Performance of Single-cycle Design

$$\text{CPU time}_{X,P} = \text{Instructions executed}_P * \text{CPI}_{X,P} * \text{Clock cycle time}_X$$

CPI = 1 for a single-cycle design

- At the start of the cycle, PC is updated (PC + 4, or PC + 4 + offset × 4)
- New instruction loaded from memory, control unit sets the datapath signals appropriately so that
  - registers are read,
  - ALU output is generated,
  - data memory is accessed,
  - branch target addresses are computed, and
  - register file is updated
- In a single-cycle datapath everything must complete within one clock cycle, before the next clock cycle

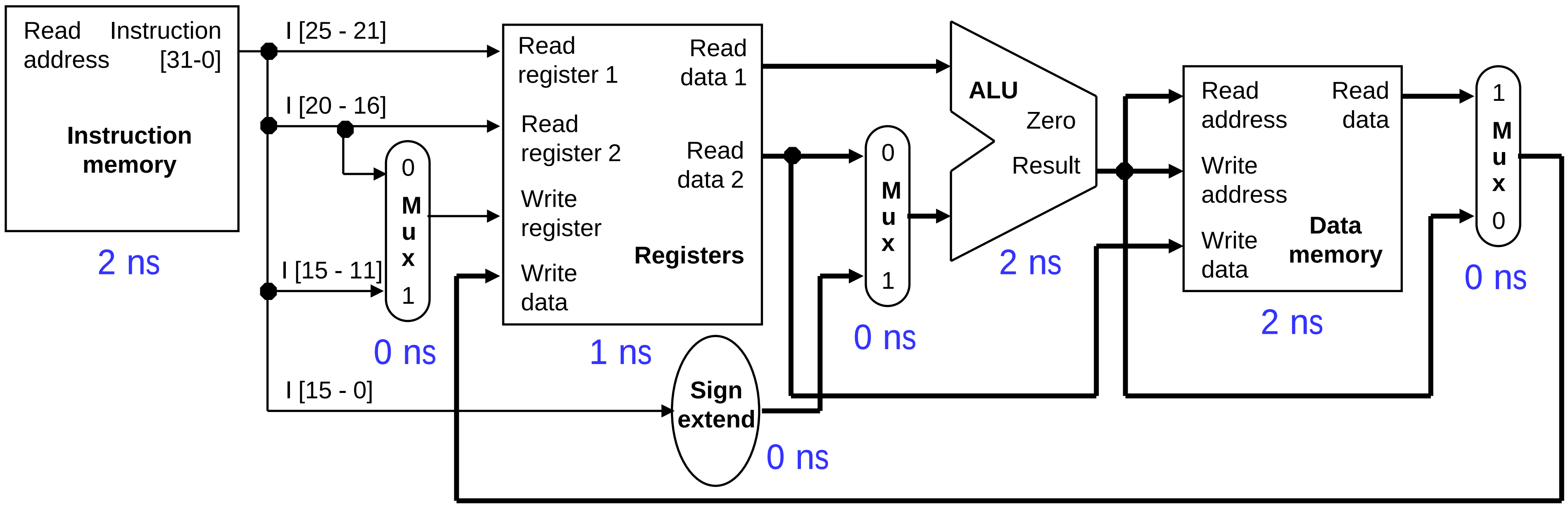
How long is that clock cycle?

Components of the data-path

- Each component of the datapath has an associated *delay* (latency)

$$\left.\begin{array}{ll}\text{reading the instruction memory} & \text{2ns} \\ \text{reading the register file} & \text{1ns} \\ \text{ALU computation} & \text{2ns} \\ \text{accessing data memory} & \text{2ns} \\ \text{writing to the register file} & \text{1ns}\end{array}\right\} \text{8ns}$$

- The cycle time has to be large enough to accommodate the *slowest* instruction



How bad is this?

- With these same component delays, a sw instruction would need 7ns, and beq would need just 5ns
- Let’s consider the **gcc** instruction mix:

Instruction	Frequency
Arithmetic	48%
Loads	22%
Stores	11%
Branches	19%

- With a single-cycle datapath, each instruction would require 8ns
- But if we could execute instructions as fast as possible, the average time per instruction for **gcc** would be:

(48% x 6ns) + (22% x 8ns) + (11% x 7ns) + (19% x 5ns) = 6.36ns

- The single-cycle datapath is about 1.26 times slower!

