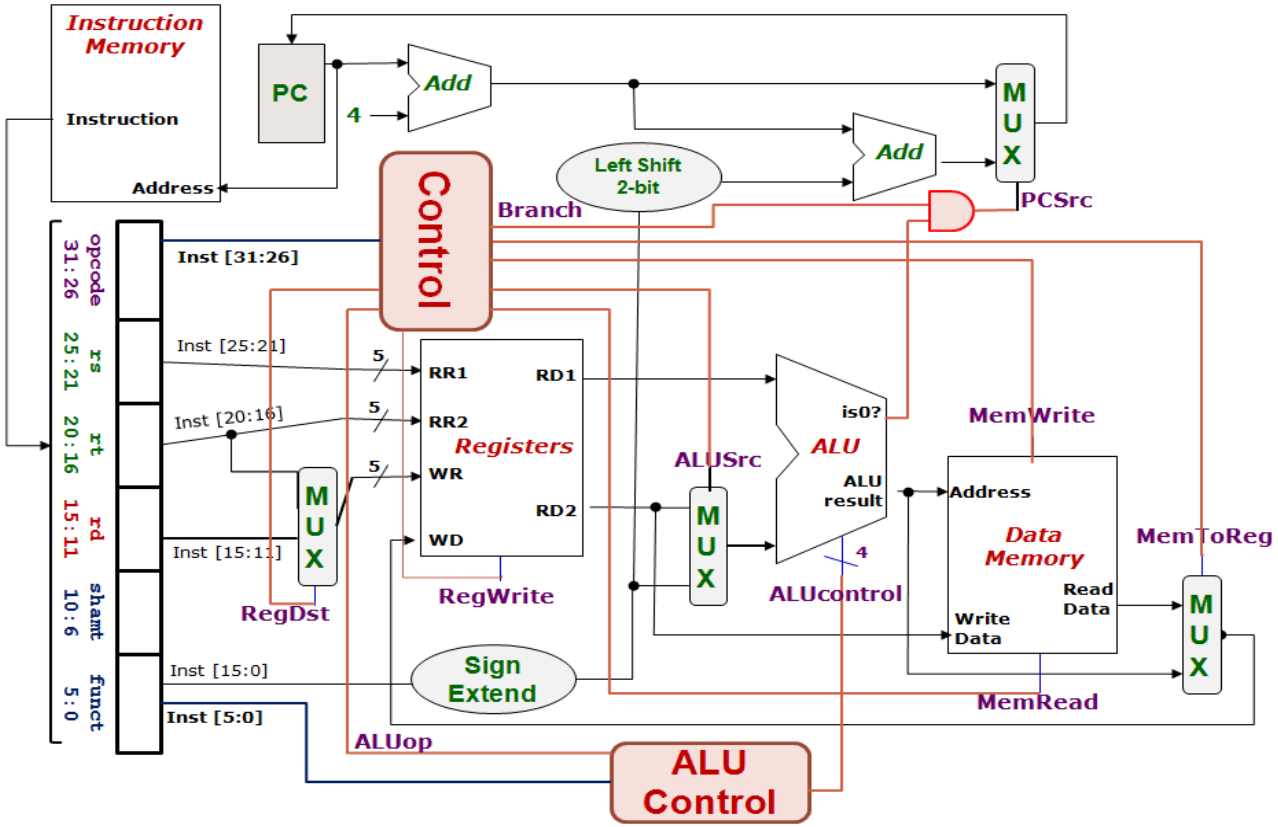
ELT3047 Computer Architecture

Lecture 9: Pipelined Processor Design

Hoang Gia Hung
Faculty of Electronics and Telecommunications
University of Engineering and Technology, VNU Hanoi

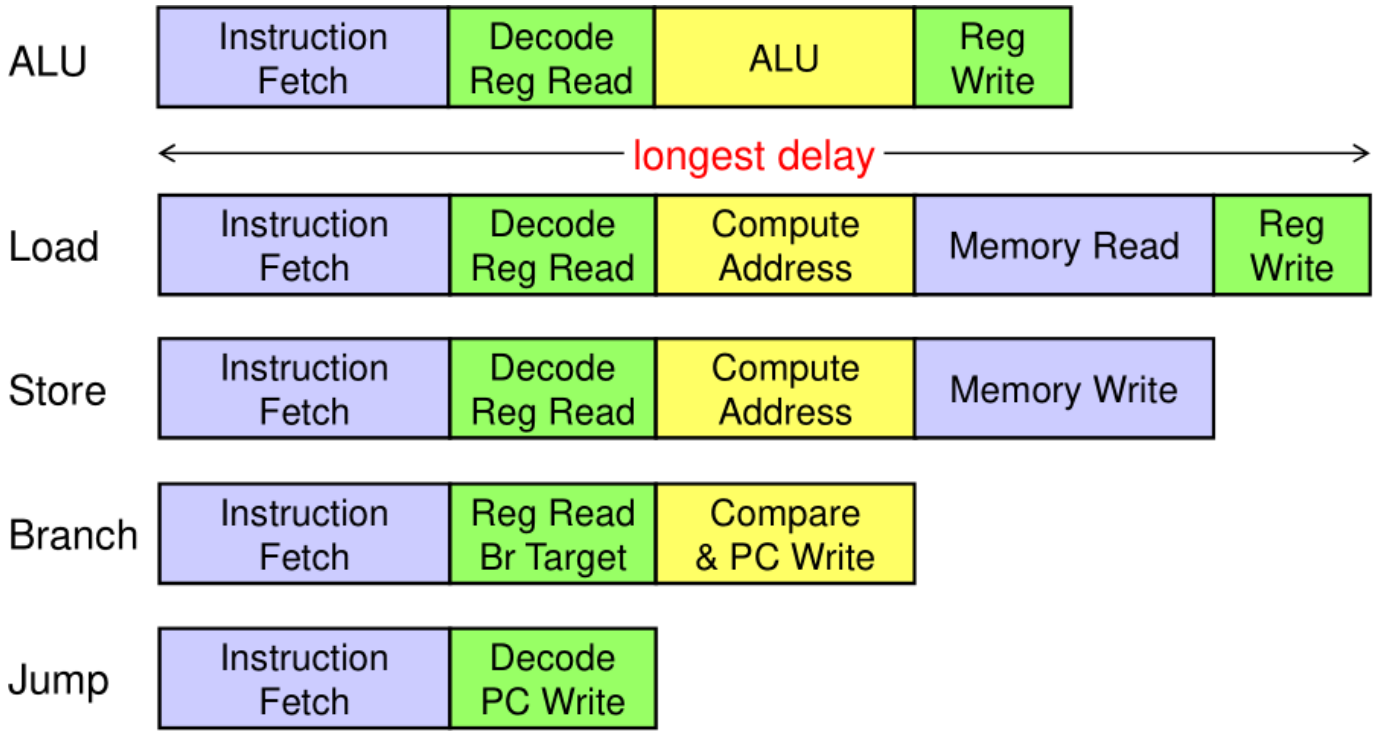
Last lecture review

- Typical design steps for control units:
  - Fill in truth table
    - **Input**: opcode; **Output**: various control signals as discussed
  - Derive simplified expression for each signal



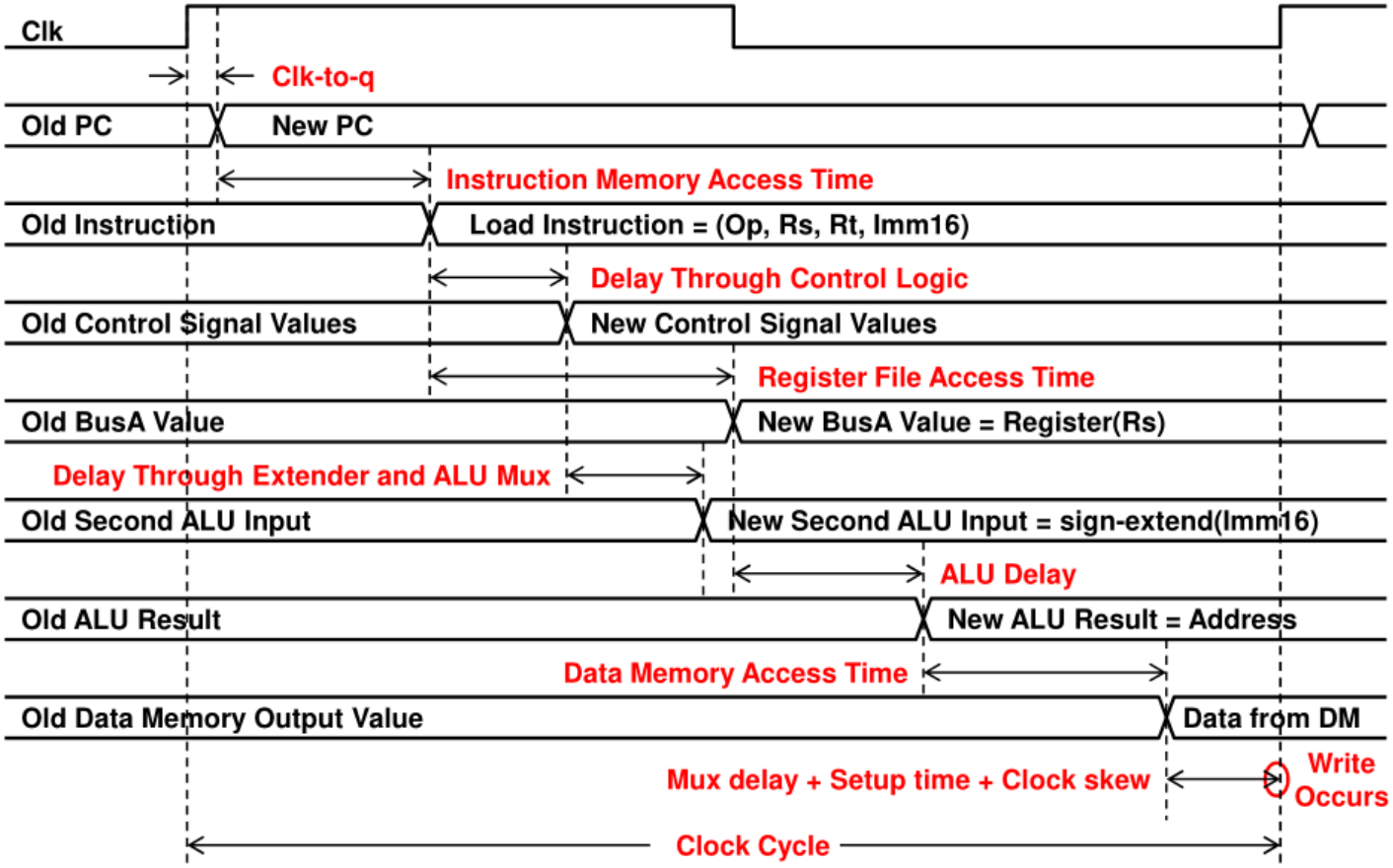
Drawbacks of Single Cycle Processor

- Long cycle time
  - All instructions take as much time as the slowest instruction



