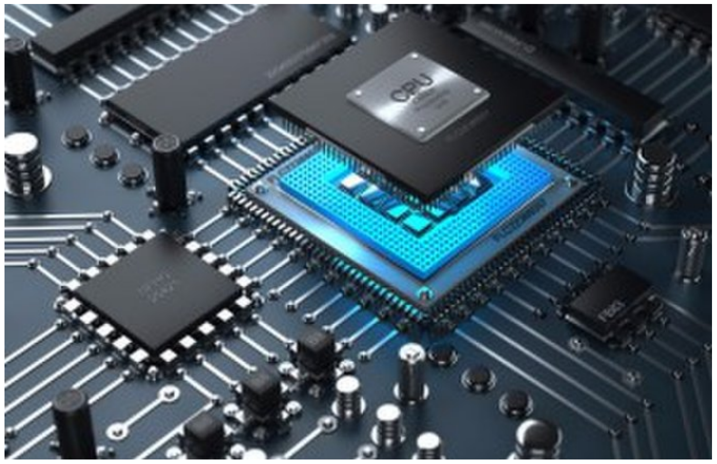


Pipelining 1

CS 154: Computer Architecture

Lecture #14

Winter 2020

Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB

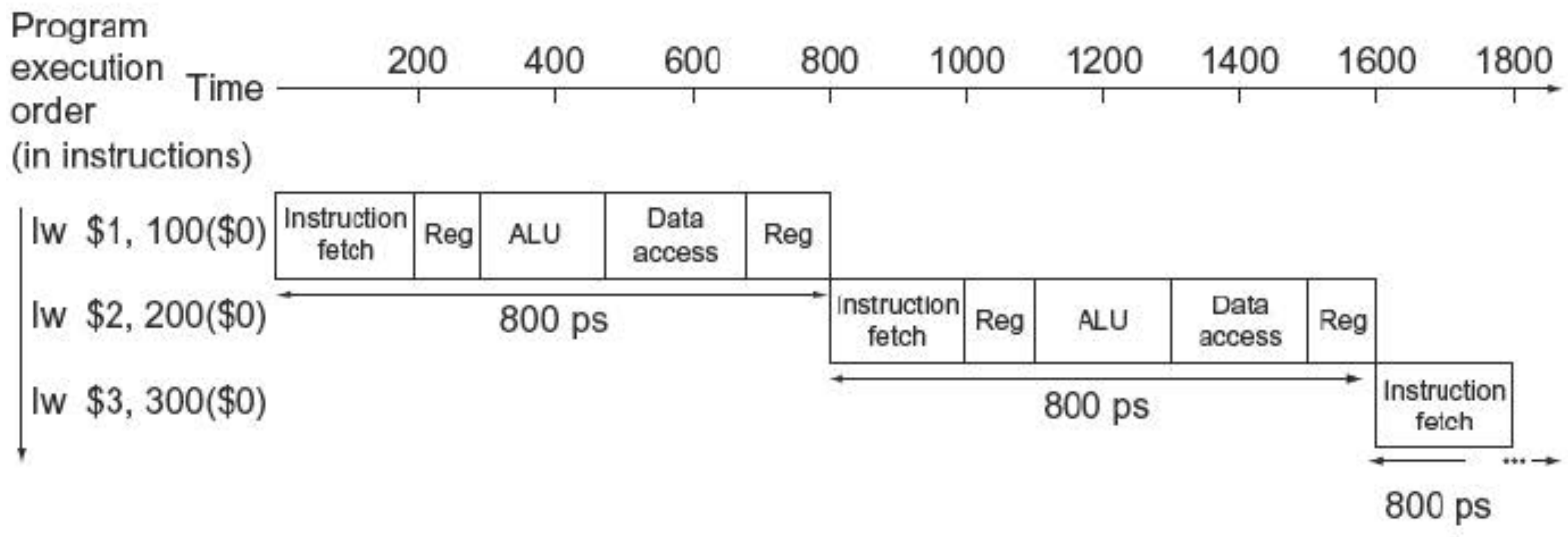
Administrative

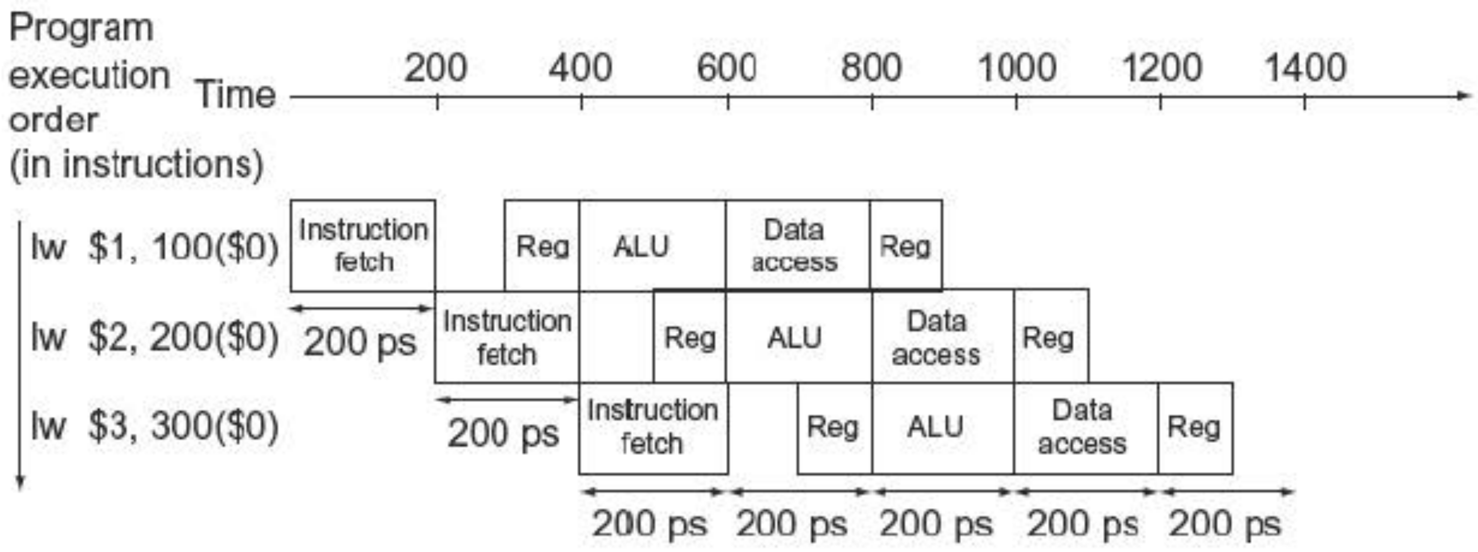
- Talk is on **MONDAY, MARCH 9th** in our usual class
  - Will take attendance…
- We "only" have 3 more lectures… ☹ *please everybody control your emotions…*
  - Pipelining and start w/ Memory Caching
- Final Exam info:
  - **Tuesday, March 17th at 12:00** (**not 12:30!!!**) in this classroom
    - Arrive 10 mins early – randomized seating…
  - Cumulative Exam
  - Will allow some notes – exact details to follow
  - Study guide/example Qs will be issued by this weekend

Lecture Outline

- Data Hazards in Pipelining
- Pipeline Designs and Operation
- Examples with some Instructions
- Diagramming Pipelined Instructions
- Control Lines for Pipelines

Comparison of Per-Instruction Time





Hazards

- Situations that *prevent* starting the next instruction in the next cycle

- **Structure hazards**
  - A required resource is busy
- **Data hazard**
  - Need to wait for previous instruction to complete its data read/write
- **Control hazard**
  - Deciding on control action depends on previous instruction

Structure Hazards

- **Conflict for use of a resource**

- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
  - Would cause a pipeline "bubble"

- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

Data Hazards

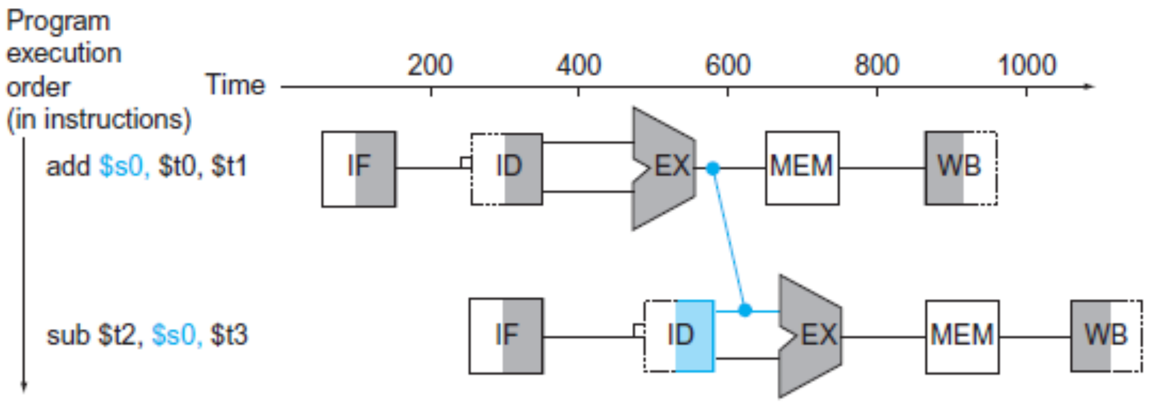
Forwarding: Use result when it is computed

- Don’t wait for it to be stored in a register
- Requires extra connections in the datapath

- **An instruction depends on completion of data access by a previous instruction**

Example 1:

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

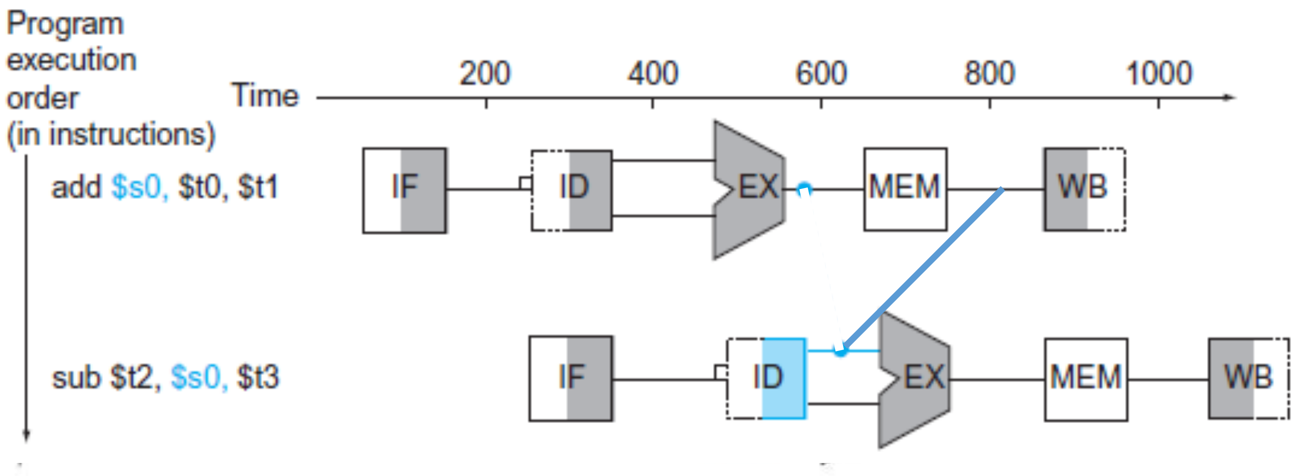


“Forwarding”: possible if destination stage is later in time than source stage

Data Hazards

Example 2:

```
lw $s0, 20($t1)
sub $t2, $s0, $t3
```



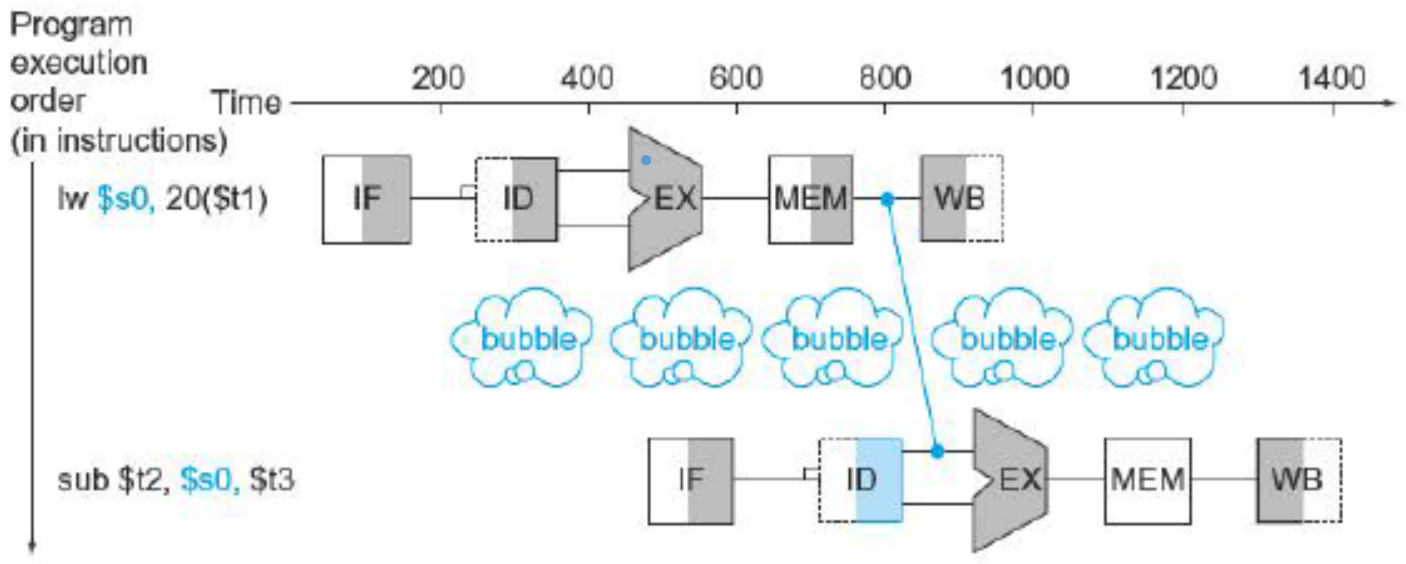
Cannot "forward" *back* in time!!