Improving performance

- Two ideas for improving performance:

  1. Spilt each instruction into multiple steps, each taking 1 cycle
     - steps: IF (instruction fetch), ID (instruction decode), EX (execute ALU operation), MEM (memory access), WB (register write-back)
     - slow instructions take more cycles than fast instructions
     - known as a *multi-cycle* implementation

  2. Crucial observation: each instruction uses only a *portion* of the datapath in each step
     - can *overlap* instructions; each uses one portion of the datapath
     - known as a *pipelined* implementation

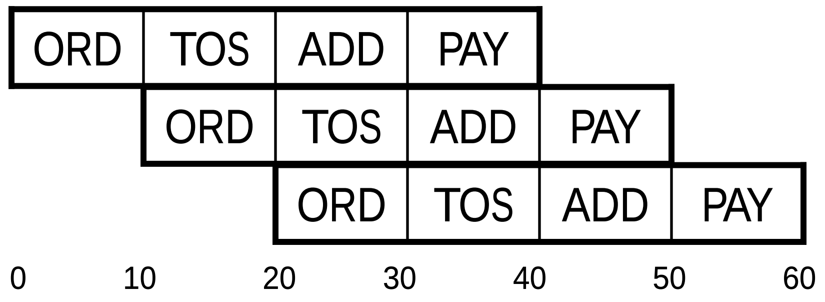
- Examples of pipelining: any assembly process (cars, sandwiches), multiple loads of laundry (washer + dryer can be pipelined), etc.

Pipelining: Example

- Assembling a sandwich: Order, Toast (optional), Add extras, Pay
  - ORD (8 seconds)
  - TOS (0 or 10 seconds)
  - ADD (0 to 10 seconds)
  - PAY (5 seconds)

A *single* sandwich takes between 13 and 33 seconds

- We can assemble sandwiches every 10 seconds with pipelining:



Pipelining lessons

- **Pipelining can *increase throughput* (#sandwiches per hour), but...**

1. Every sandwich must use *all* stages
   - prevents clashes in the pipeline
2. Every stage must take the same amount of time
   - limited by the *slowest* stage (in this example, 10 seconds)

- These two factors *decrease the latency* (time per sandwich)!

- For an optimal *k*-stage pipeline:
  1. every stage does useful work
  2. stage lengths are balanced

- Under these conditions, we *nearly* achieve the optimal speedup: *k*
  - "nearly" because there is still the *fill* and *drain* time

Pipelining not just Multiprocessing

- Pipelining does involve parallel processing, but in a specific way

- Both multiprocessing and pipelining relate to the processing of multiple "things" using multiple "functional units"
  - **In multiprocessing**, each thing is processed entirely by a single functional unit
    - e.g. multiple lanes at the supermarket
  - In **pipelining**, each thing is broken into a **sequence of pieces**, where each piece is handled by a **different** (specialized) functional unit
    - e.g. checker vs. bagger

- Pipelining and multiprocessing are not mutually exclusive
  - Modern processors do both, with multiple pipelines (e.g. superscalar)

- Pipelining is a general-purpose efficiency technique; used elsewhere in CS:
  - Networking, I/O devices, server software architecture

Pipelining MIPS

- Executing a MIPS instruction can take up to five stages

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

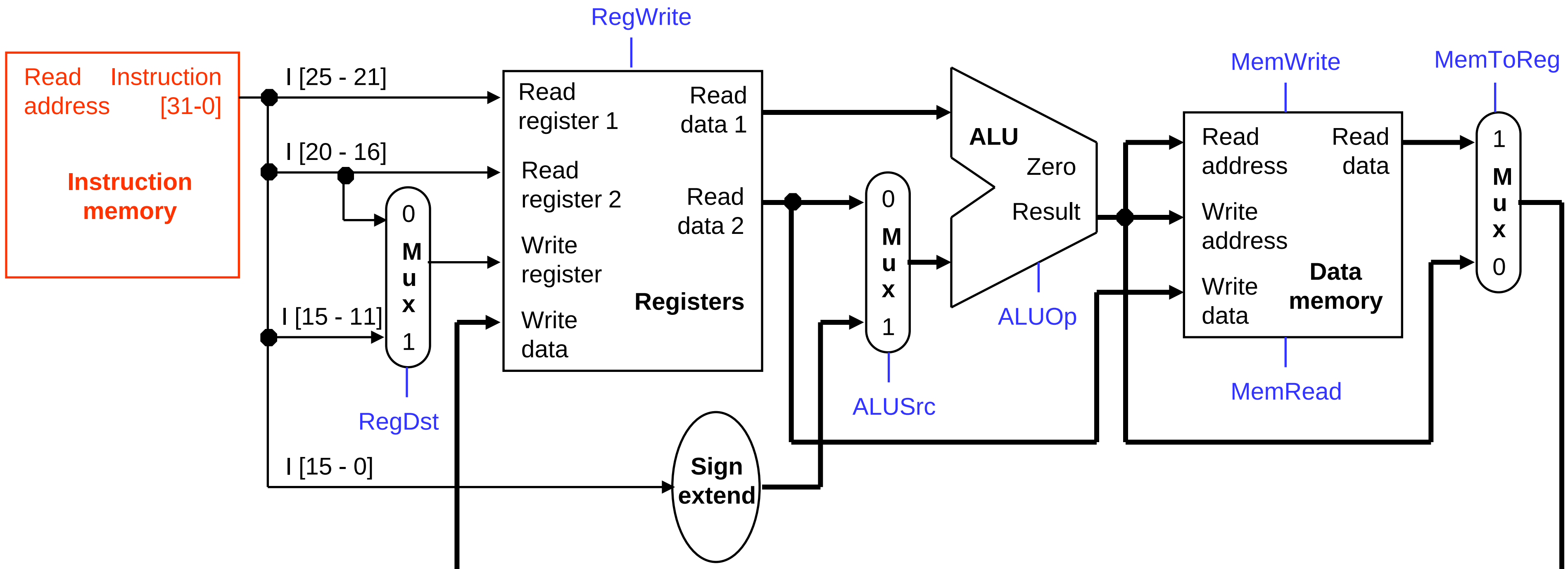
- Not all instructions need all five stages and stages have different lengths

