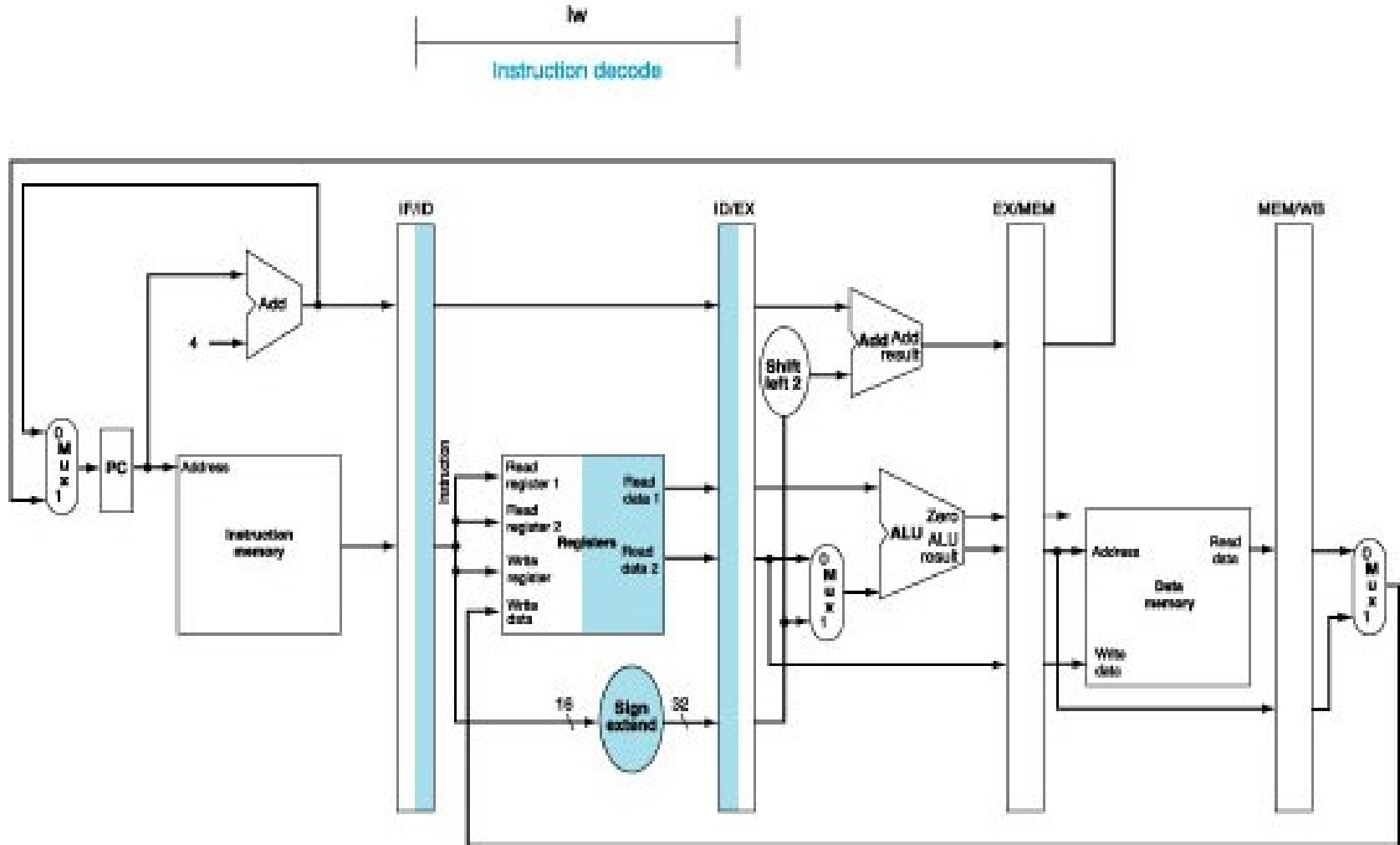


# The walk-through of the load instruction (1/5)

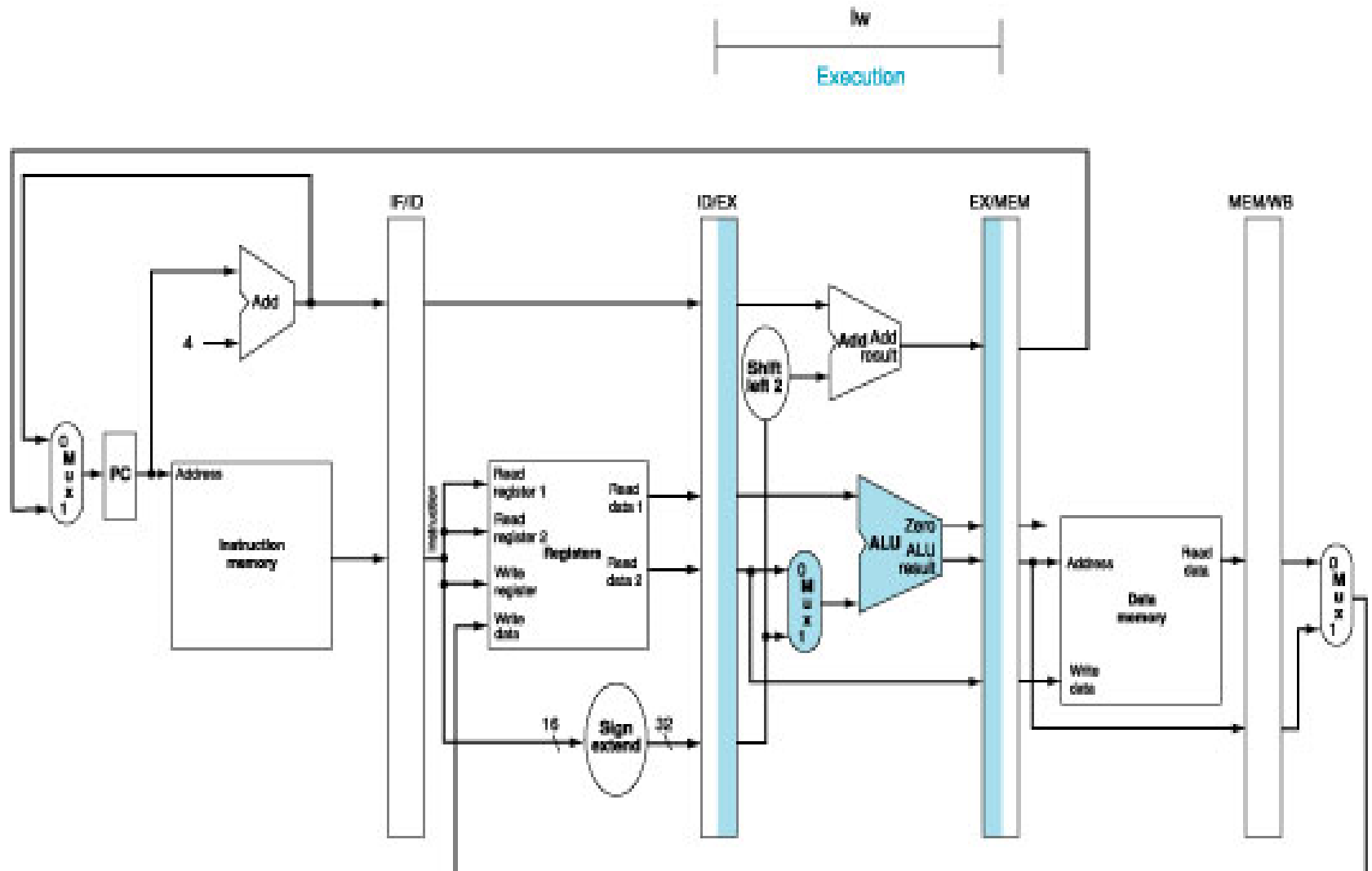




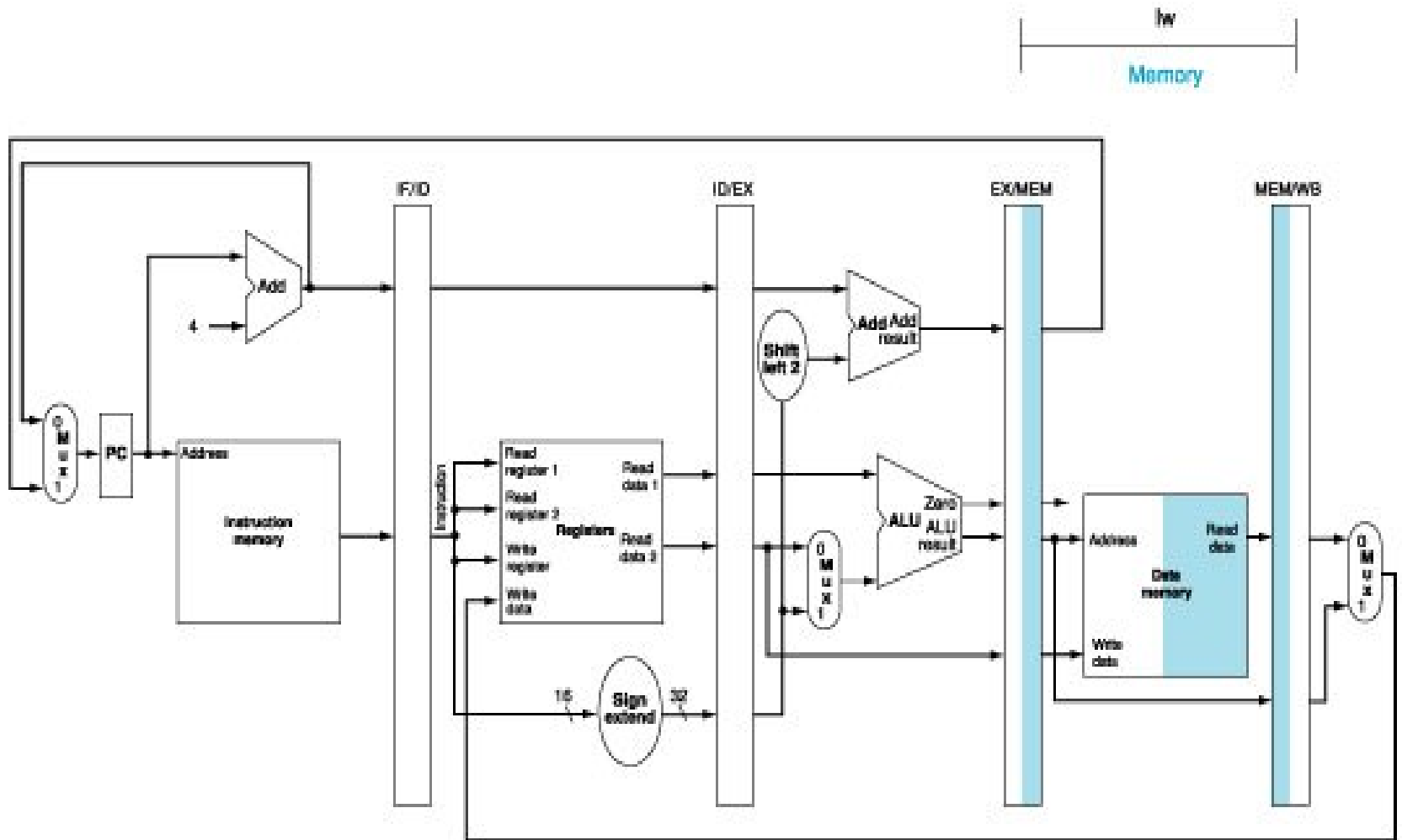
# The walk-through of the load instruction (2/5)



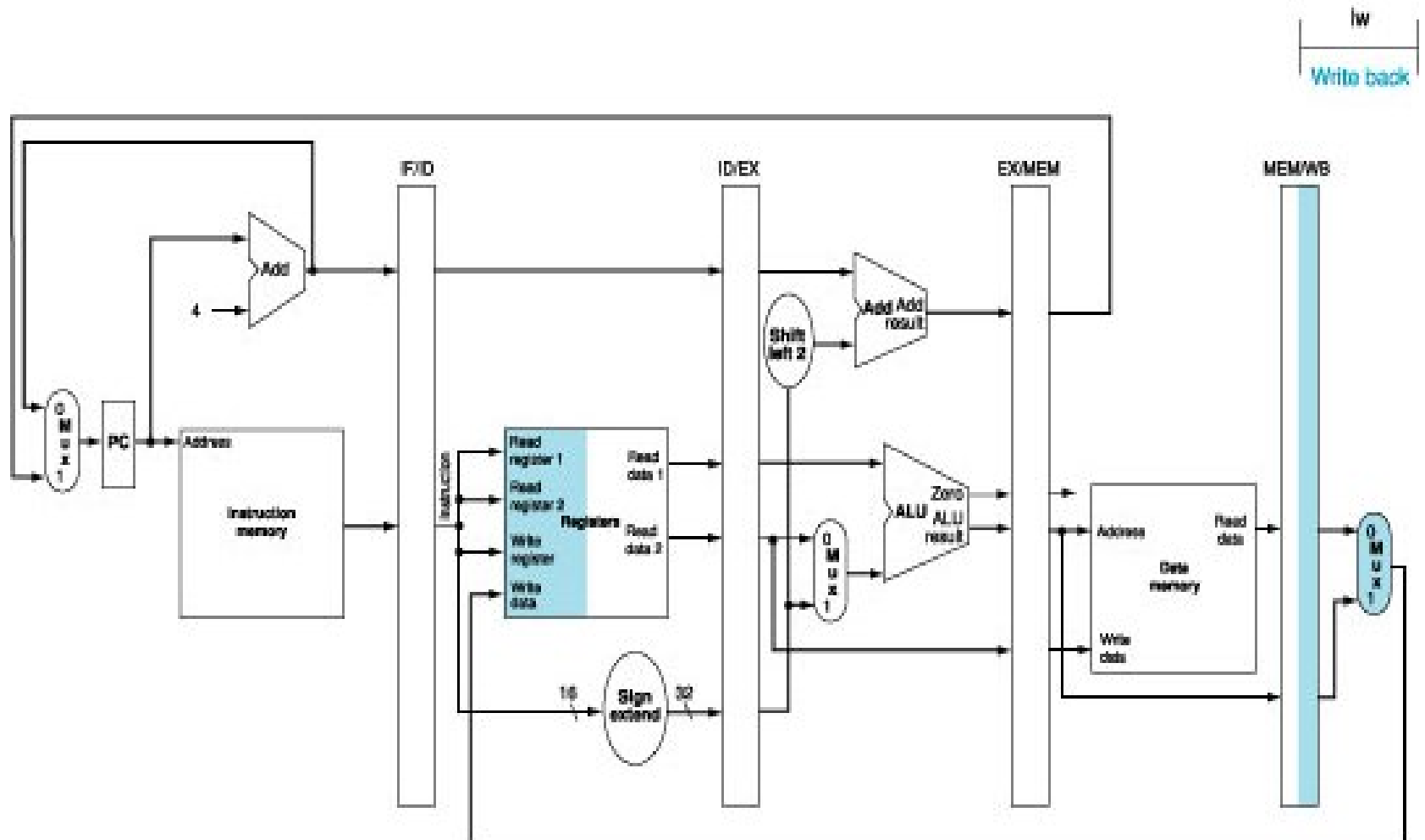
# The walk-through of the load instruction (3/5)



# The walk-through of the load instruction (4/5)



# The walk-through of the load instruction (5/5)



# Pipelining the MIPS ISA



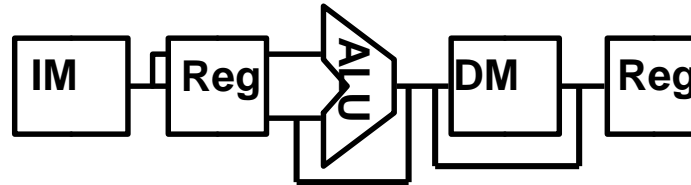
## ❑ What makes it easy

- all instructions are the same length (32 bits)
  - can fetch in the 1<sup>st</sup> stage and decode in the 2<sup>nd</sup> stage
- few instruction formats (three) with **symmetry** across formats
  - can begin reading register file in 2<sup>nd</sup> stage
- memory operations can occur only in loads and stores
  - can use the execute stage to calculate memory addresses
- each MIPS instruction writes at most one result (i.e., changes the machine state) and does so near the end of the pipeline (MEM and WB)

## ❑ What makes it hard

- **structural hazards**: what if we had only one memory?
- **control hazards**: what about branches?
- **data hazards**: what if an instruction's input operands depend on the output of a previous instruction?