"Forwarding": not possible in this example, UNLESS …

Data Hazards

Example 2:

```
lw $s0, 20($t1)
sub $t2, $s0, $t3
```



"Forwarding": not possible in this example, UNLESS we put in a stalling instruction (aka **pipeline stall** or **bubble**) ***between them***

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- Example: C++ code for **a = b + e;**
  c = b + f;

Original

```
lw $tl, 0($t0)
lw $t2, 4($t0)
add $t3, $tl, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $tl, $t4
sw $t5, 16($t0)
```

Move here and save 2 cycles

Re-Ordered

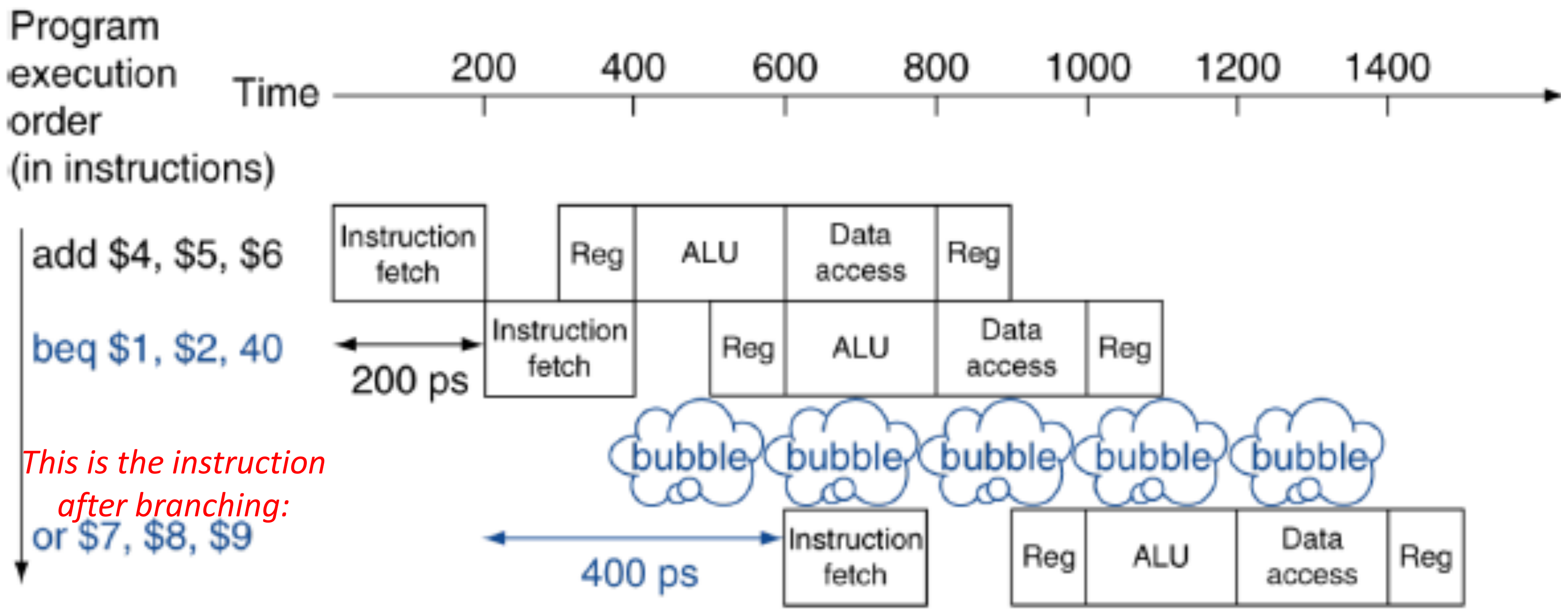
```
lw $tl, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($t0)
add $t3, $tl, $t2
sw $t3, 12 ($t0)
add $t5, $tl,$t4
sw $t5, 16 ($t0)
```

Control Hazard

- **Comes from need to make a decision based on the results of one instruction while others are executing**
  - Think of the laundry example if we were worried that the soap amount was enough based on how clean the loads come out…
- **Branch** determines flow of control
  - Fetching next instruction *depends on branch outcome*
  - **Pipeline can't always fetch correct instruction** because it is still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Needs added hardware to do it in ID stage

Possible Solution: Stall on Branch

- Wait until branch outcome is determined before fetching next instruction
  - Since the pipeline cannot possibly know what the next instruction should be, since it ***only just received*** the branch instruction from memory



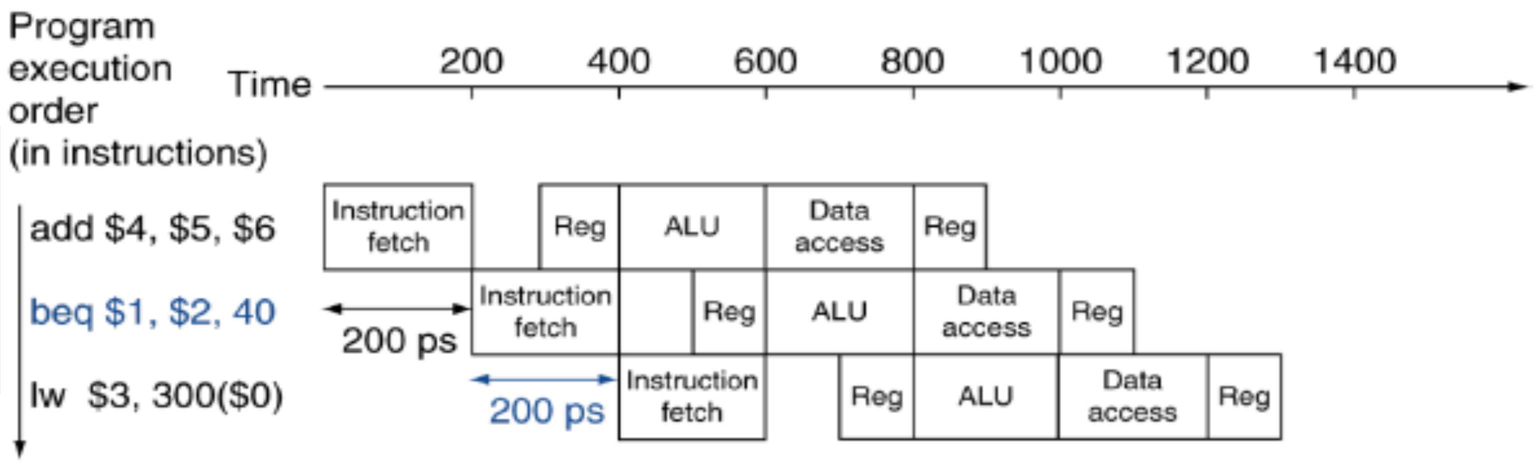
Better Solution: *Predict* the Branch!

- Predict outcome of branch
  - Only stalls if prediction is wrong
  - This option does not slow down the pipeline when it’s correct!

- In MIPS pipeline,
  it is simplest to predict if branches are **not** taken
  - When correct, the pipeline proceeds at full speed
  - When branches <u>*are*</u> taken, then the pipeline purposely stalls
  - Fetch instruction *after* branch, with no delay

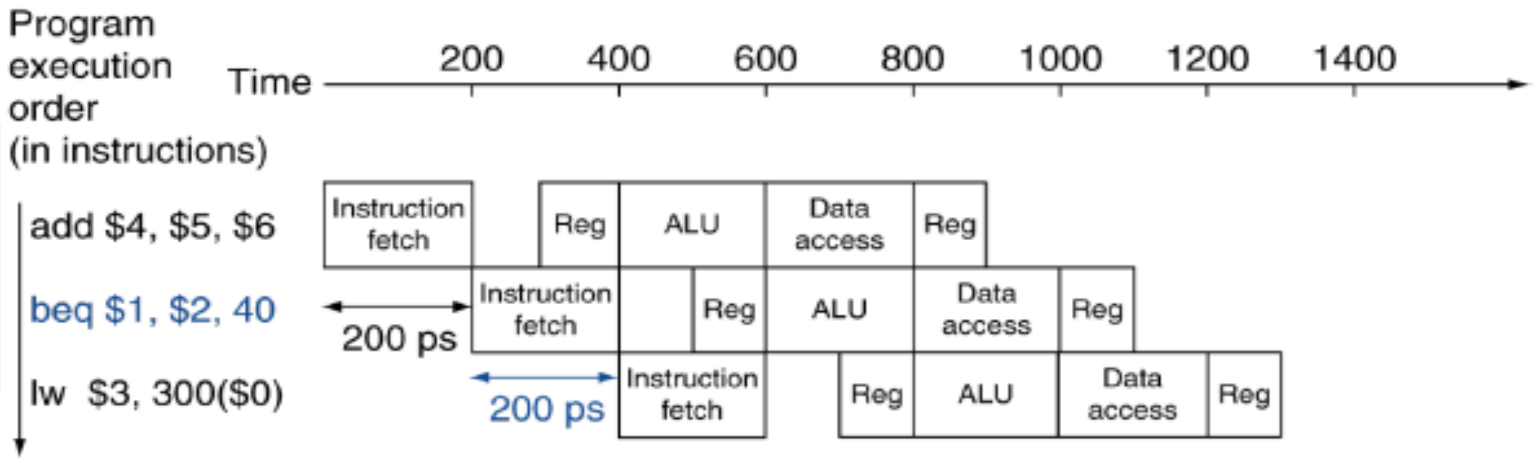
MIPS with Predict Not Taken

Prediction Correct that "branch not needed"

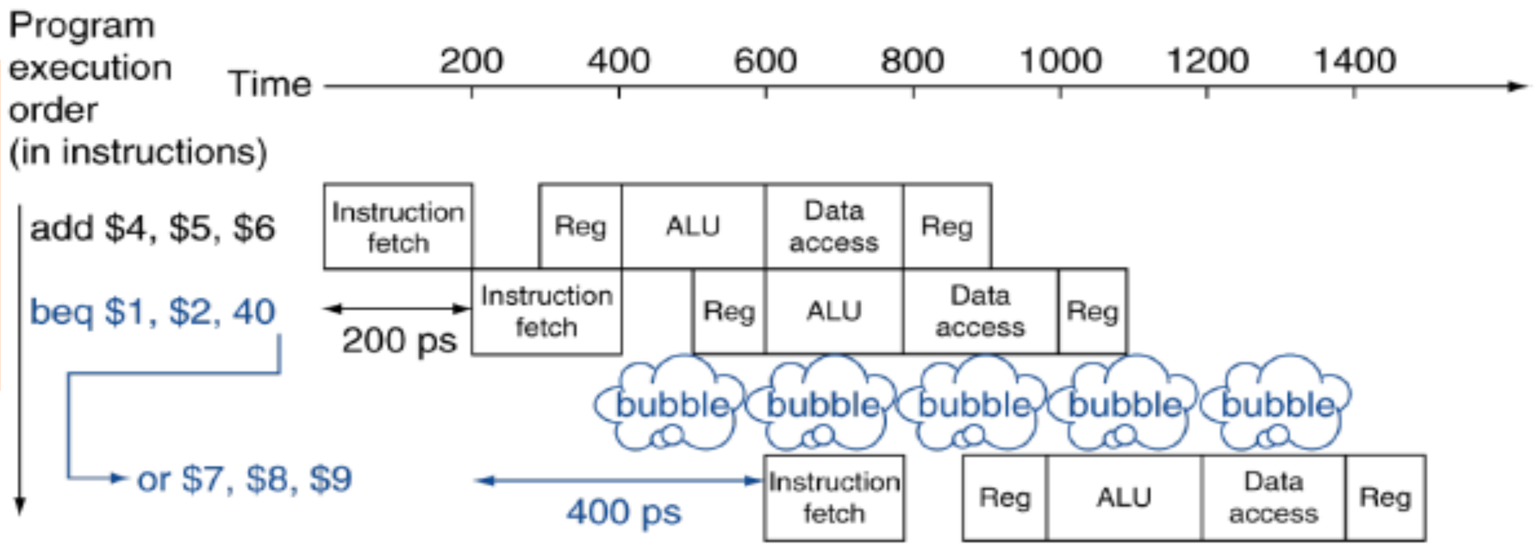


MIPS with Predict Not Taken

Prediction Correct that "branch not needed"



Prediction Incorrect that "branch not needed" (simplified e.g.)



