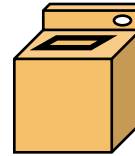


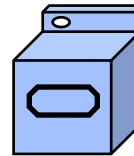
A relevant question

❑ Assuming you've got:

– One washer (takes 30 minutes)



– One drier (takes 40 minutes)



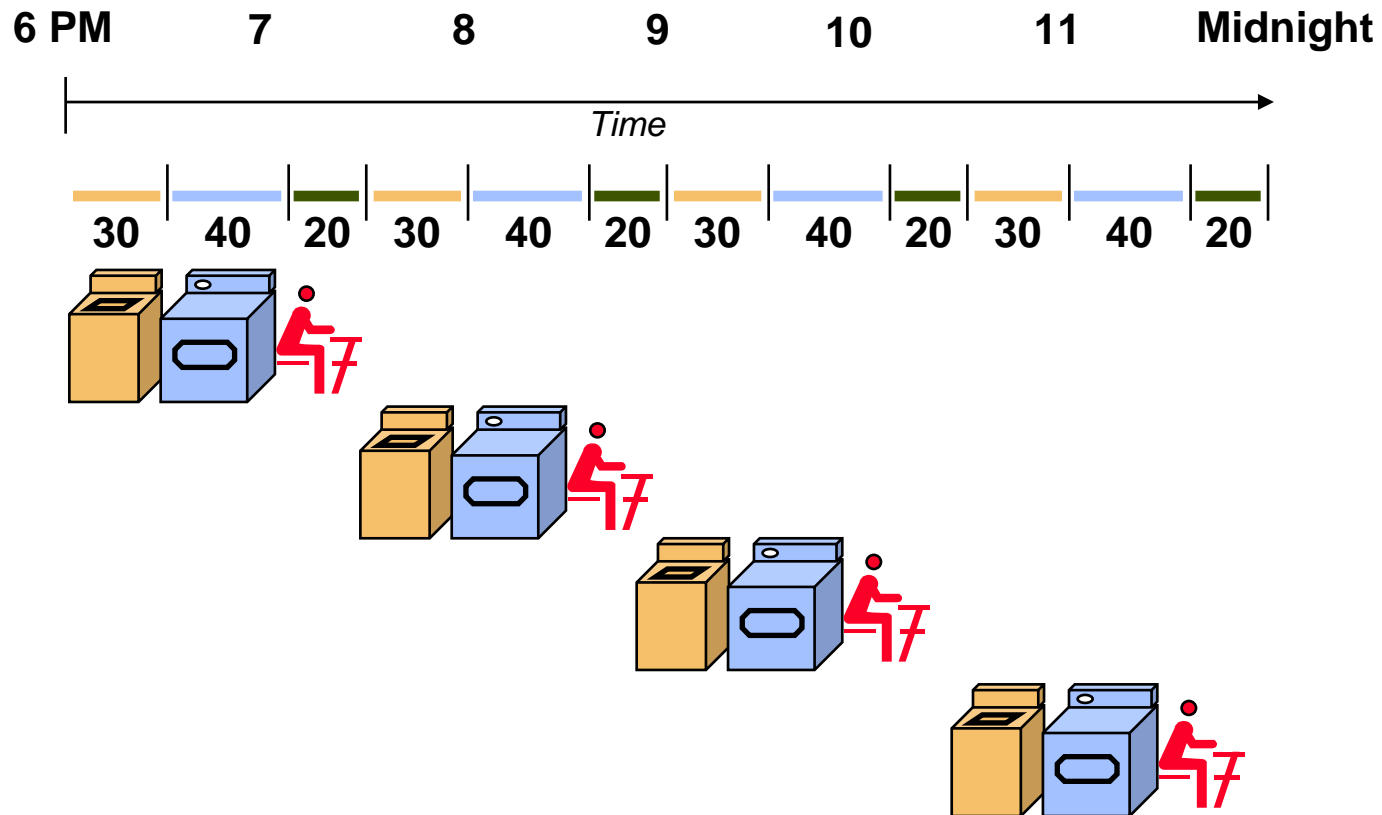
– One “folder” (takes 20 minutes)



❑ It takes 90 minutes to wash, dry, and fold 1 load of laundry.

– How long does 4 loads take?

