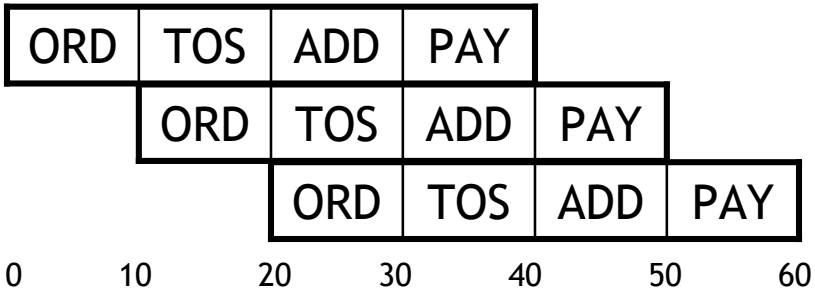
Example from section

- Assembling a sandwich:
  - 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

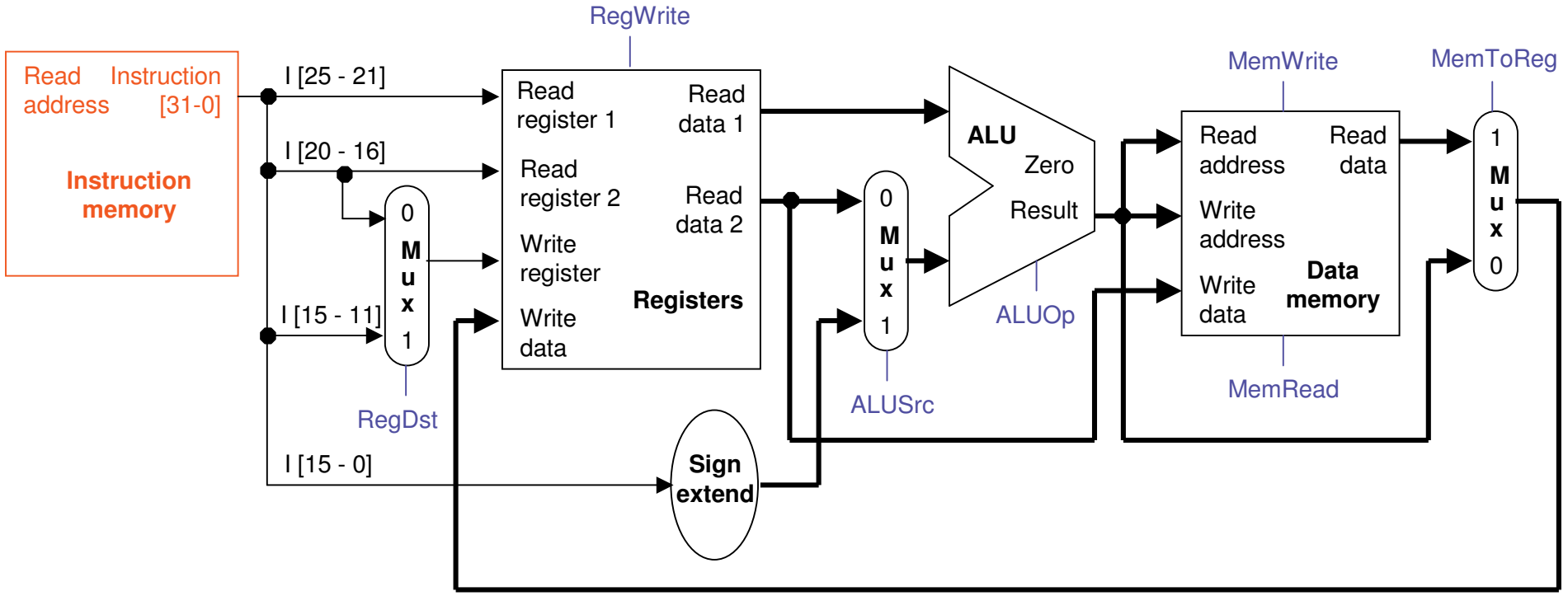
- Not all instructions need all five stages…