More-Realistic Branch Predictions

- **Static branch prediction**
  - Based on “typical” branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
  - Not the best, but better than always predicting no-branch…

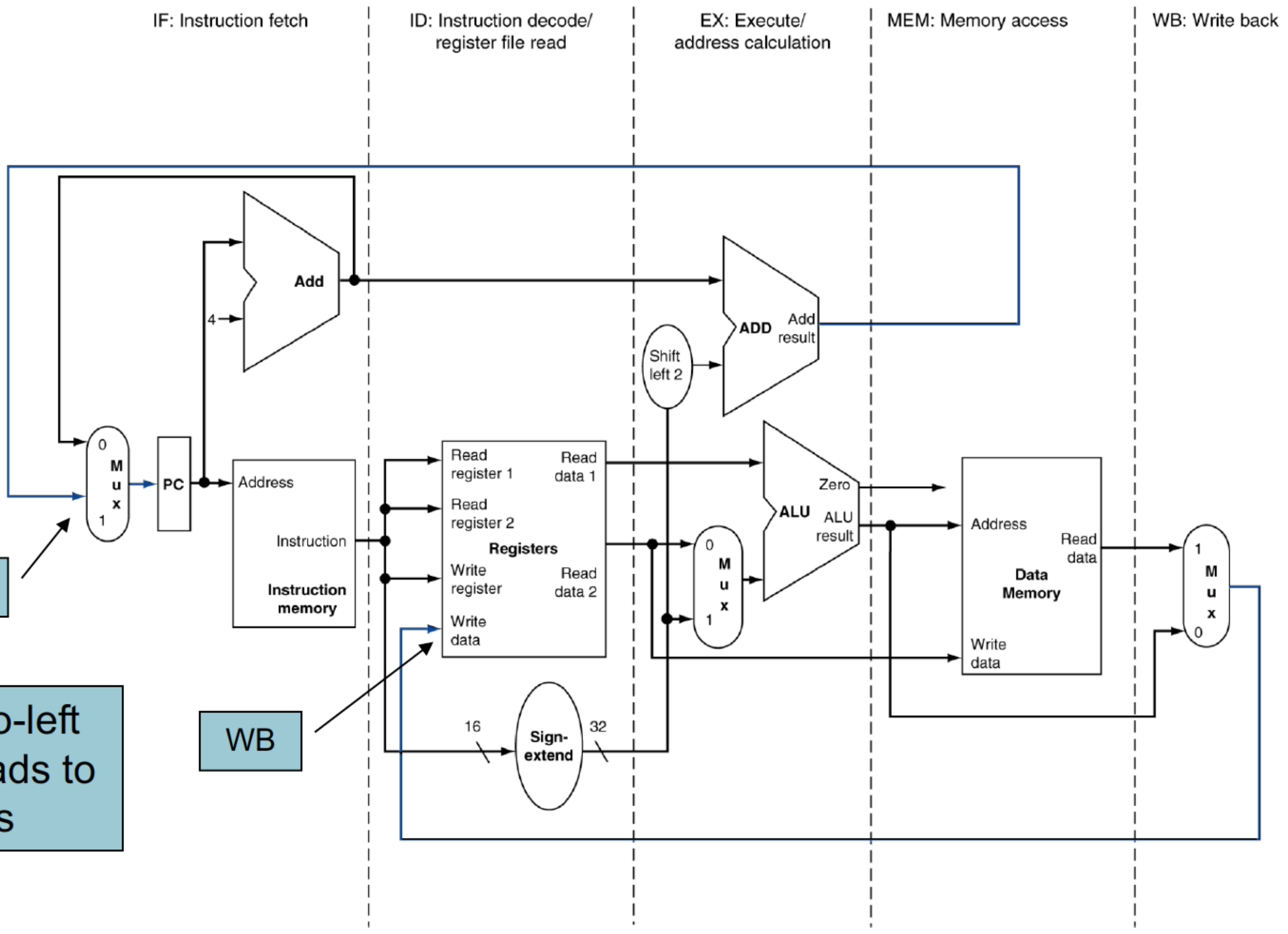
- **Dynamic branch prediction**
  - **Hardware** measures actual branch behavior
    - e.g., it record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history
    - Tends to be accurate ~90% of the time

Pipeline Summary

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction still has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

MIPS Pipelined Datapath: Requirements

Based on Single-Cycle Datapath Design…



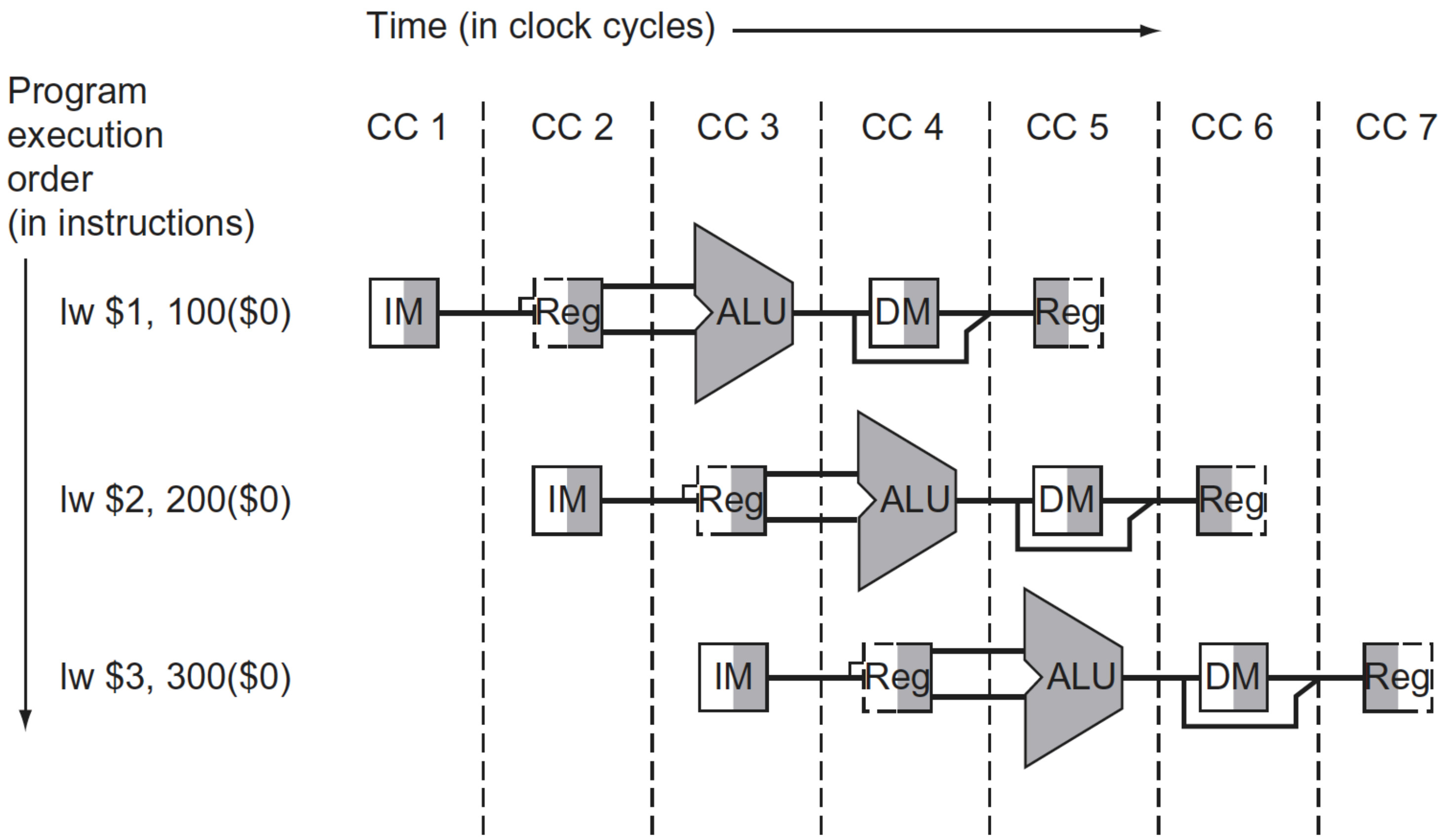
MEM

WB

Right-to-left flow leads to hazards

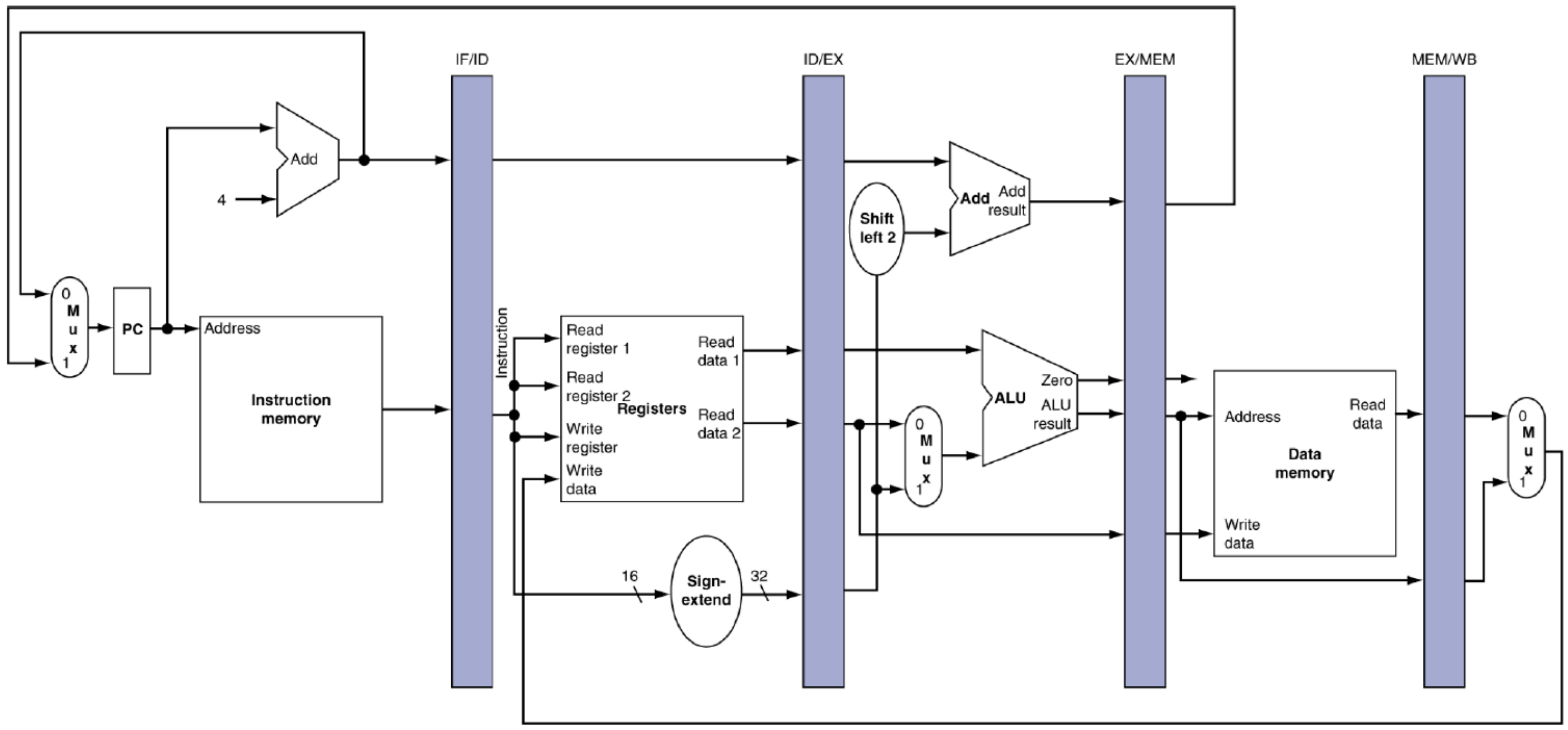
Example

- Instructions executed using the single-cycle datapath assuming pipelined execution.
  - In laundry analogy, we might have a basket between each pair of stages to hold the clothes for the next step