# Graphically Representing MIPS Pipeline

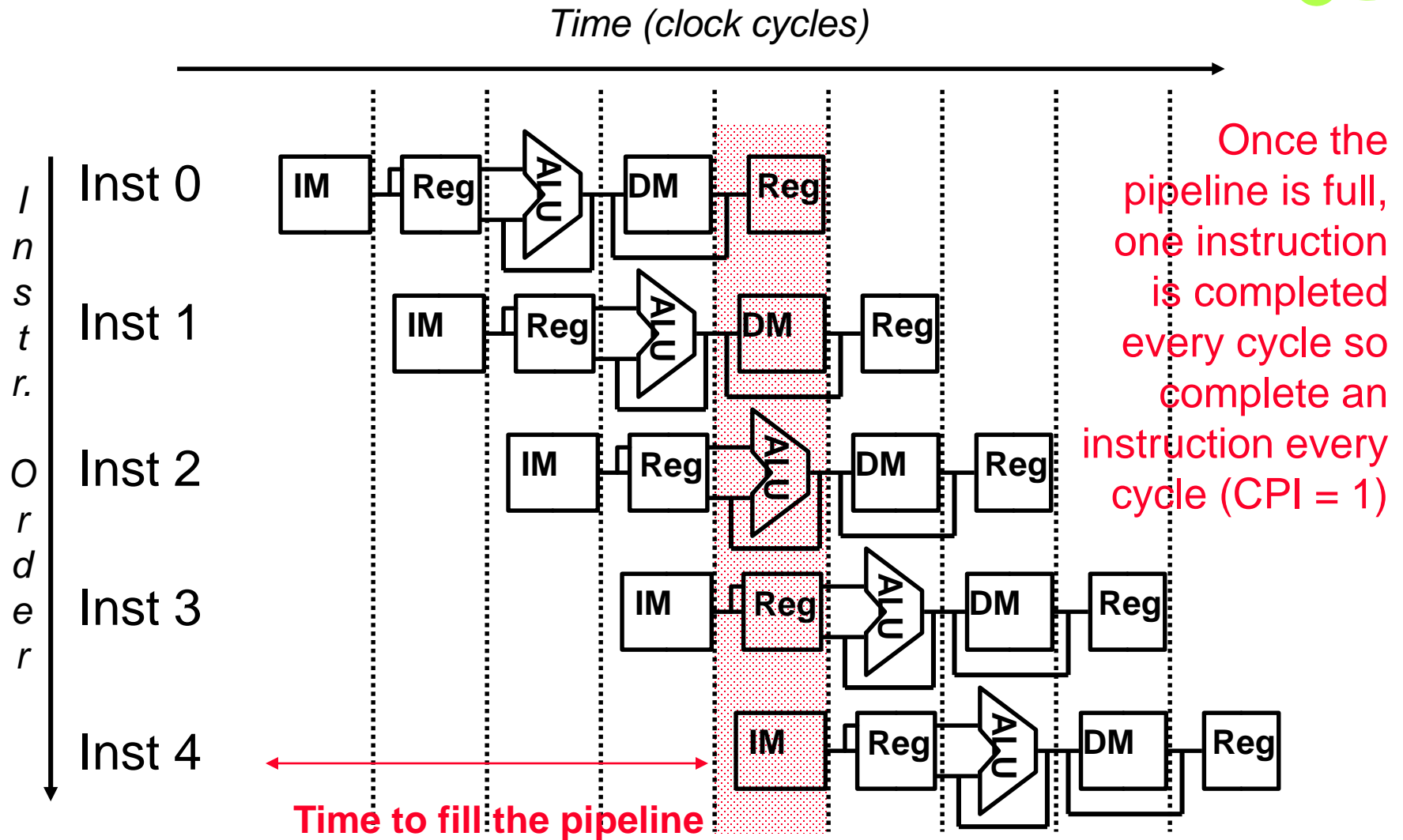


□ Can help with answering questions like:

- How many cycles does it take to execute this code?
- What is the ALU doing during cycle 4?
- Is there a hazard, why does it occur, and how can it be fixed?



# Why Pipeline? For Performance!



# Can Pipelining Get Us Into Trouble?



## □ Yes: Pipeline Hazards

- **structural hazards**: attempt to use the same resource by two different instructions at the same time
- **data hazards**: attempt to use data before it is ready
  - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
- **control hazards**: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
  - branch and jump instructions, exceptions

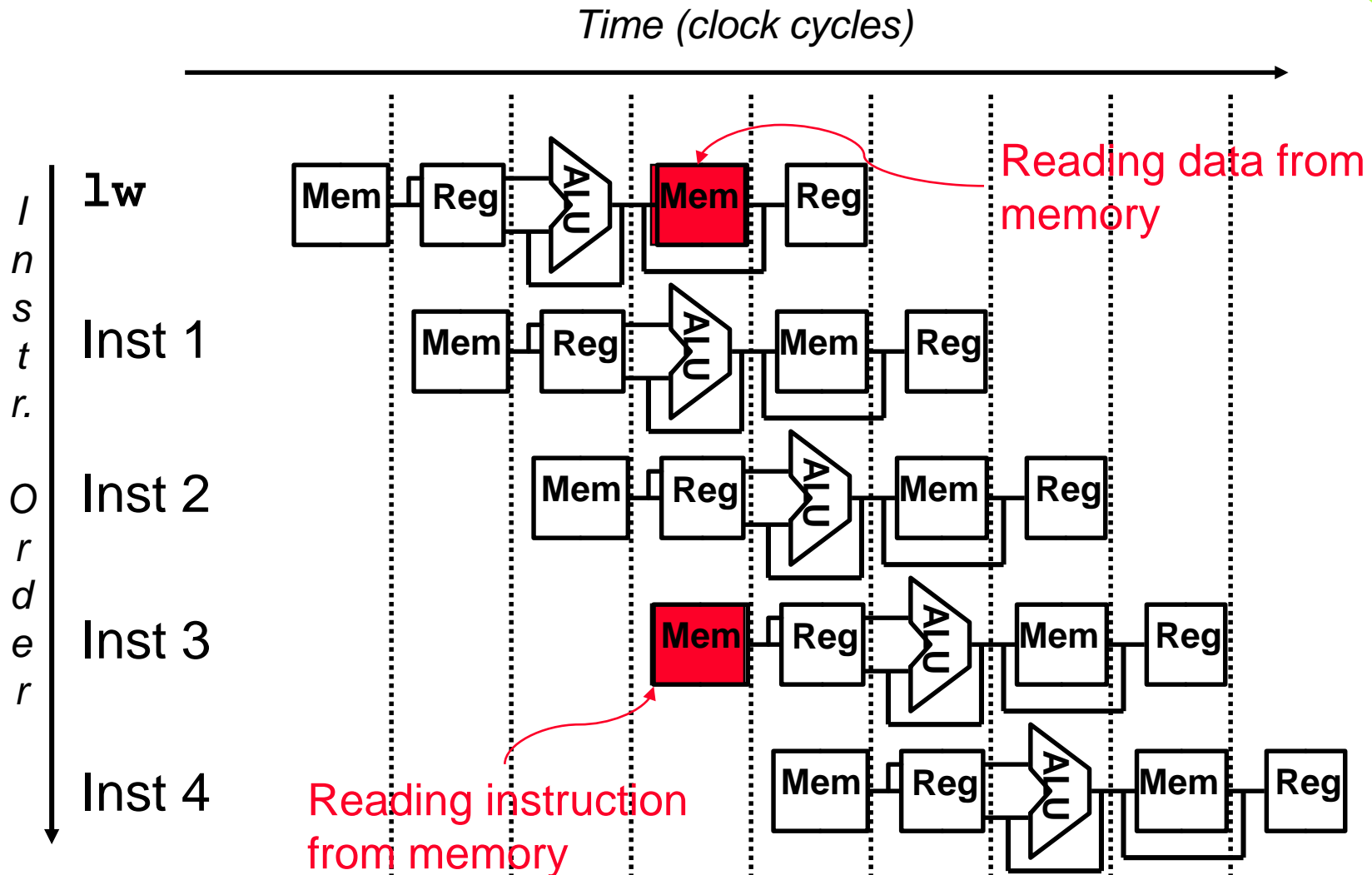
□ Pipeline control must **detect** the hazard