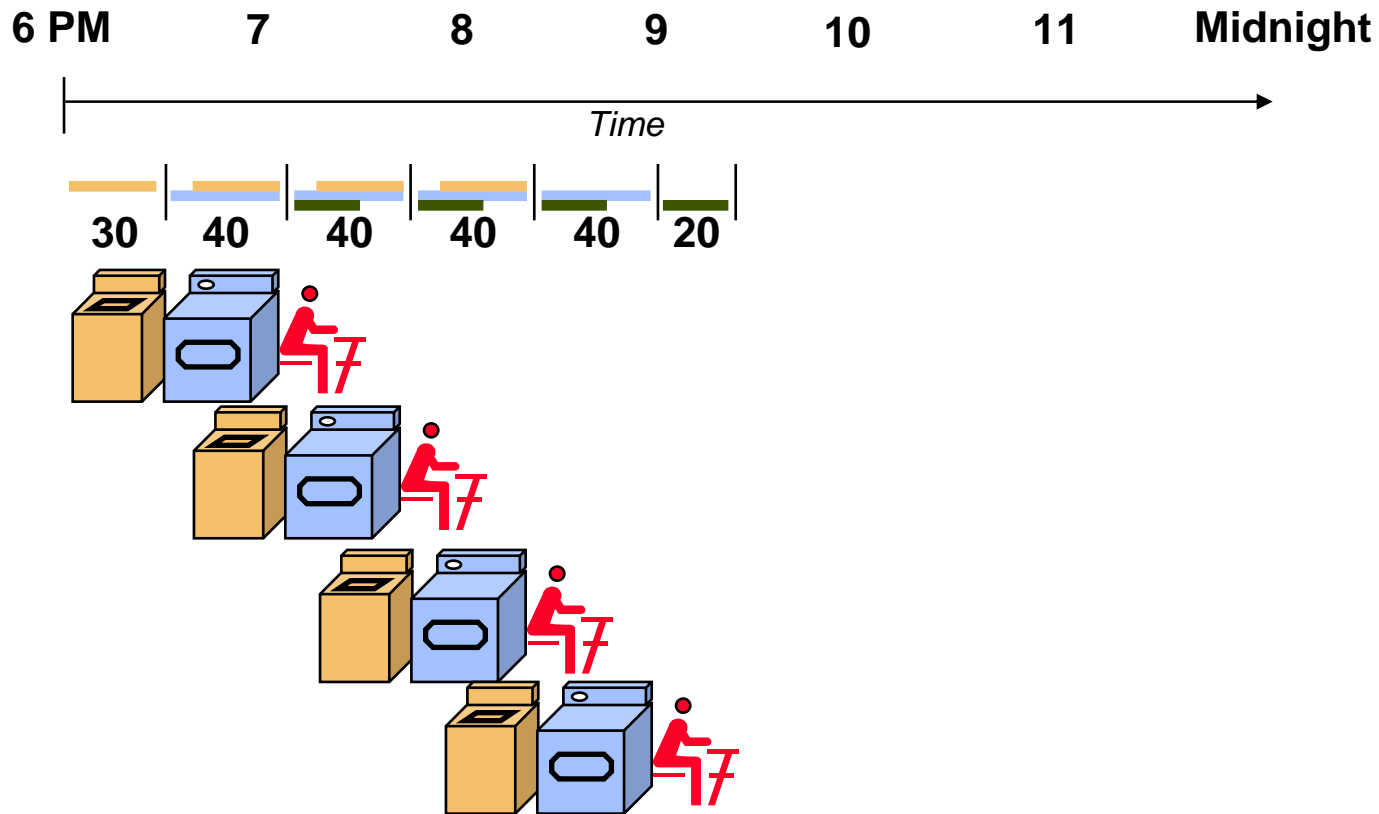
The slow way



□ If each load is done sequentially it takes 6 hours

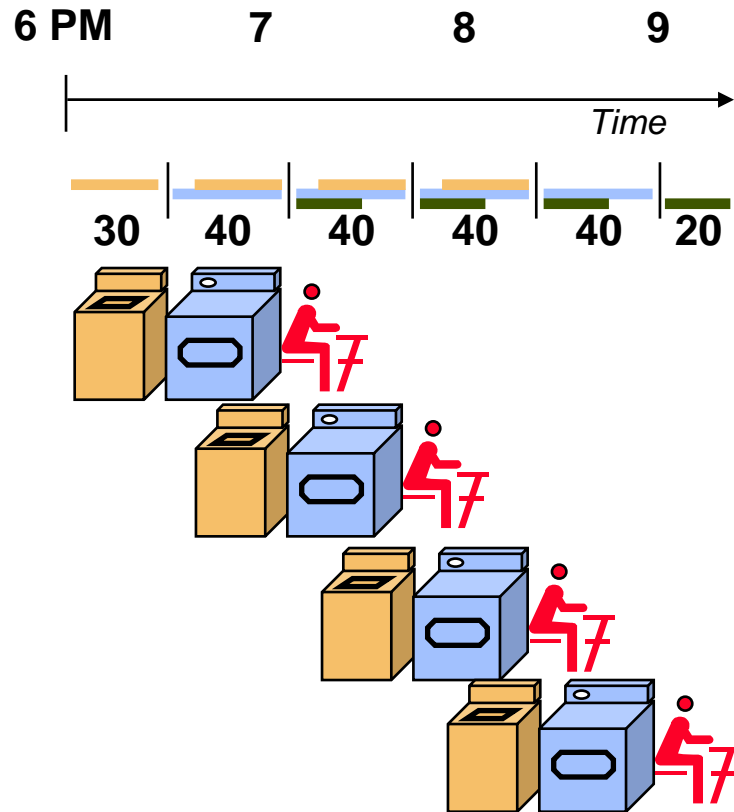
Laundry Pipelining

- Start each load as soon as possible
 - Overlap loads



- Pipelined laundry takes 3.5 hours

Pipelining Lessons



- ❑ Pipelining doesn't help **latency** of single load, it helps **throughput** of entire workload
- ❑ Pipeline rate limited by **slowest** pipeline stage
- ❑ **Multiple** tasks operating simultaneously using different resources
- ❑ Potential speedup = **Number pipe stages**
- ❑ Unbalanced lengths of pipe stages reduces speedup
- ❑ Time to "**fill**" pipeline and time to "**drain**" it reduces speedup

Pipelining

- ❑ **Pipelining is a general-purpose efficiency technique**
 - It is not specific to processors
- ❑ **Pipelining is used in:**
 - Assembly lines
 - Fast food restaurants
- ❑ **Pipelining** gives the best of both worlds and is used in just about every modern processor.

Instruction execution review

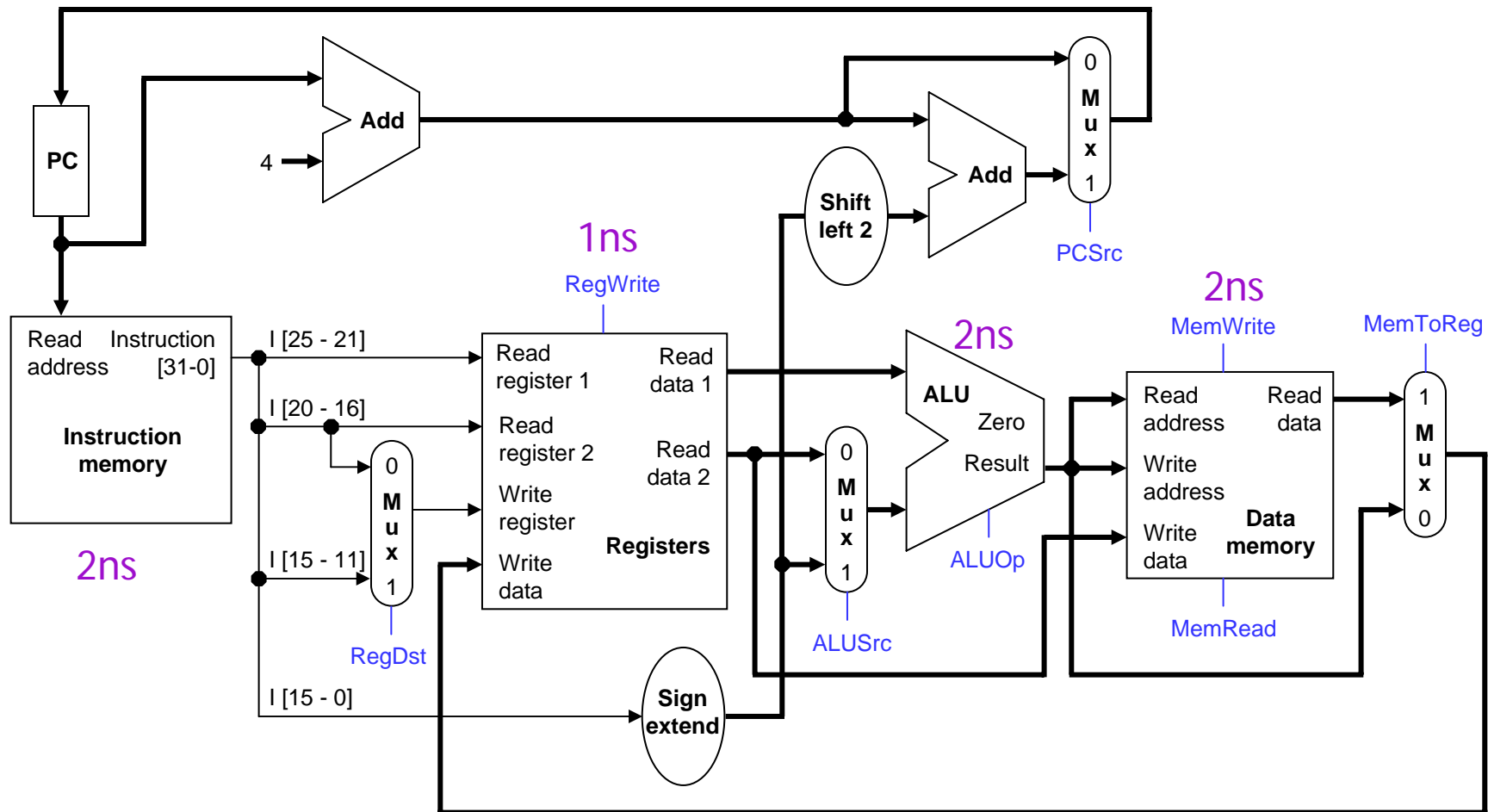
- ❑ Executing a MIPS instruction can take up to five steps.

Step	Name	Description
Instruction Fetch	IF	Read an instruction from memory.
Instruction Decode	ID	Read source registers and generate control signals.
Execute	EX	Compute an R-type result or a branch outcome.
Memory	MEM	Read or write the data memory.
Writeback	WB	Store a result in the destination register.

- ❑ However, as we saw, not all instructions need all five steps.