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

- Clock cycle time determined by length of slowest stage (2ns here)

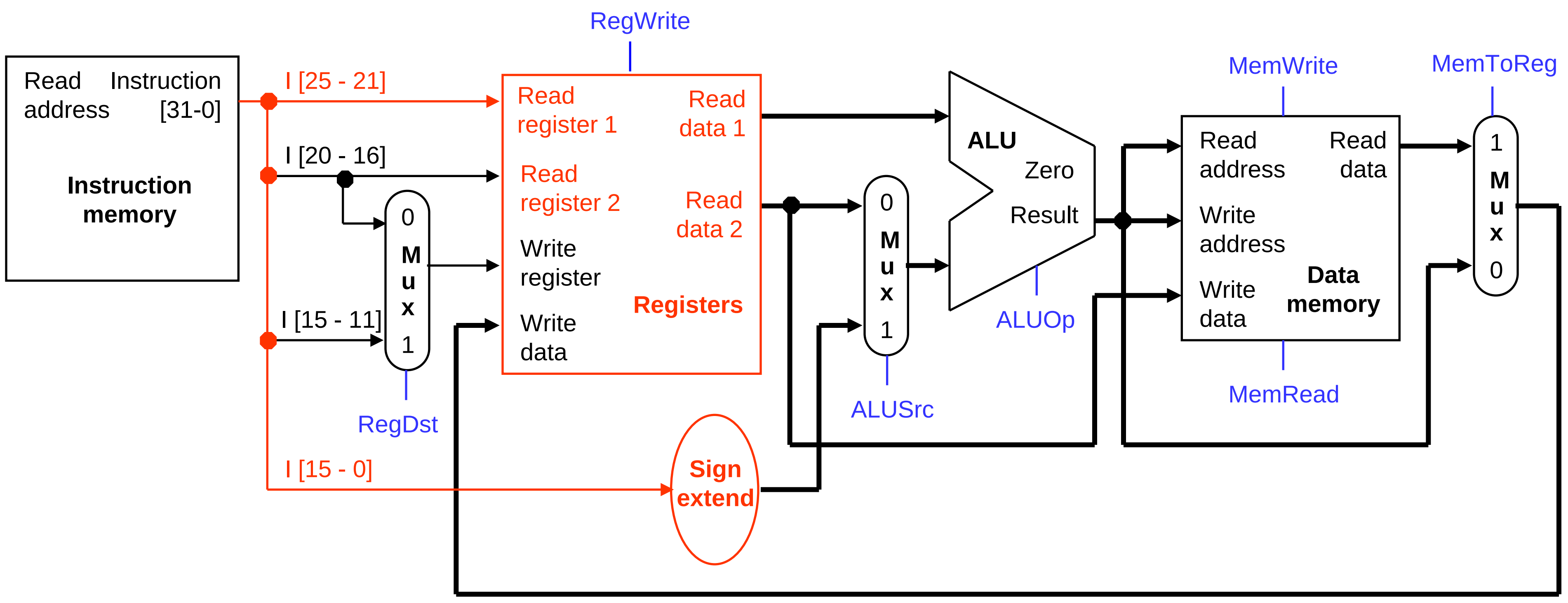
Instruction Fetch (IF)