□ And take action to **resolve** hazards

□ Can always resolve hazards by **waiting**

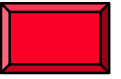


# A Single Memory Would Be a Structural Hazard

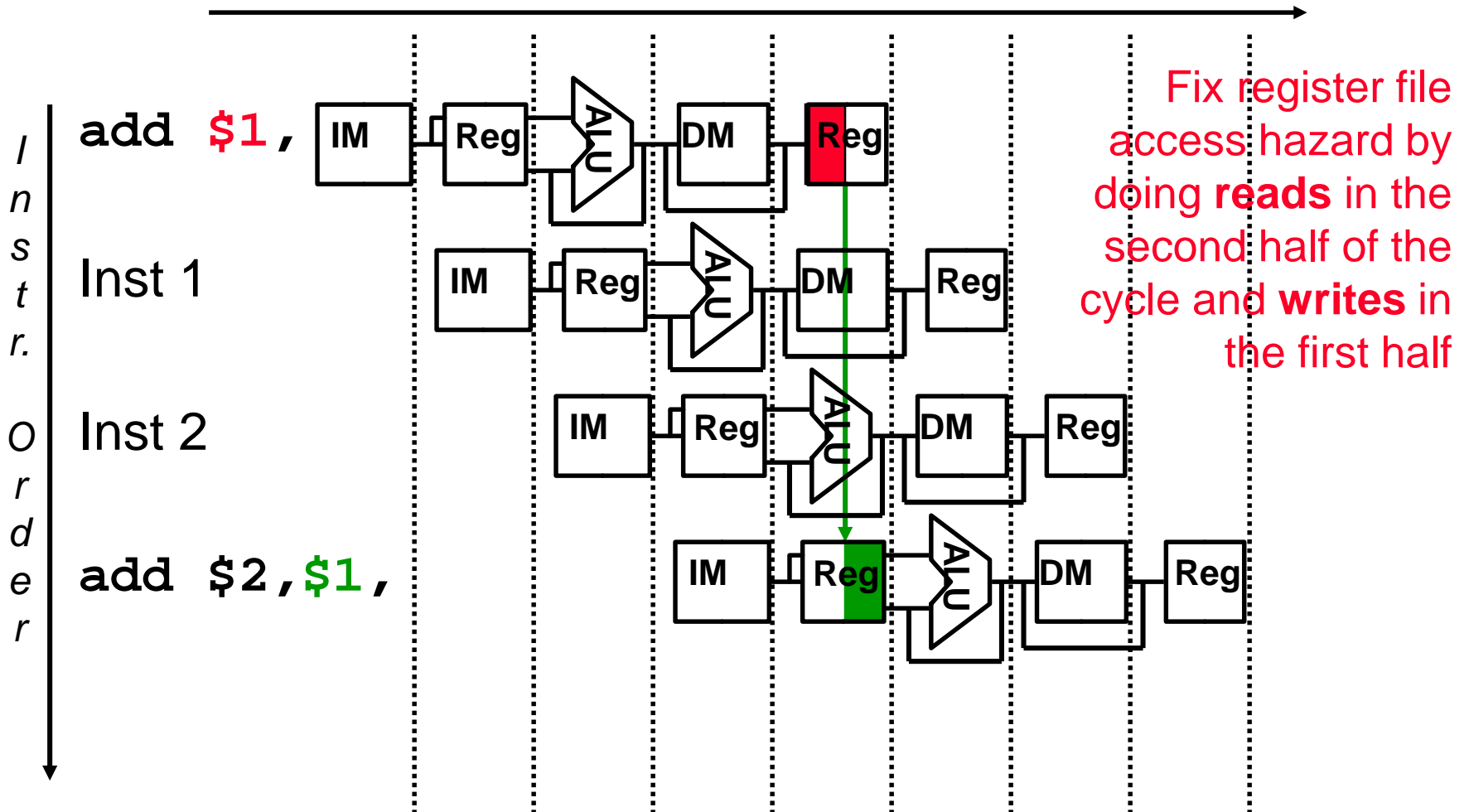


❑ Can fix with separate instr and data memories

# How About Register File Access?



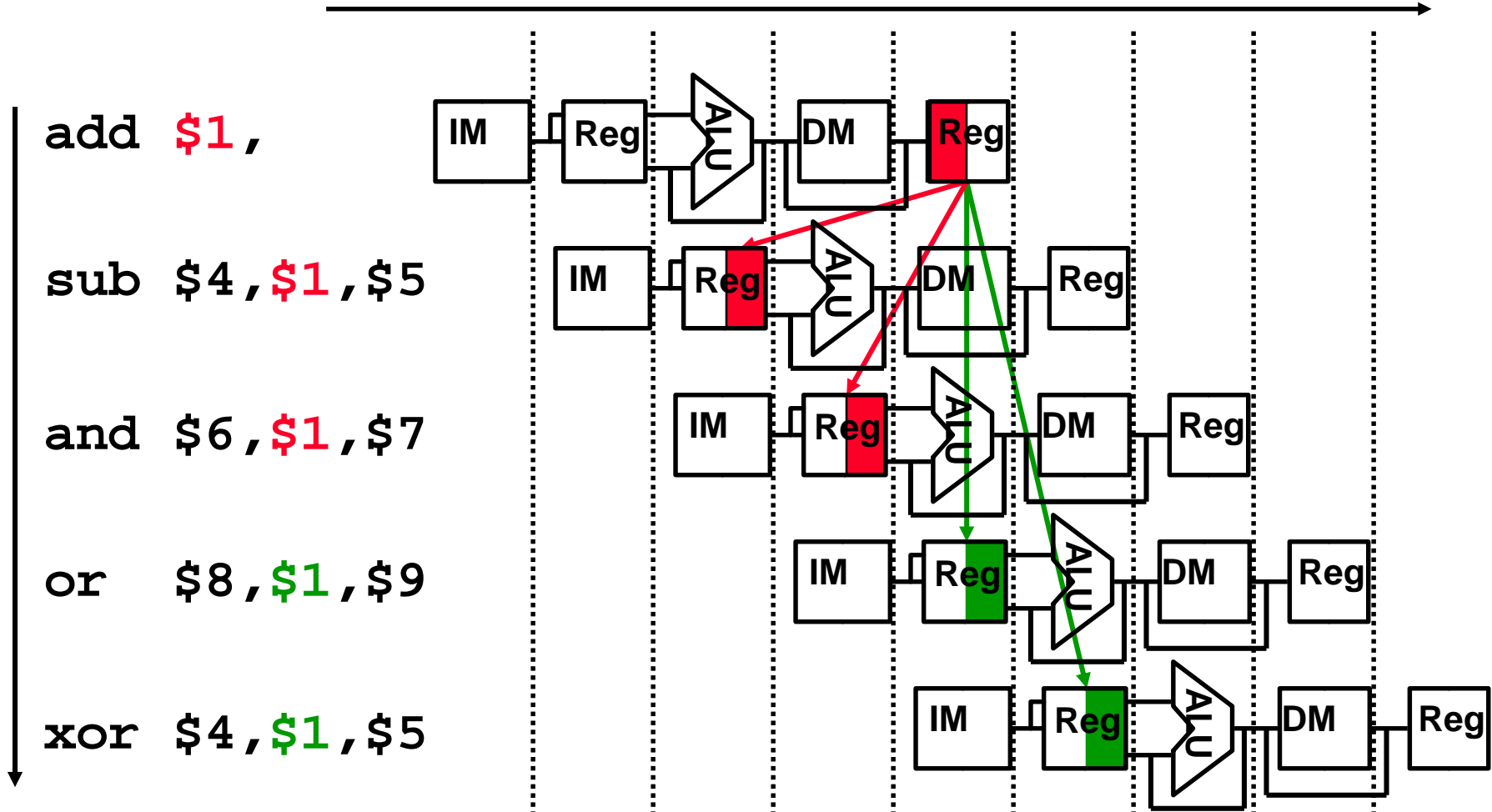
Time (clock cycles)