Worst Case Timing

- Slowest instruction: **load**



- Cycle time is **longer than needed** for other instructions

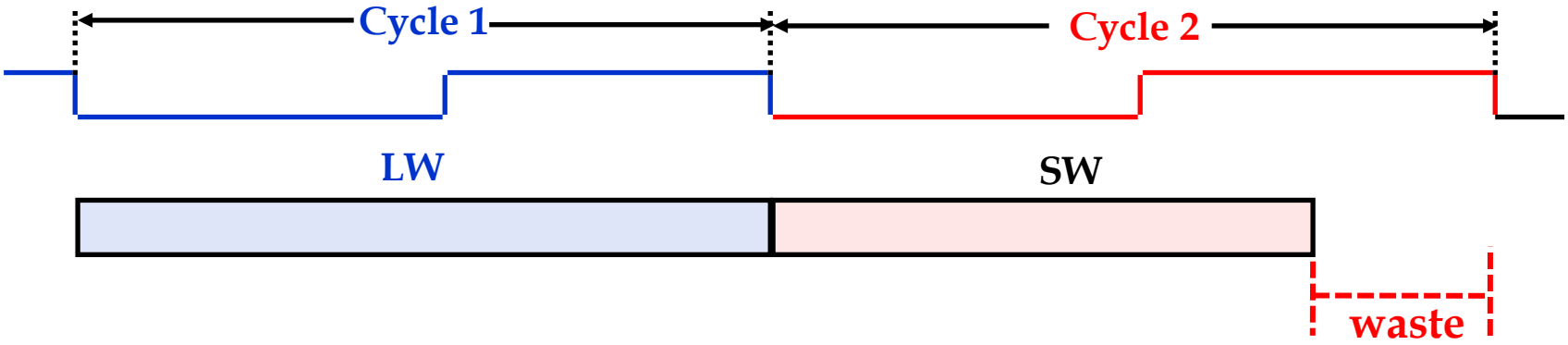
Multicycle Implementation

- Break instruction execution into five steps
  - Instruction fetch
  - Instruction decode, register read, target address for jump/branch
  - Execution, memory address calculation, or branch outcome
  - Memory access or ALU instruction completion
  - Load instruction completion
- **One clock cycle per step** (clock cycle is reduced)
  - First 2 steps are the same for all instructions

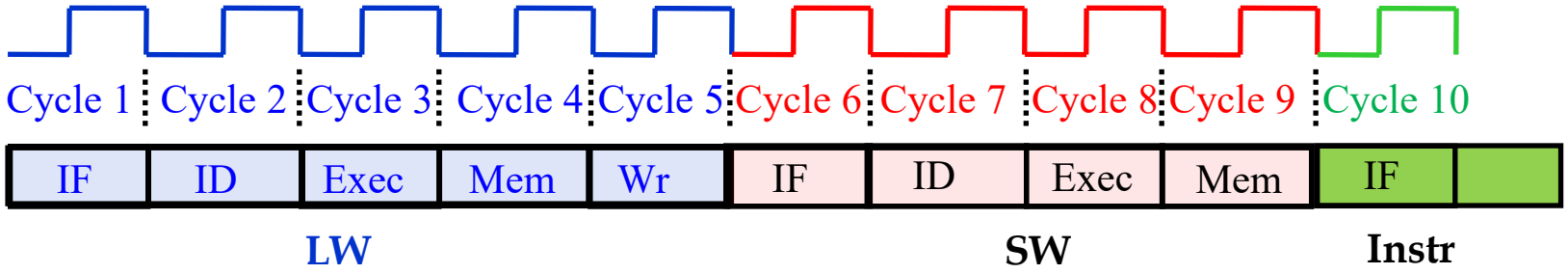
Instruction	# cycles	Instruction	# cycles
ALU & Store	4	Branch	3
Load	5	Jump	2

Single cycle vs. multicycle example

- Single cycle



- Multicycle



  - Shorter clock cycle time: constrained by longest step, not longest instruction
  - Higher overall performance: simpler instructions take fewer cycles, less waste

Single cycle vs. Multicycle

- Assume the following operation times for components:
  - Instruction and data memories: 200 ps
  - LU and adders: 180 ps
  - Decode and Register file access (read or write): 150 ps
  - Ignore the delays in PC, mux, extender, and wires
- Assume the following instruction mix:
  - 40% ALU, 20% Loads, 10% stores, 20% branches, & 10% jumps
- Which of the following would be faster and by how much?
  - Single-cycle implementation for all instructions
  - Multicycle implementation optimized for every class of instructions

Example solution

Instruction Class	Instruction Memory	Register Read	ALU Operation	Data Memory	Register Write	Total
ALU	200	150	180		150	680 ps
Load	200	150	180	200	150	880 ps
Store	200	150	180	200		730 ps
Branch	200	150	180 ← Compare and write PC			530 ps
Jump	200	150 ← Decode and write PC				350 ps

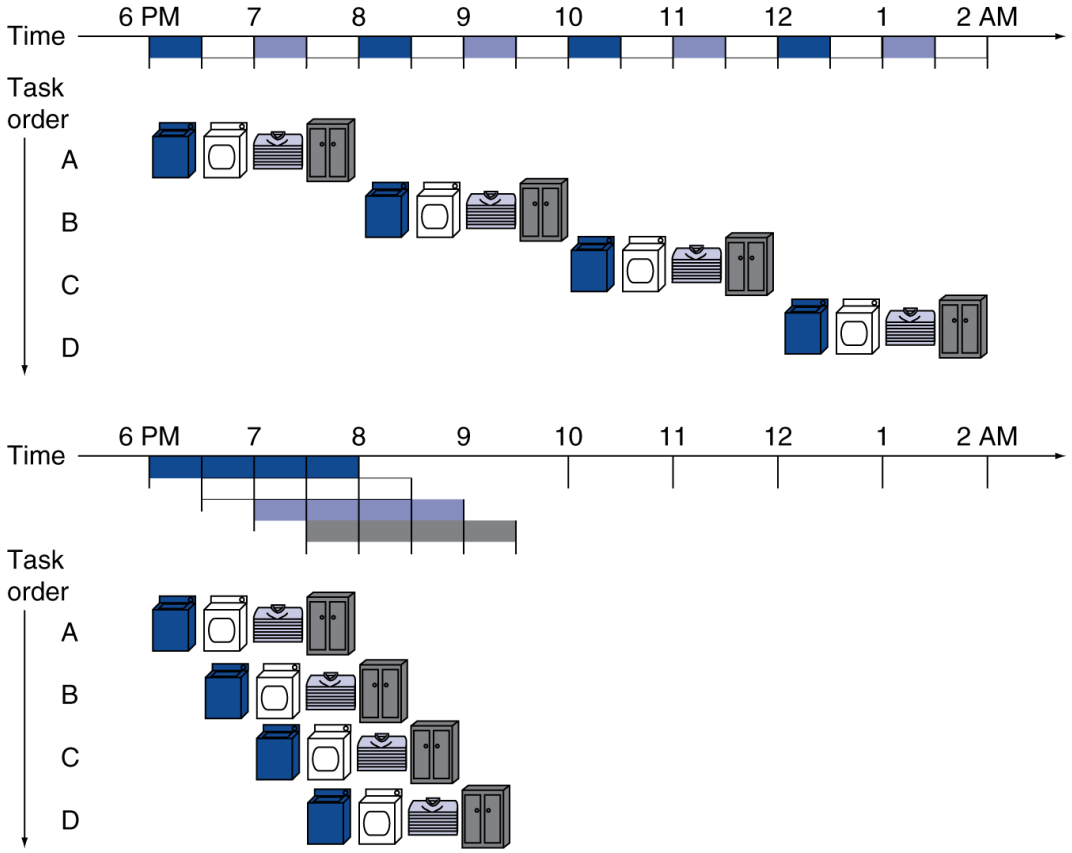
- For fixed single-cycle implementation:
  - Clock cycle = 880 ps determined by longest delay (load instruction)
- For multi-cycle implementation:
  - Clock cycle = max (200, 150, 180) = 200 ps (maximum delay at any step)
  - Average CPI = 0.4×4 + 0.2×5 + 0.1×4+ 0.2×3 + 0.1×2 = 3.8
- Speedup = 880 ps / (3.8 × 200 ps) = 880 / 760 = 1.16

The idea of pipelining

- Multicycle improves performance over single cycle, but can you see limitations of the multi-cycle design?
  - Some HW resources are idle during different phases of the instruction cycle, e.g. “Fetch” logic is idle when an instruction is being “decoded” or “executed”
  - Most of the datapath is idle when a memory access is happening
- Can we do better?
  - Yes: More concurrency → Higher instruction throughput (i.e., more “work” completed in one cycle)
- Idea: when an instruction is using some resources in its processing phase, process **other** instructions on **idle** resources
  - E.g., when an instruction is being decoded, fetch the next instruction
  - E.g., when an instruction is being executed, decode another instruction
  - E.g., when an instruction is accessing data memory (`lw/sw`), execute the next instruction
  - E.g., when an instruction is writing its result into the register file, access data memory for the next instruction