- While IF is executing, the rest of the datapath is sitting idle…



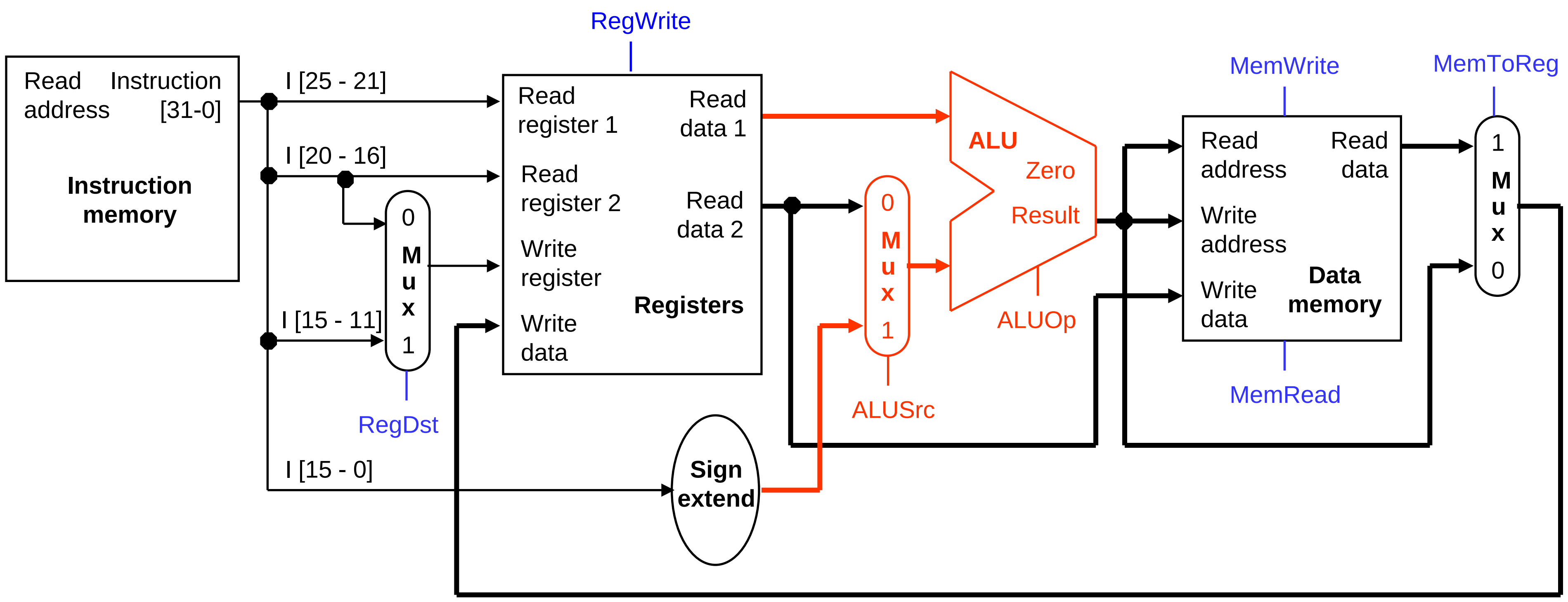
Instruction Decode (ID)

- Then while ID is executing, the IF-related portion becomes idle…



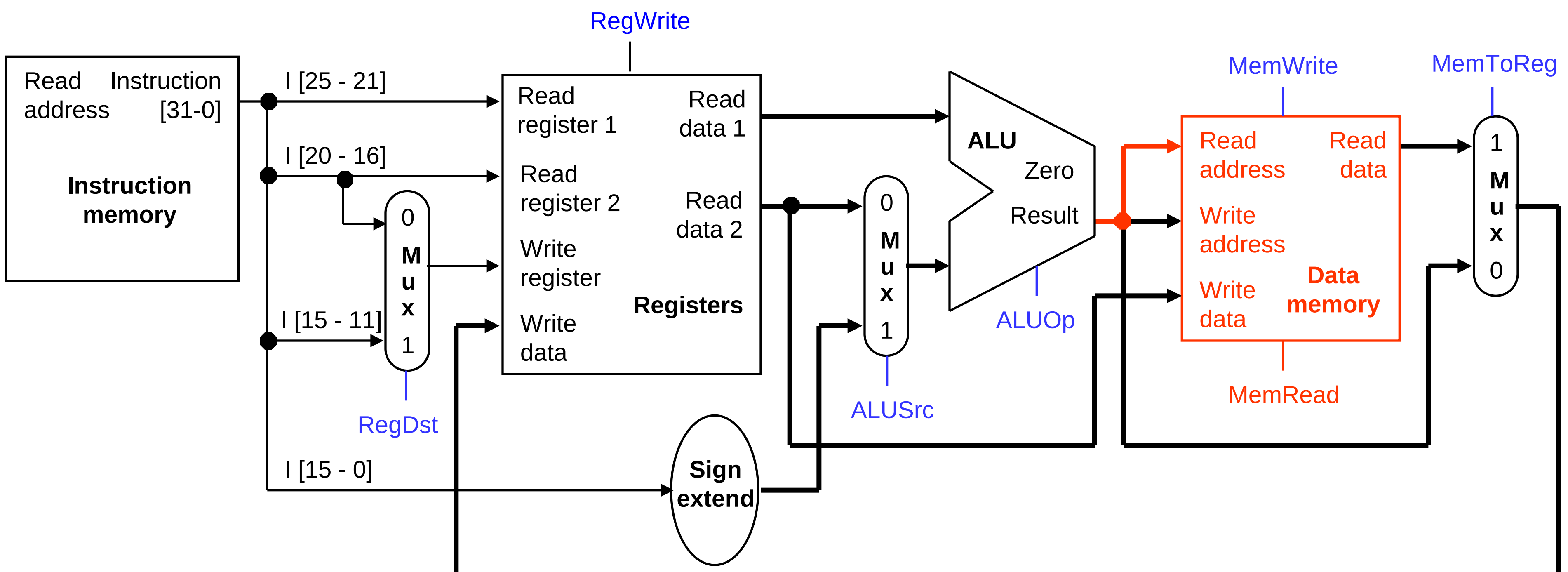
Execute (EX)

- ..and so on for the EX portion...