# Register Usage Can Cause Data Hazards

□ Dependencies backward in time cause **hazards**

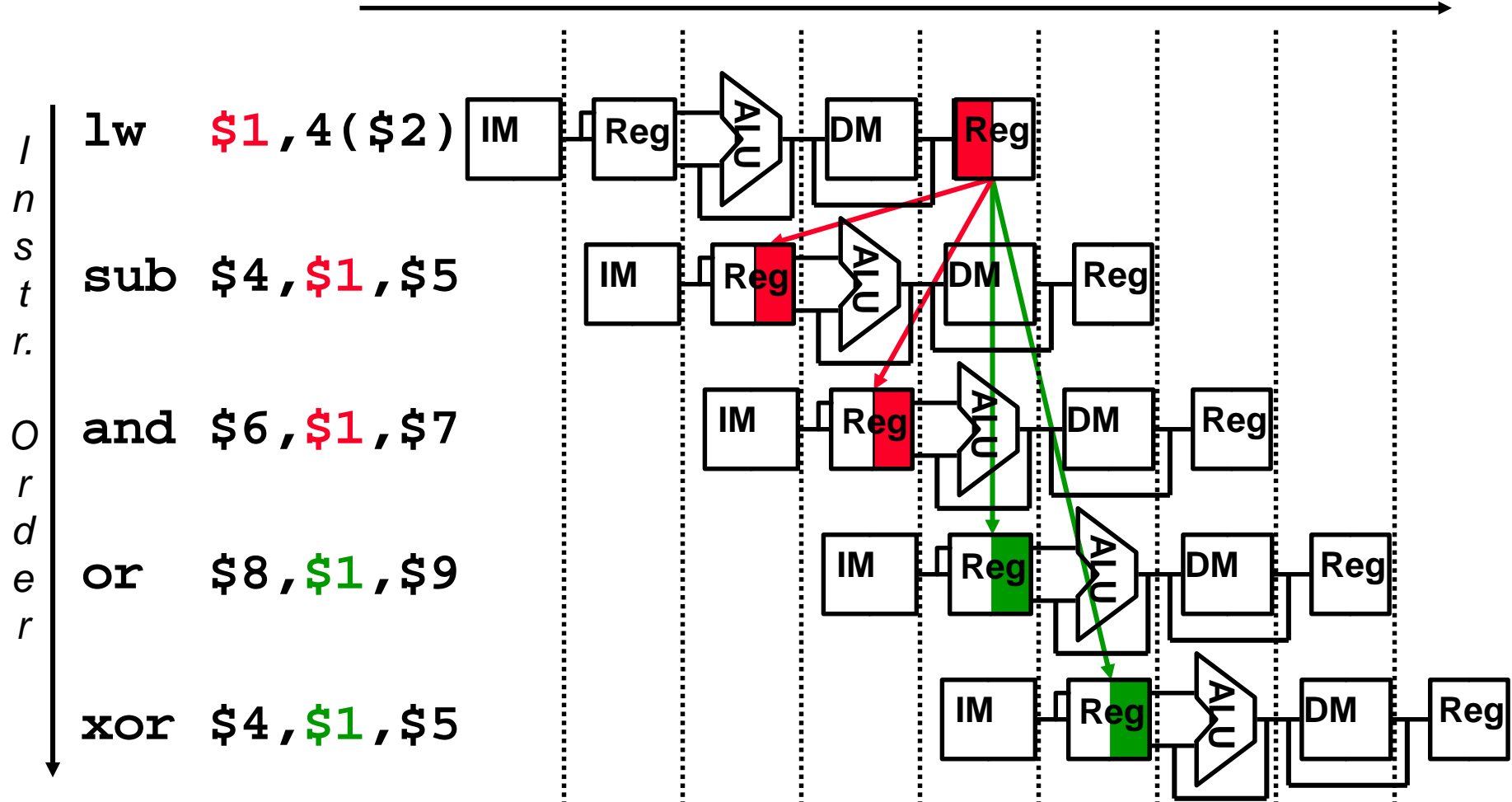


□ **Read before write data hazard**



# Loads Can Cause Data Hazards

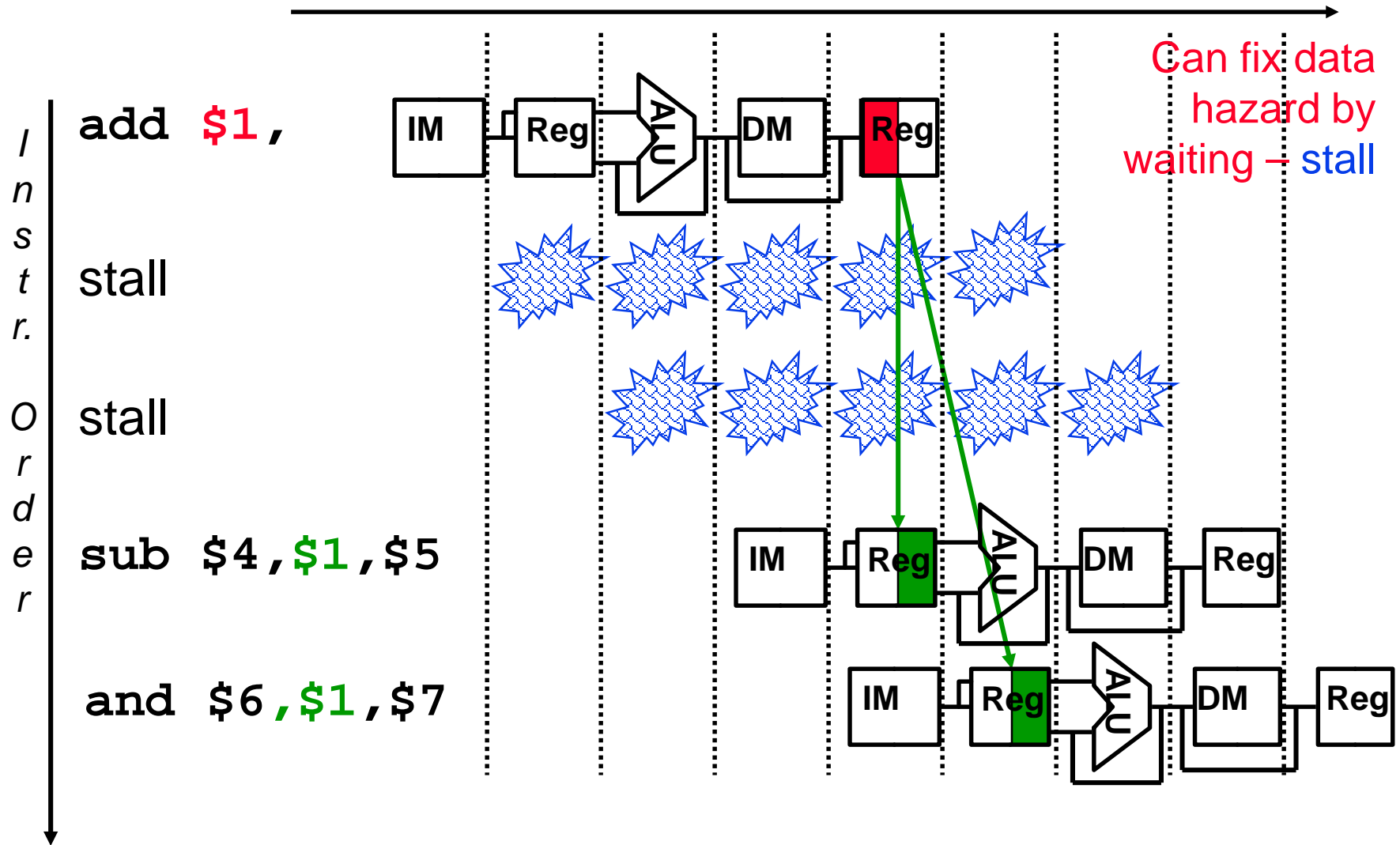
□ Dependencies backward in time cause **hazards**