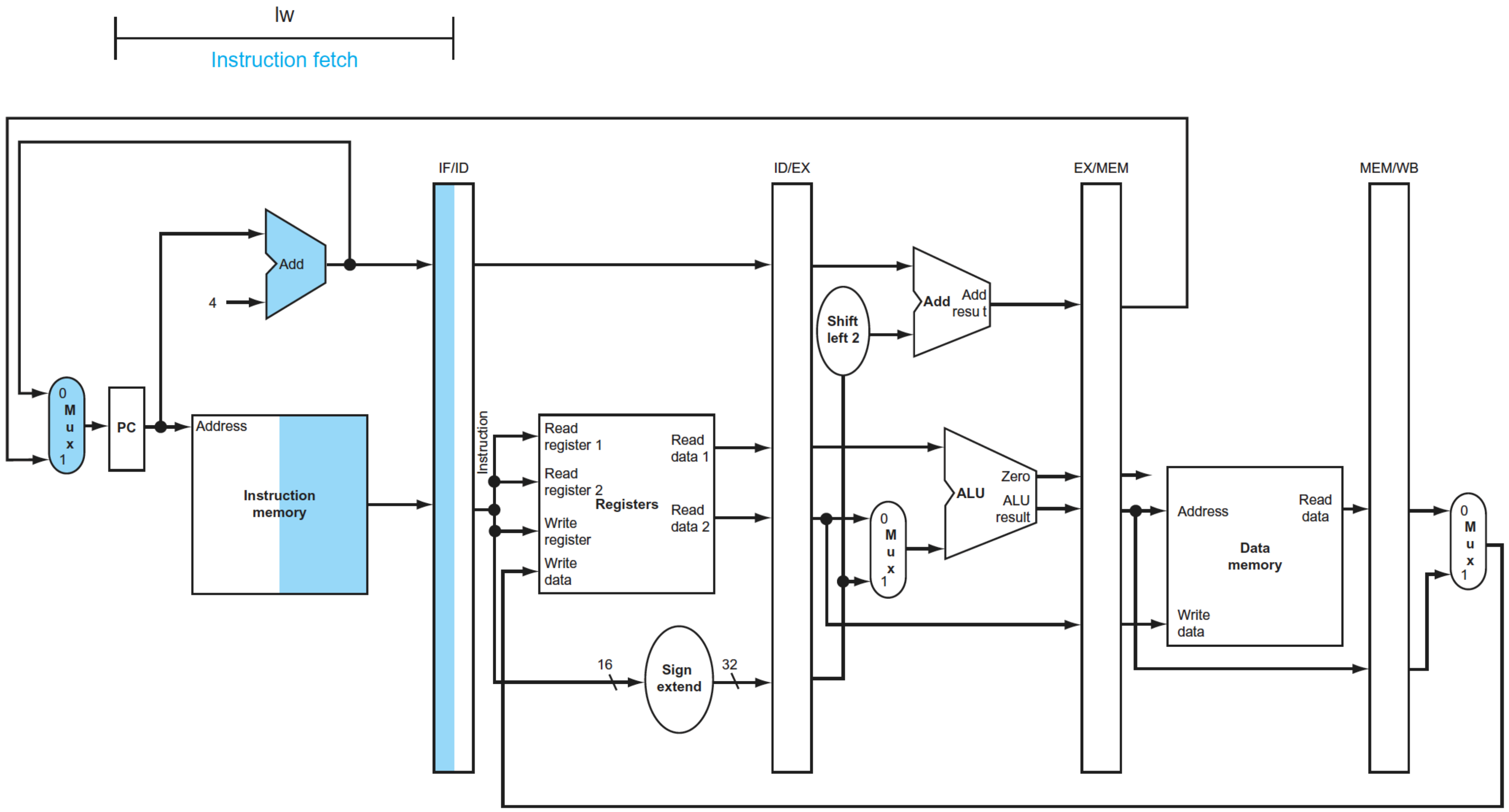
MIPS Pipelined Datapath

Need registers between stages to hold information produced in previous cycle



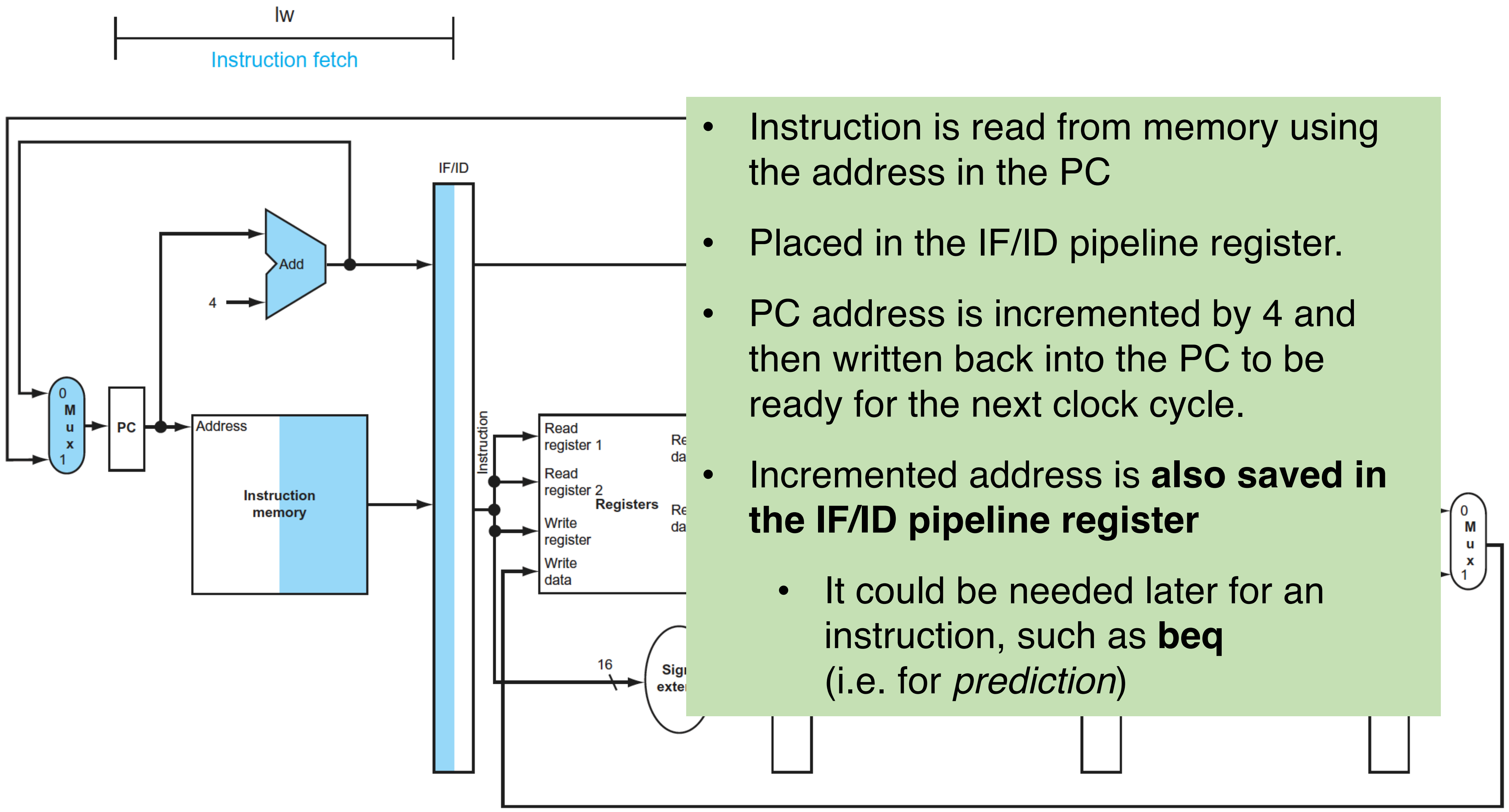
MIPS Pipelined Datapath for lw instruction: IF

- *Highlight the right half of registers or memory when they are being read*
- *Highlight the left half when they are being written*



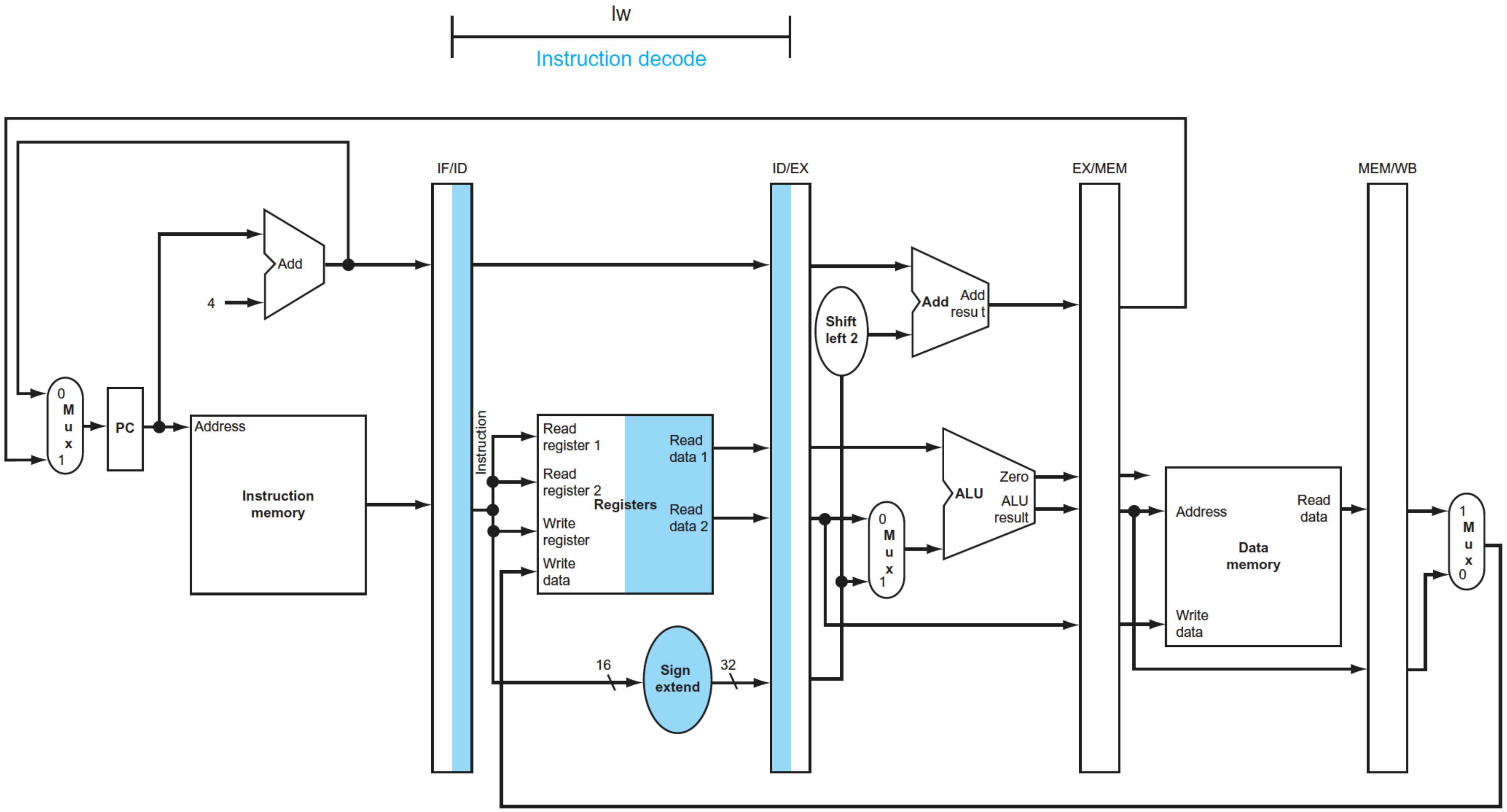
MIPS Pipelined Datapath for lw instruction: IF

- *Highlight the right half of registers or memory when they are being read*
- *Highlight the left half when they are being written*