A laundry analogy

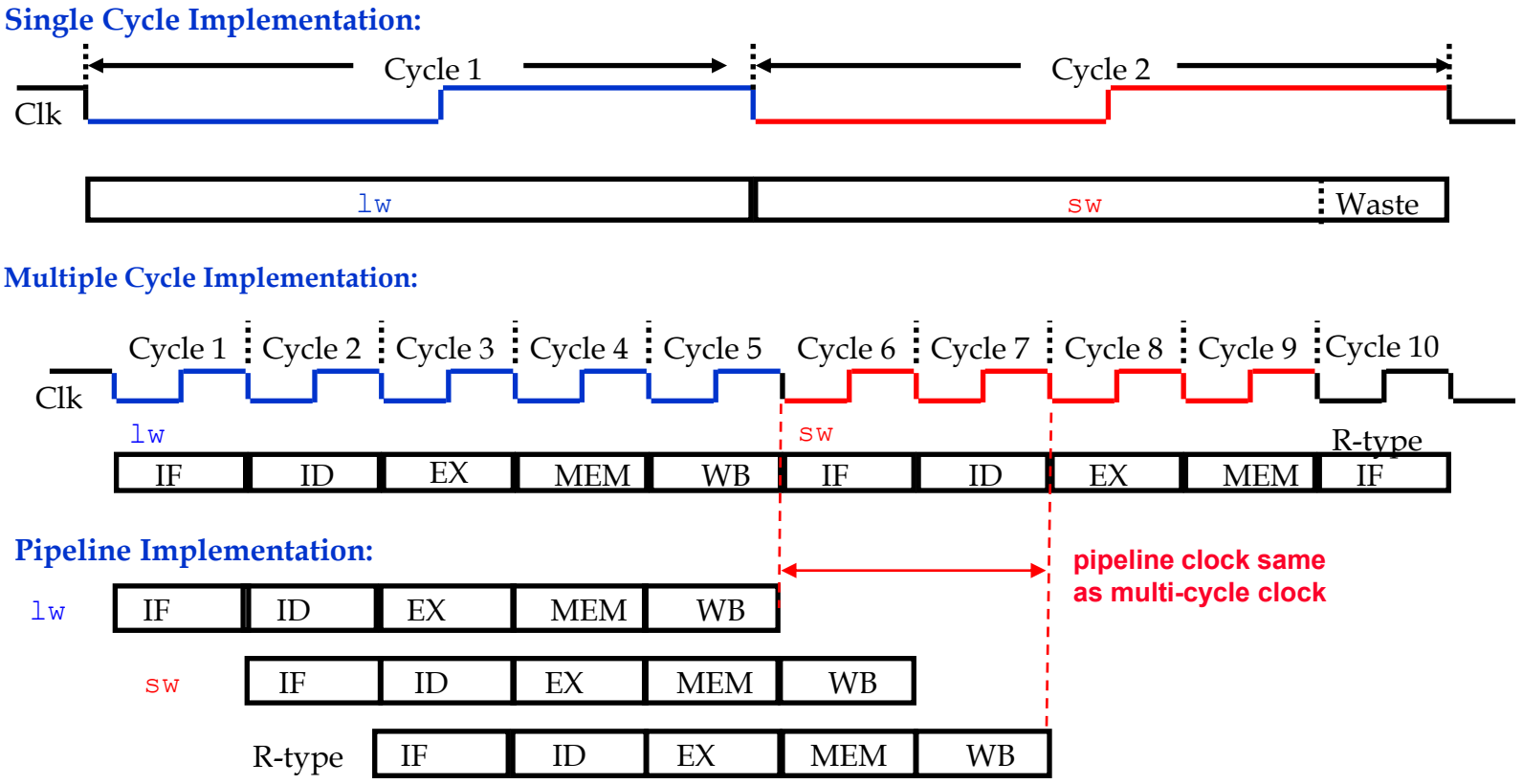
- Sequential laundry: wash-dry-fold-put away cycle
- Pipelined laundry: start the next load at each step completion
  - Parallelism improves performance. How much?



- Four loads
  - Speedup $= 8/3.5 = 2.3$
- Non-stop ($n \to \infty$ loads)
  - Speedup $= \frac{2n}{0.5n+1.5} \approx 4$
    = number of **stages**

Single-cycle vs multi-cycle vs pipeline

- Five stages, one step per stage
  - Each step requires 1 clock cycle → steps enter/leave pipeline at the rate of one step per clock cycle



Pipeline performance

- Ideal pipeline assumptions
  - **Identical operations**, e.g. four laundry steps are repeated for all loads
  - **Independent operations**, e.g. no dependency between laundry steps
  - **Uniformly partitionable suboperations** (that do not share resources), e.g. laundry steps have uniform latency.
- Ideal pipeline speedup
  - $\text{Time between instructions}_{pipelined} = \frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$
  - Speedup is due to increased **throughput** (*), **latency** (*) does not decrease
- Speedup for non-ideal pipelines is less
  - External/internal fragmentation, pipeline stalls.

- ✓ Latency = execution time (delay or response time) = the total time from start to finish of ONE instruction
- ✓ Throughput (or **execution bandwidth**) = the total amount of work done in a given amount of time

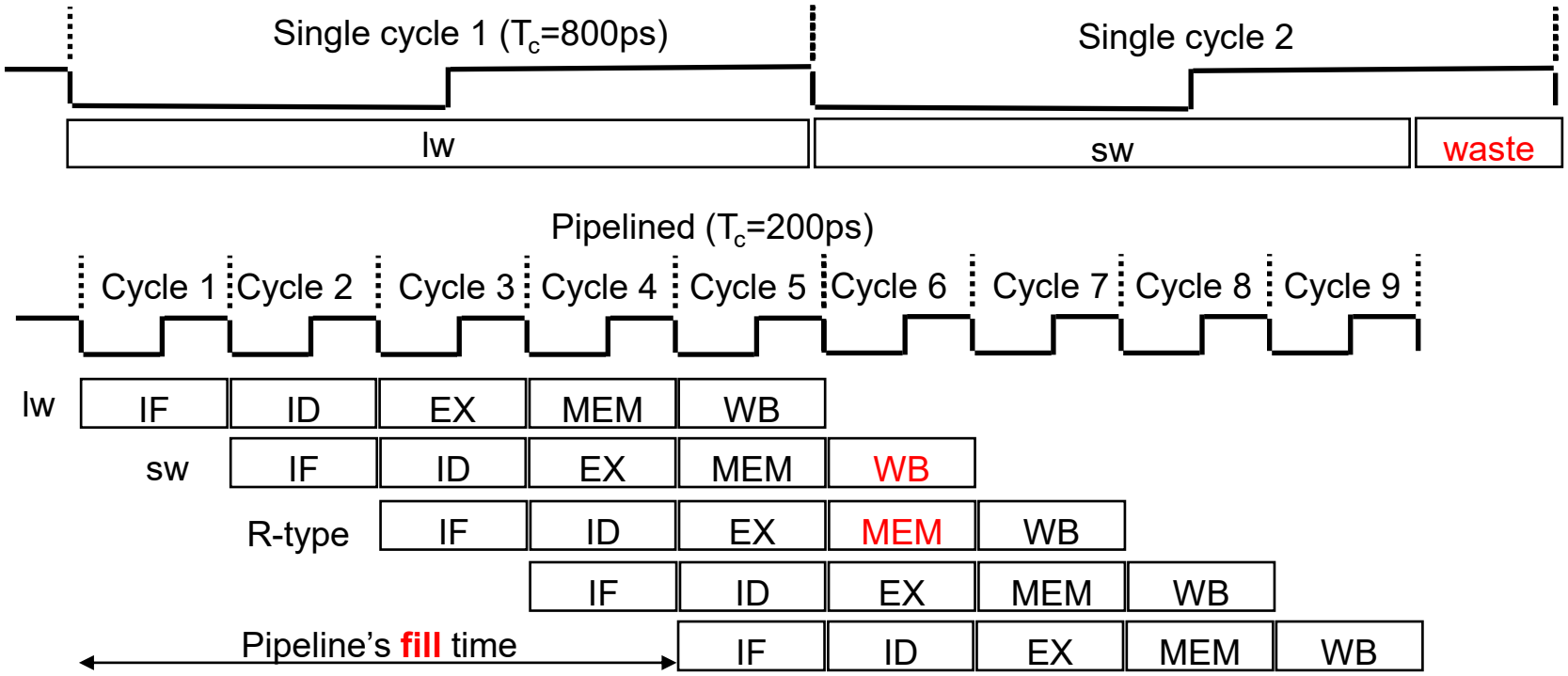
Example: An MIPS pipelined processor performance

- Assume time for stages is
  - ✓ 100ps for register read or write
  - ✓ 200ps for other stages

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

- Compare pipelined datapath with single-cycle datapath

Pipelined performance example solution



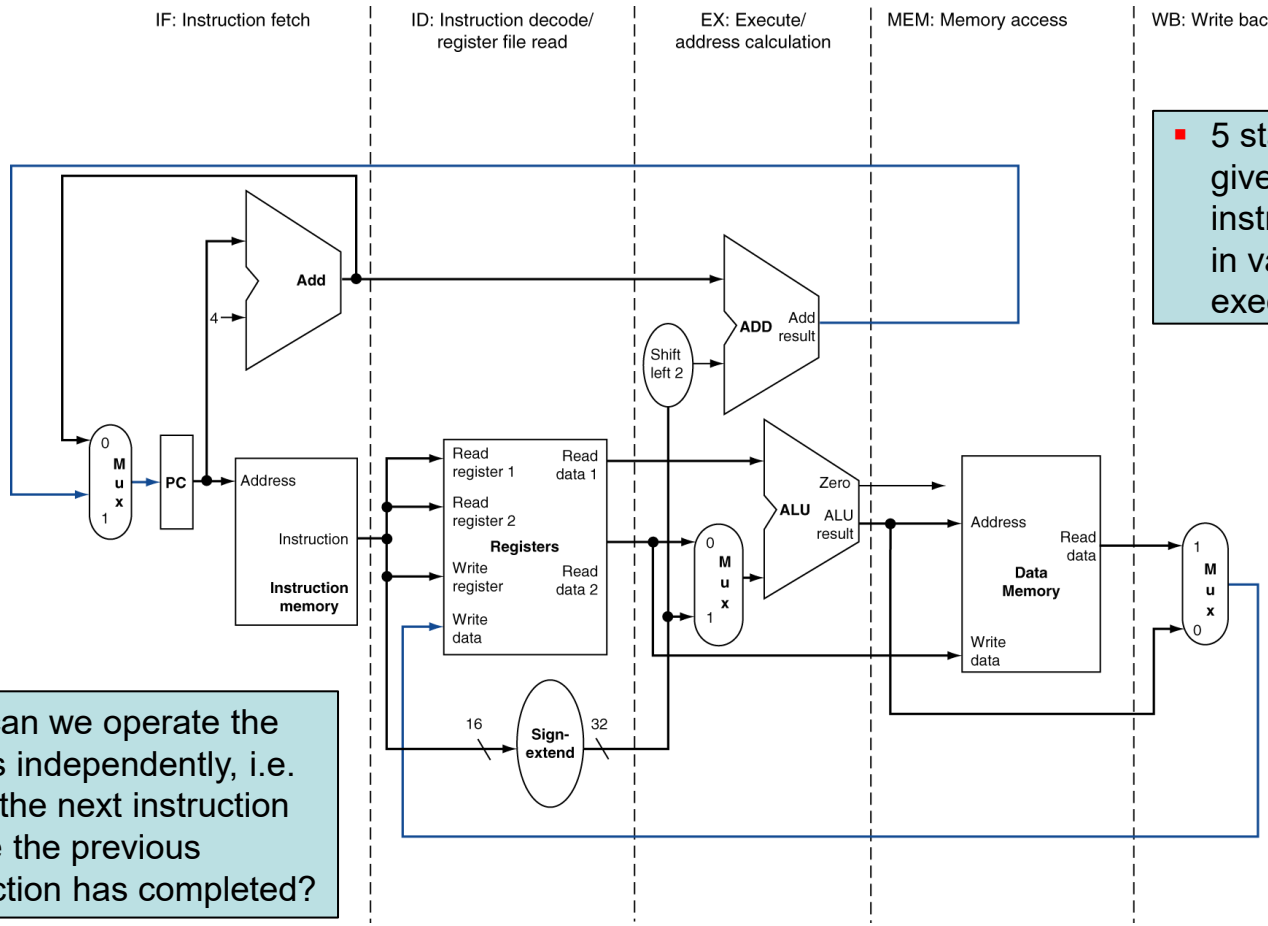
- Time btw 1st and 5th instructions: **single cycle = 3200ps** (4 x 800ps) vs **pipelined = 800ps** (4 x 200ps) → speedup = 4.
  - Execution time for **5 instructions**: **4000ps** vs **1800ps** ≈ 2.22 times speedup → Why shouldn't the speedup be 5 (#stages)? What's wrong?
  - Think of real programs which execute **billions** of instructions.

