Pipelining

- We've seen two possible implementations of the MIPS architecture.
  - A single-cycle datapath executes each instruction in just one clock cycle, but the cycle time may be very long.
  - A multicycle datapath has much shorter cycle times and higher clock rates, but each instruction requires many cycles to execute.
- A third approach, pipelining, yields the best of both worlds and is used in every modern desktop processor.
  - Cycle times are short so clock rates are high.
  - But we can still execute an instruction in about one clock cycle!
- Today we introduce pipelining and its benefits, and on Wednesday we'll show a pipelined datapath and control unit.
- After spring break we'll talk about what makes pipelining difficult in real life and what to do about it.

Instruction execution review

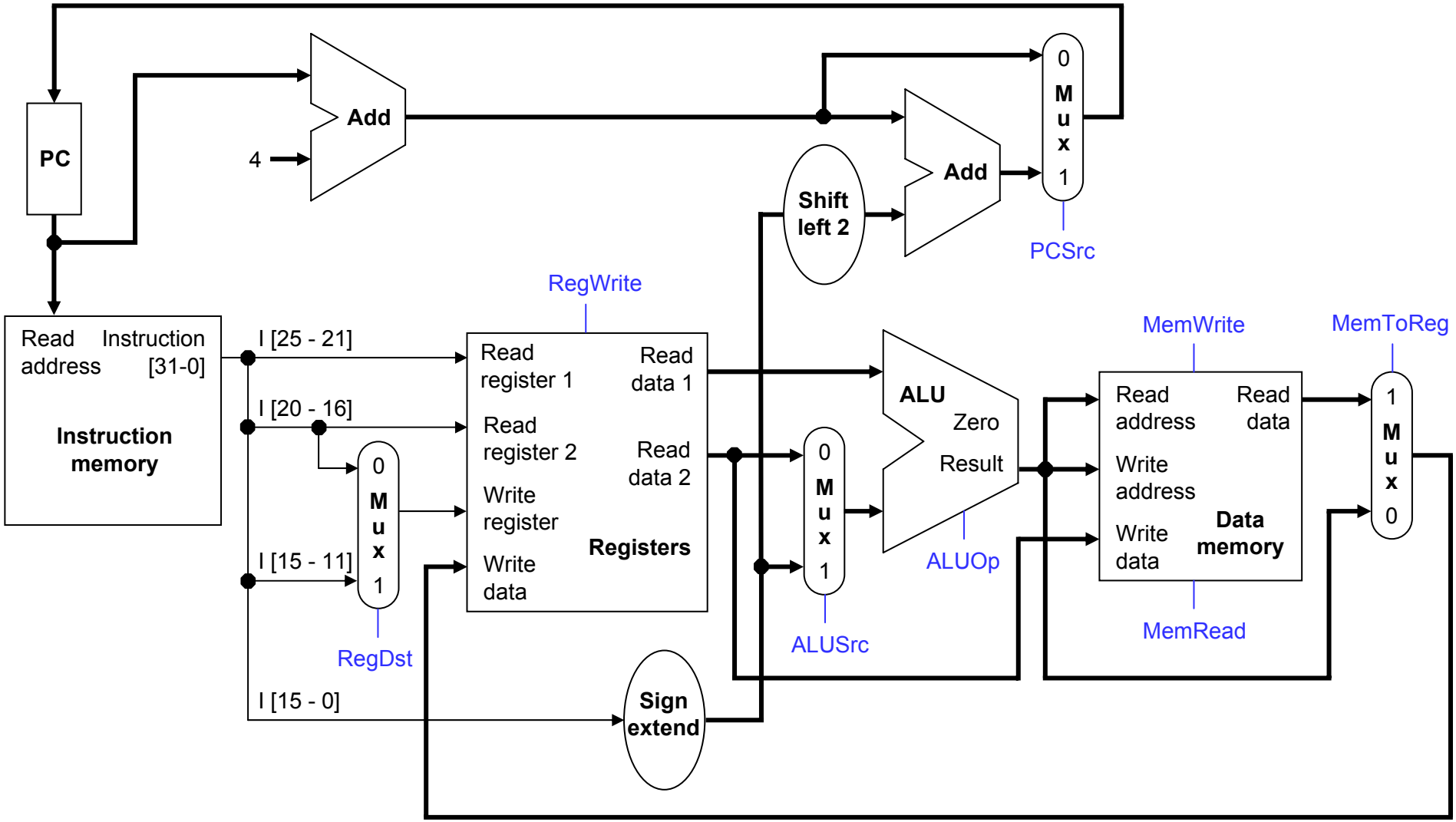
- Executing a MIPS instruction can take up to five steps.

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 target.
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