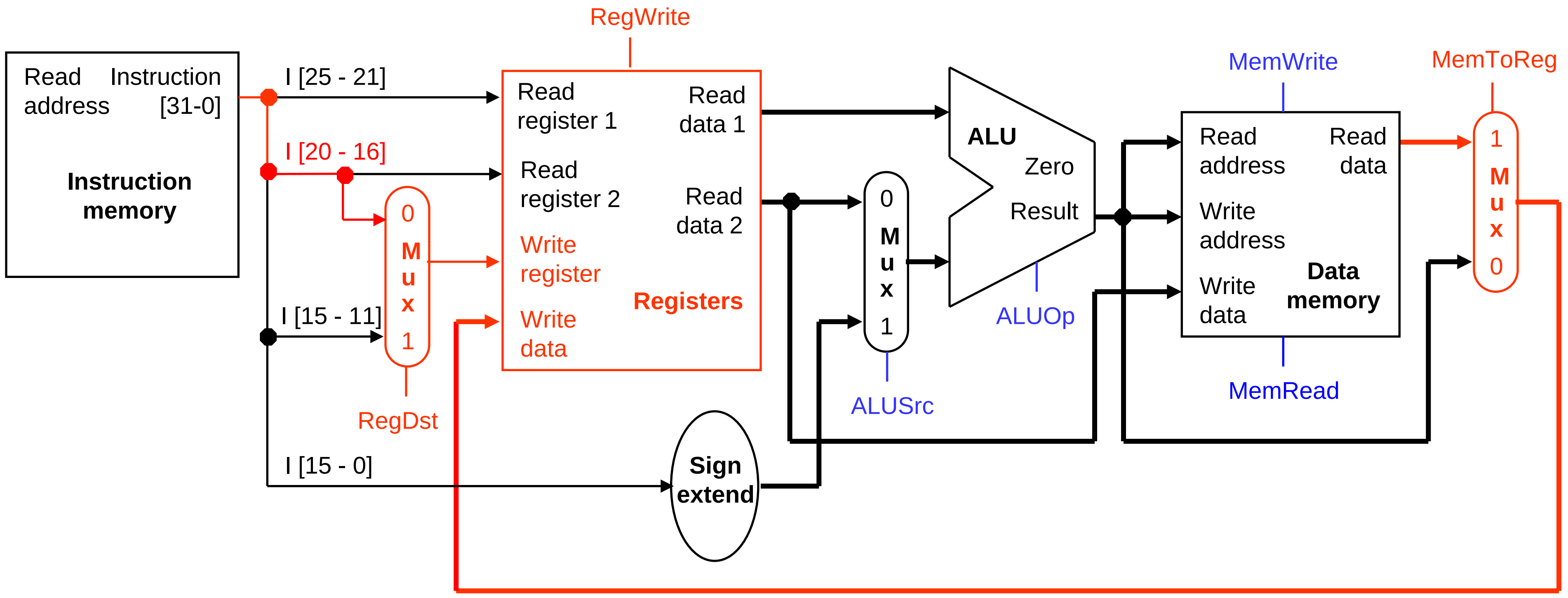
Memory (MEM)