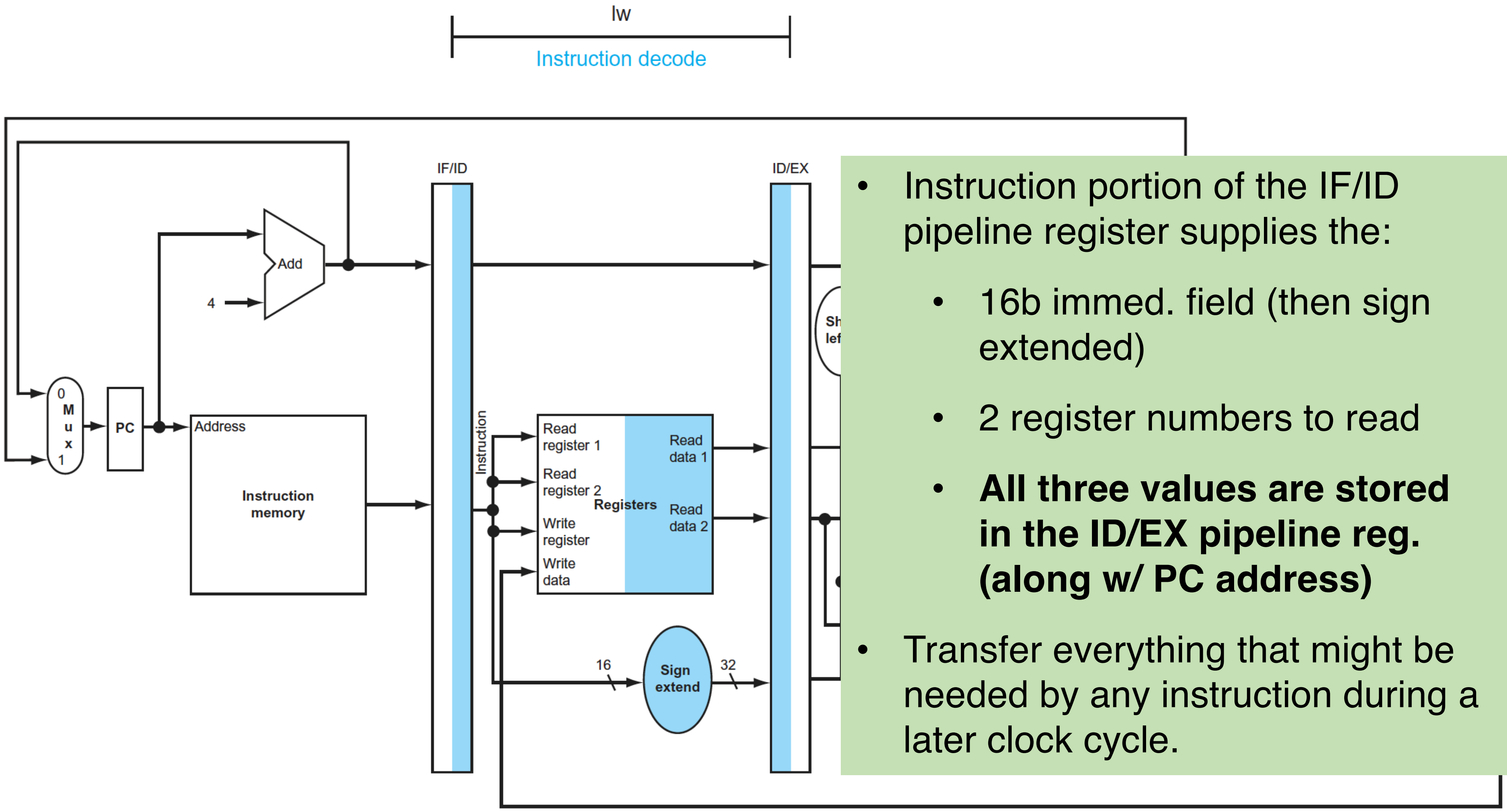


- Instruction is read from memory using the address in the PC
- Placed in the IF/ID pipeline register.
- PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle.
- Incremented address is **also saved in the IF/ID pipeline register**
  - It could be needed later for an instruction, such as **beq** (i.e. for *prediction*)

MIPS Pipelined Datapath for lw instruction: ID

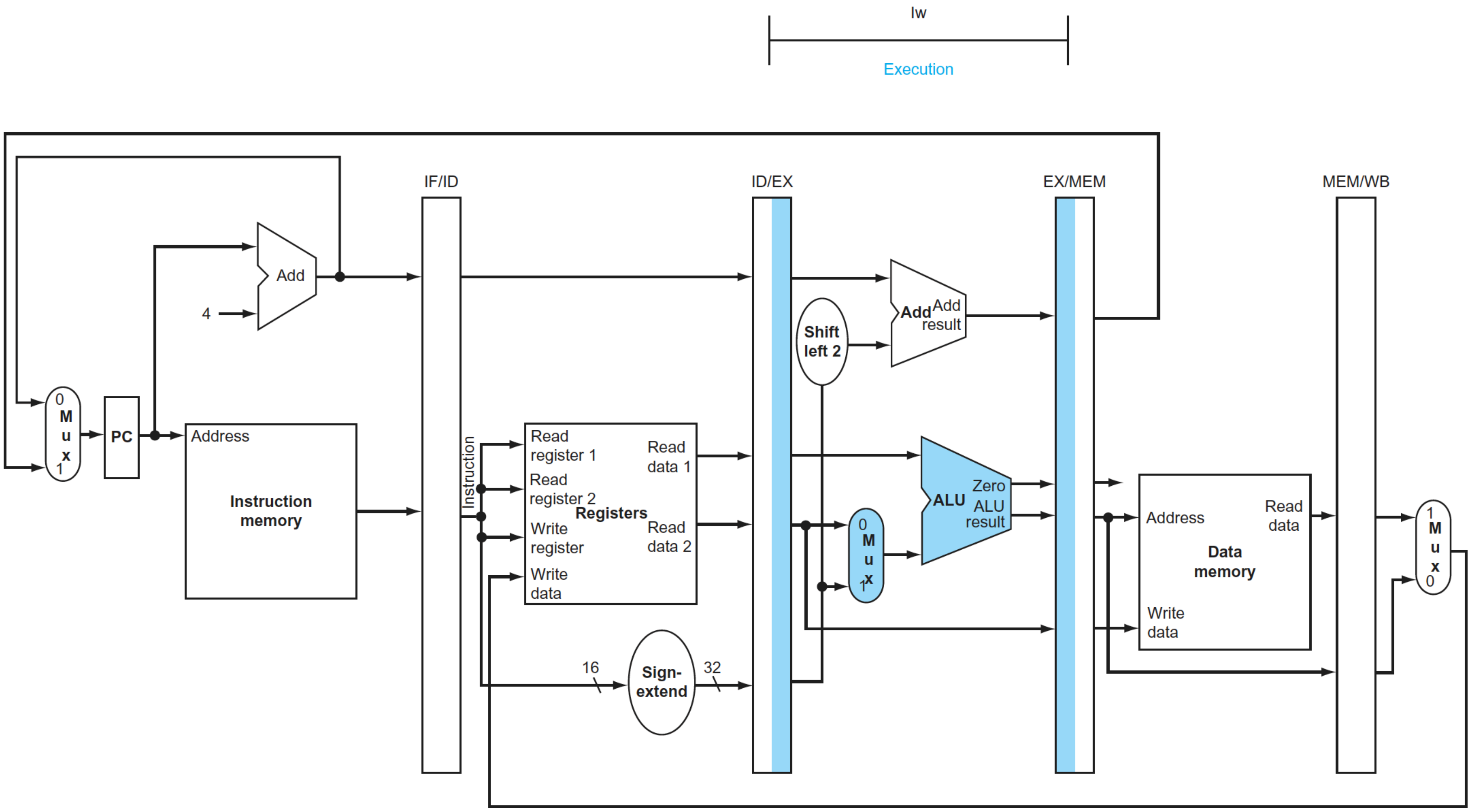


MIPS Pipelined Datapath for lw instruction: ID

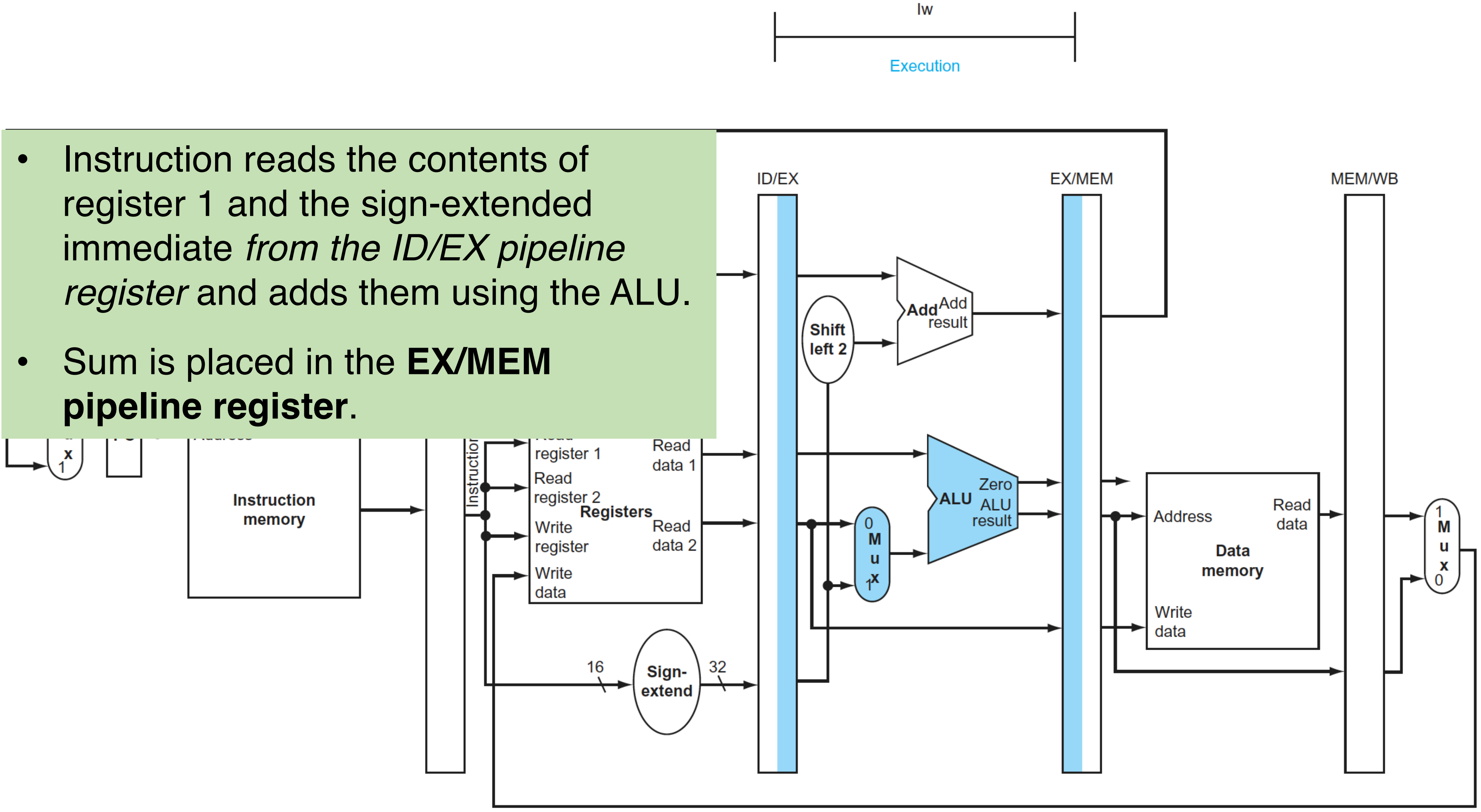


- Instruction portion of the IF/ID pipeline register supplies the:
  - 16b immed. field (then sign extended)
  - 2 register numbers to read
  - **All three values are stored in the ID/EX pipeline reg. (along w/ PC address)**
- Transfer everything that might be needed by any instruction during a later clock cycle.