Instruction	Steps required				
beq	IF	ID	EX		
R-type	IF	ID	EX		WB
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

Single-cycle datapath diagram



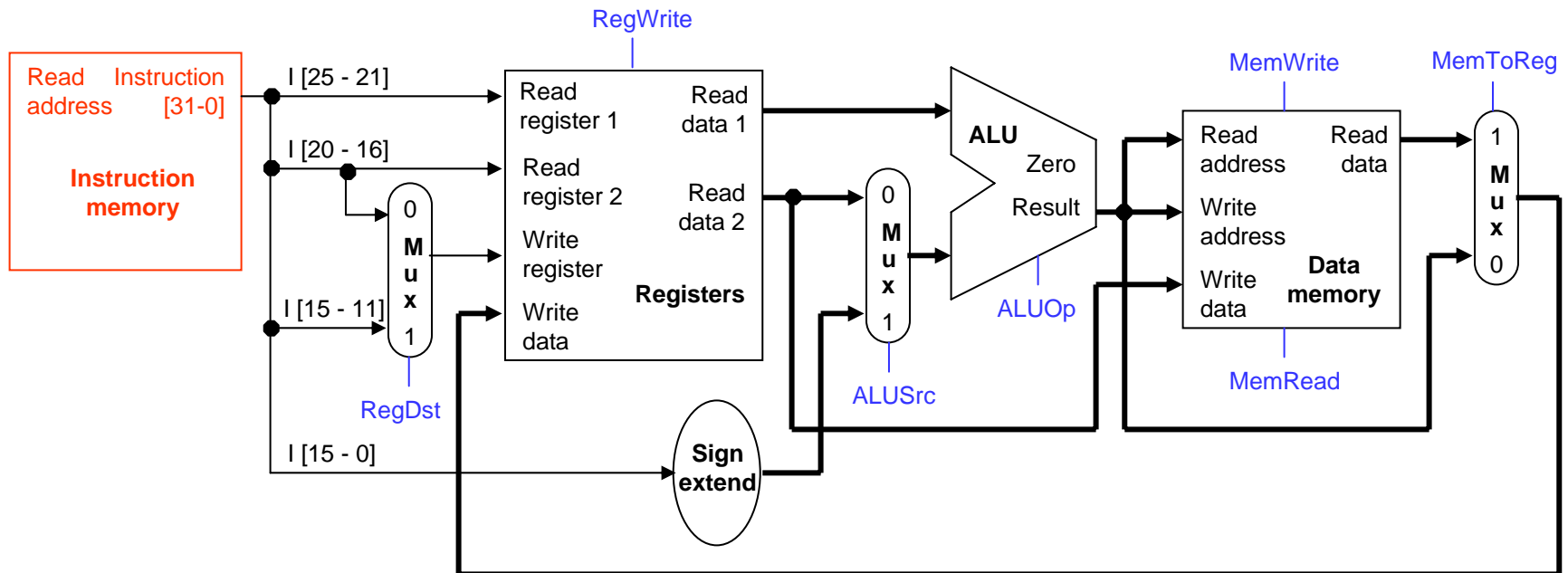
❑ How long does it take to execute each instruction?

Single-cycle review

- ❑ **All five execution steps occur in one clock cycle.**
- ❑ **Each hardware element can only be used once per clock cycle.**
 - A “lw” or “sw” must access memory twice (in the IF and MEM stages), so there are separate instruction and data memories.
 - There are multiple adders, since each instruction increments the PC (IF) *and* performs another computation (EX). On top of that, branches also need to compute a target address.

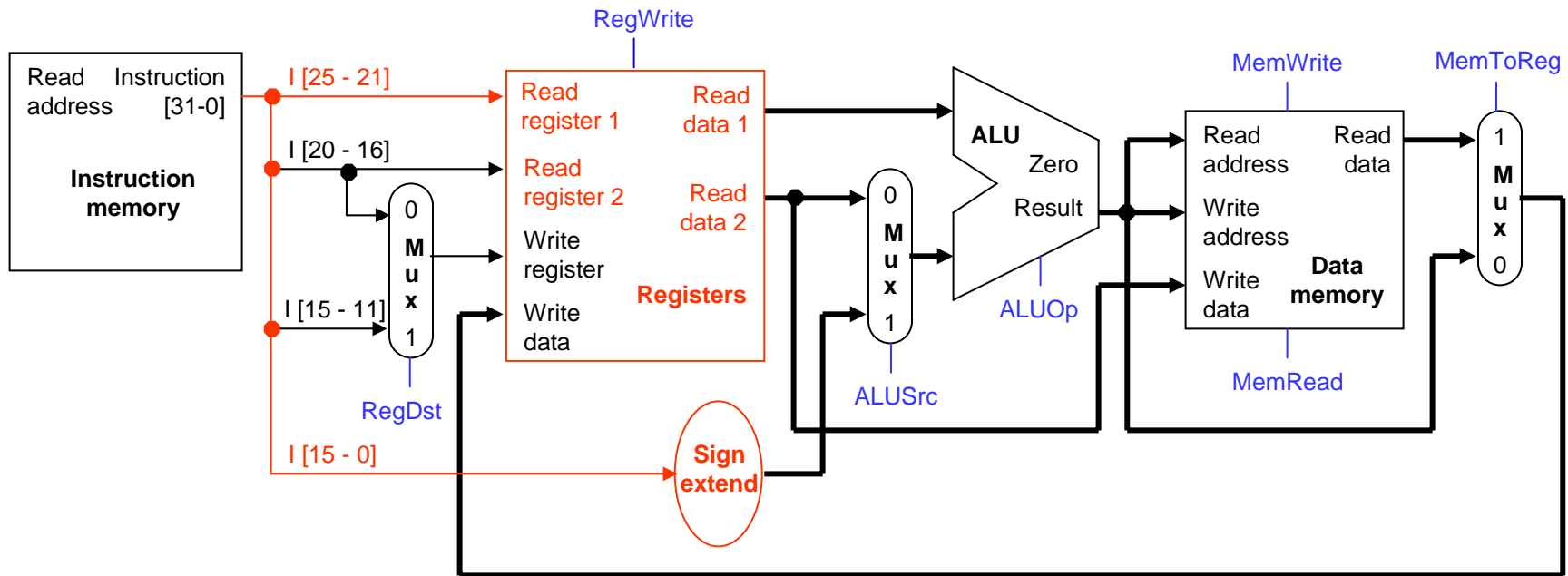
Review: Instruction Fetch (IF)

- ❑ Let's quickly review how `lw` is executed in the single-cycle datapath.
- ❑ We'll ignore PC incrementing and branching for now.
- ❑ In the Instruction Fetch (IF) step, we read the instruction memory.



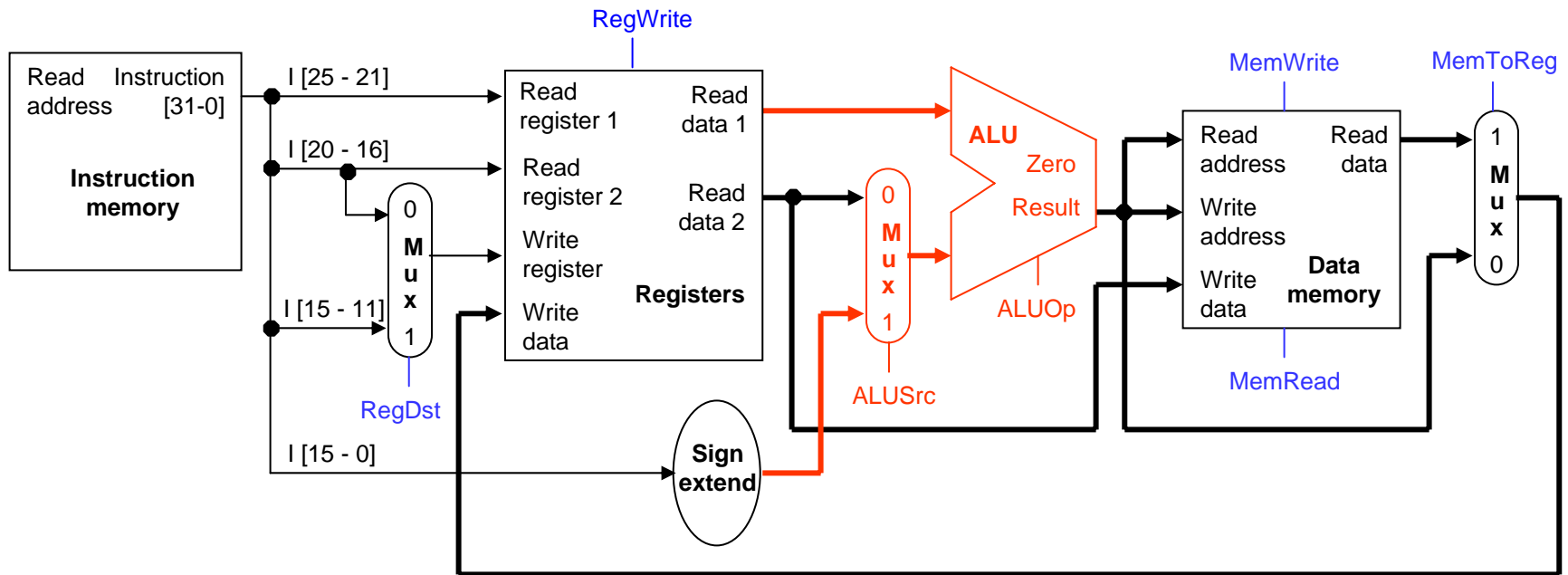
Instruction Decode (ID)

- ❑ The Instruction Decode (ID) step reads the source register from the register file.