MIPS ISA supports for pipelining

- What makes it easy
  - **All** instructions are 32-bits
    - Easier to fetch and decode in one cycle: fetch in the 1st stage and decode in the 2nd stage
    - c.f. x86: 1- to 17-byte instructions
  - **Few and regular** instruction formats
    - Can decode and read registers in one step
  - Memory operations occur **only** in loads and stores
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Operands must be **aligned** in memory
    - Memory access takes only one cycle
  - Each instruction writes **at most** one result (i.e., changes the machine state) and does it in the **last few** pipeline stages (MEM or WB)
- What's makes it hard? (later)

Ideas from the Single-Cycle Datapath

- How to pipeline a single-cycle datapath? Think of the simple datapath as a linear sequence of stages.

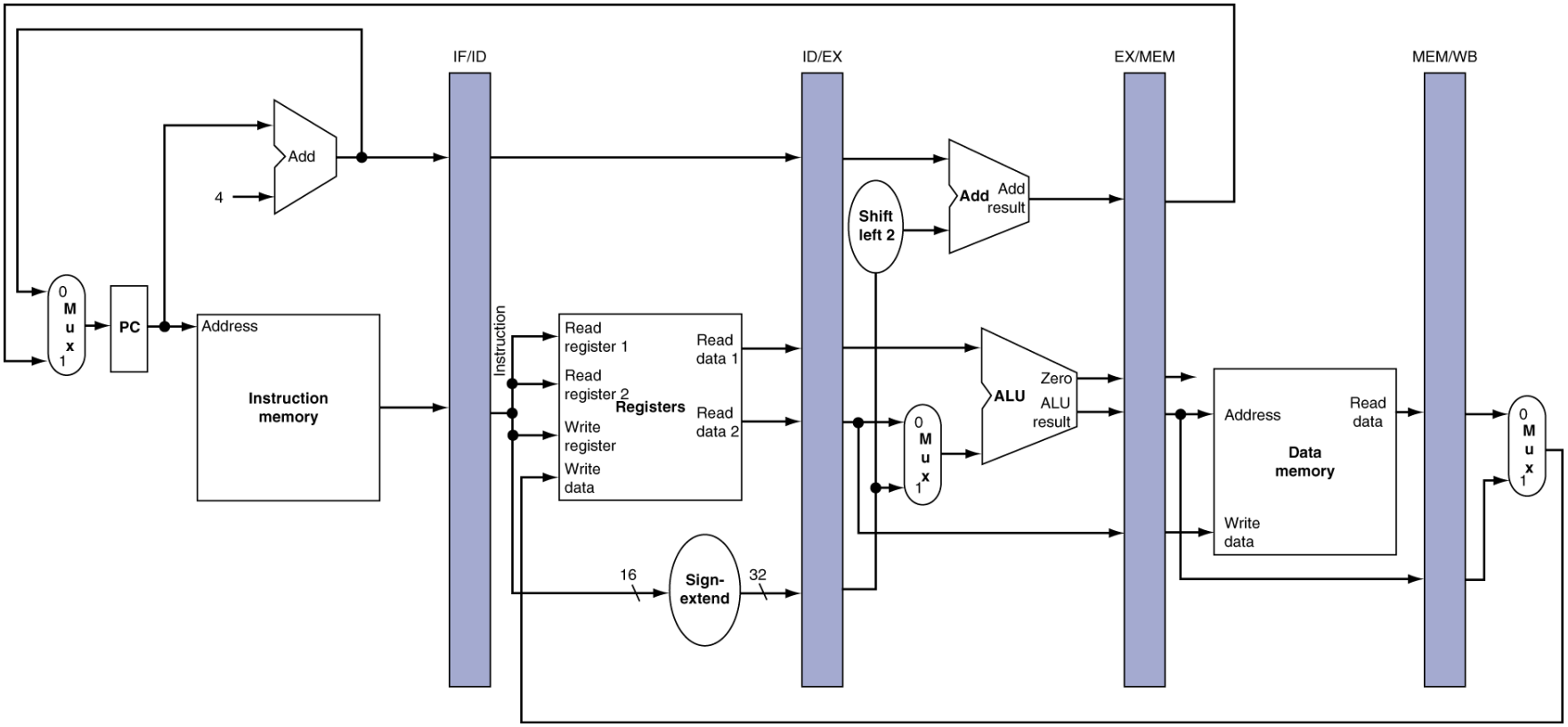


- 5 stages → on any given cycle up to 5 instructions will be in various points of execution.

- How can we operate the stages independently, i.e. begin the next instruction before the previous instruction has completed?

Pipelined Datapath

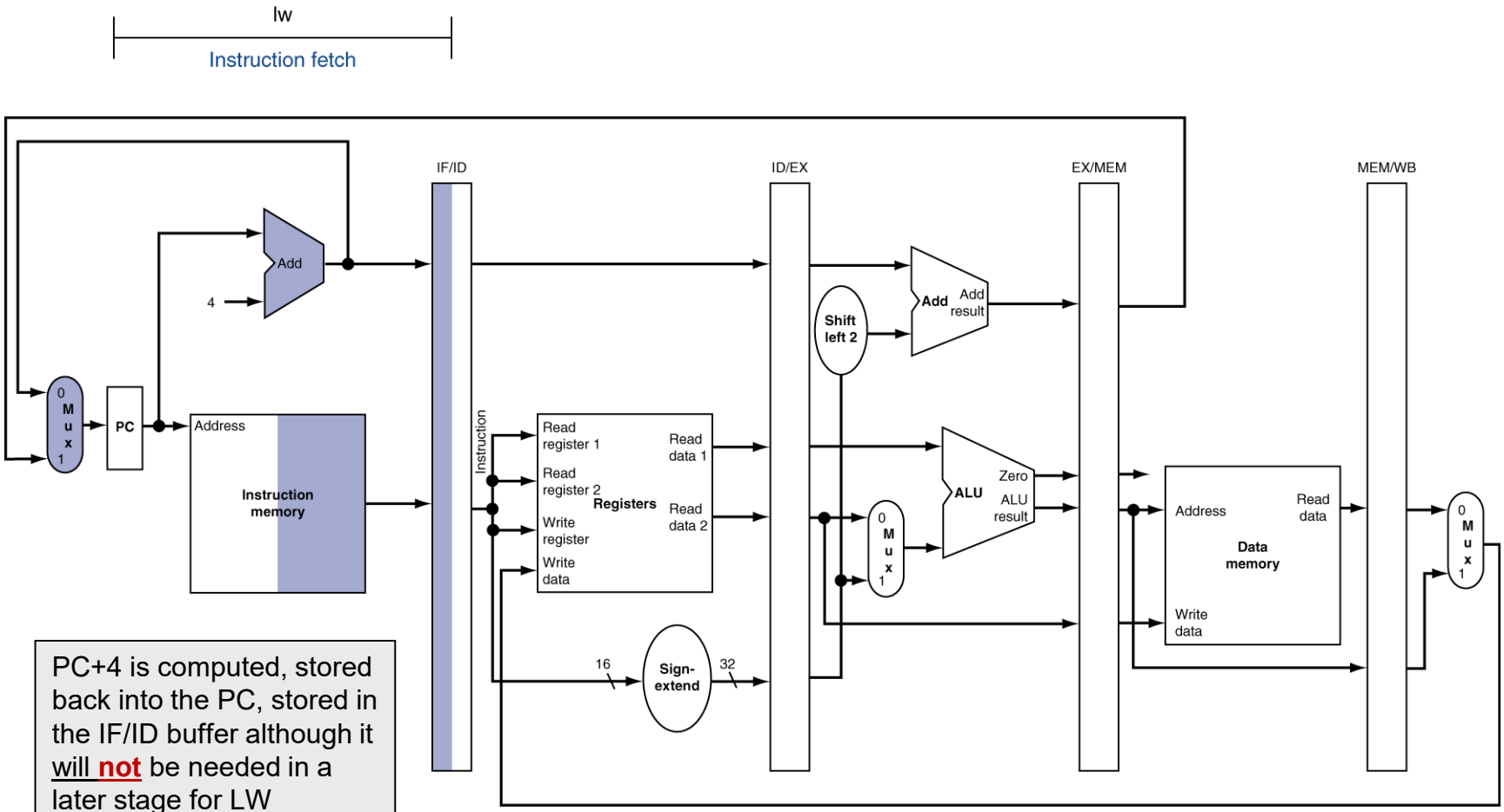
- Add **state registers** between each pipeline stage
  - To **isolate** information between cycles



Pipeline operation

- **Cycle-by-cycle flow** of instructions through the pipelined datapath
  - Same **clock edge updates** all pipeline registers, register file, and data memory (for store instruction)
  - "**Single-clock-cycle**" pipeline diagram
    - ✓ Shows pipeline usage in a single cycle
    - ✓ Highlight resources used
  - c.f. "**multi-clock-cycle**" diagram (later)
    - ✓ Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for load to verify the proposed datapath