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

…but a single-cycle datapath must accommodate all 5 stages in *one* clock

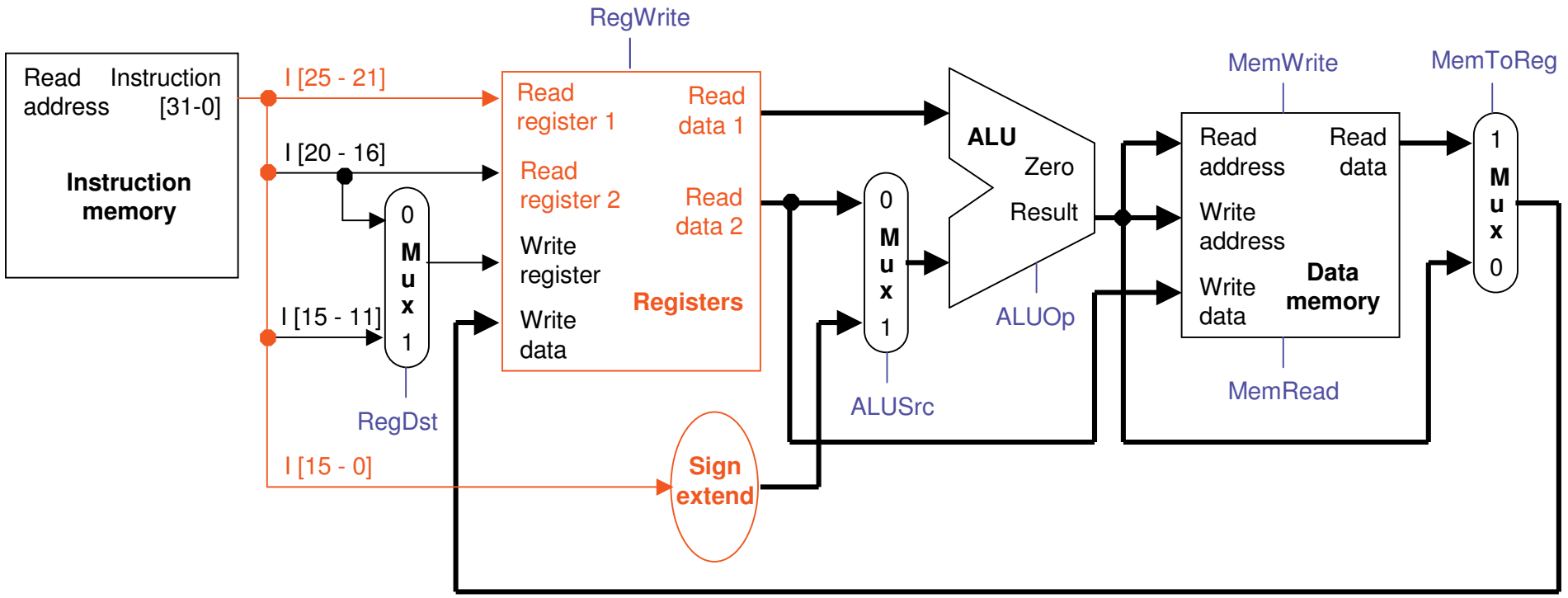
Instruction Fetch (IF)