MIPS Pipelined Datapath for lw instruction: EX

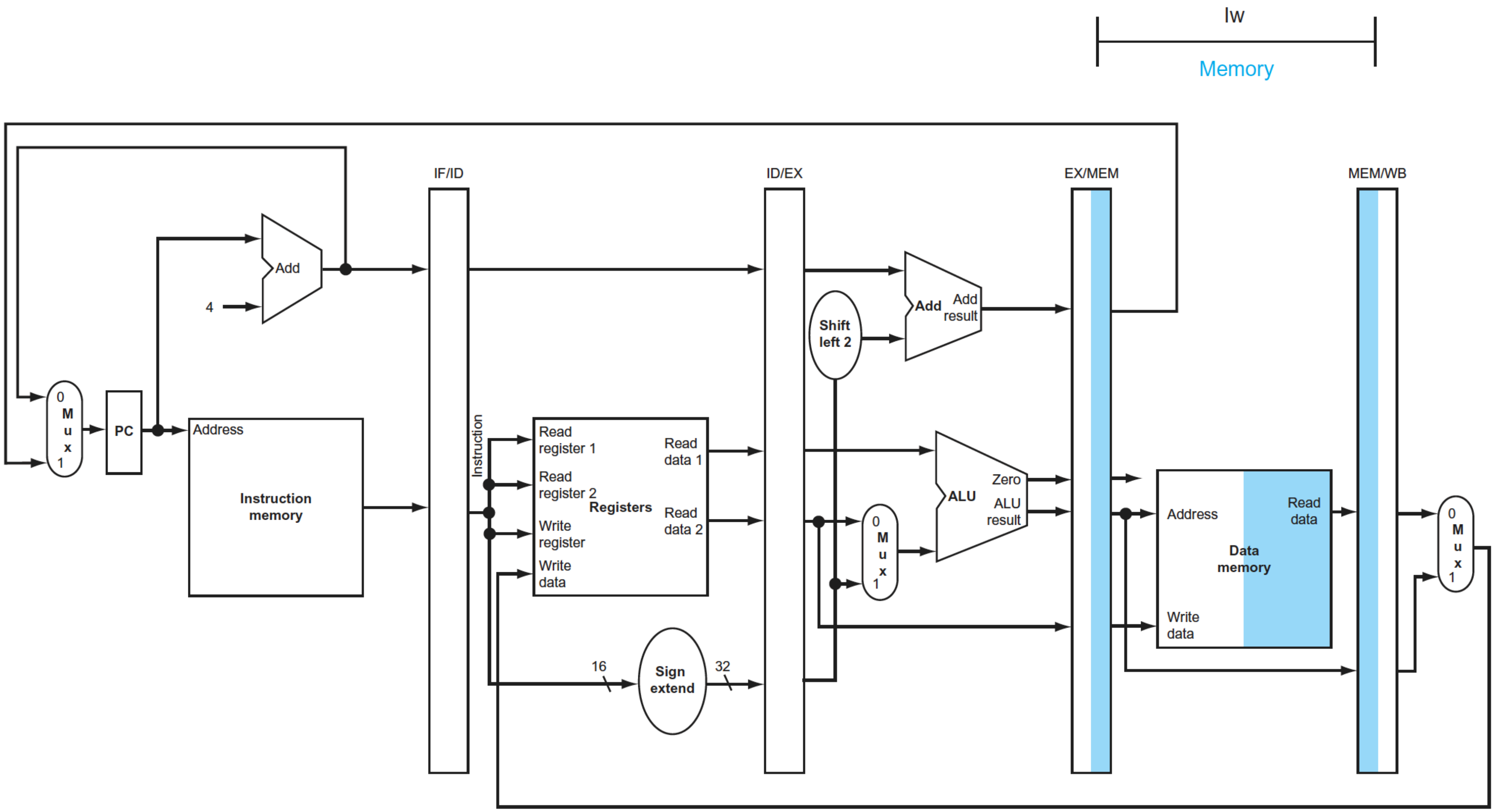


MIPS Pipelined Datapath for lw instruction: EX



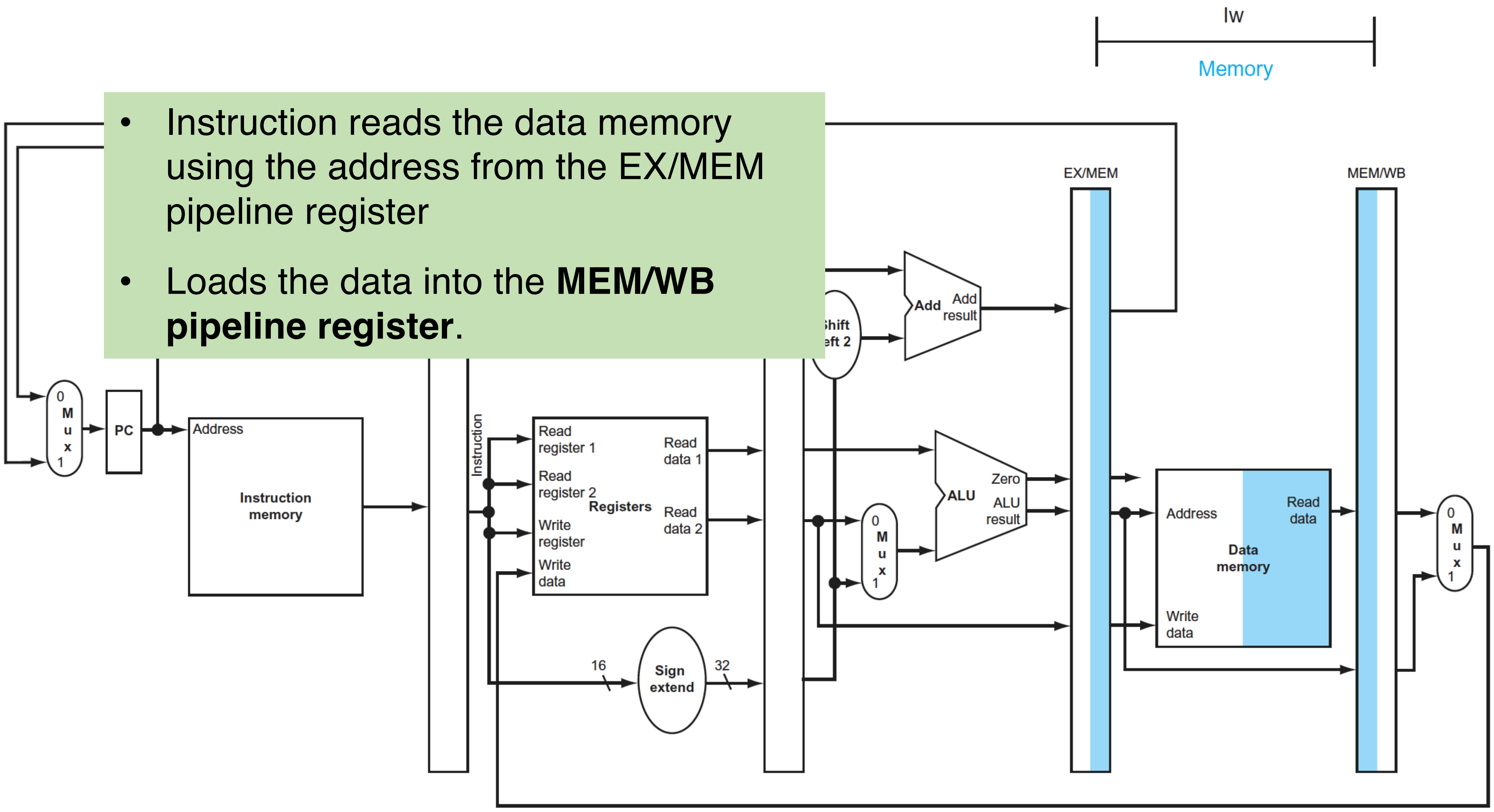
- Instruction reads the contents of register 1 and the sign-extended immediate *from the ID/EX pipeline register* and adds them using the ALU.
- Sum is placed in the **EX/MEM pipeline register**.

MIPS Pipelined Datapath for lw instruction: MEM

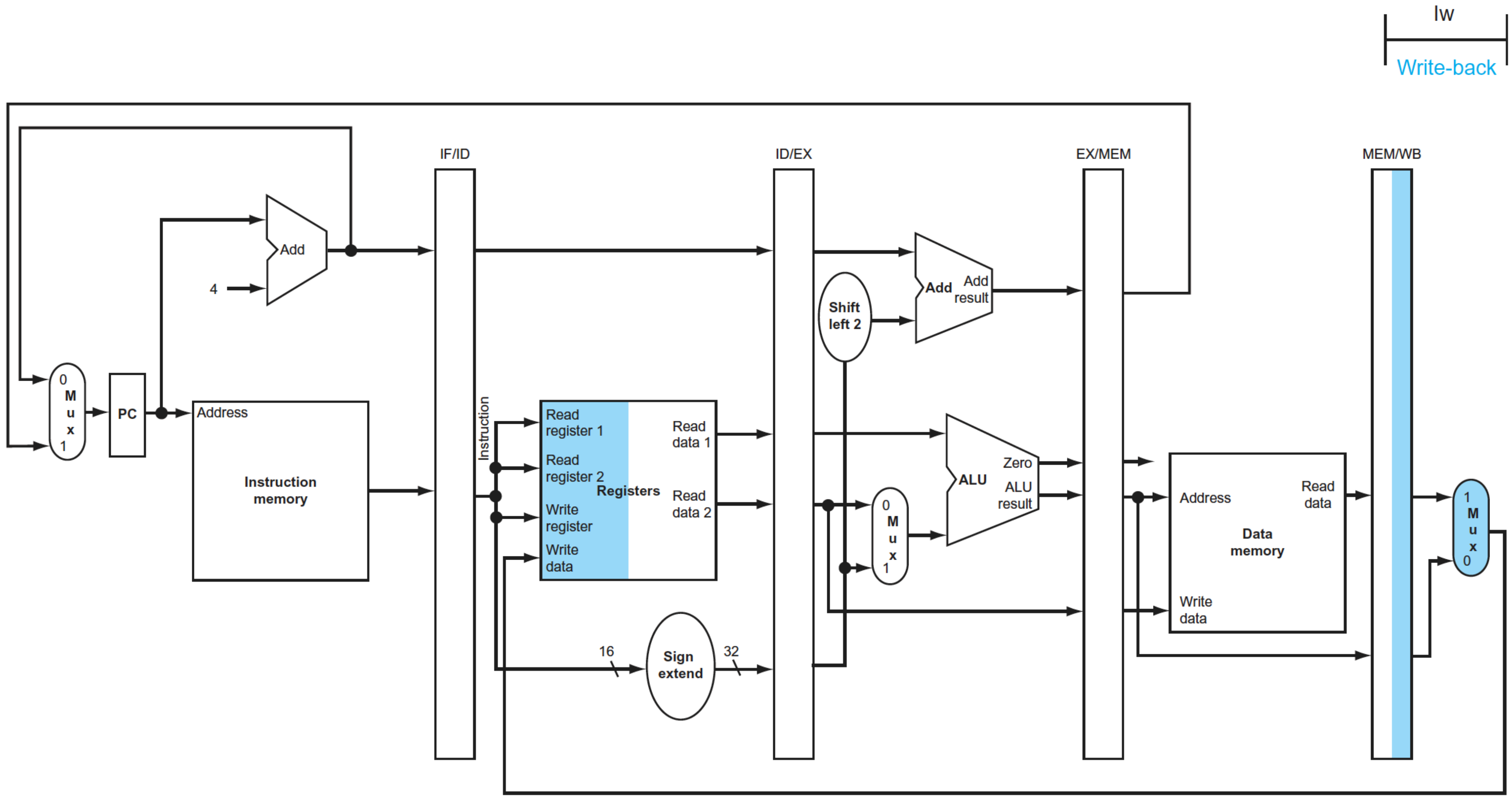


MIPS Pipelined Datapath for lw instruction: MEM

- Instruction reads the data memory using the address from the EX/MEM pipeline register
- Loads the data into the **MEM/WB pipeline register**.