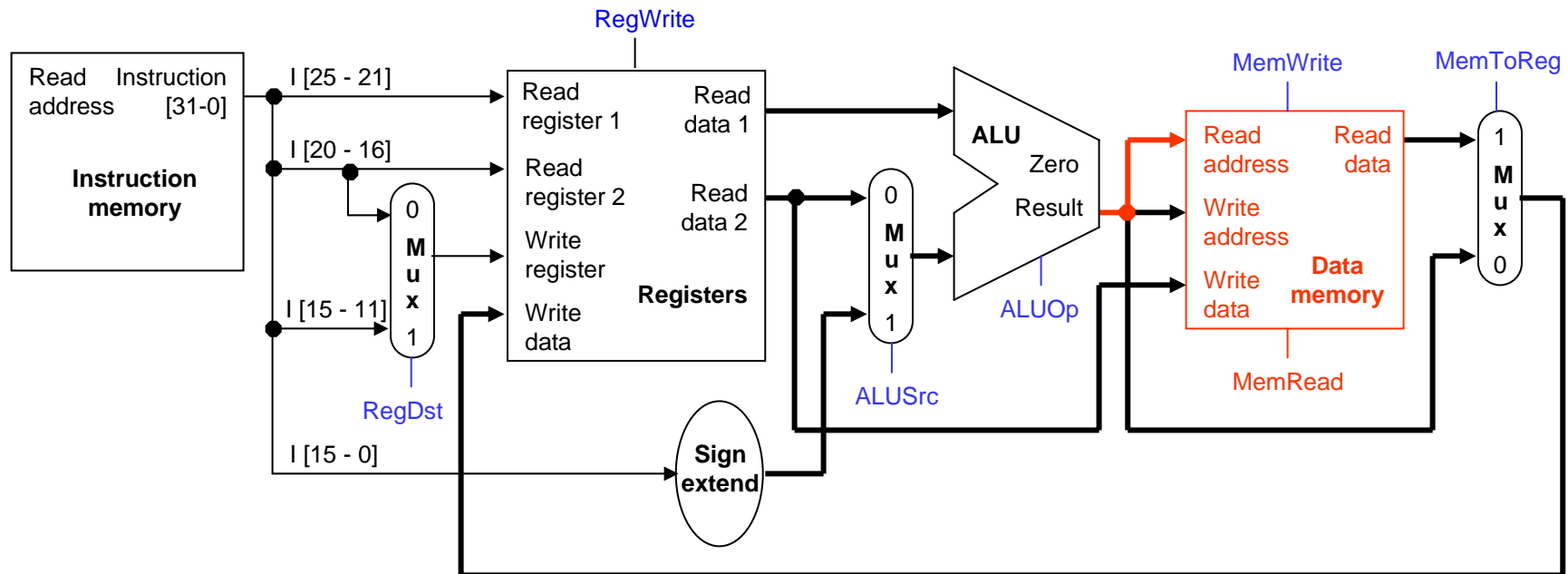
Execute (EX)

- ❑ The third step, Execute (EX), computes the effective memory address from the source register and the instruction's constant field.



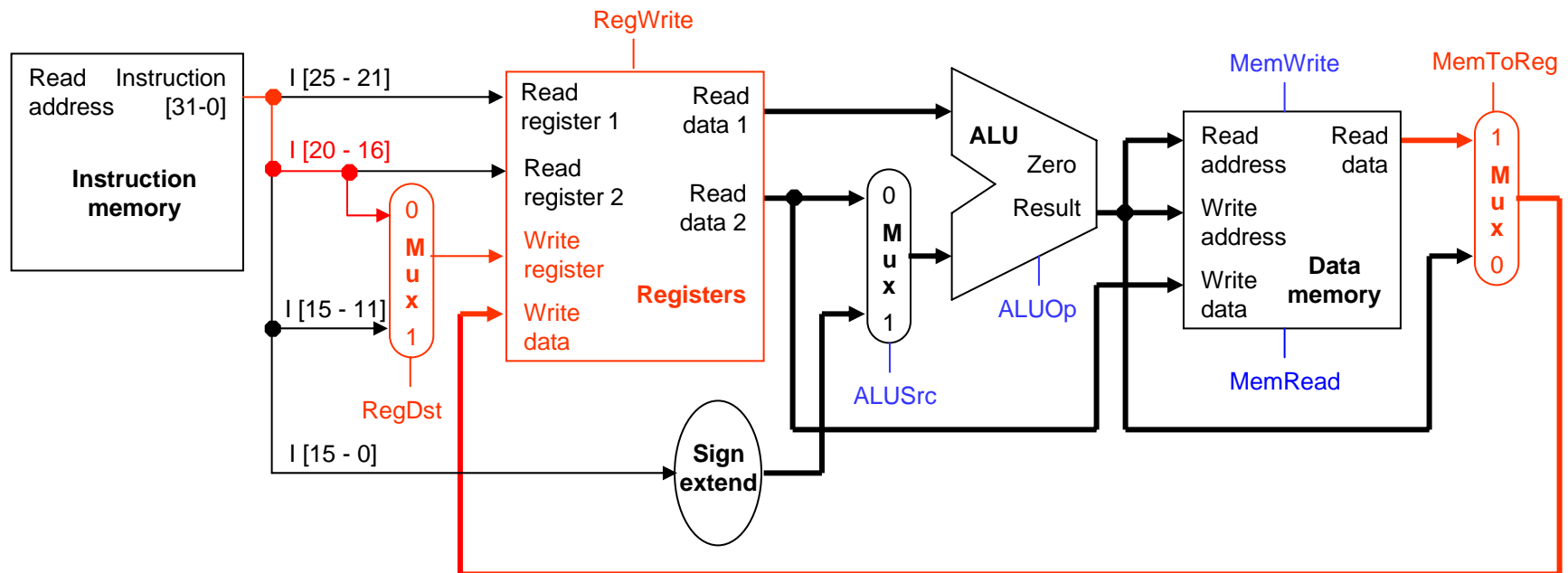
Memory (MEM)

- ❑ The Memory (MEM) step involves reading the data memory, from the address computed by the ALU.



Writeback (WB)

- Finally, in the Writeback (WB) step, the memory value is stored into the destination register.