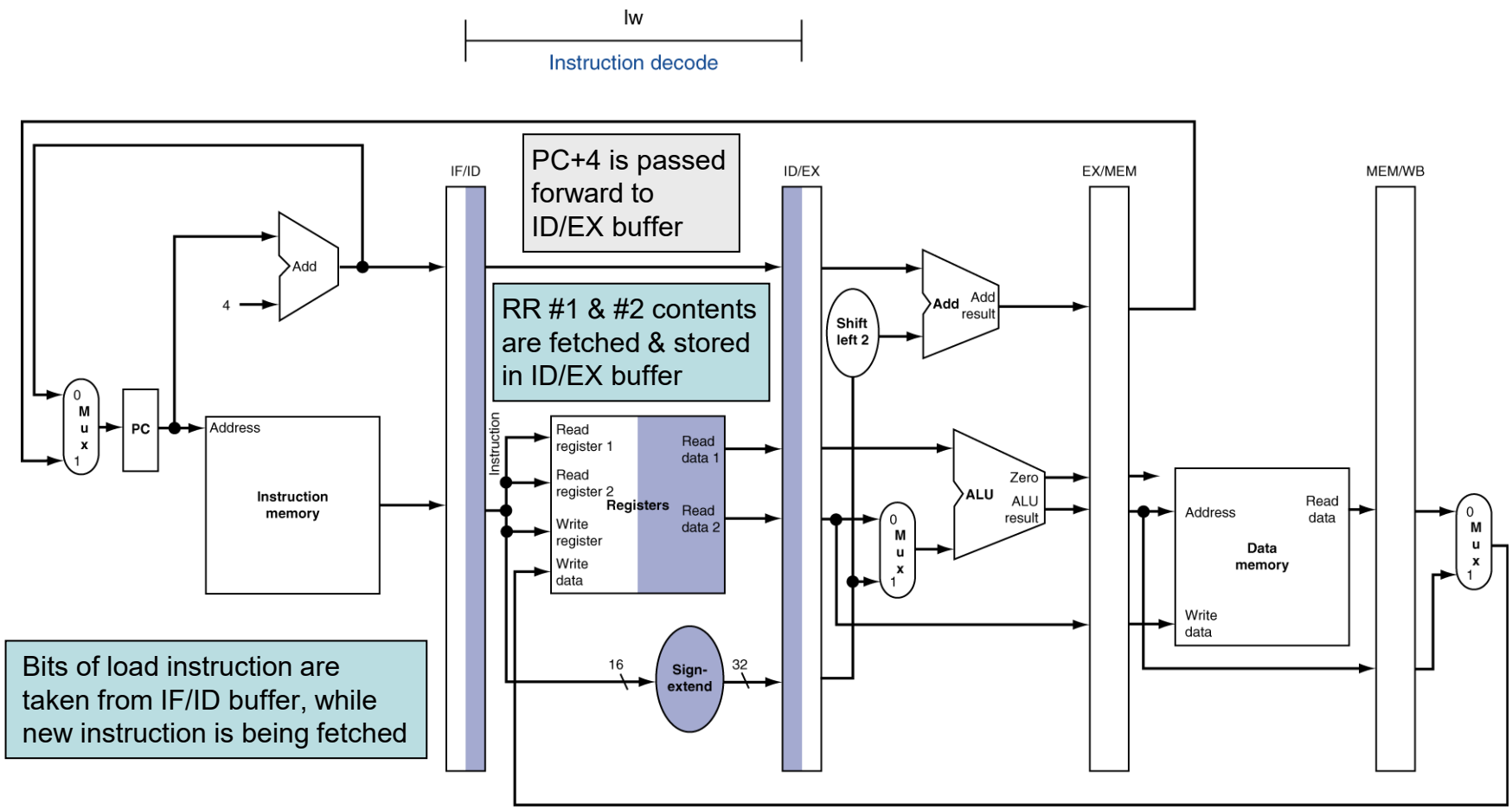
IF for Load, Store, …



PC+4 is computed, stored back into the PC, stored in the IF/ID buffer although it will **not** be needed in a later stage for LW

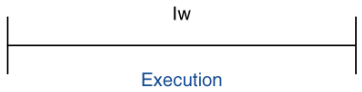
Instruction word is fetched from memory, and stored in the IF/ID buffer because it **will** be needed in the next stage.

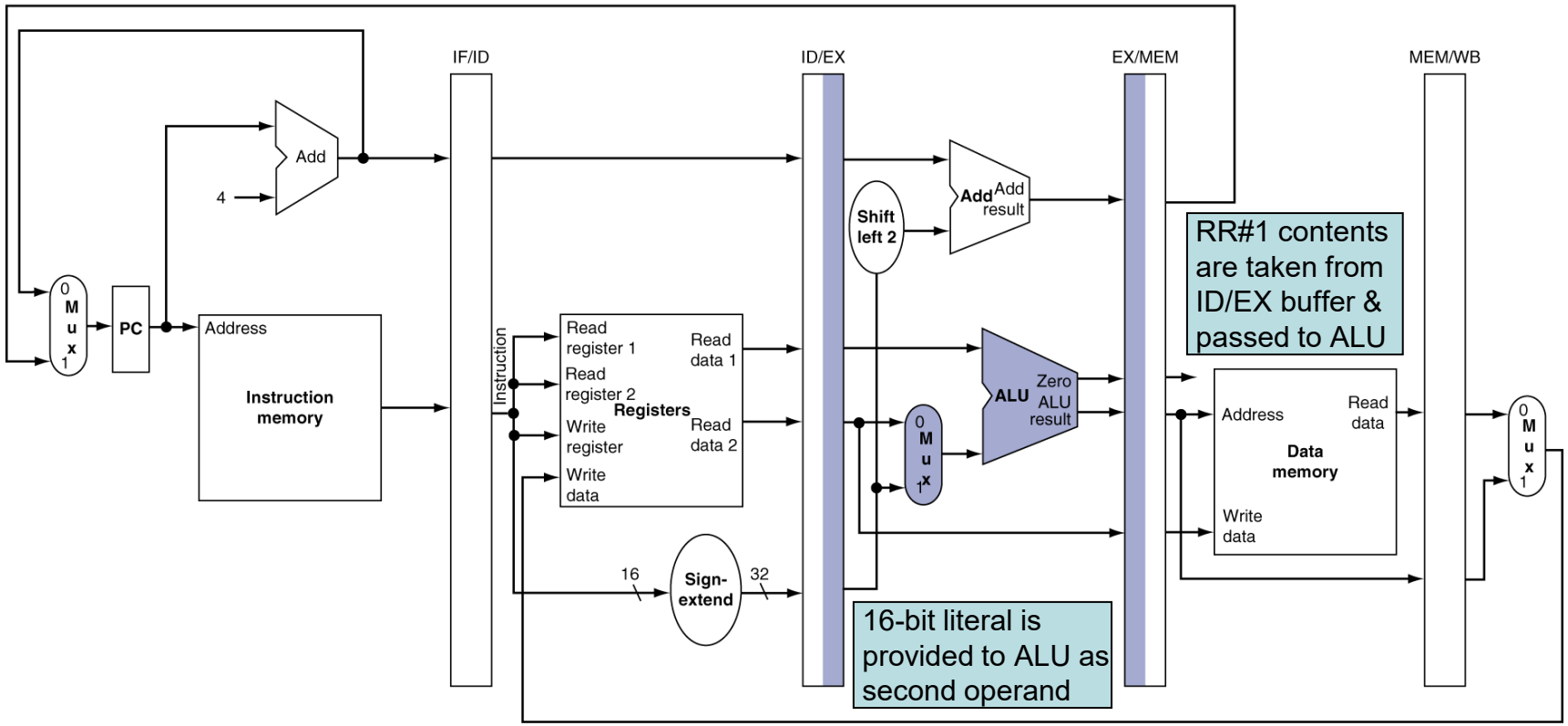
ID for Load, Store, …



EX for Load

PC+4 is taken from ID/EX buffer and added to branch offset, then the computed branch target address is stored in EX/MEM buffer (won't be needed, though)





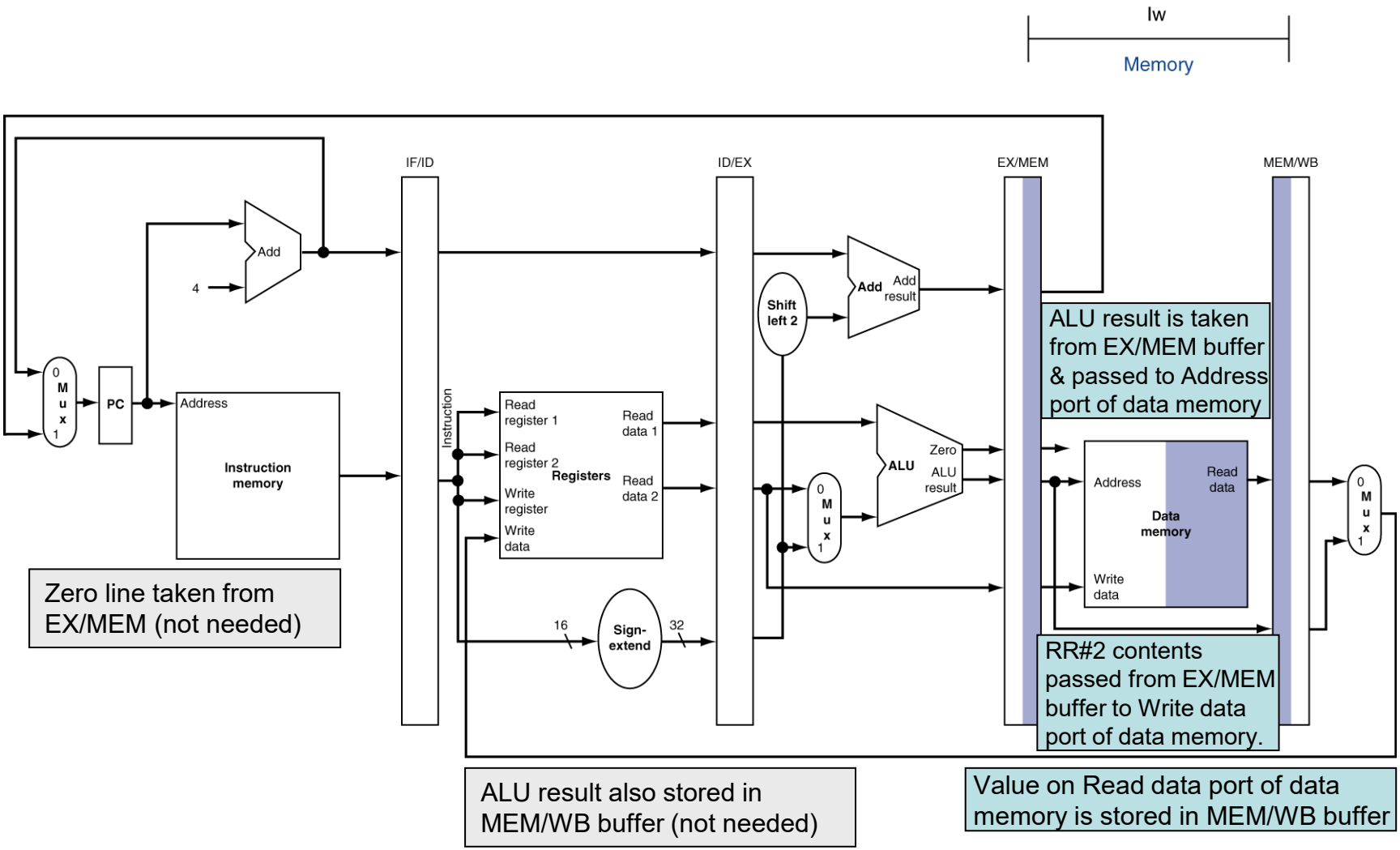
RR#1 contents are taken from ID/EX buffer & passed to ALU