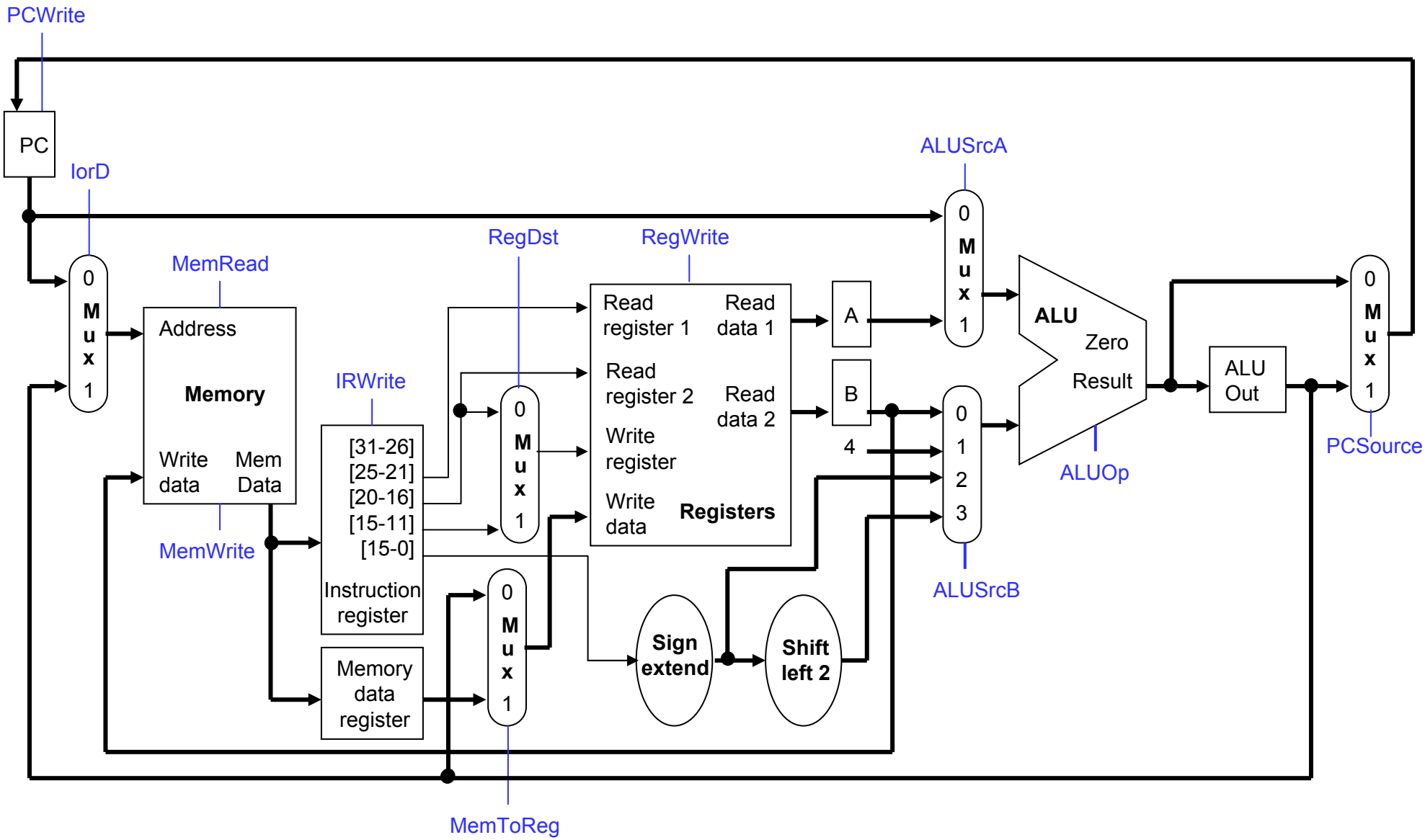
Single-cycle datapath diagram



Single-cycle review

- All five execution steps occur in one clock cycle.
- This means the cycle time must be long enough to accommodate all the steps of the most complex instruction—a "lw" in our instruction set.
  - If the register file has a 1ns latency and the memories and ALU have a 2ns latency, "lw" will require 8ns.
  - Thus *all* instructions will take 8ns to execute.
- 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.

Multicycle datapath diagram



Multicycle review

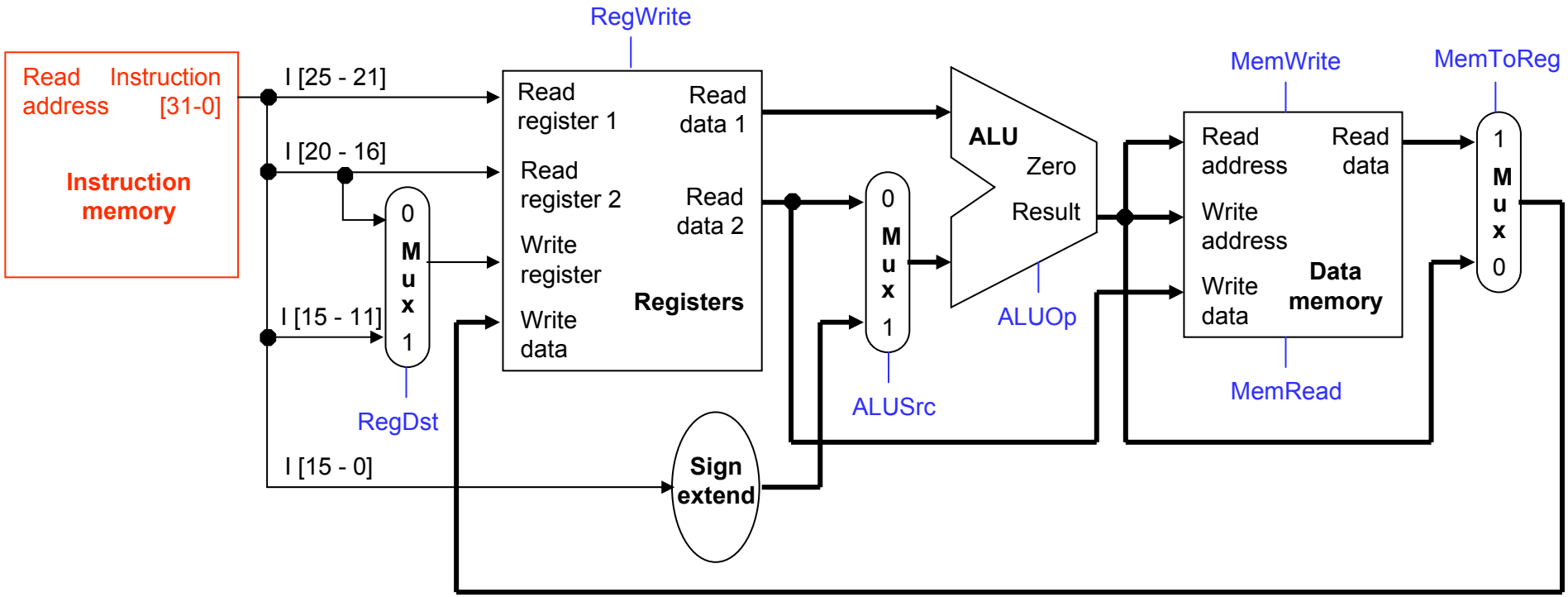
- Instruction execution is split into five stages, each taking one clock cycle.
- The cycle time is shorter, since each stage is relatively simple. With a 1ns delay for registers and 2ns for the memory and ALU, the cycle time would be 2ns.
- Only necessary stages are executed, so more complex instructions will not slow down simpler ones.

Instruction	Steps required					Cycles
beq	IF	ID	EX			3
R-type	IF	ID	EX		WB	4
sw	IF	ID	EX	MEM		4
lw	IF	ID	EX	MEM	WB	5

- The actual CPI will depend on the particular instruction mix.
- We can get by with only one memory and one ALU, because they can be reused in different clock cycles of a single instruction execution.