- …the MEM portion…



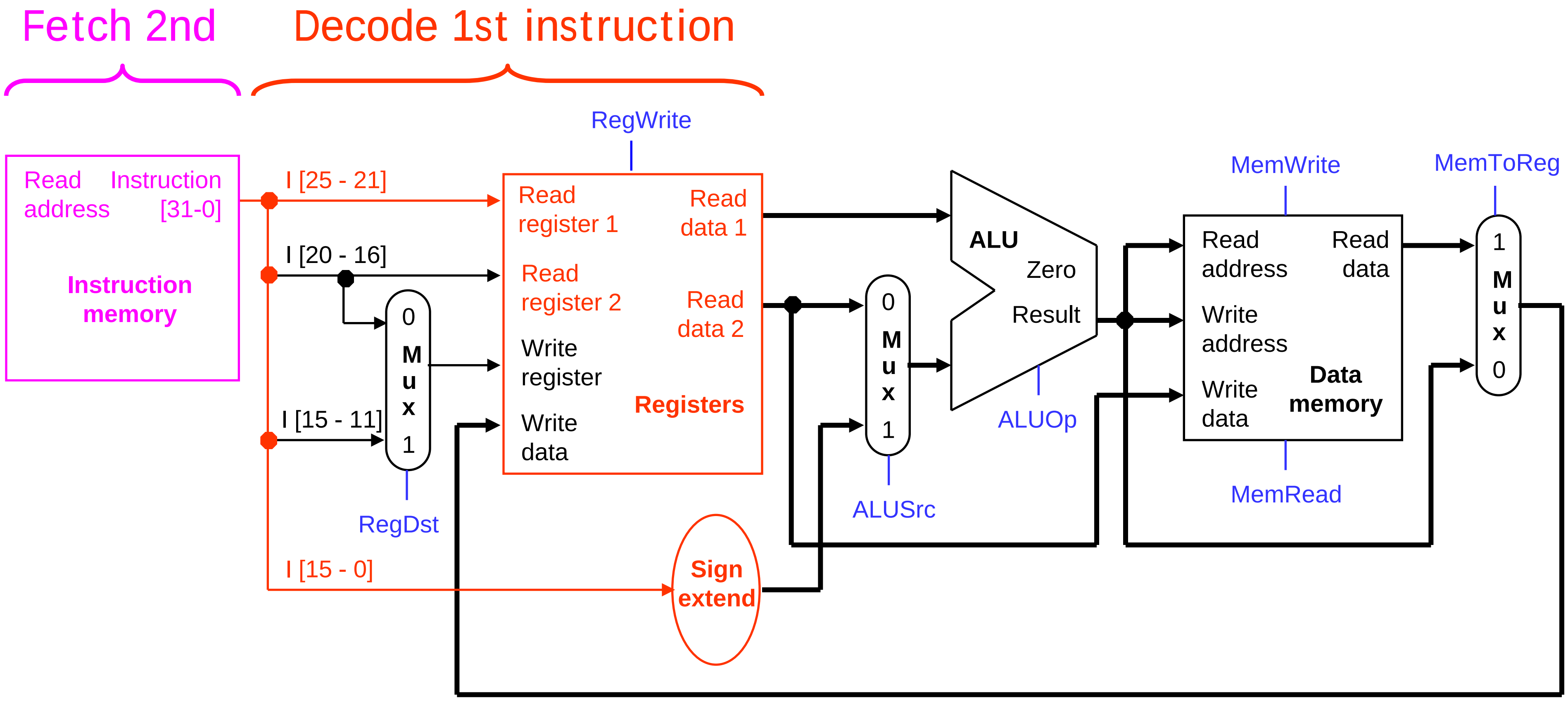
Writeback (WB)

- …and the WB portion
- What about the "clash" with the IF stage over the register file?
- Answer: Register file is *written* on the positive edge, but *read* later in the clock cycle. Hence, there is no clash