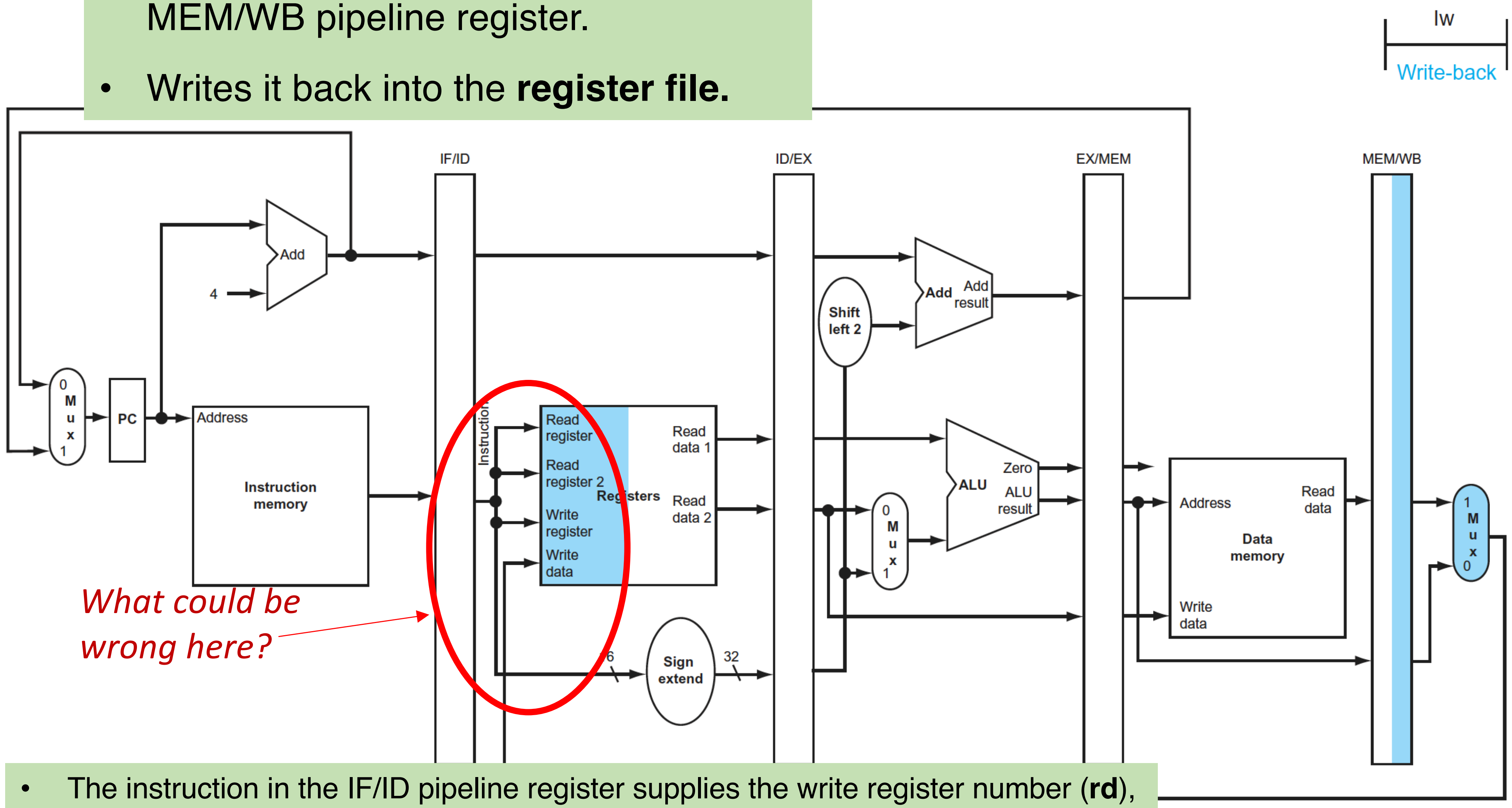


MIPS Pipelined Datapath for lw instruction: WB



MIPS Pipelined Datapath for lw instruction: WB

- Instruction reads data from the MEM/WB pipeline register.
- Writes it back into the **register file.**

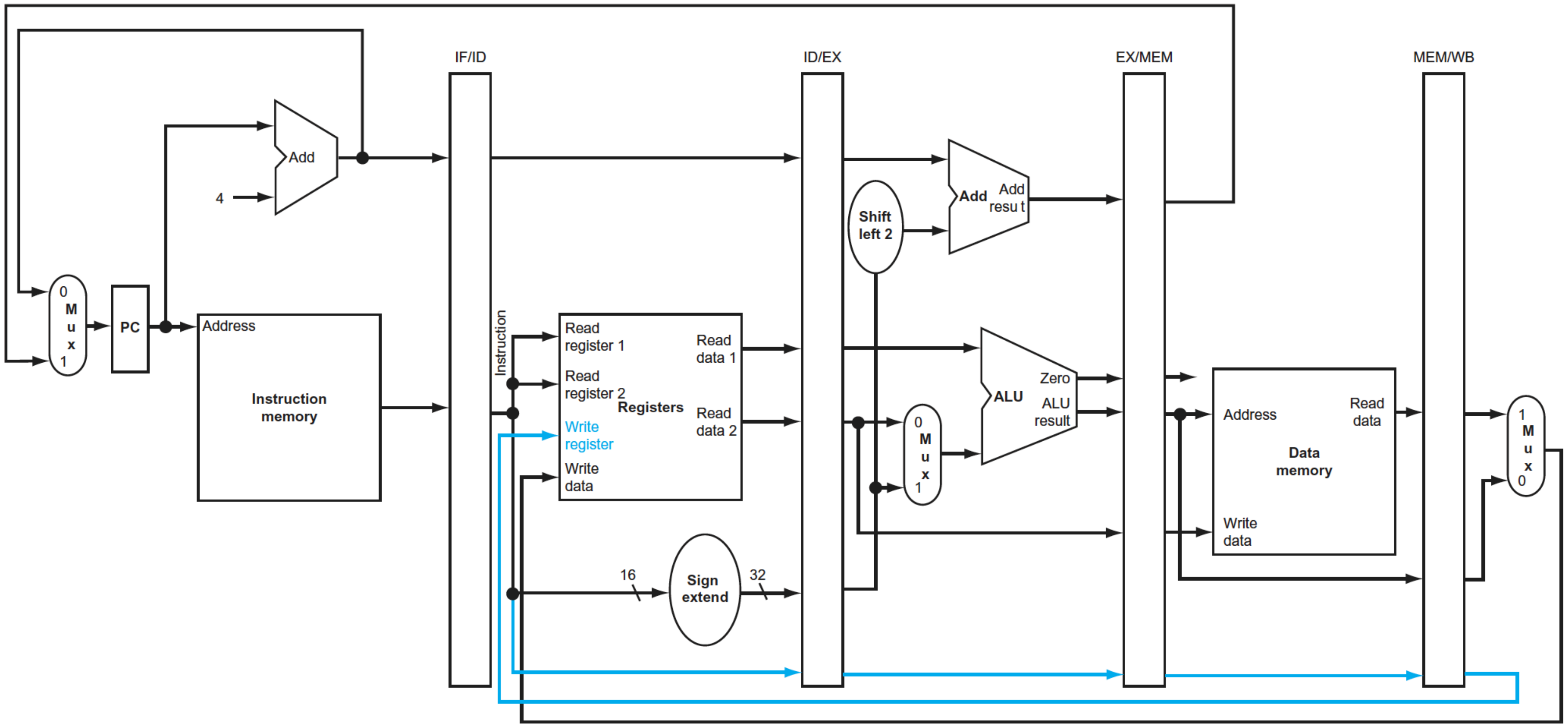


What could be wrong here?

- The instruction in the IF/ID pipeline register supplies the write register number (**rd**), **BUT** the "writing" occurs *at the end* of the load instruction!
- We need to *preserve* the **rd** number in the load instruction…

Corrected Pipelined Datapath

Otherwise load instruction wouldn't work properly…



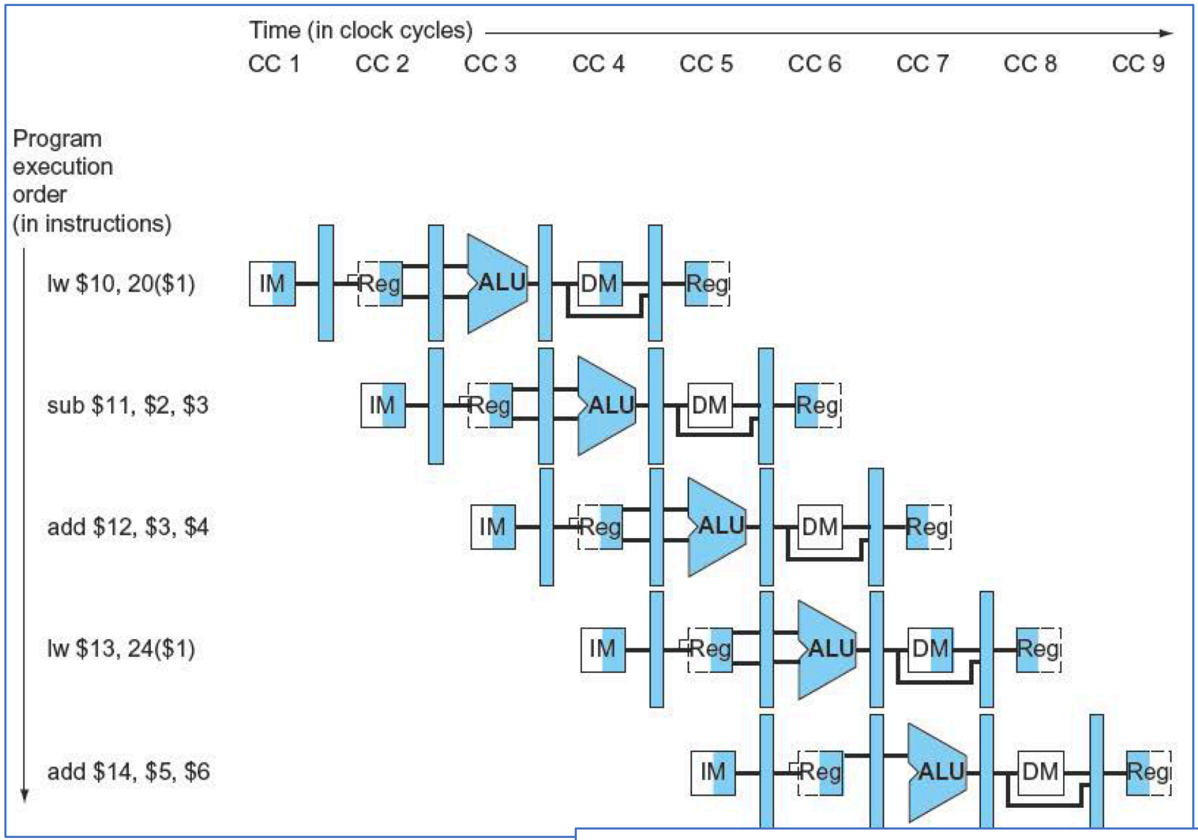
Summary (so far)

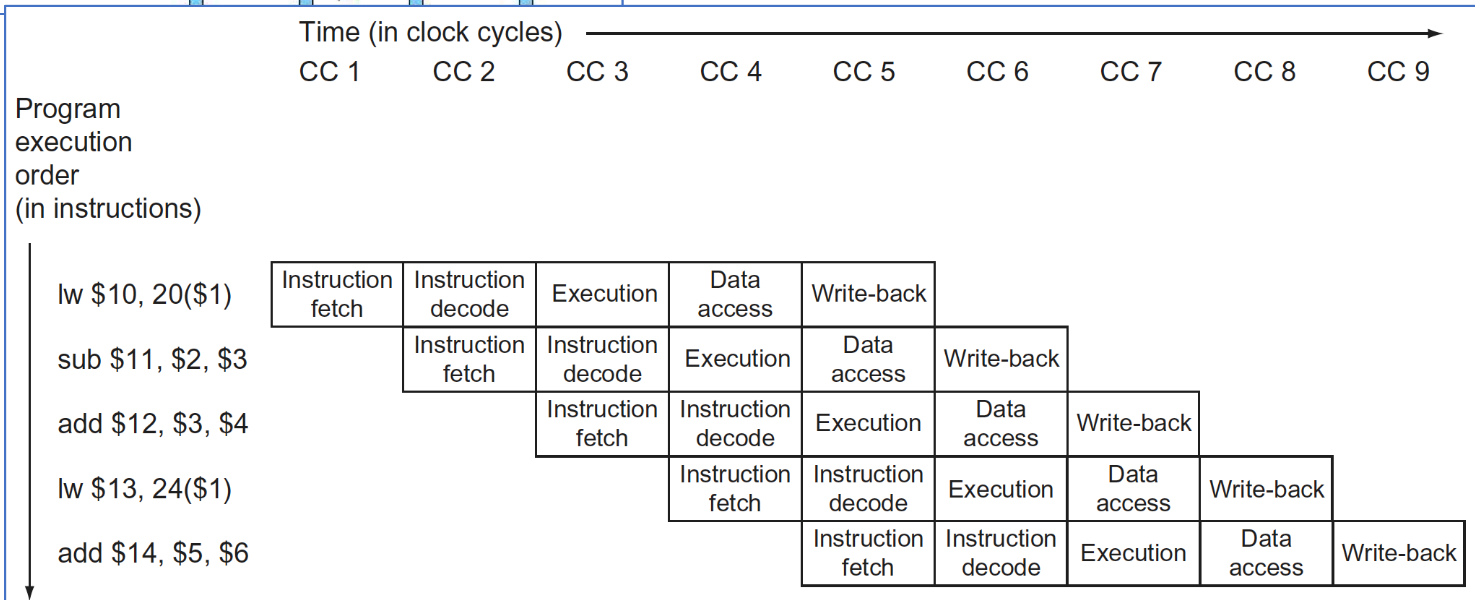
- To pass something from an early pipe stage to a later pipe stage, the information **must be placed in a pipeline register**
  - Otherwise, the information is lost when the next instruction enters that pipeline stage.