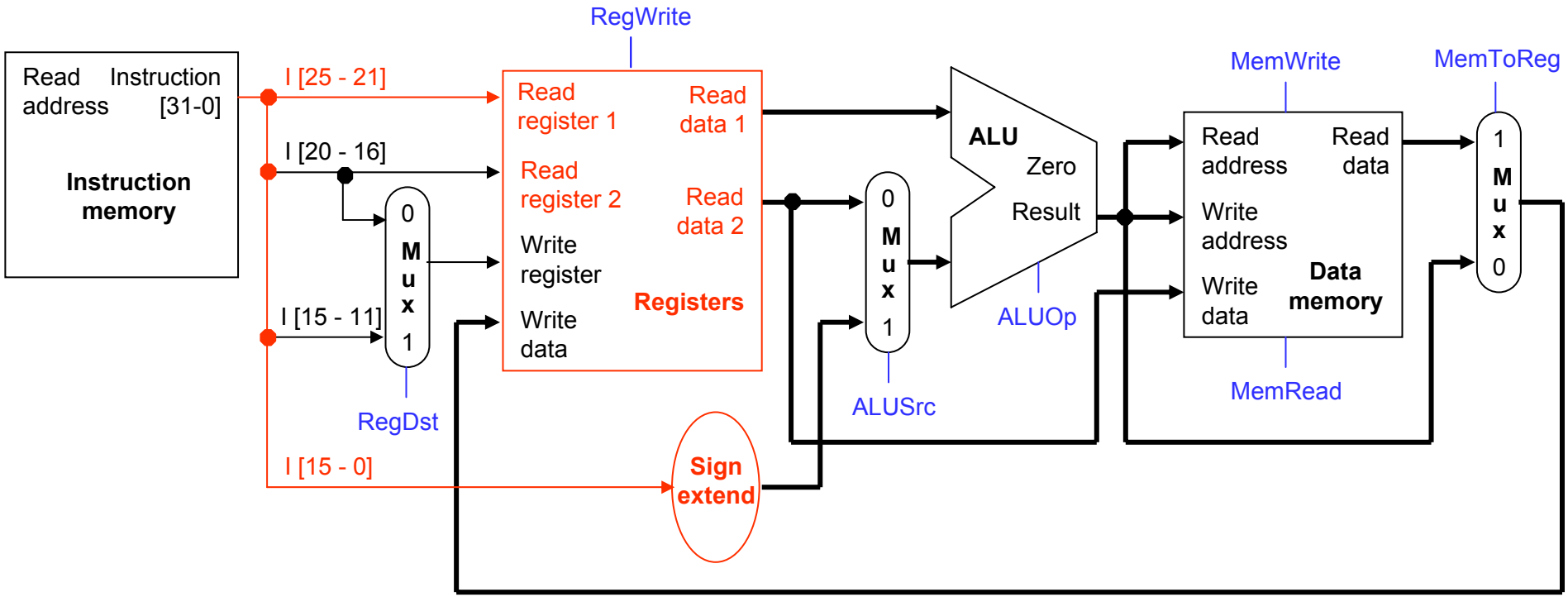
Example: Instruction Fetch (IF)

- 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.



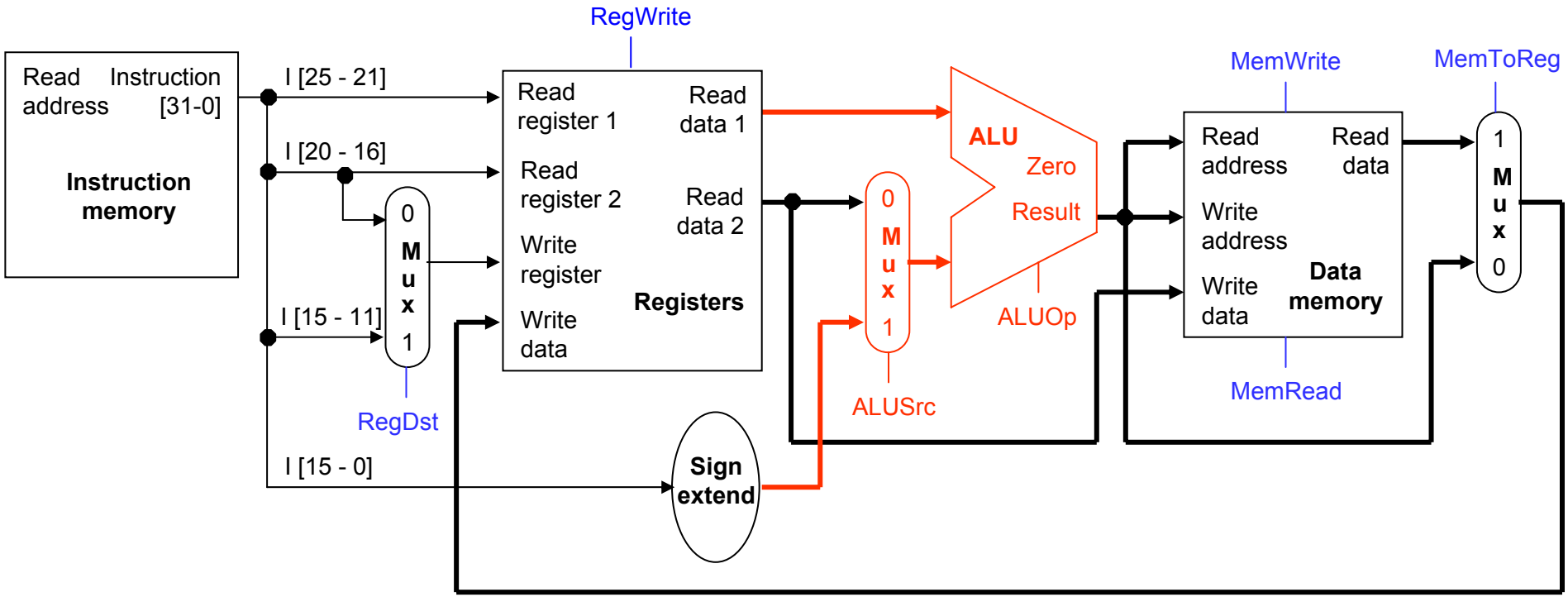
Instruction Decode (ID)