A bunch of lazy functional units

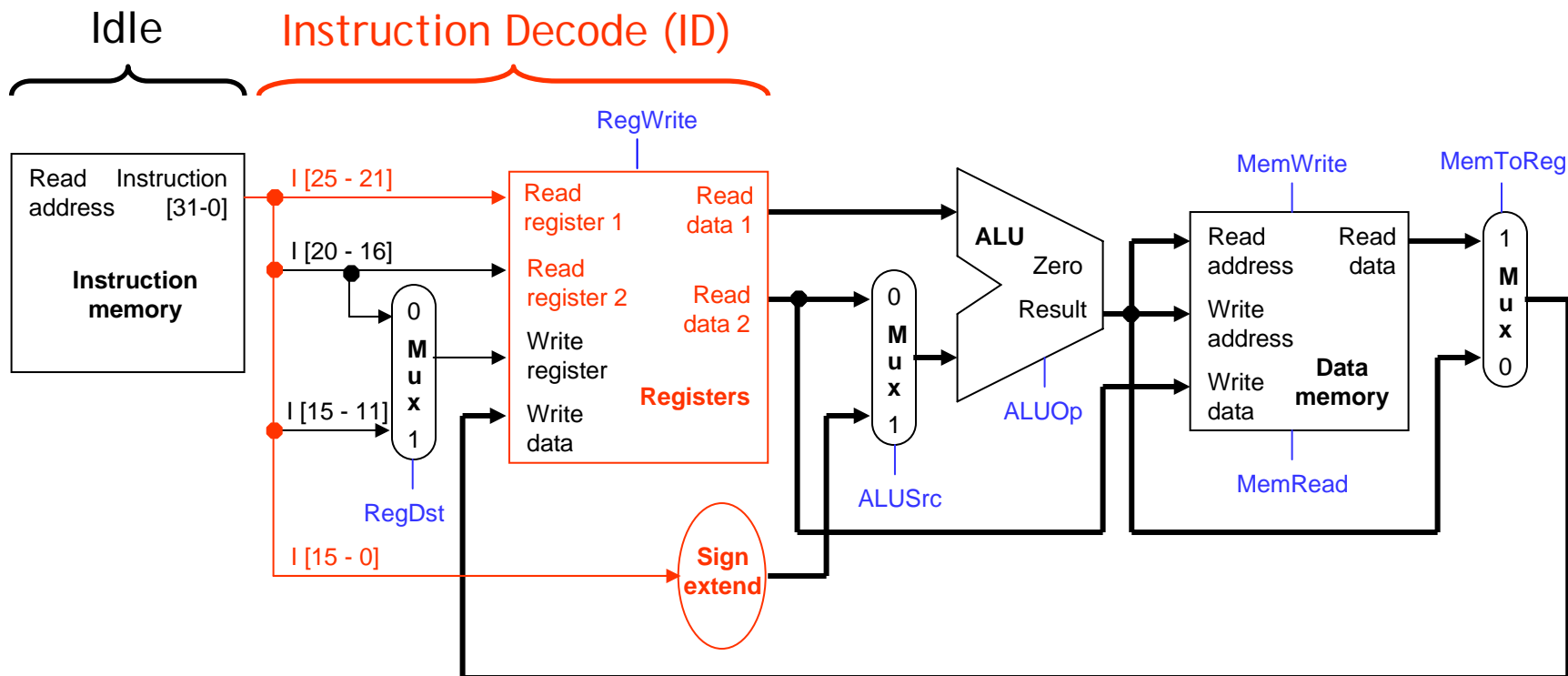
- ❑ Notice that each execution step uses a different functional unit.

- ❑ In other words, the main units are idle for most of the 8ns cycle!
 - The instruction RAM is used for just 2ns at the start of the cycle.
 - Registers are read once in ID (1ns), and written once in WB (1ns).
 - The ALU is used for 2ns near the middle of the cycle.
 - Reading the data memory only takes 2ns as well.

- ❑ That's a lot of hardware sitting around doing nothing.

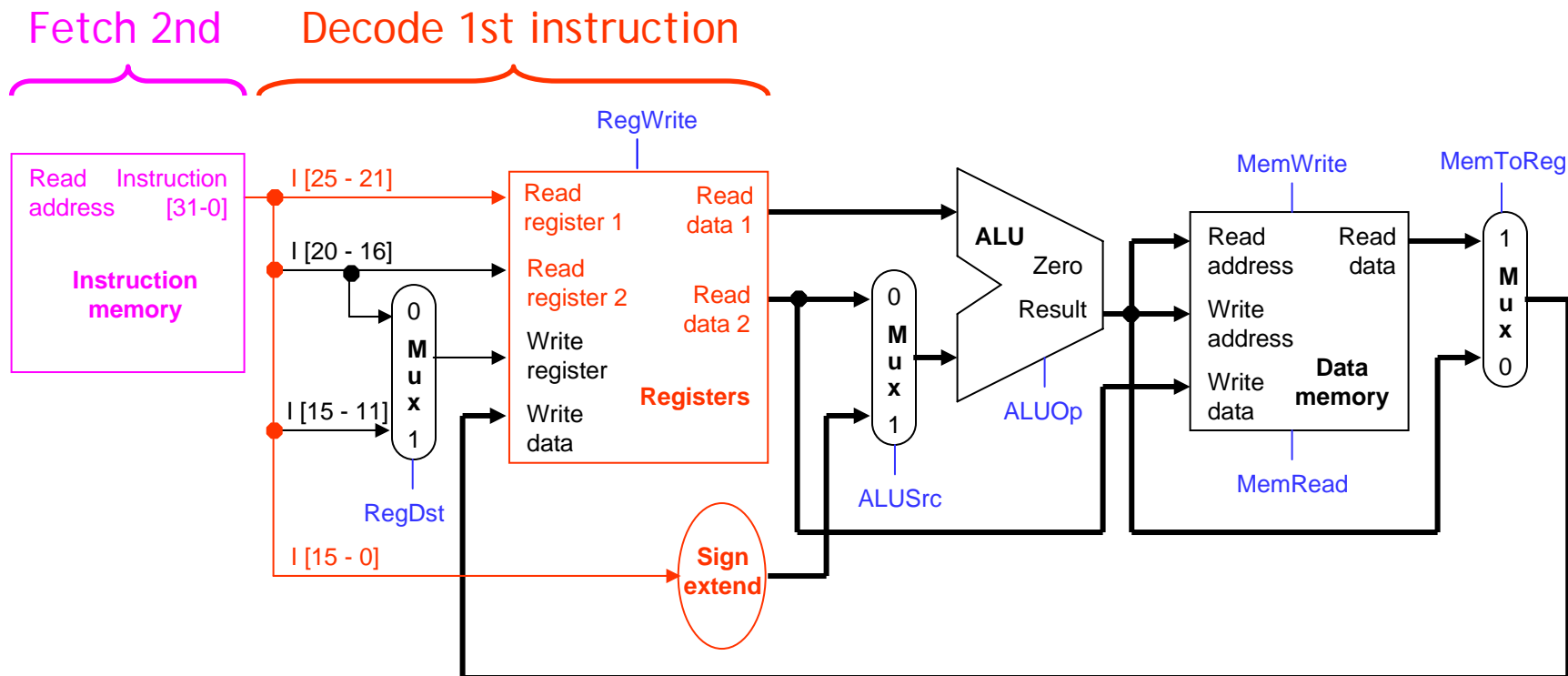
Putting those slackers to work

- ❑ We shouldn't have to wait for the entire instruction to complete before we can re-use the functional units.
- ❑ For example, the instruction memory is free in the Instruction Decode step as shown below, so...