- Each logical component of the datapath can be used **only within a single pipeline stage**.

- We have seen 2 types of representations:
  - multiple-clock-cycle pipeline diagrams *Overview, fewer details*
  - single-clock-cycle pipeline diagrams *Full details*

Multiple-Clock-Cycle Pipeline Diagrams of Five Instructions

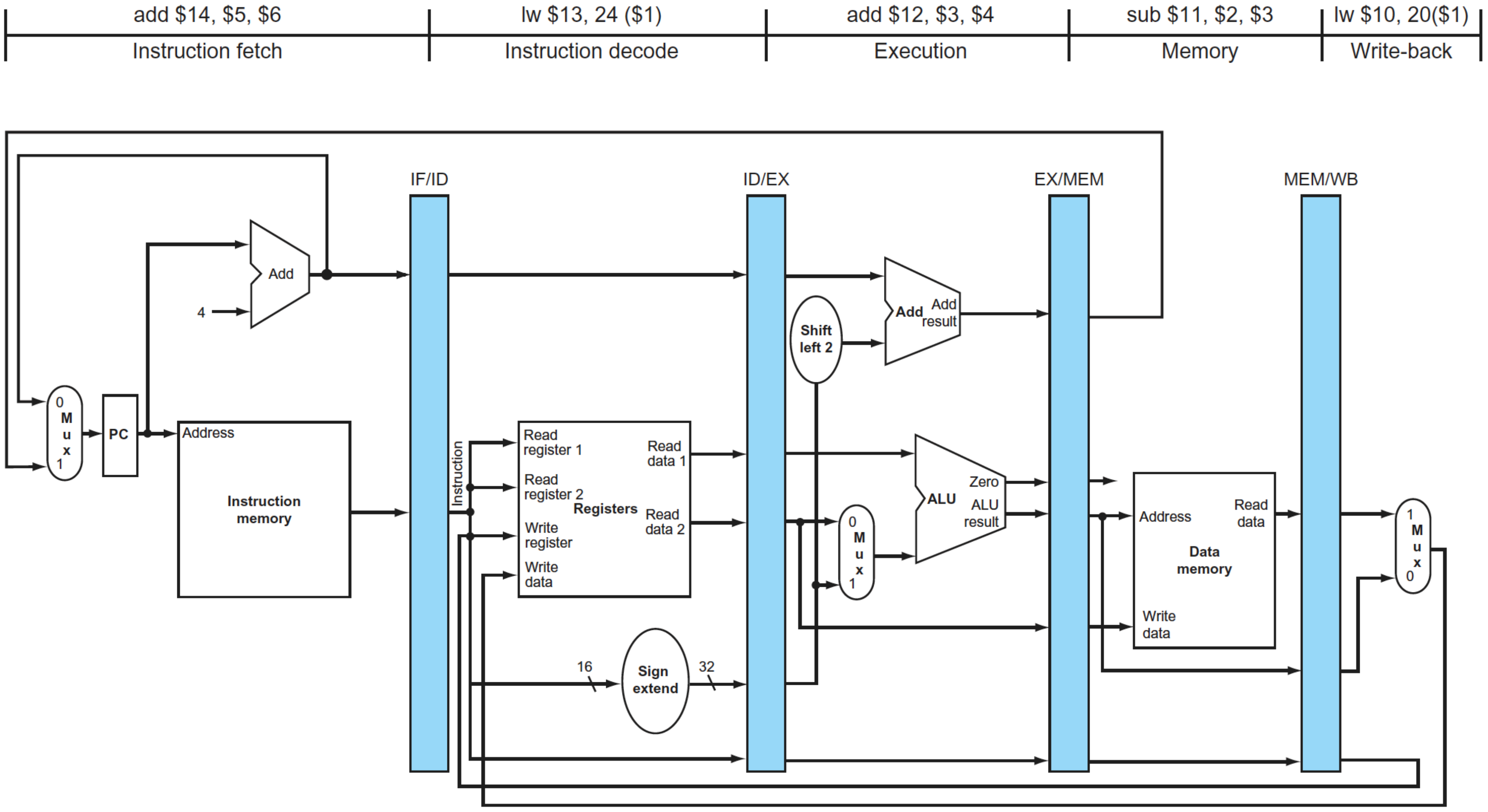



Time (in clock cycles)

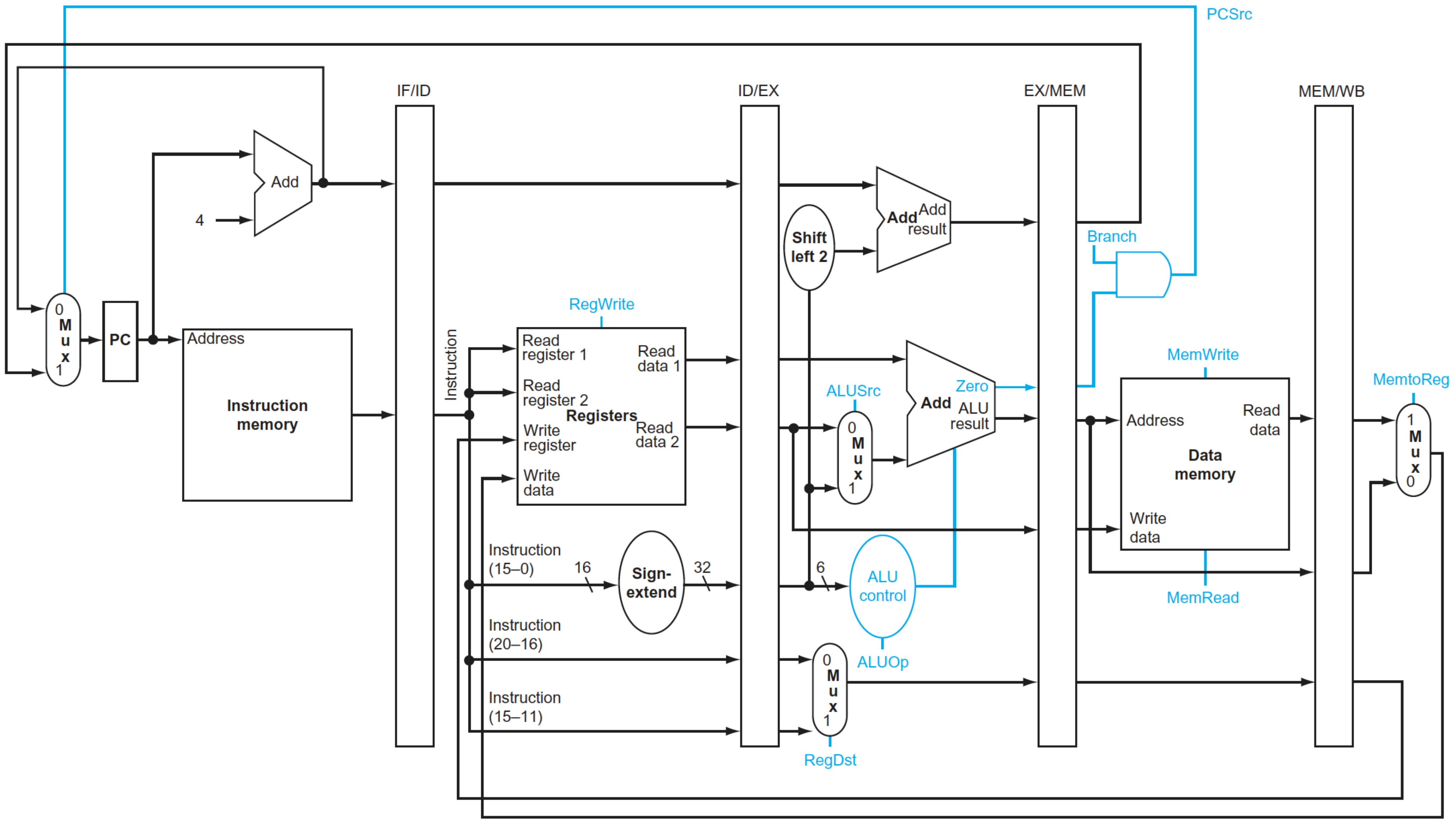
Program execution order (in instructions)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write-back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write-back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write-back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write-back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write-back

Single-Clock-Cylce Pipeline Diagram of Five Instructions *Corresponding to Clock Cycle 5*



Simplified Pipeline Control Diagram (seems familiar?)

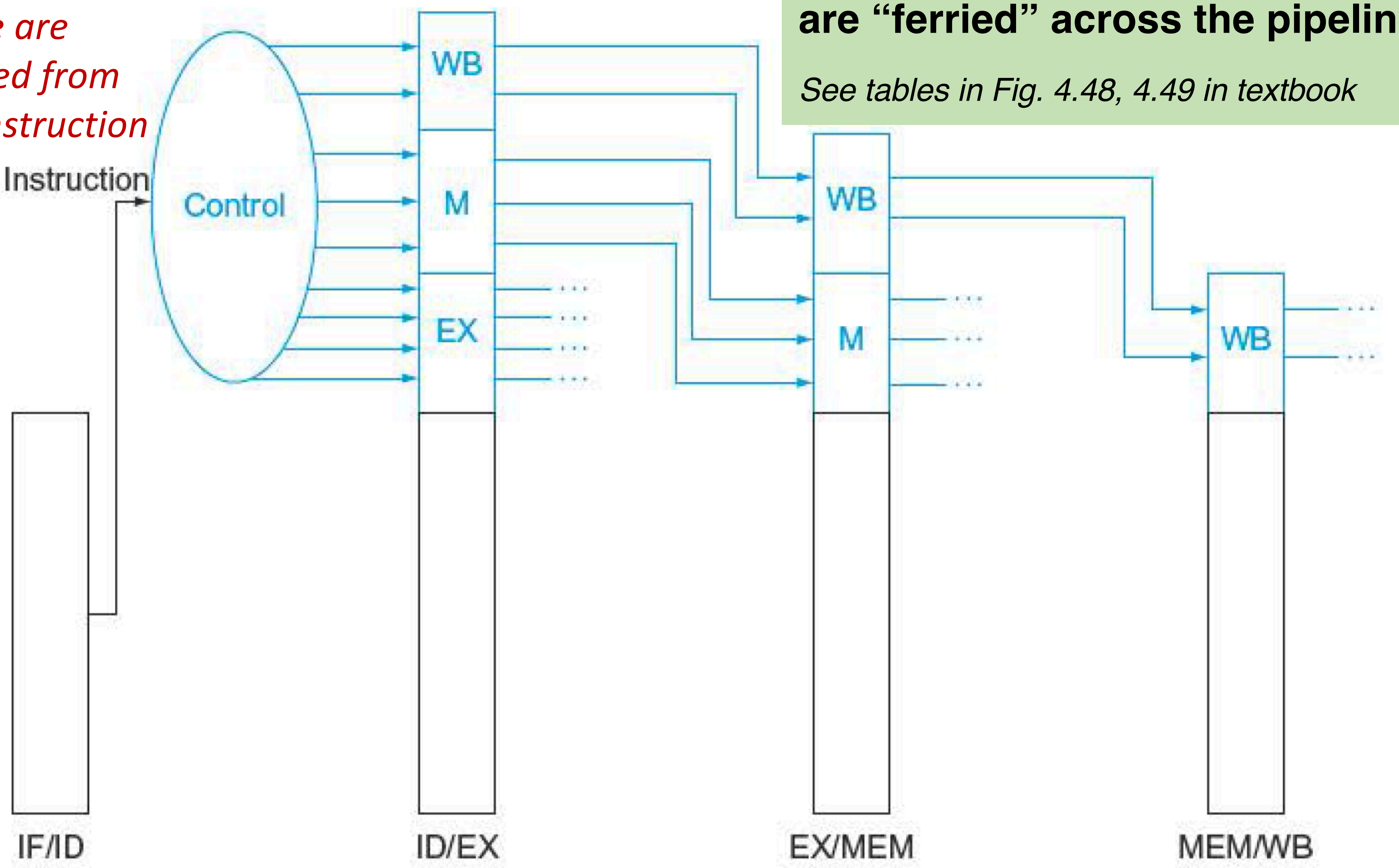


Control Lines for the Last 3 Pipeline Stages

Same control signals that we learned earlier, but this time they are “ferried” across the pipelines

See tables in Fig. 4.48, 4.49 in textbook

These are derived from the instruction



YOUR TO-DOs for the Week

- Finish Lab 7 by Sunday

- New Lab 8 (last one!) will be issued later on this week
  - Due next week, which is last week of classes… ☹

</LECTURE>