16-bit literal is provided to ALU as second operand

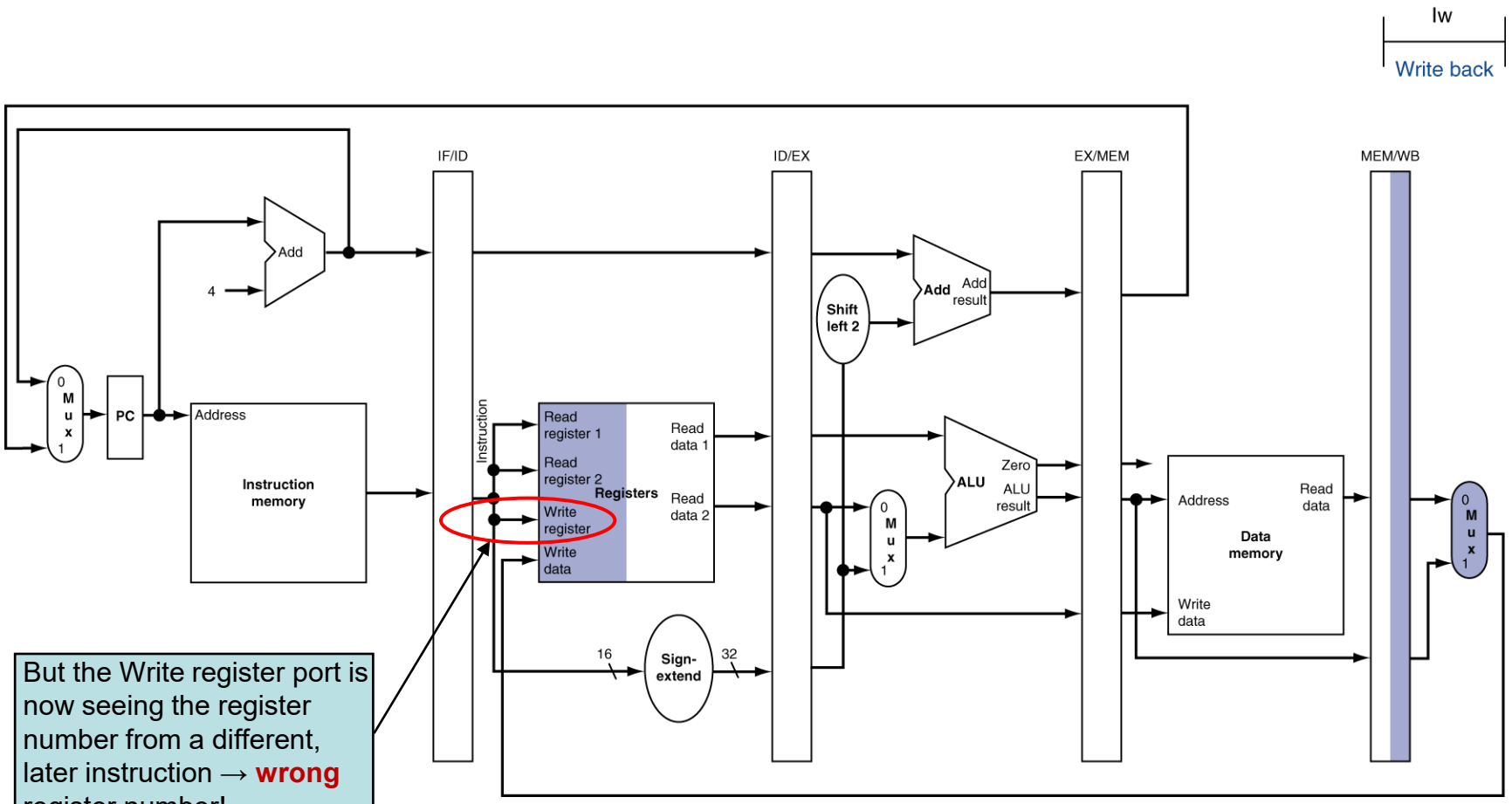
Read register #2 is passed forward to EX/MEM buffer (but won't be needed, though)

ALU result and Zero line are stored in EX/MEM buffer for use as memory address in next stage

MEM for Load


lw
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
16
Sign-extend
32
Address
Read data
Data memory
Write data
ALU result is taken from EX/MEM buffer & passed to Address port of data memory
Zero line taken from EX/MEM (not needed)
RR#2 contents passed from EX/MEM buffer to Write data port of data memory.
ALU result also stored in MEM/WB buffer (not needed)
Value on Read data port of data memory is stored in MEM/WB buffer

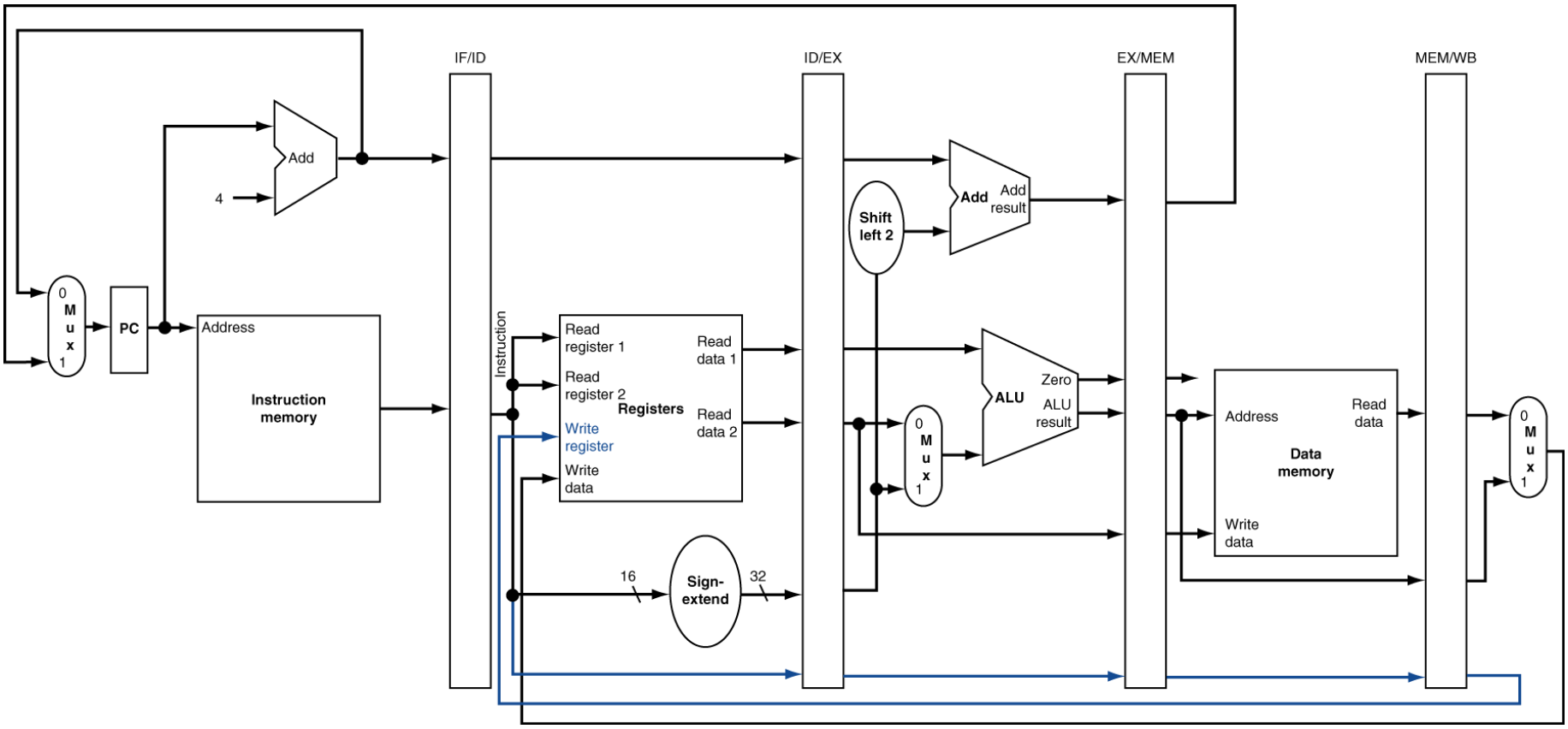
WB for Load



But the Write register port is now seeing the register number from a different, later instruction → **wrong** register number!

Value from data memory is selected and passed back to register file.

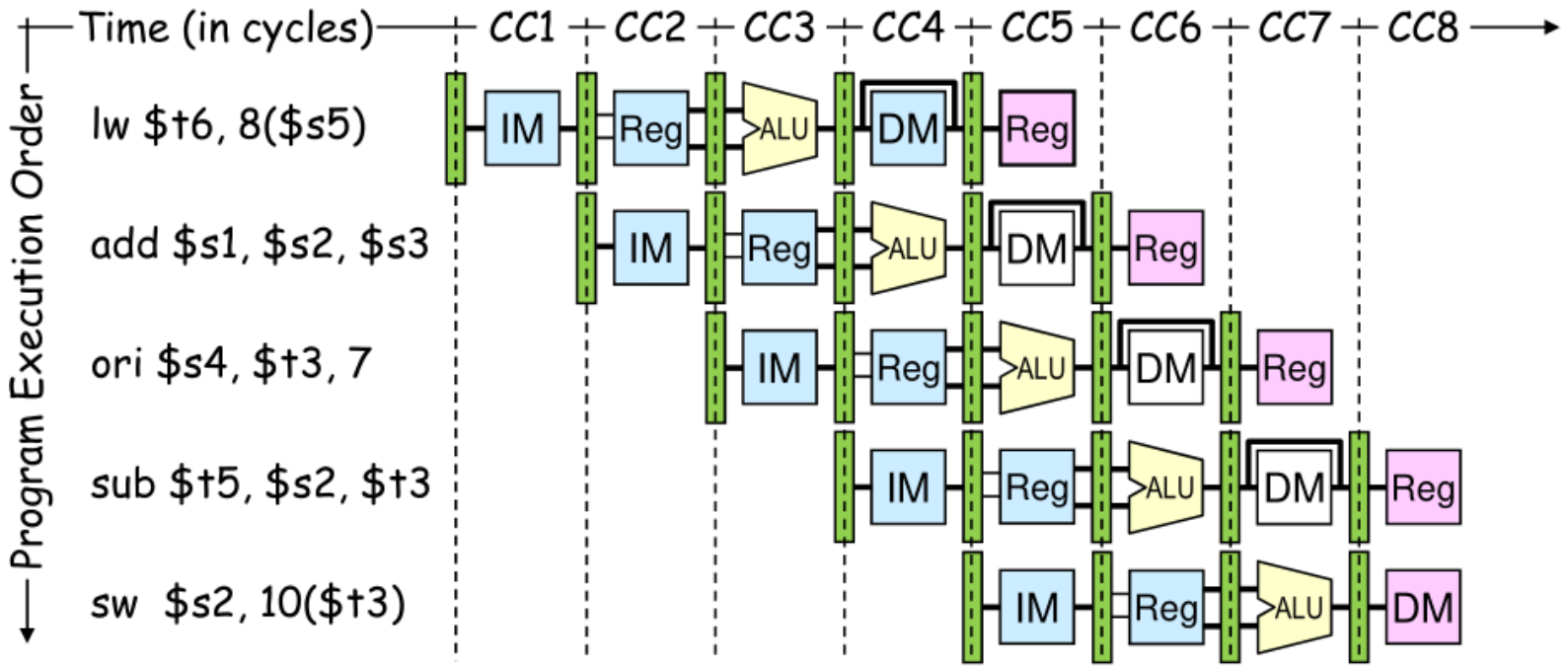
Corrected Datapath for Load



So we fix the problem by passing the Write register number from the load instruction through the various inter-stage buffers, and then feed it back, just in time → adding five more bits to the last three pipeline registers.