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



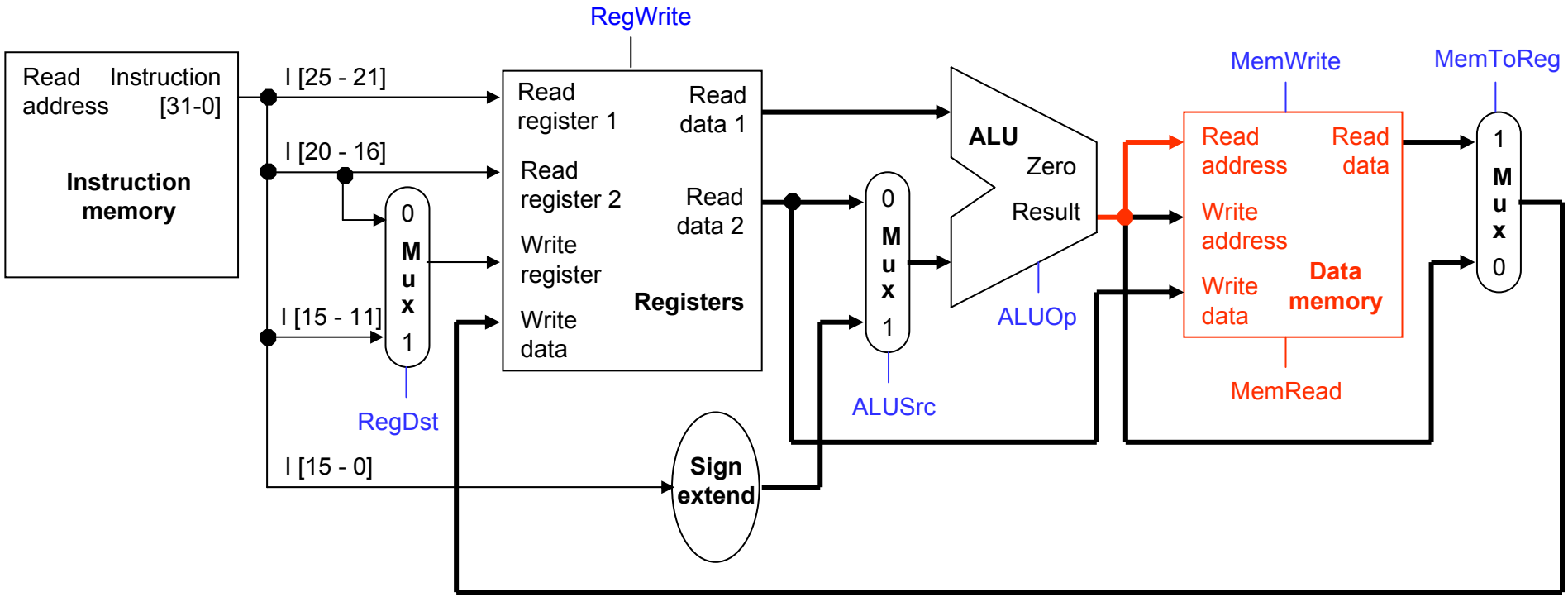
Execute (EX)

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



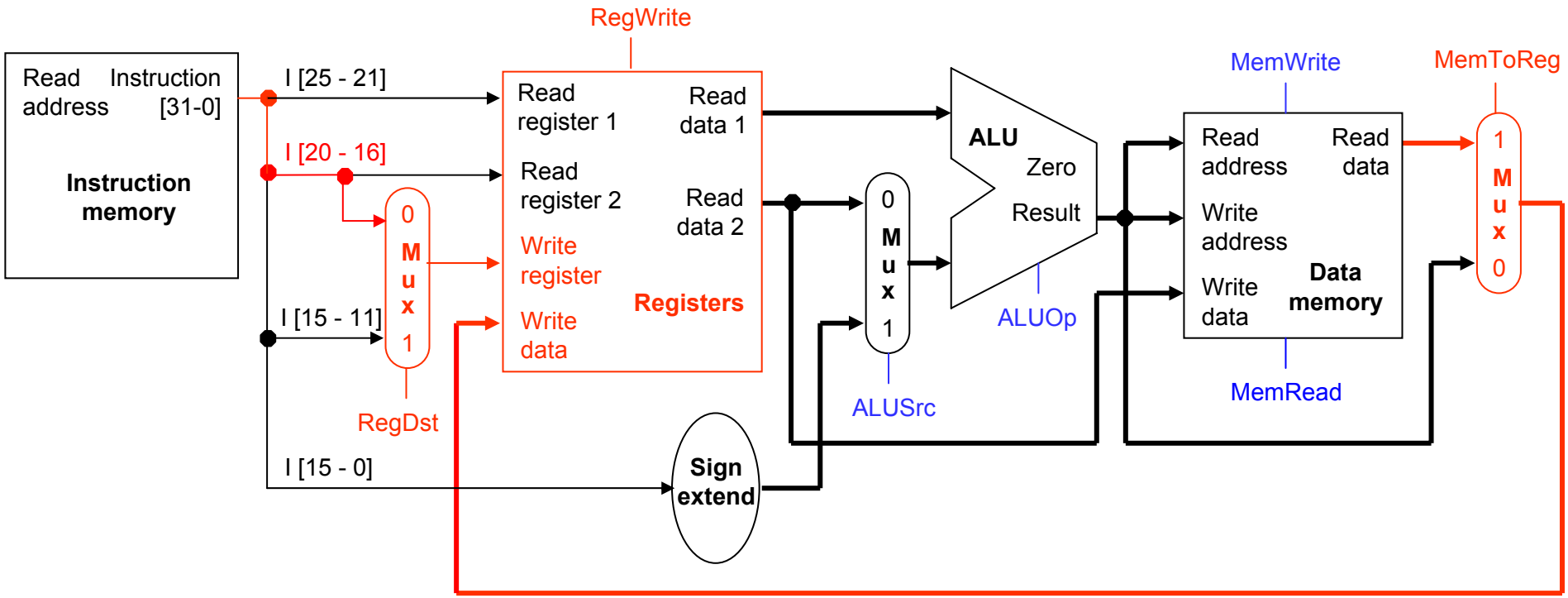
Memory (MEM)

- 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.