□ **Load-use data hazard**

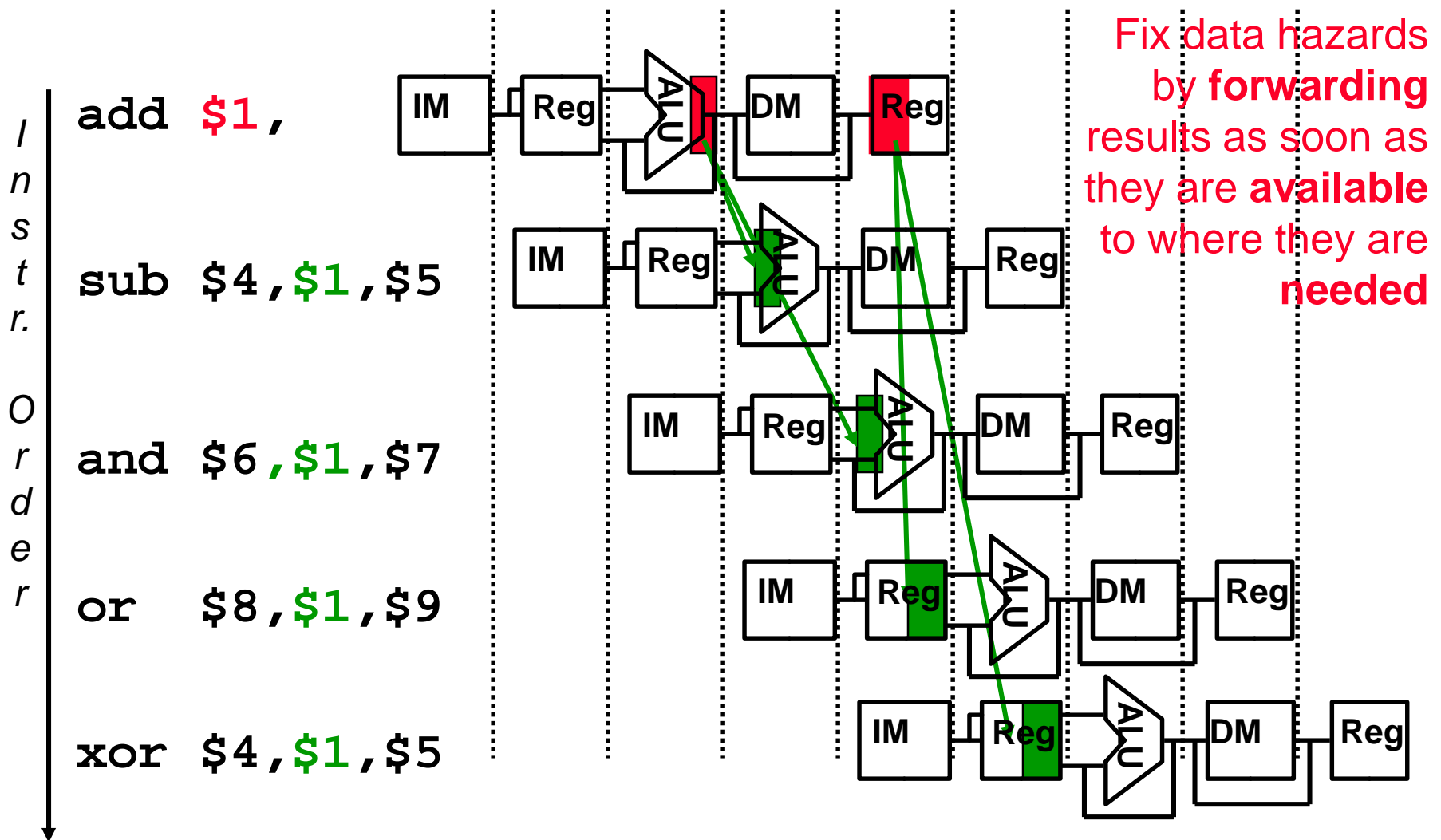


# One Way to “Fix” a Data Hazard

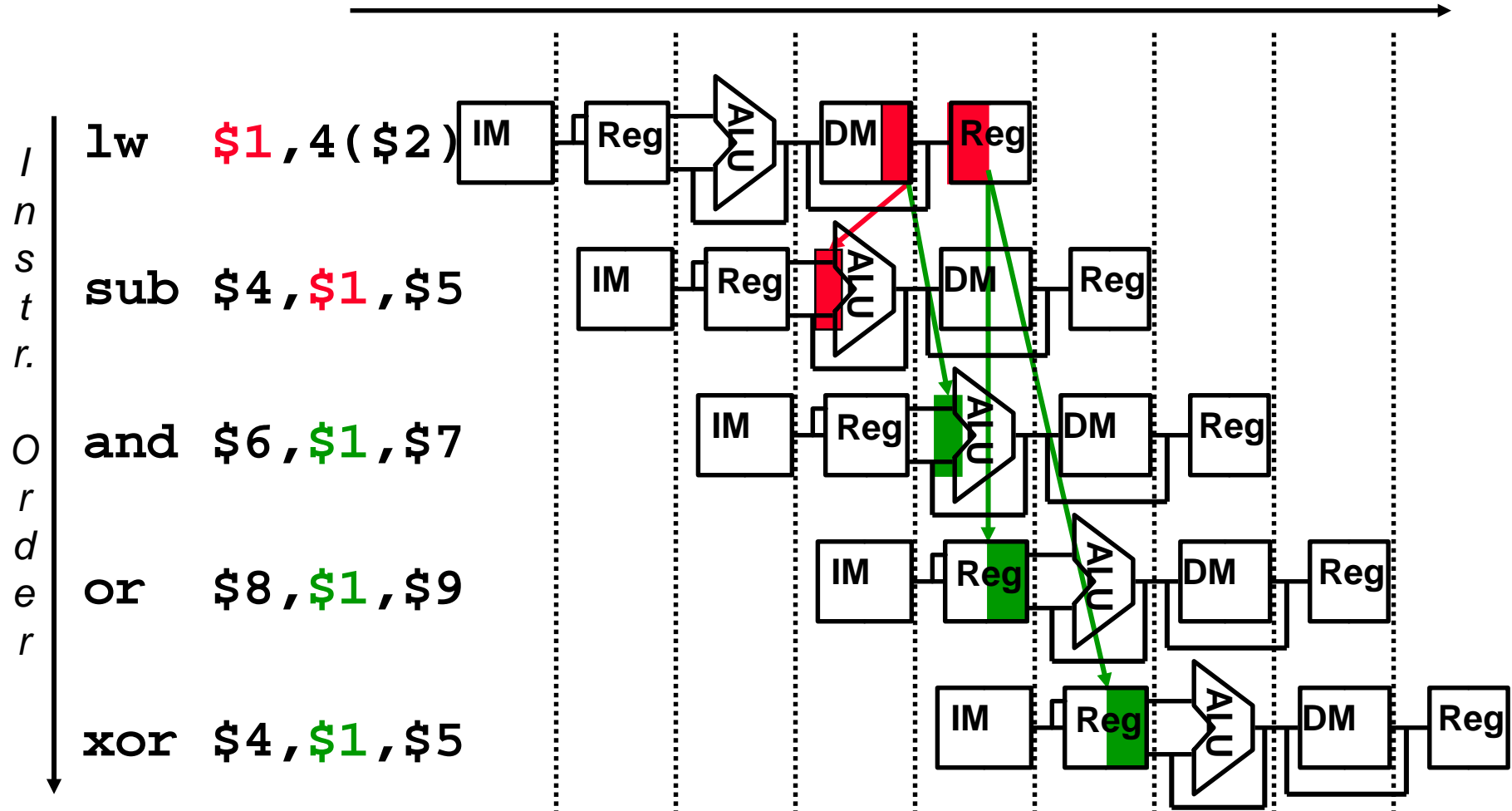
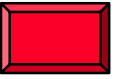




# Another Way to “Fix” a Data Hazard



# Forwarding with Load-use Data Hazards



❑ Will still need **one stall cycle** even with forwarding

# Control Hazards



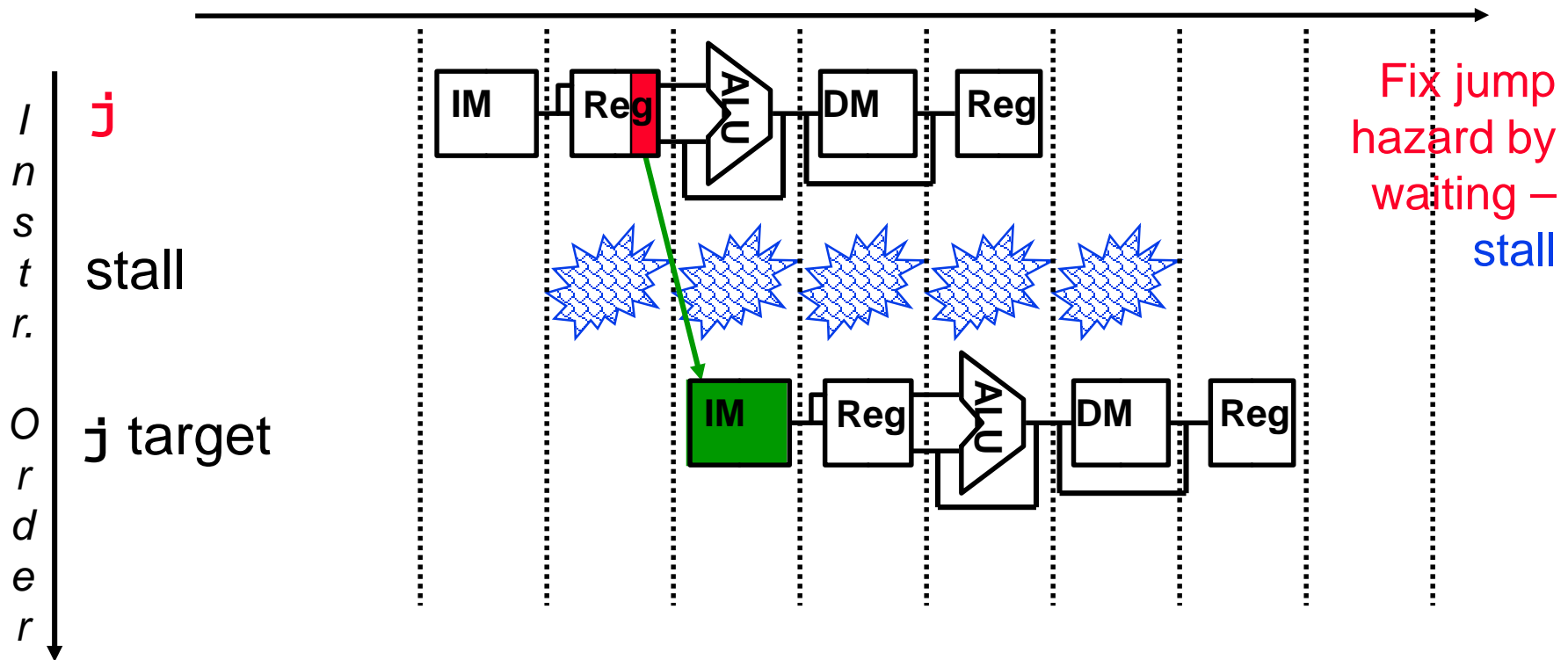
- ❑ When the flow of instruction addresses is not sequential (i.e.,  $PC = PC + 4$ )
  - Conditional branches (beq, bne)
  - Unconditional branches (j, jal, jr)
  - Exceptions
- ❑ Possible “solutions”
  - Stall (impacts performance)
  - Move branch decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
  - Delayed decision (requires compiler support)
  - Predict and hope for the best !
- ❑ Control hazards occur less frequently than data hazards, but there is *nothing* as effective against control hazards as forwarding is for data hazards