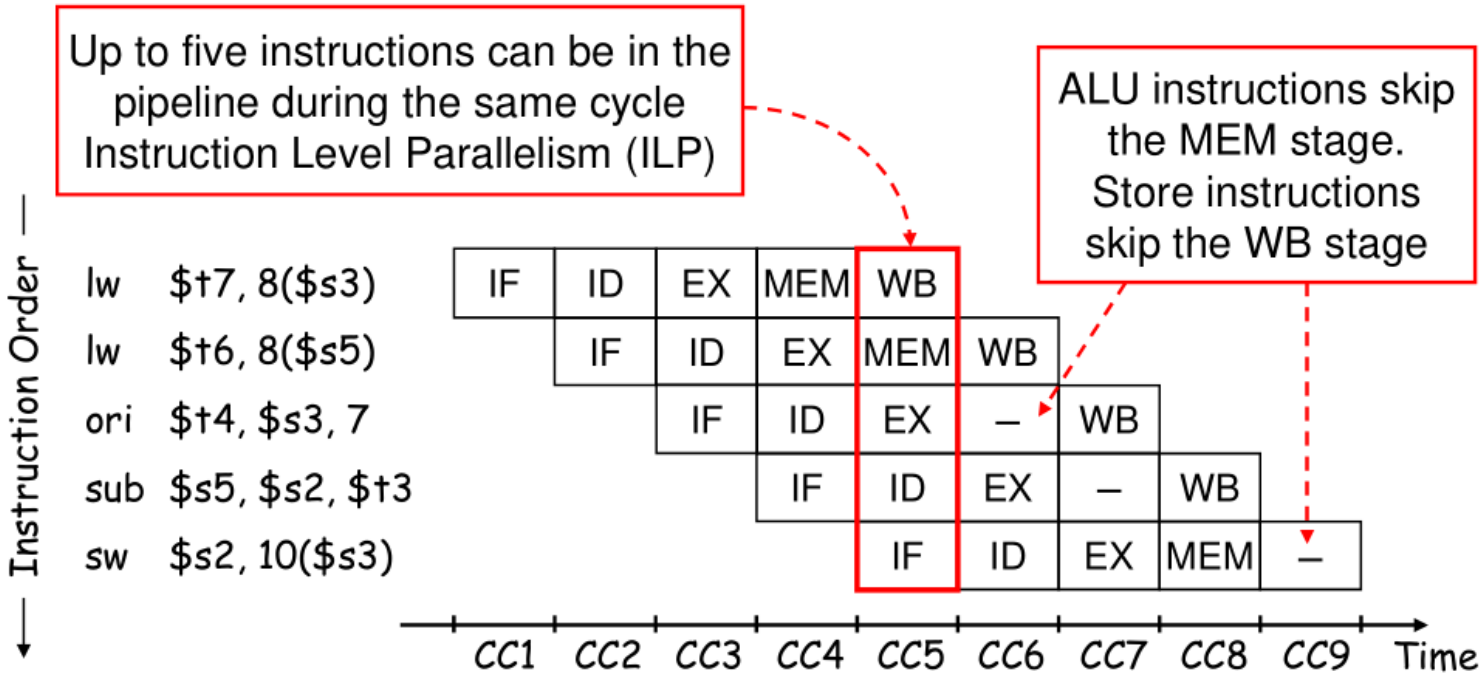
Multi-Cycle Pipeline Diagram (1)

- Shows the **complete execution** of instructions in a single figure
  - Instructions are listed in instruction execution order from top to bottom
  - Clock cycles move from left to right
  - Figure shows the use of resources at each stage and each cycle



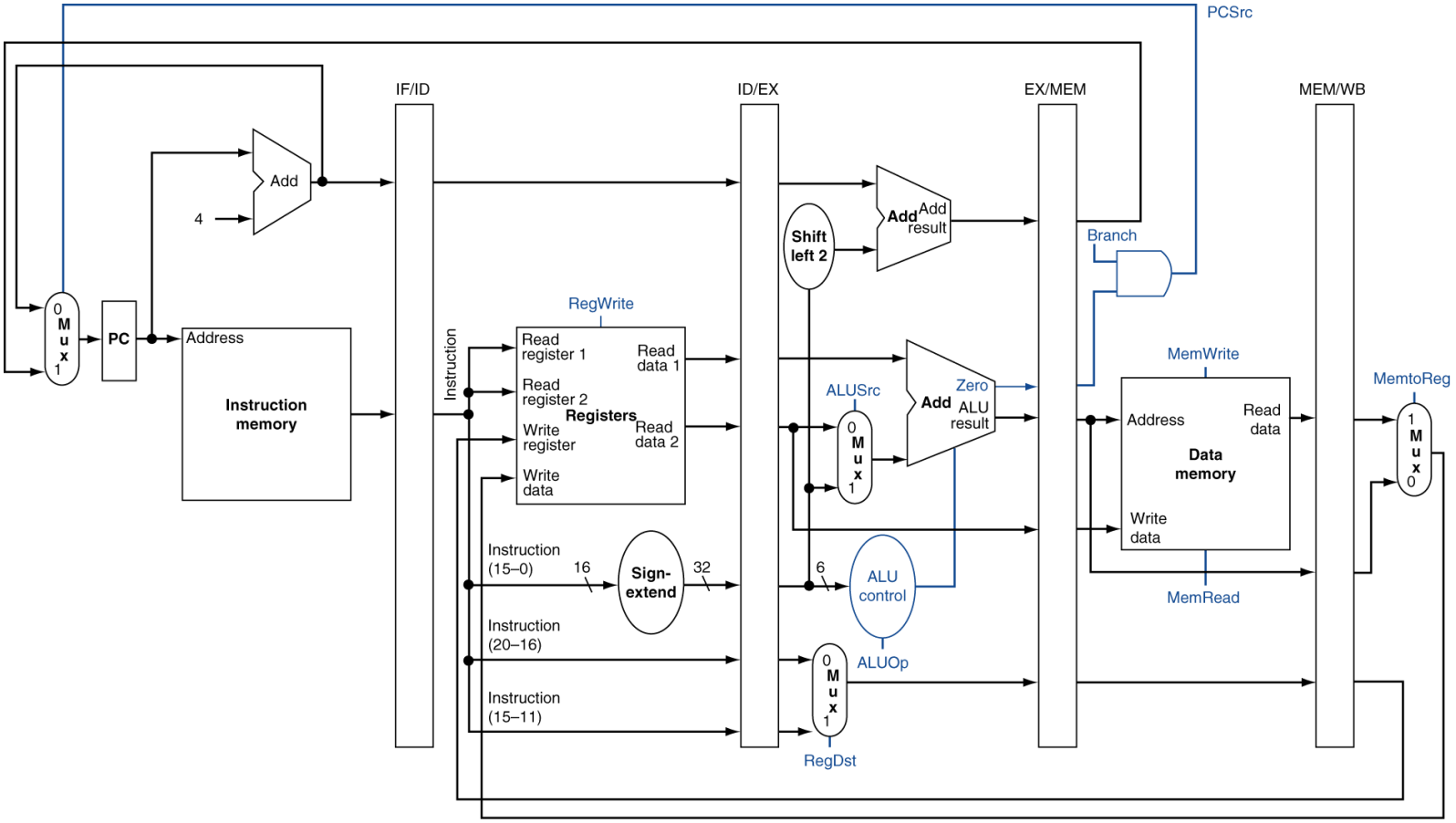
Multi-Cycle Pipeline Diagram (2)

- 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? (later)



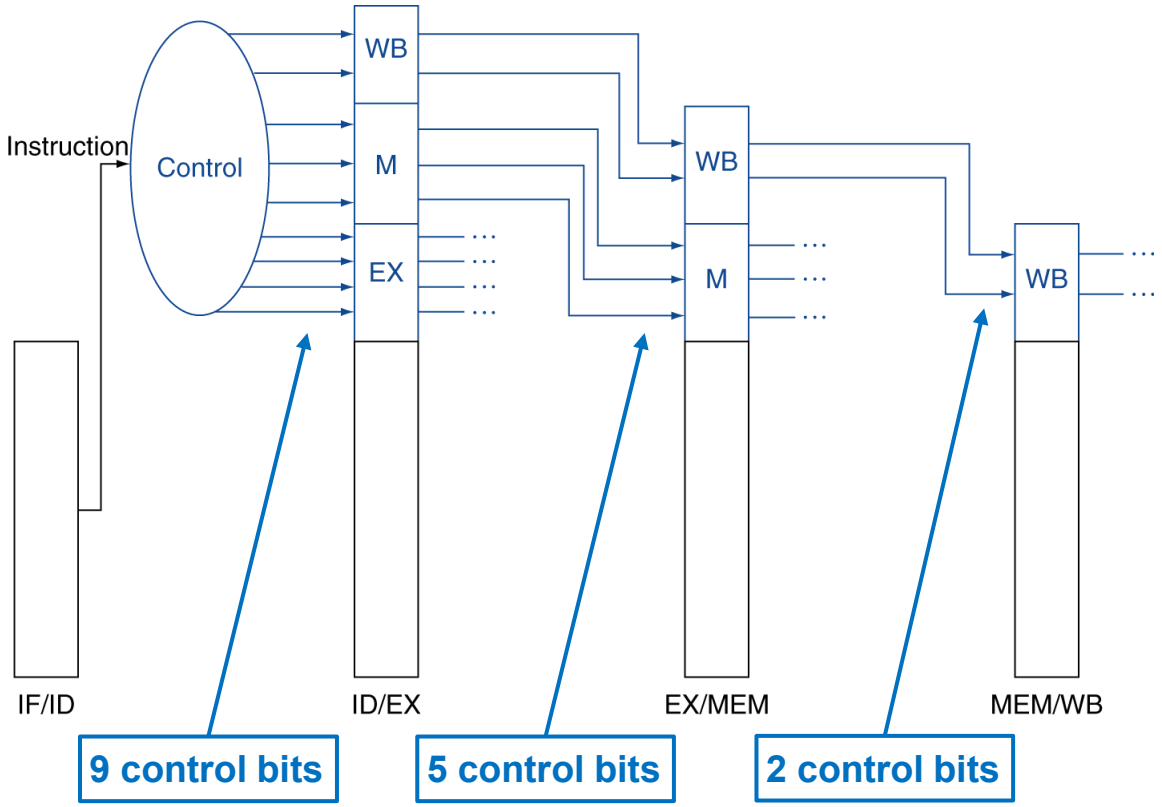
Pipelined control: control points

- Same control points as in the single-cycle datapath



Pipelined control: settings (1)