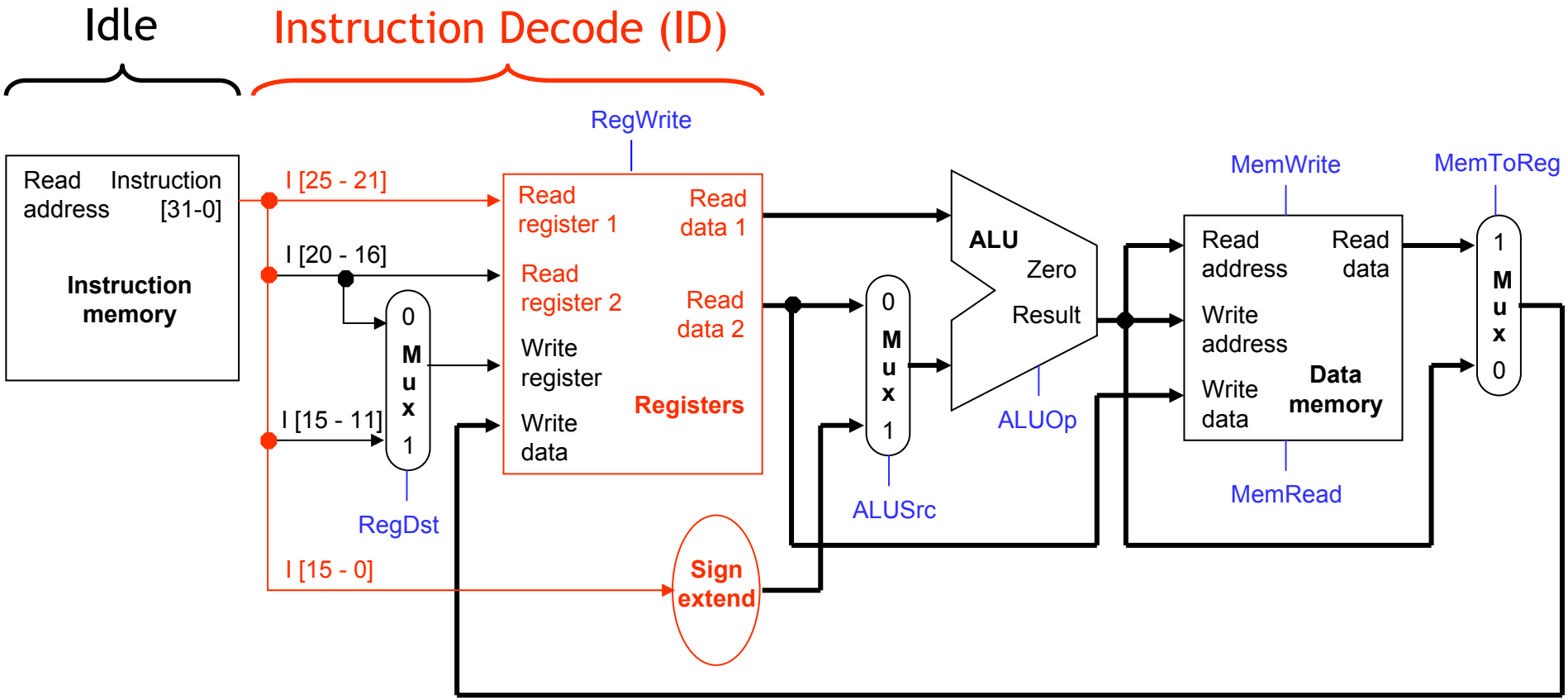


A bunch of lazy functional units

- 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 expensive hardware sitting around doing nothing.

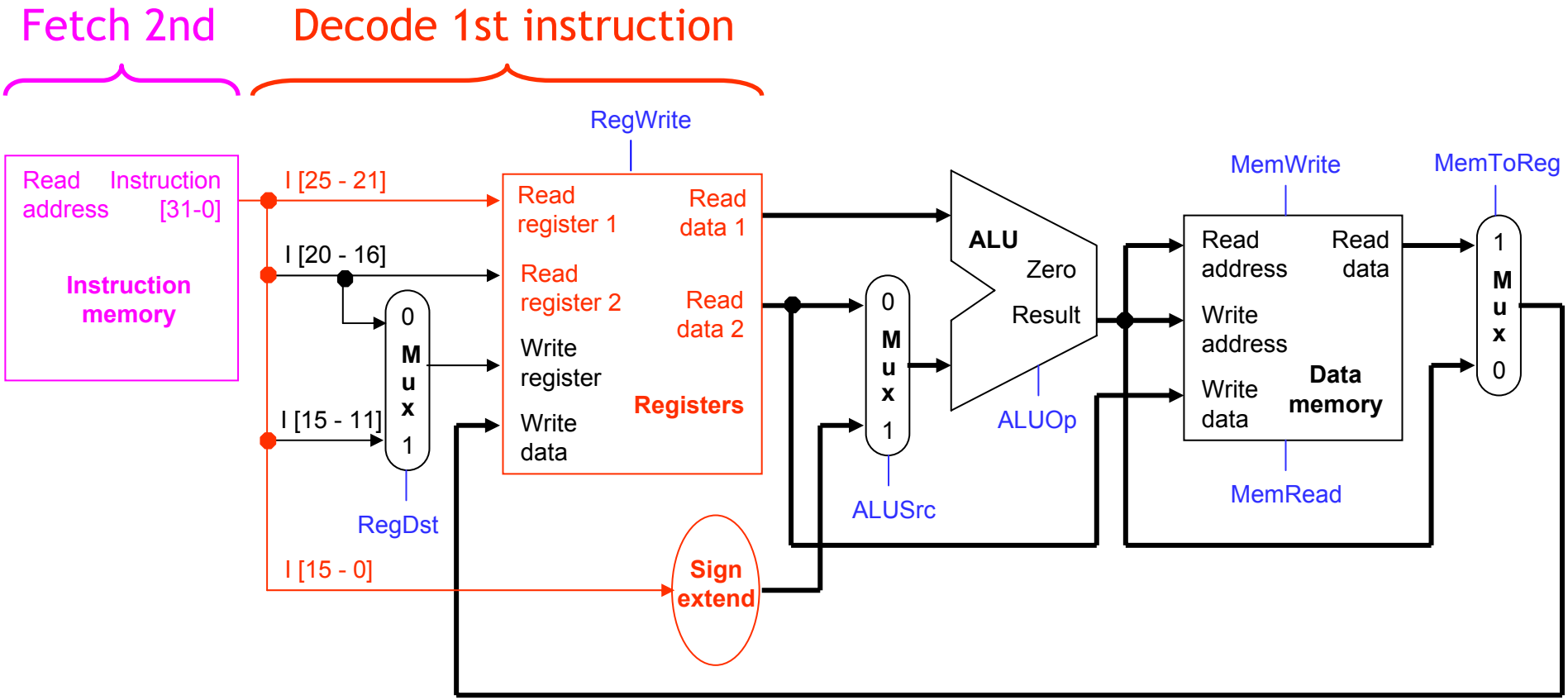
Putting those slackers to work

- 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...