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



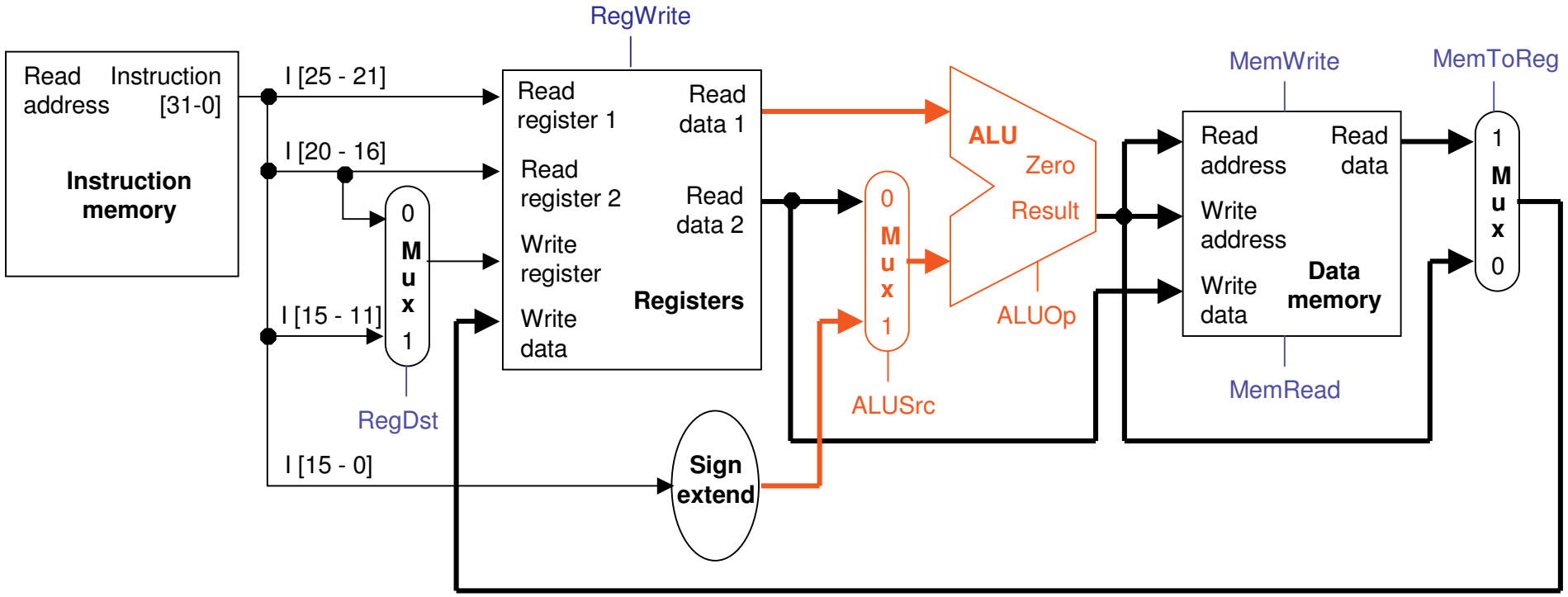
Instruction Decode (ID)

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



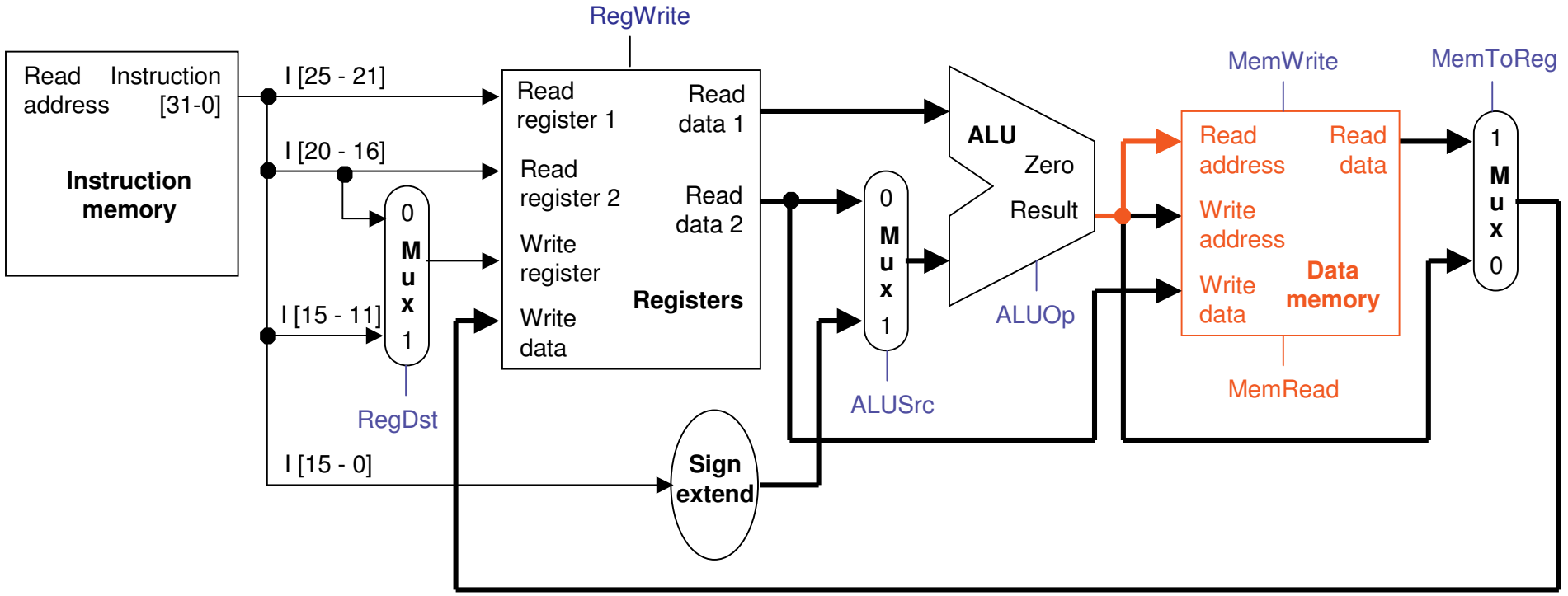
Execute (EX)