# Jumps Incur One Stall

- ❑ Jumps not decoded until ID, so one **stall** is needed

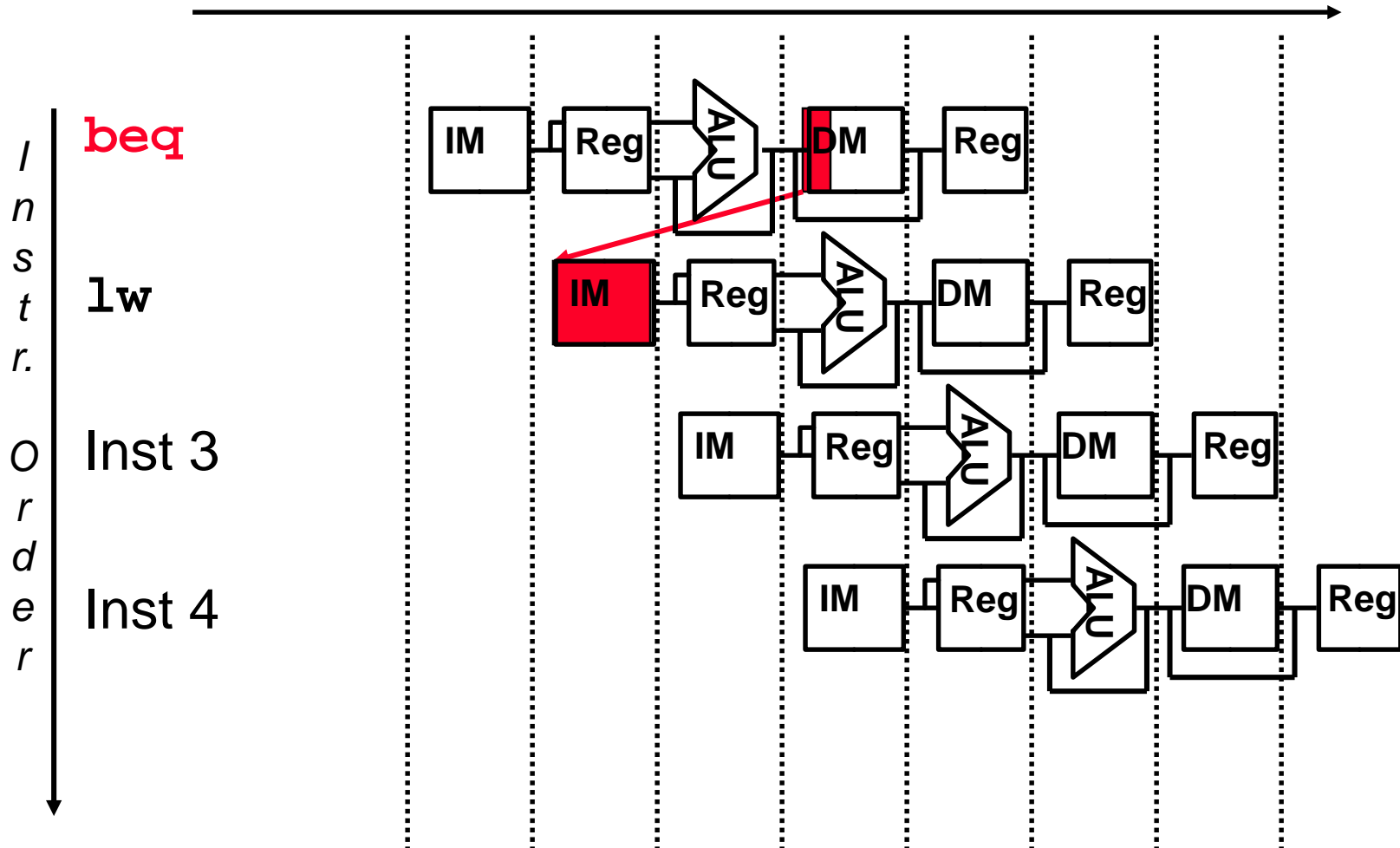


- ❑ Fortunately, jumps are very infrequent – only 3% of the SPECint instruction mix