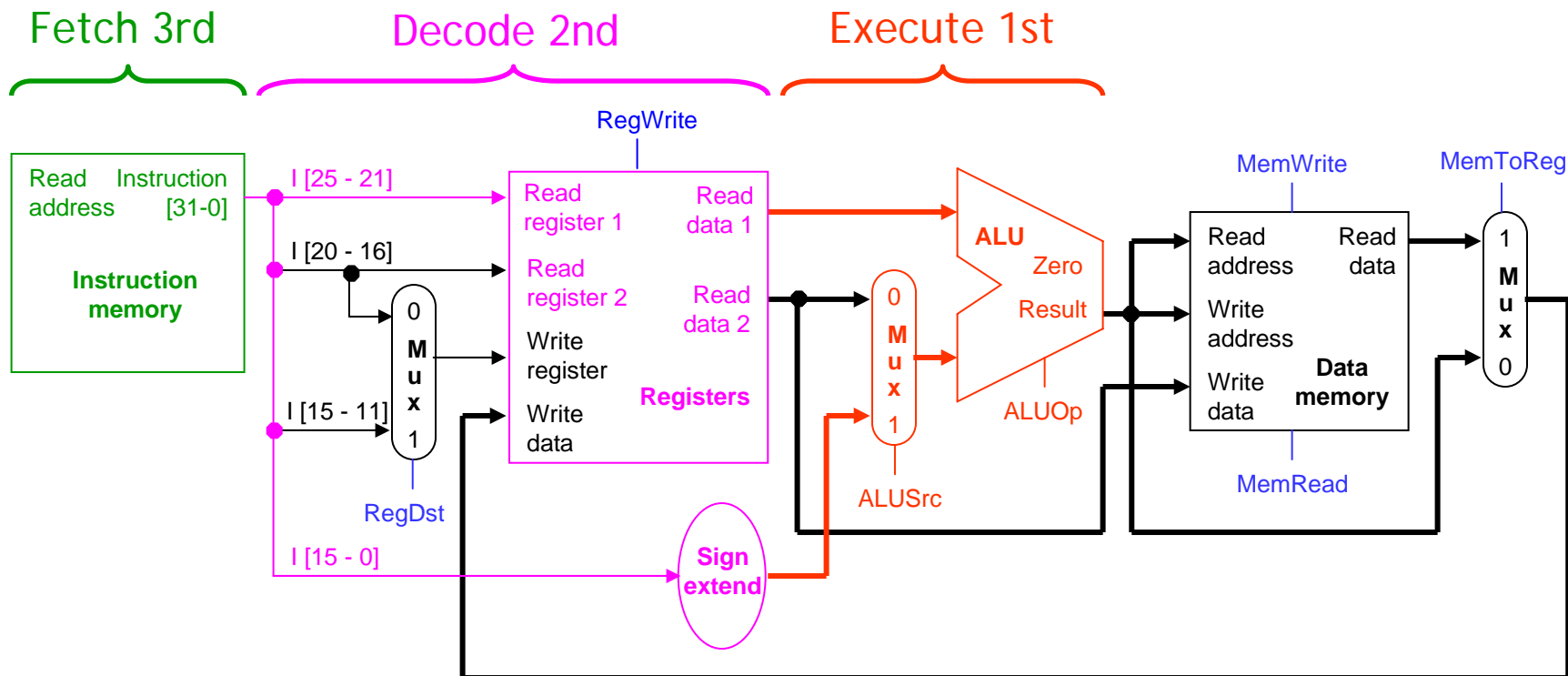
Decoding and fetching together

- Why don't we go ahead and fetch the *next* instruction while we're decoding the first one?



Executing, decoding and fetching

- ❑ Similarly, once the first instruction enters its Execute stage, we can go ahead and decode the second instruction.
- ❑ But now the instruction memory is free again, so we can fetch the third instruction!



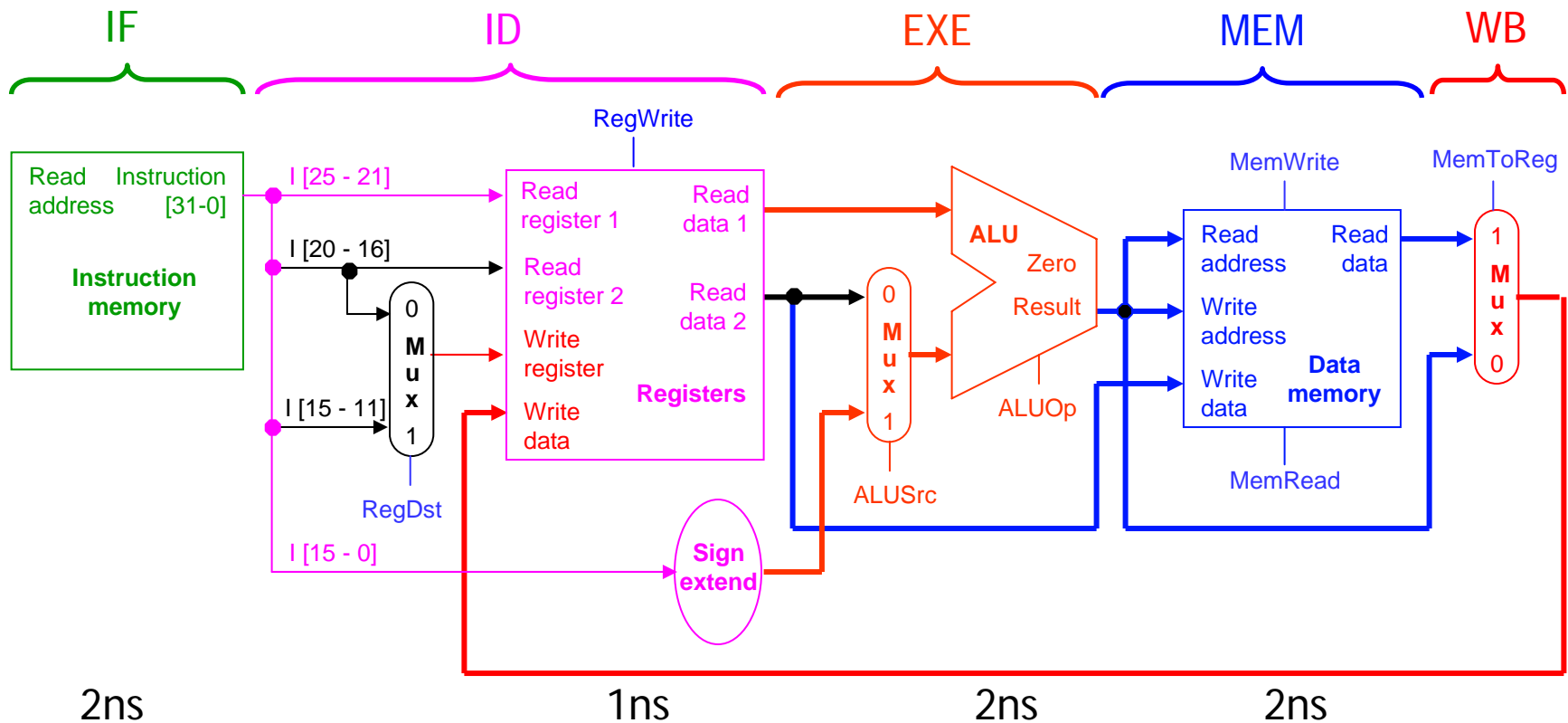
Making Pipelining Work

- ❑ We'll make our pipeline 5 stages long, to handle load instructions as they were handled in the multi-cycle implementation
 - Stages are: IF, ID, EX, MEM, and WB
- ❑ We want to support executing 5 instructions simultaneously: one in each stage.

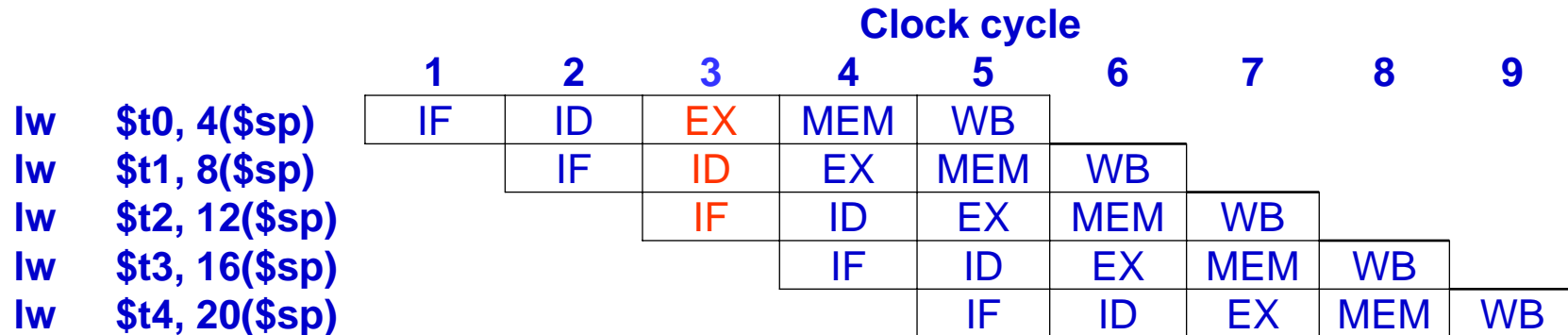


Break datapath into 5 stages

- ❑ Each stage has its own functional units.
- ❑ Each stage can execute in 2ns
 - Just like the multi-cycle implementation