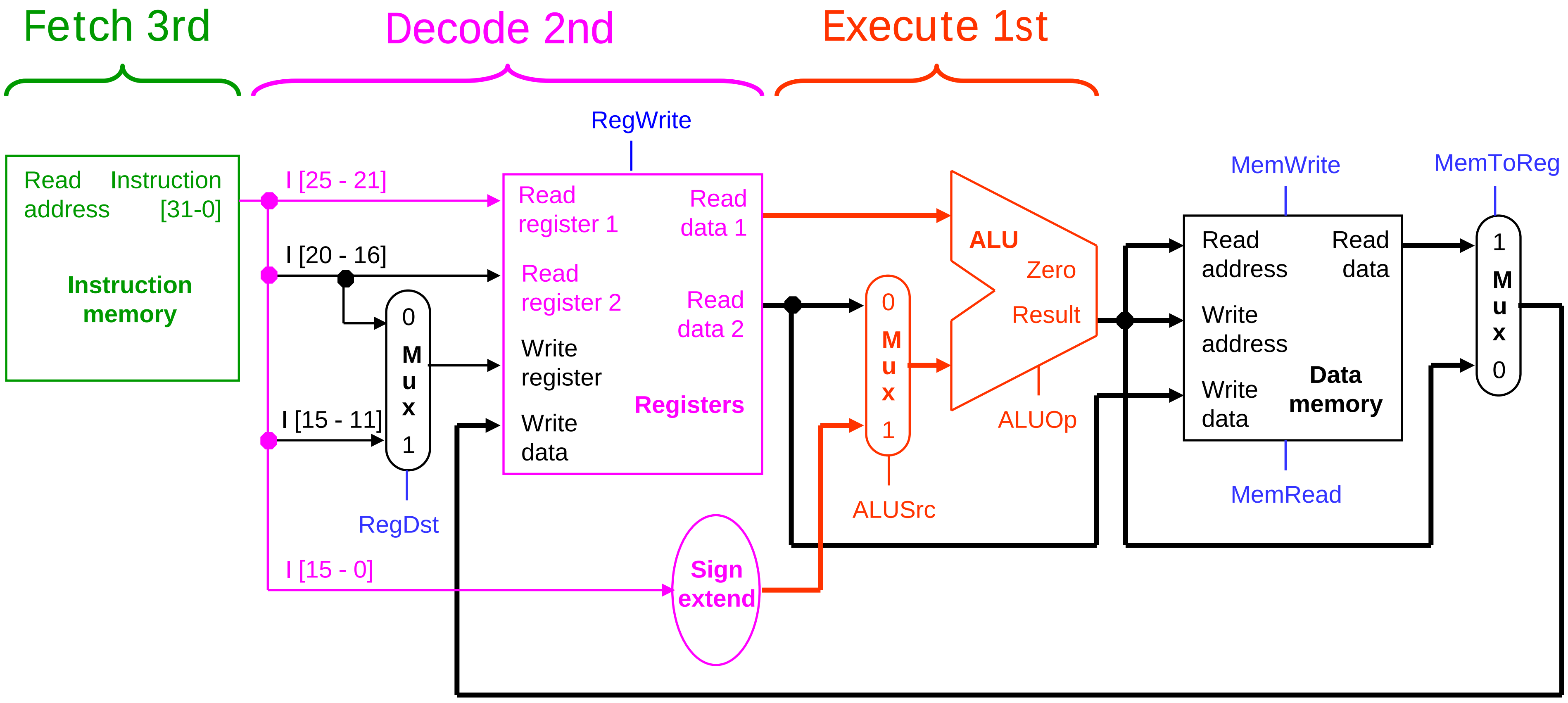
Decoding and fetching together

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



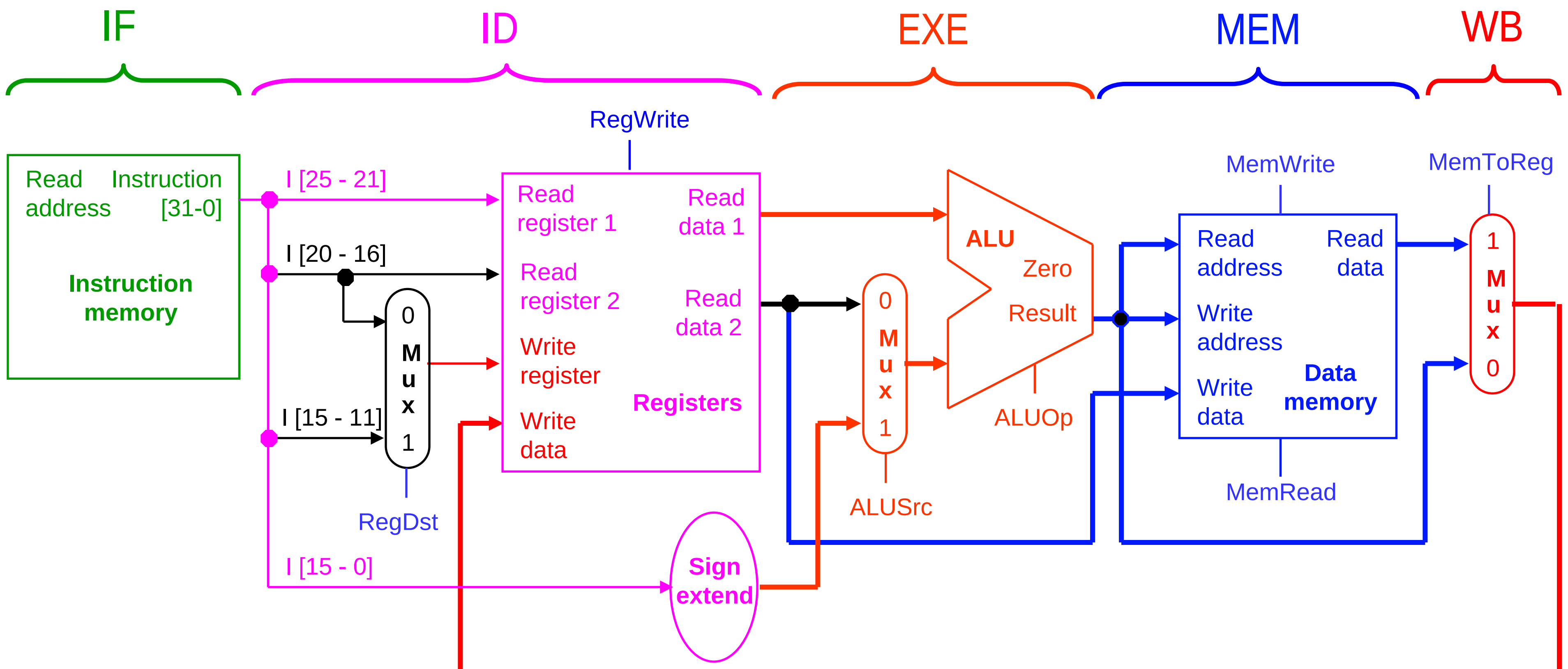
Executing, decoding and fetching

- 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!