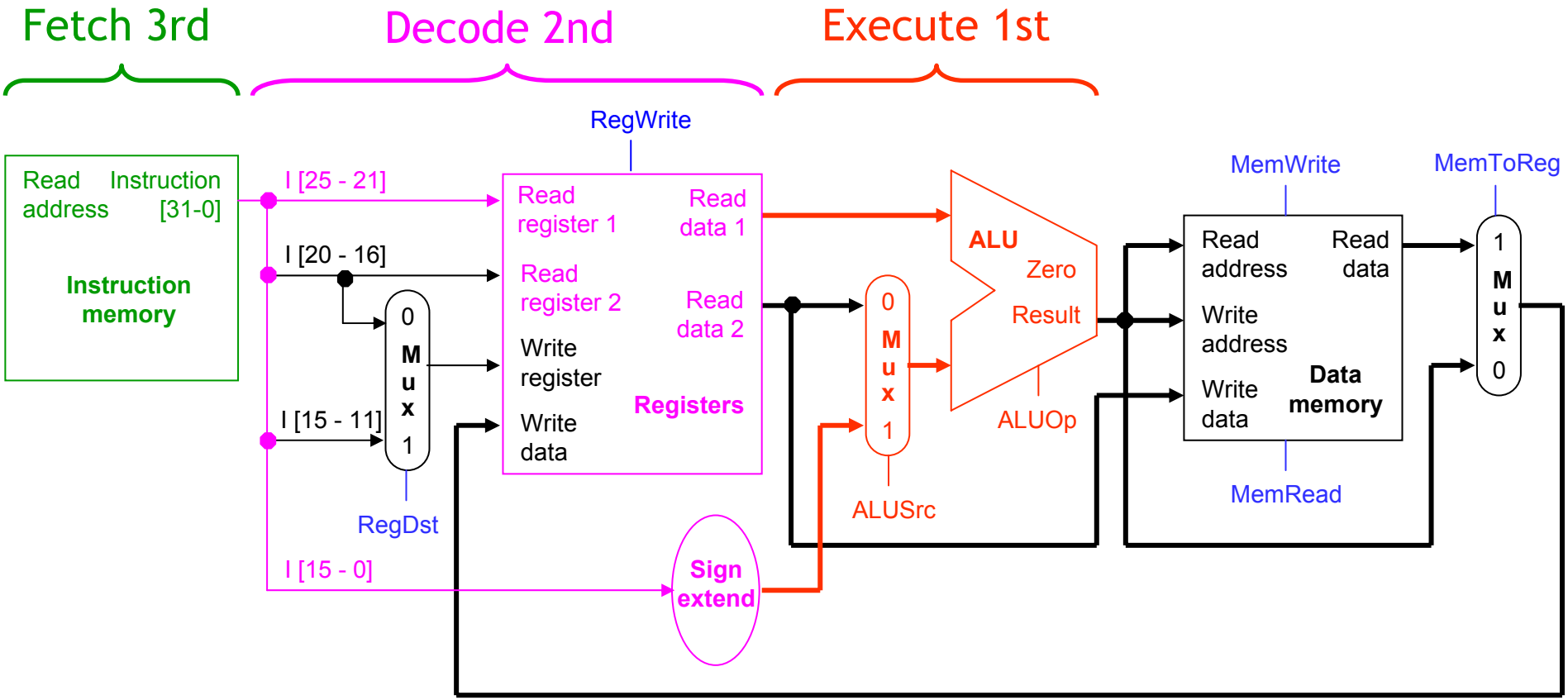
Decoding and fetching together

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



Executing, decoding and fetching

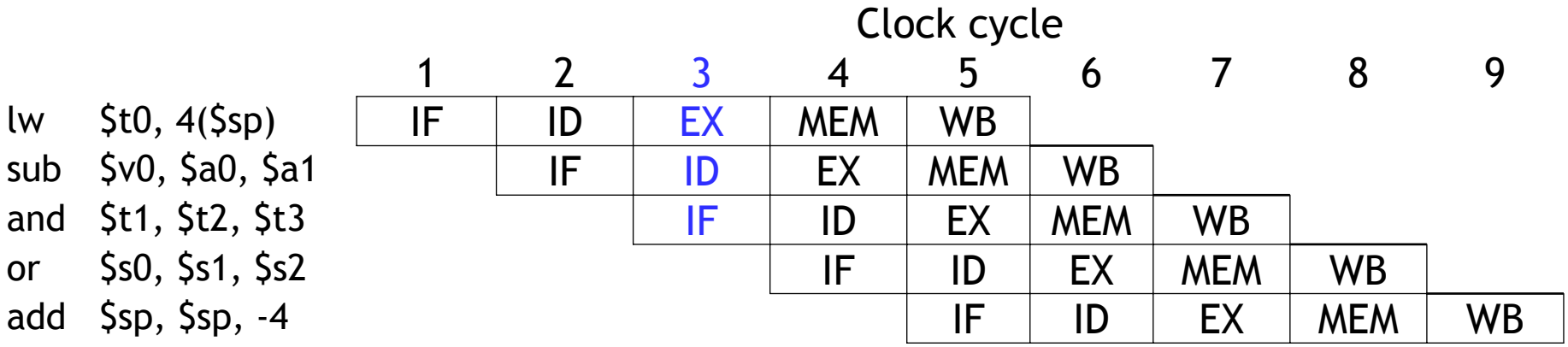
- Similarly, once the first instruction enters its Execute stage, we can go ahead and decode the second instruction.
- But now the instruction memory is free again, so we can fetch the third instruction!



Working hard

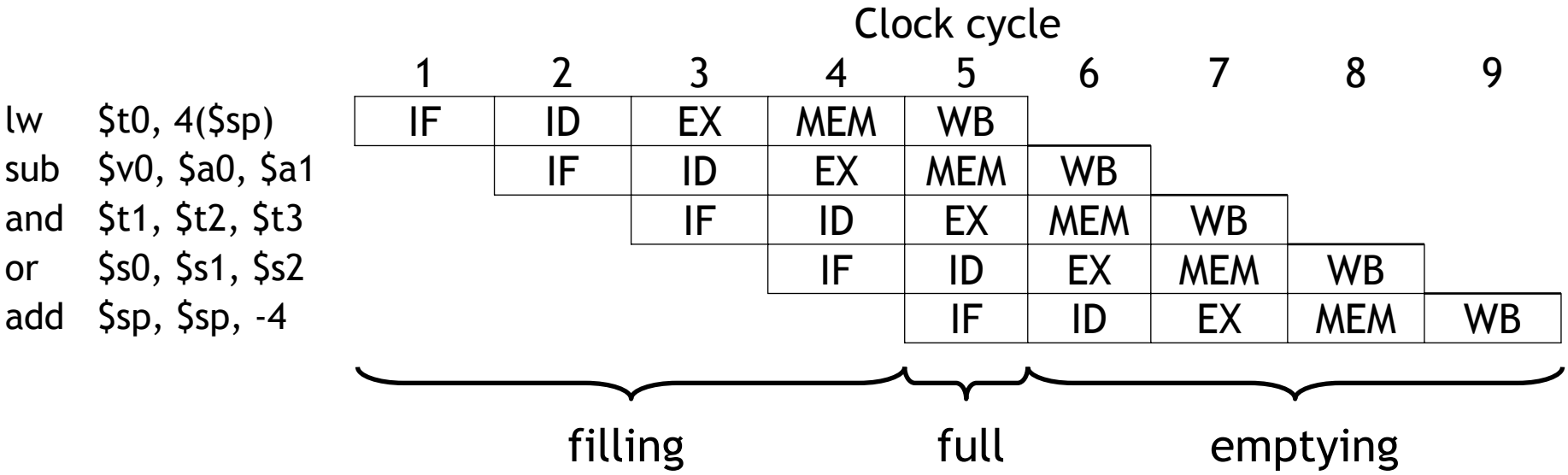
- The idea behind pipelining is to maximize the usage of the hardware by overlapping the execution of several instructions.
- Each of our five execution steps uses a different functional unit.
- Conceivably we can have up to five instructions executing at the same time—one in each of the IF, ID, EX, MEM and WB stages.
- To make this work, we will go back to the single-cycle datapath with its multiple adders and memories, so many instructions can execute together without interference.