Pipelining Loads



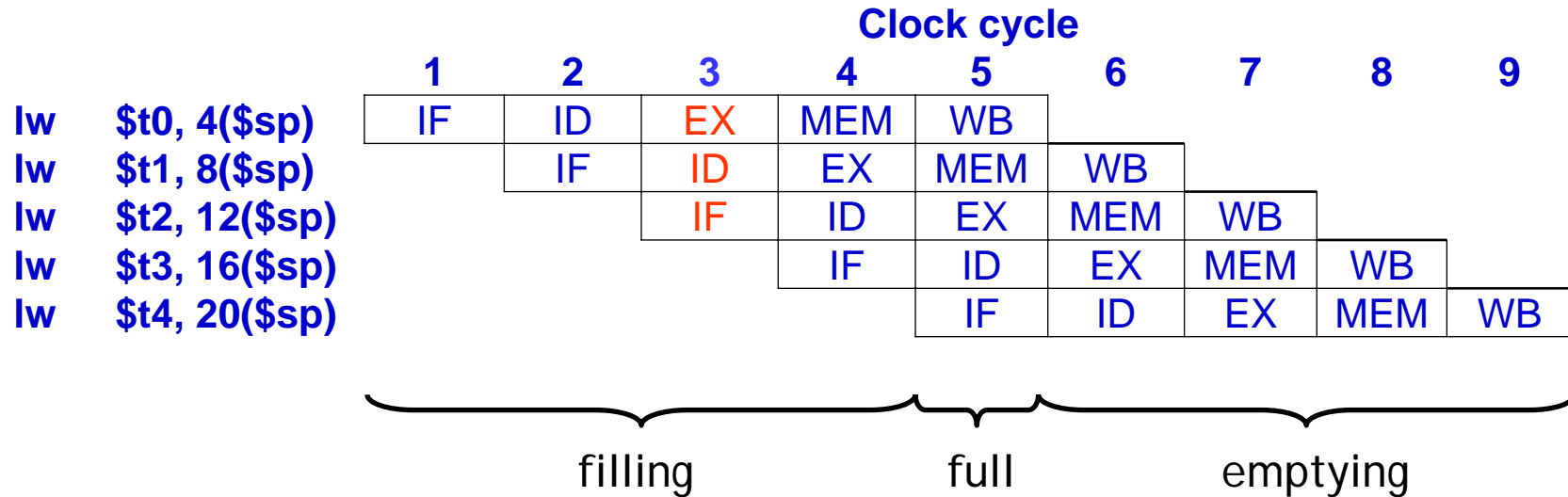
□ **A pipeline diagram shows the execution of a series of instructions.**

- The instruction sequence is shown vertically, from top to bottom.
- Clock cycles are shown horizontally, from left to right.
- Each instruction is divided into its component stages. (We show five stages for every instruction, which will make the control unit easier.)

□ **This clearly indicates the overlapping of instructions. For example, there are three instructions active in the third cycle above.**

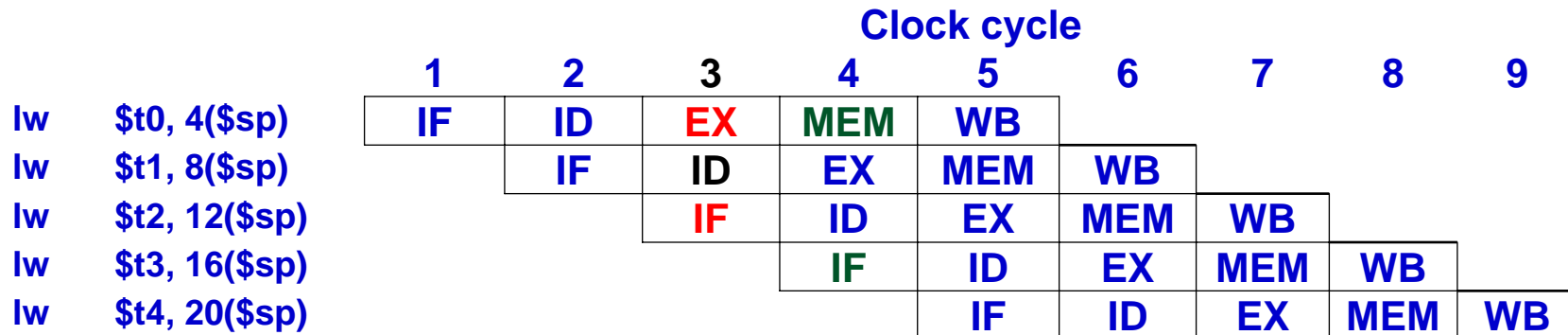
- The “lw \$t0” instruction is in its Execute stage.
- Simultaneously, the “lw \$t1” is in its Instruction Decode stage.
- Also, the “lw \$t2” instruction is just being fetched.

Pipelining terminology



- ❑ The **pipeline depth** is the number of stages—in this case, five.
- ❑ In the first four cycles here, the pipeline is **filling**, since there are unused functional units.
- ❑ In cycle 5, the pipeline is **full**. Five instructions are being executed simultaneously, so all hardware units are in use.
- ❑ In cycles 6-9, the pipeline is **emptying**.

Pipeline Datapath: Resource Requirements



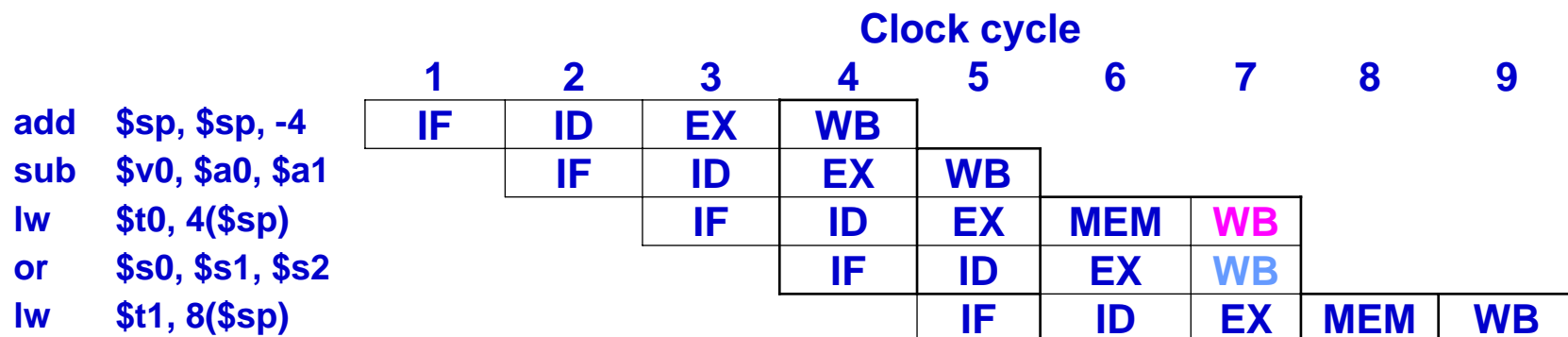
- ❑ We need to perform several operations in the same cycle.
 - Increment the PC and add registers at the same time.
 - Fetch one instruction while another one reads or writes data.

- ❑ What does that mean for our hardware?

Pipelining other instruction types

- ❑ R-type instructions only require 4 stages: IF, ID, EX, and WB
 - We don't need the MEM stage

- ❑ What happens if we try to pipeline loads with R-type instructions?