Break datapath into 5 stages

- Each stage has its own functional units
- *Full* pipeline ⇒ the datapath is simultaneously working on 5 instructions!



newest

oldest

A pipeline diagram

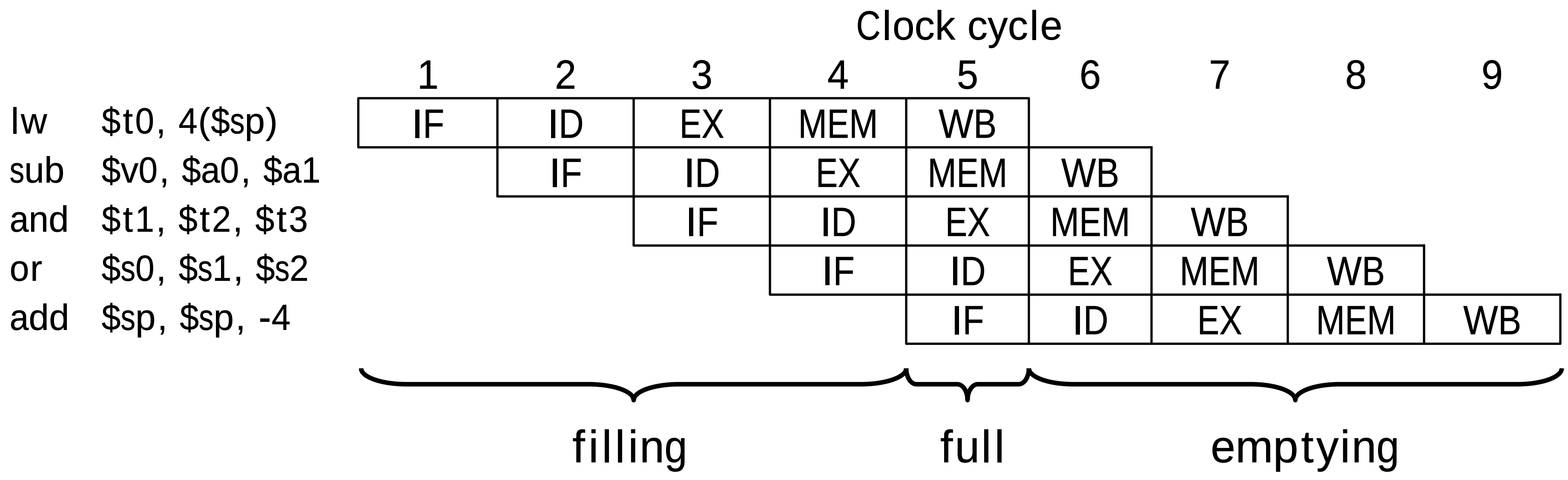
Clock cycle

		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
sub	$v0, $a0, $a1		IF	ID	EX	MEM	WB			
and	$t1, $t2, $t3			IF	ID	EX	MEM	WB		
or	$s0, $s1, $s2				IF	ID	EX	MEM	WB	
addi	$sp, $sp, -4					IF	ID	EX	MEM	WB

- 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

- *Example*: In cycle 3, there are three active instructions:
  - 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



Clock cycle

		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
sub	$v0, $a0, $a1		IF	ID	EX	MEM	WB			
and	$t1, $t2, $t3			IF	ID	EX	MEM	WB		
or	$s0, $s1, $s2				IF	ID	EX	MEM	WB	
add	$sp, $sp, -4					IF	ID	EX	MEM	WB

- The pipeline depth is the number of stages—in this case, five
- The pipeline is filling in the first four cycles (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/draining

Pipelining Performance

	Clock cycle								
	1	2	3	4	5	6	7	8	9
lw $t0, 4($sp)	IF	ID	EX	MEM	WB				
lw $t1, 8($sp)		IF	ID	EX	MEM	WB			
lw $t2, 12($sp)			IF	ID	EX	MEM	WB		
lw $t3, 16($sp)				IF	ID	EX	MEM	WB	
lw $t4, 20($sp)					IF	ID	EX	MEM	WB

$\underbrace{\hspace{8em}}_{\text{filling}}$

- How many cycles to execute *N* instructions on a *k* stage pipeline?

- Solution 1: *k* – 1 cycles to fill the pipeline + one cycle per instruction
  = *k* – 1 + *N* cycles

- Solution 2: *k* cycles for the first instruction + one cycle for each of the remaining *N* – 1 instructions

- When *N* = 1000, how much faster is a 5-stage pipeline (2ns clock cycle) vs. a single cycle implementation (8ns clock cycle)?

Pipeline Datapath: Resource Requirements

		Clock cycle								
		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
lw	$t1, 8($sp)		IF	ID	EX	MEM	WB			
lw	$t2, 12($sp)			IF	ID	EX	MEM	WB		
lw	$t3, 16($sp)				IF	ID	EX	MEM	WB	
lw	$t4, 20($sp)					IF	ID	EX	MEM	WB

- 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?
  - Separate ADDER and ALU
  - Two memories (instruction memory and data memory)