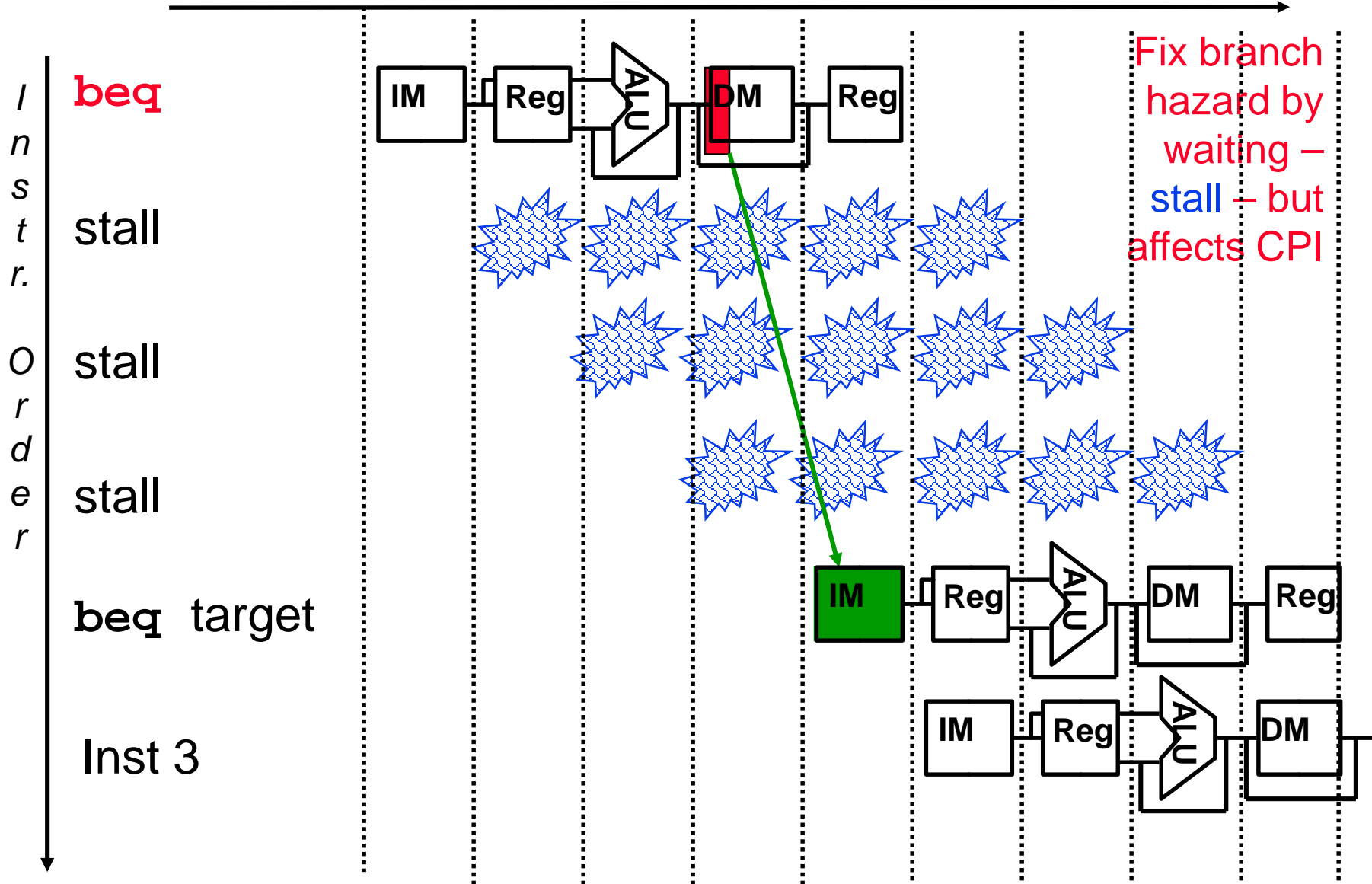


# Branches Cause Control Hazards

- Dependencies backward in time cause **hazards**



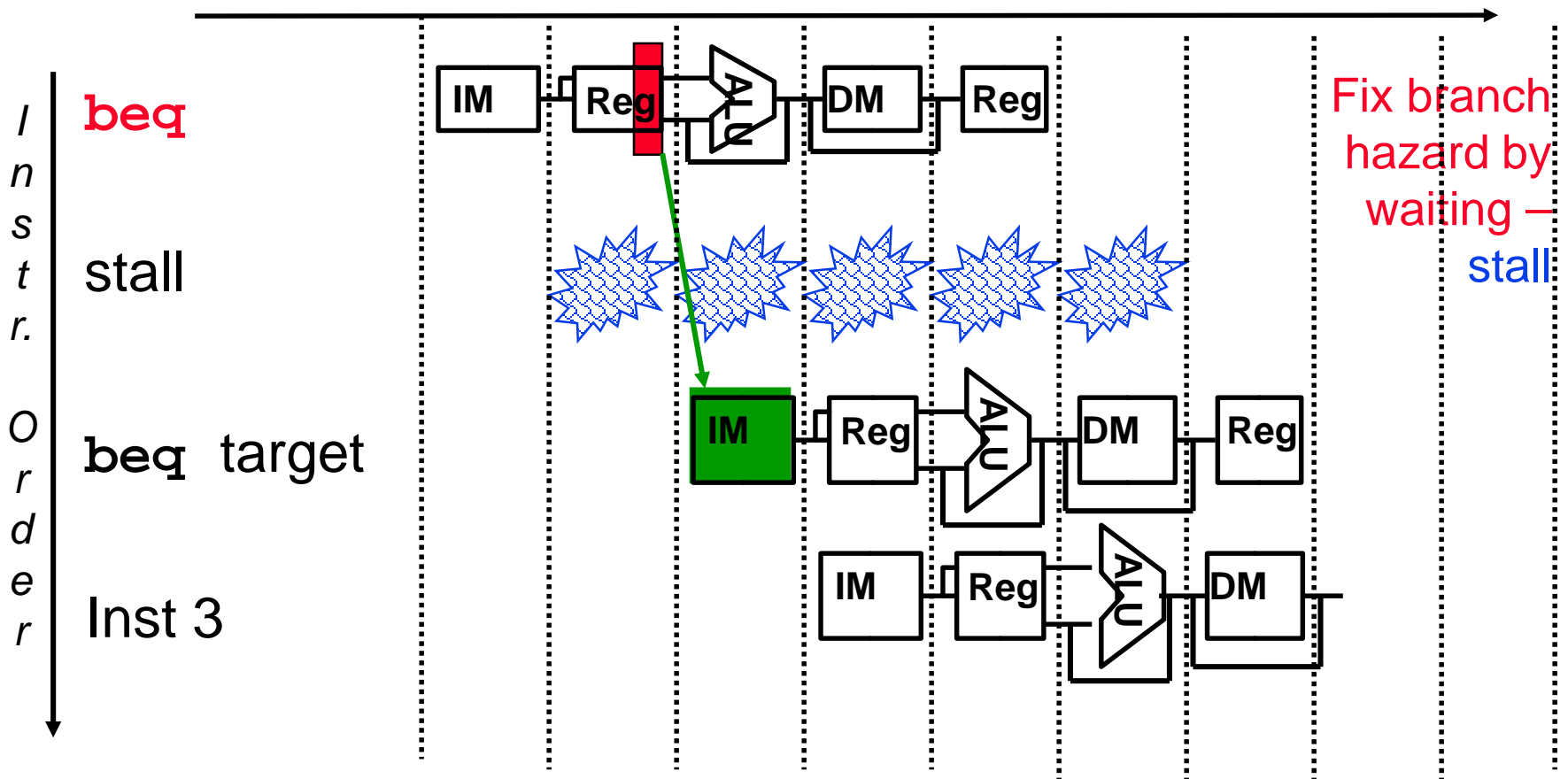
# One Way to “Fix” a Branch Control Hazard



# Another Way to “Fix” a Branch Control Hazard

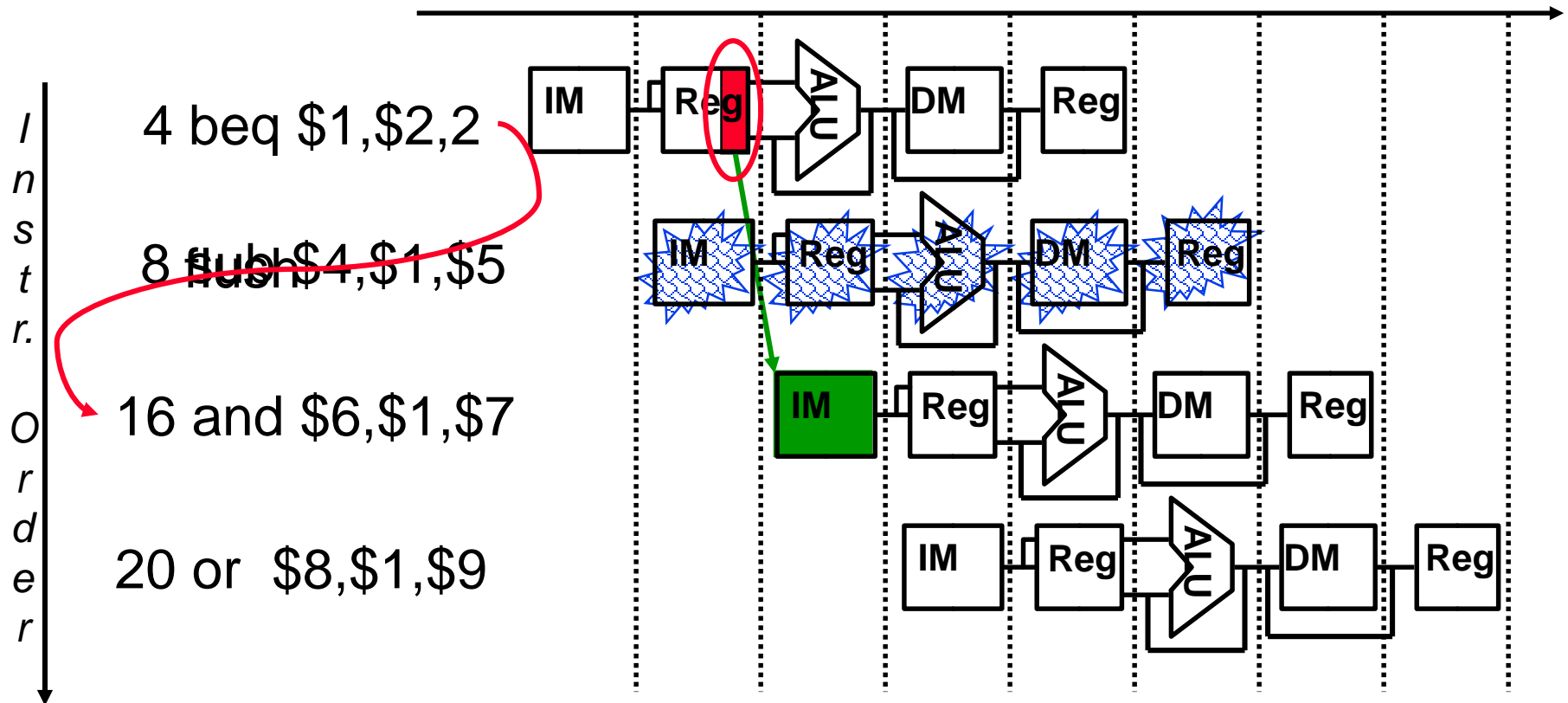


- ❑ Move branch decision hardware back to as **early** in the pipeline as possible – i.e., during the decode cycle



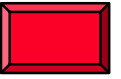
## Yet Another Way to “Fix” a Control Hazard

- ❑ “Predict branches are always not taken – and take corrective action when wrong (i.e., taken)”



- ❑ To flush, set `IF.Flush` to zero the instruction field of the IF/ID pipeline register (turning it into a `noop`)

# Two “Types” of Stalls

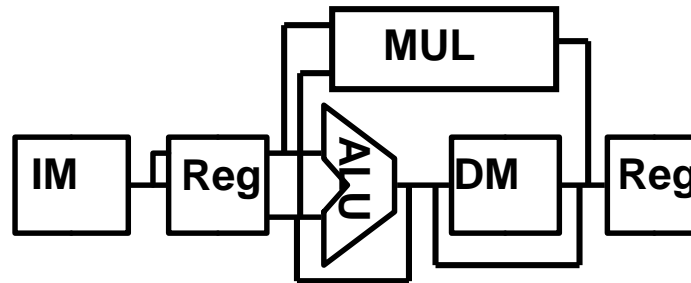


- ❑ Noop instruction (or bubble) **inserted** between two instructions in the pipeline (e.g., load-use hazards)
  - Keep the instructions *earlier* in the pipeline (later in the code) from progressing down the pipeline for a cycle (“bounce” them in place with write control signals)
  - Insert `noop` instruction by zeroing control bits in the pipeline register at the appropriate stage
- ❑ Flushes (or instruction squashing) were an instruction in the pipeline is **replaced** with a `noop` instruction (as done for instructions located sequentially after `j` and `beq` instructions)
  - Zero the control bits for the instruction to be flushed

## Many Other Pipeline Structures Are Possible

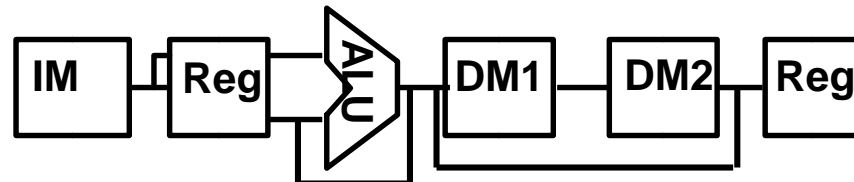
### ❑ What about the (slow) multiply operation?

- Make the clock twice as slow or ...
- let it take two cycles (since it doesn't use the DM stage)



### ❑ What if the data memory access is twice as slow as the instruction memory?

- make the clock twice as slow or ...
- let data memory access take two cycles (and keep the same clock rate)





# Pipelining Summary

- ❑ All modern day processors use pipelining
- ❑ Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- ❑ Potential speedup: a really fast clock cycle and able to complete one instruction every clock cycle (CPI)
- ❑ Pipeline rate limited by **slowest** pipeline stage
  - Unbalanced pipe stages makes for inefficiencies
  - The time to “**fill**” pipeline and time to “**drain**” it can impact speedup for deep pipelines and short code runs
- ❑ Must detect and resolve hazards
  - Stalling negatively affects CPI (makes CPI less than the ideal of 1)