- Load uses Register File's Write Port during its 5th stage
- R-type uses Register File's Write Port during its 4th stage

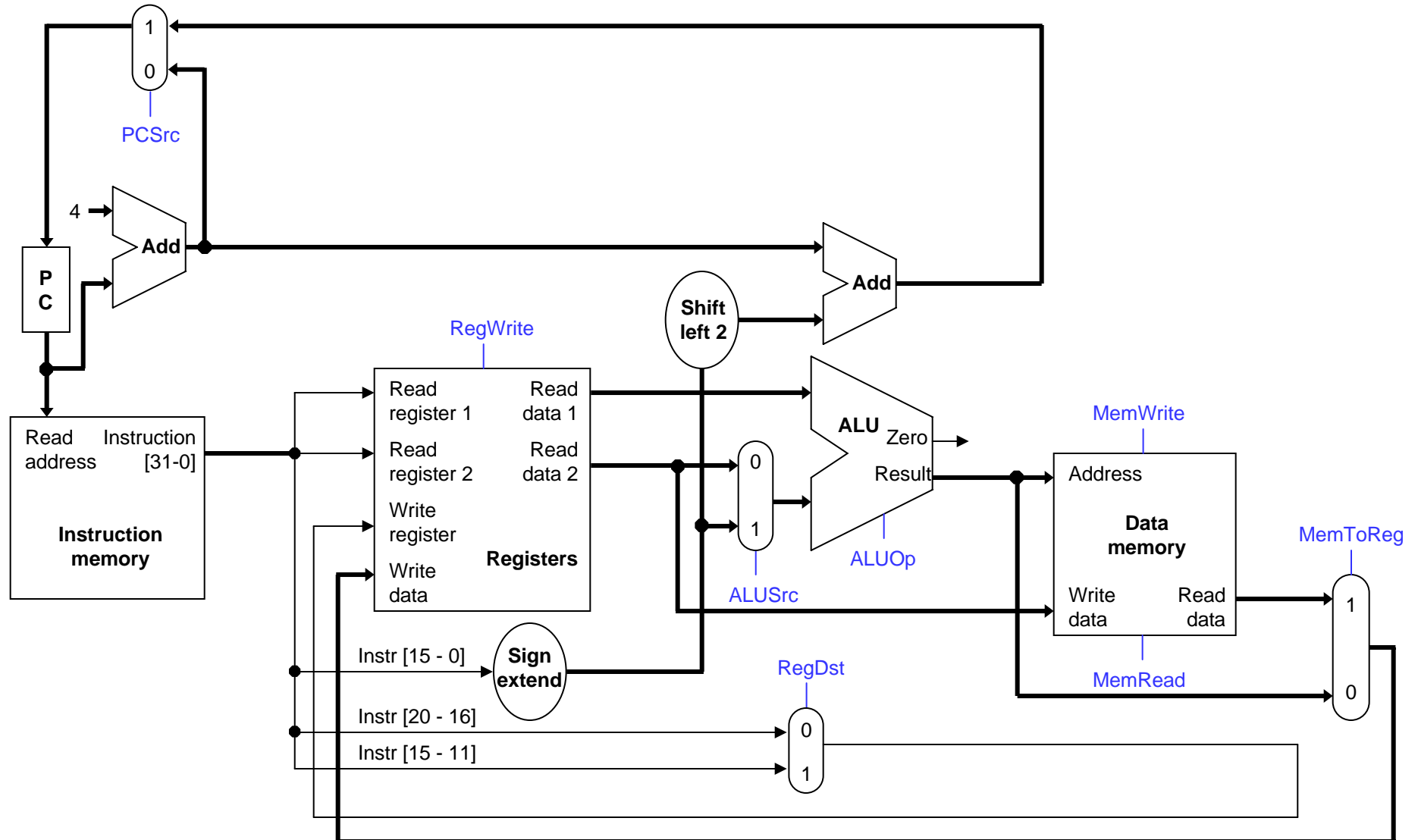
One register file is enough

- ❑ We need only one register file to support both the **ID** and **WB** stages.



- ❑ Reads and writes go to **separate ports** on the register file.
- ❑ We already took advantage of this property in our single-cycle CPU.

Single-cycle datapath, slightly rearranged



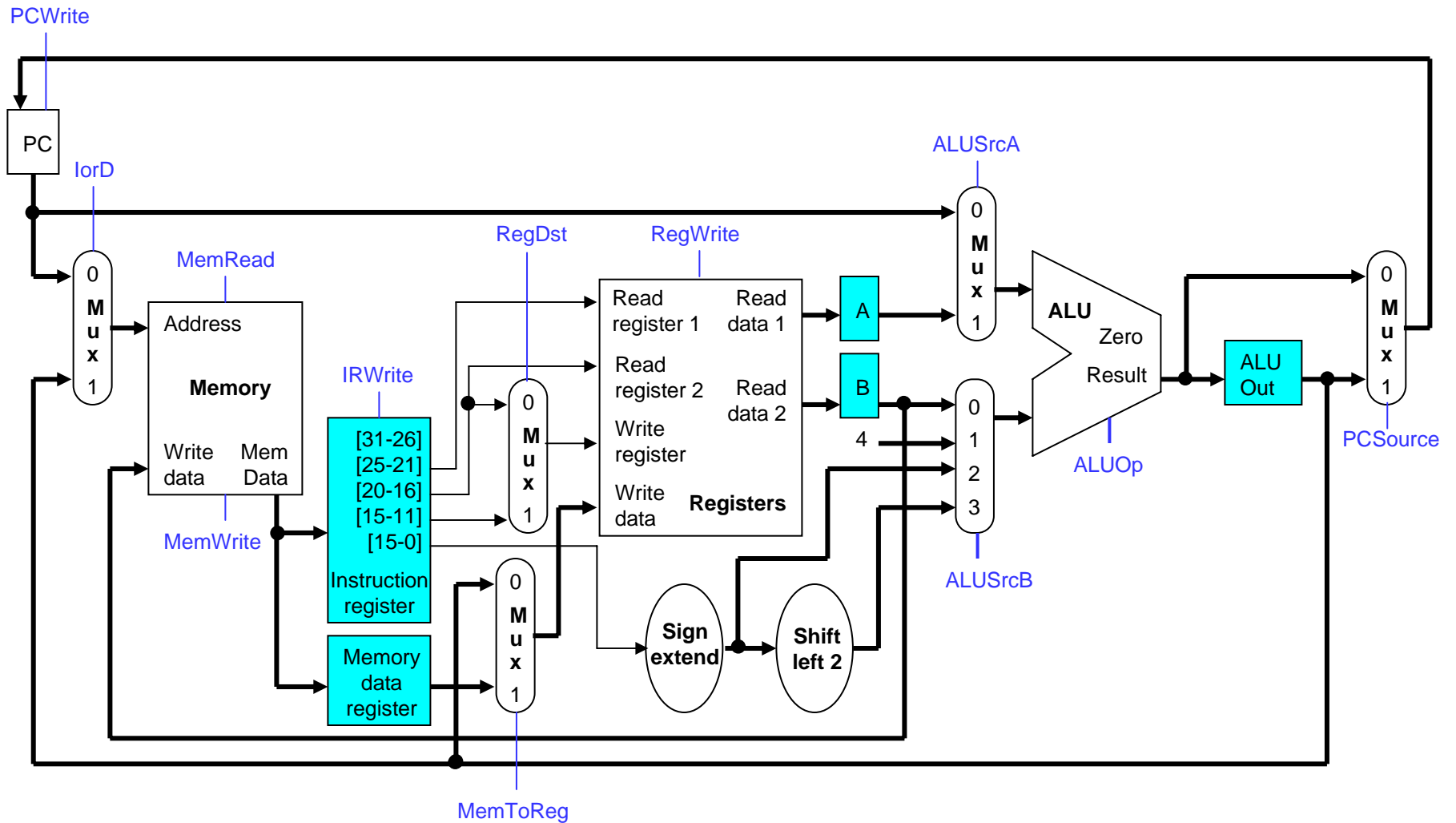
Multiple cycles

- ❑ **In pipelining, we also divide instruction execution into multiple cycles.**

- ❑ **Information computed during one cycle may be needed in a later cycle.**
 - The instruction read in the IF stage determines which registers are fetched in the ID stage, what constant is used for the EX stage, and what the destination register is for WB.
 - The registers read in ID are used in the EX and/or MEM stages.
 - The ALU output produced in the EX stage is an effective address for the MEM stage or a result for the WB stage.

- ❑ **We added several intermediate registers to the multicycle datapath to preserve information between stages, as highlighted on the next slide.**

Registers added to the multi-cycle



Pipeline registers

- ❑ We'll add intermediate registers to our pipelined datapath too.
- ❑ There's a lot of information to save, however. We'll simplify our diagrams by drawing just one big **pipeline register** between each stage.
- ❑ The registers are named for the stages they connect.

IF/ID

ID/EX

EX/MEM

MEM/WB

- ❑ No register is needed after the WB stage, because after WB the instruction is done.

Pipelined datapath