A pipeline diagram



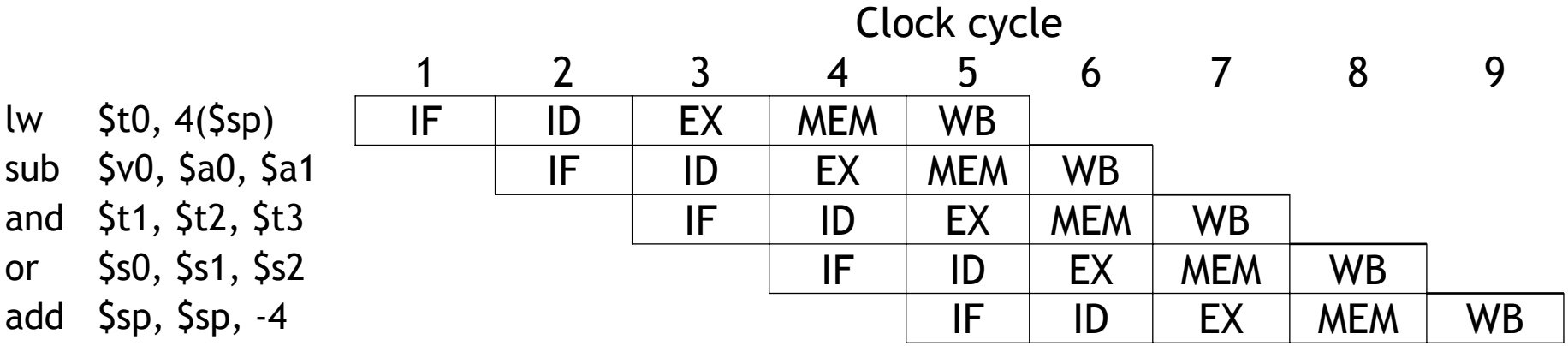
- A pipeline diagram shows the execution of a series of instructions.
  - The instruction sequence is shown vertically, from top to bottom.
  - Clock cycles are shown horizontally, from left to right.
  - Each instruction is divided into its component stages. (We show five stages for every instruction, 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" instruction is in its Execute stage.
  - Simultaneously, the "sub" is in its Instruction Decode stage.
  - Also, the "and" instruction is just being fetched.

Pipeline terminology



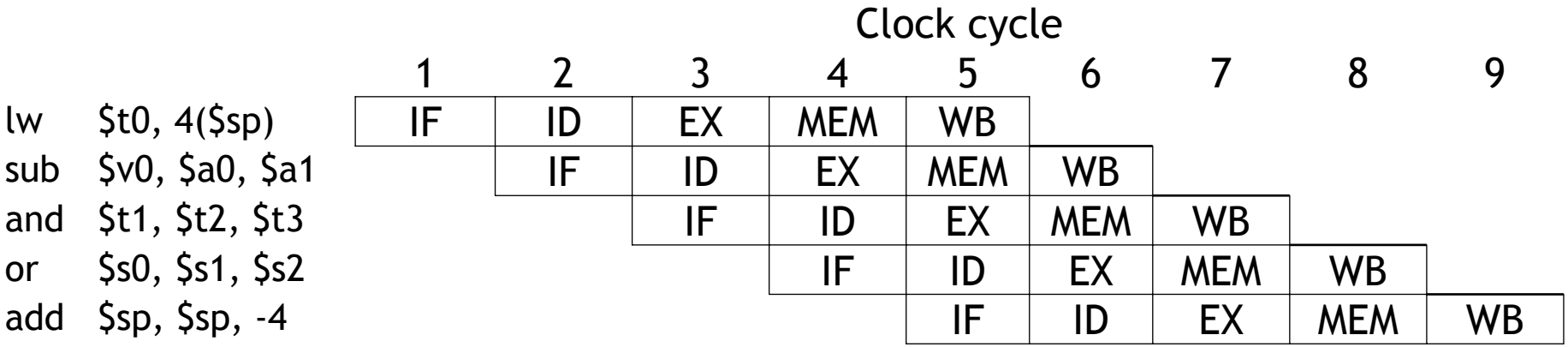
- The pipeline depth is the number of stages—in this case, five.
- In the first four cycles here, the pipeline is filling, since there are unused functional units.
- In cycle 5, the pipeline is full. Five instructions are being executed simultaneously, so all hardware units are in use.
- In cycles 6-9,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

Performance benefits of pipelining



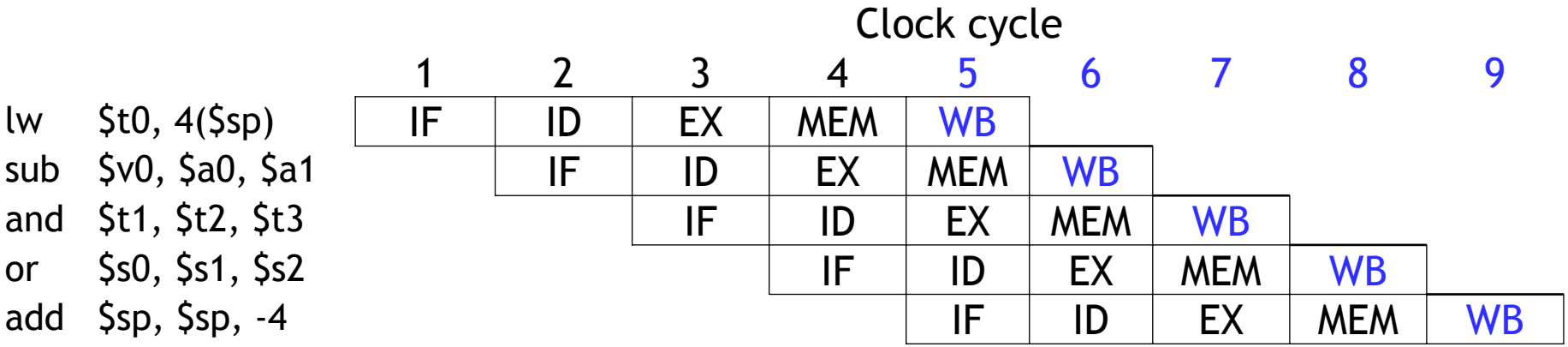
- How much time would it take to execute these five instructions?
  - We would need 40ns with a single-cycle CPU with an 8ns cycle time.
  - In a multicycle design, these instructions require 42ns.
  - With pipelining, we need just 9 cycles, and 18ns, as shown above!
- What kind of speedups are we talking about?
  - Compared to a single-cycle design, 40ns/18ns = 2.2.
  - Versus the multicycle machine, the speedup is 42ns/18ns = 2.3!