- Control signals derived from instruction & determined during ID
  - As the instruction moves → pipeline the control signals → extend the pipeline registers to include the control signals
  - Each stage uses some of the control signals



Pipelined control: settings (2)

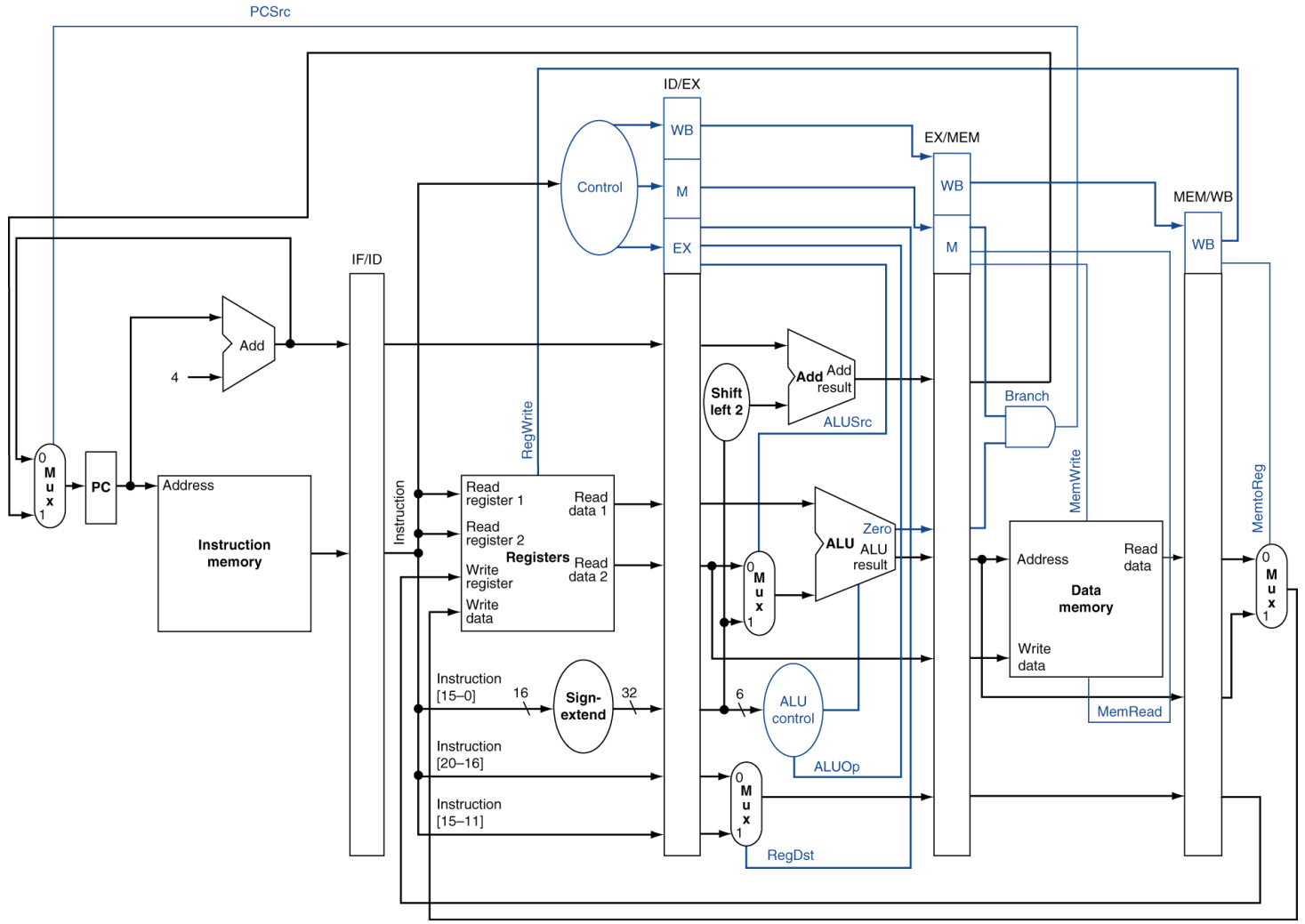
- Control signals needed in each stage

Pipeline Stage	Control signals
IF	None
ID	None
EX	RegDst, ALUOp1, ALUOp0, ALUSrc
MEM	Branch, MemRead, MemWrite
WB	MemtoReg, RegWrite

- Control signal values based on instruction type

	EX Stage				MEM Stage			WB Stage	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Brch	Mem Read	Mem Write	Reg Write	Mem toReg
R	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

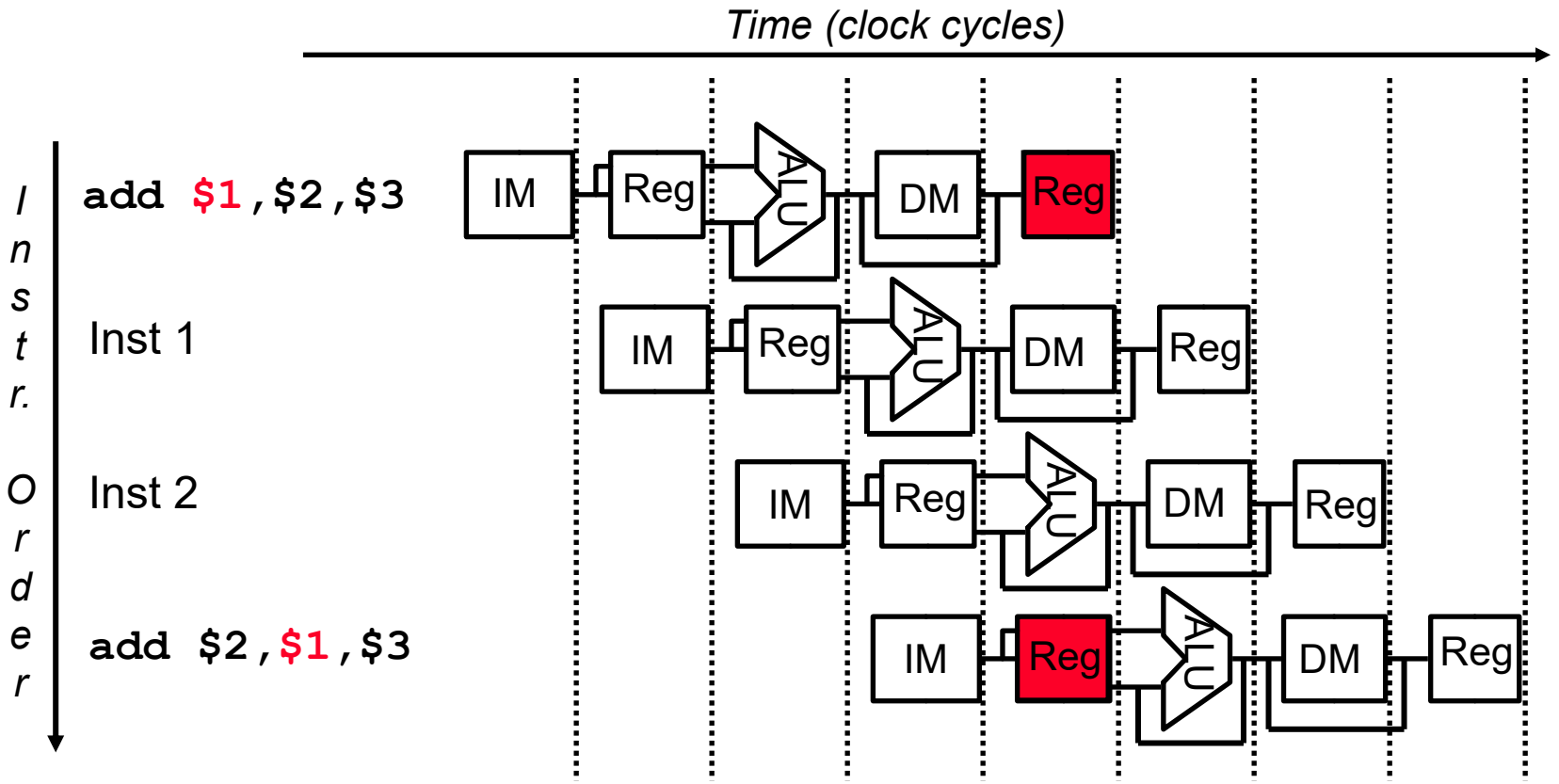
Pipelined control: complete



Can Pipelining Get Us Into Trouble?

- Yes - instruction pipeline is **not** an ideal pipeline
  - different instructions → not all need the same stages: some pipe stages idle for some instructions → external fragmentation
  - different pipeline stages → not the same latency: some pipe stages are too fast but all take the same clock cycle time → internal fragmentation
  - instructions are not independent of each other → pipeline stalls: pipeline is not always moving
- Issues in pipeline design: **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, e.g. 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 (e.g. branch and jump instructions, exceptions)

Example: structural hazards



- Two instructions are attempting to use the same register ($1) during the same cycle (CC5).

Summary

- Multi-cycle processor
  - Use one clock cycle per step → shorter clock cycle time = longest step, not longest instruction.
  - Higher performance over single-cycle processor: simpler instructions take fewer cycles → less waste
- Pipeline processor design
  - Employs **instruction parallelism**: process the next instruction on the resources available when current instructions move to subsequent phases.
  - Speedup is due to **increased throughput**: once the pipeline is full, CPI=1.
  - Datapath can be derived from that of single-cycle processor, with additional buffer registers
  - Control signals remain the same as in the single-cycle case but some of them are moved along the pipeline via inter-stage buffers.
- As the instruction pipeline is not ideal, various issues may occur including structural, data, and control **hazards**.