It gets better!



- This instruction sequence is too short.
  - Most of the cycles are spent filling or emptying the pipeline.
  - We only achieve full utilization for one clock cycle.
- Imagine a longer program with 1000 instructions.
  - It takes 4 cycles to fill the pipeline, and 1000 more cycles to complete all the instructions.
  - So it takes 1004 cycles to execute 1000 instructions.
  - That's almost one cycle per instruction, and a speedup of 3.98 over the single-cycle datapath!

Almost one cycle per instruction



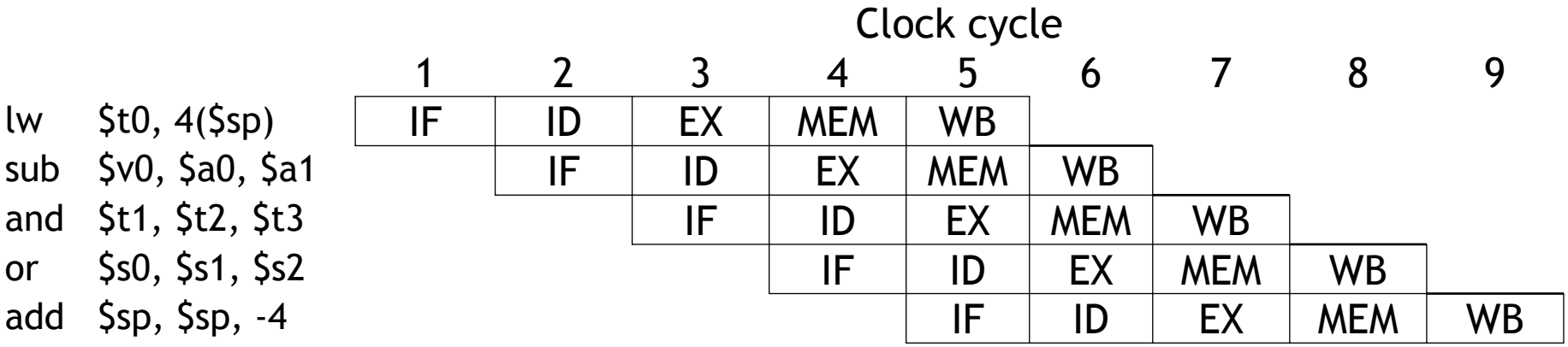
- Here's another way to visualize the benefits of pipelining.
  - In the first four stages of the diagram above, the pipeline is filling.
  - But after that, while the pipeline is full or emptying, you can see that one instruction completes on every cycle.
- So except for some initial overhead, we really are executing at the rate of one cycle per instruction.

It's GRRReeeaaAAT!

Tony the Tiger is trademark of Kellogg.
Don't sue me.

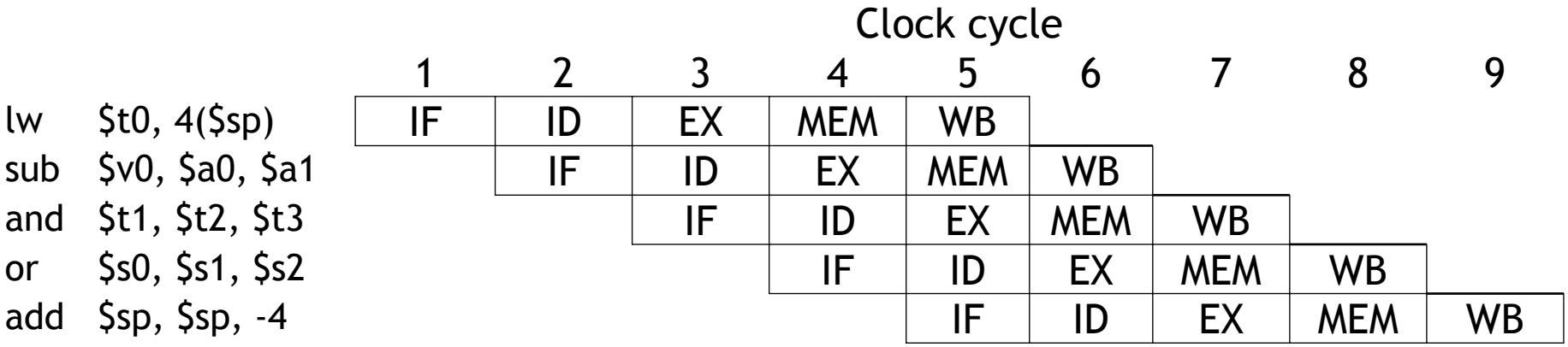
- So pipelining can really deliver the goods.
  - The CPI is nearly 1, like in a single-cycle processor.
  - But the cycle time is very short (2ns), like a multicycle datapath.
  - This adds up to excellent performance!

Ideal speedup



- In our pipeline, we can execute up to five instructions simultaneously.
  - This implies that the maximum speedup is 5 times.
  - In general, the ideal speedup equals the pipeline depth.
- Why was our speedup on the previous slide "only" 3.98 times?
  - The pipeline stages are imbalanced: a register file operation can be done in 1ns, but we must stretch that out to 2ns to keep the ID and WB stages synchronized with IF, EX and MEM.
  - Balancing the stages is one of the many hard parts in designing a pipelined processor.

The pipelining paradox



- Pipelining does *not* improve the execution time of any single instruction. Each instruction here actually takes *longer* to execute than in a single-cycle datapath (10ns vs. 8ns)!
- Instead, pipelining increases the throughput, or the amount of work done per unit time. Here, several instructions are executed together in each clock cycle.
- The result is improved execution time for a *sequence* of instructions, such as an entire program.

Summary

- Pipelining attempts to maximize hardware usage by overlapping the execution stages of several different instructions.
- Pipelining offers amazing speedup.
  - The CPU throughput is dramatically improved, because several instructions can be executing concurrently.
  - In the best case, one instruction finishes on every cycle, and the speedup is equal to the pipeline depth.
- Next time we'll see an actual datapath and control unit for a pipelined processor, so we can see how to make all these wild ideas work.