- ..and so on for the EX portion…



Memory (MEM)

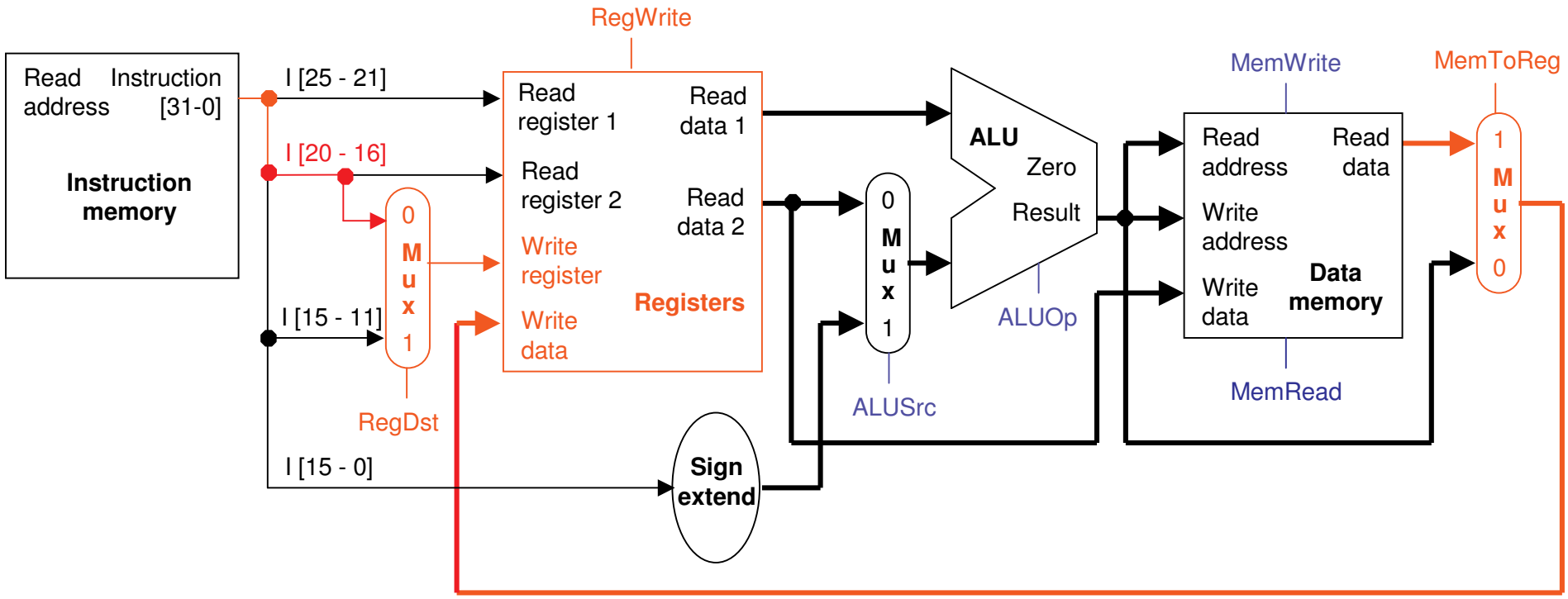
- …the MEM portion…



Writeback (WB)

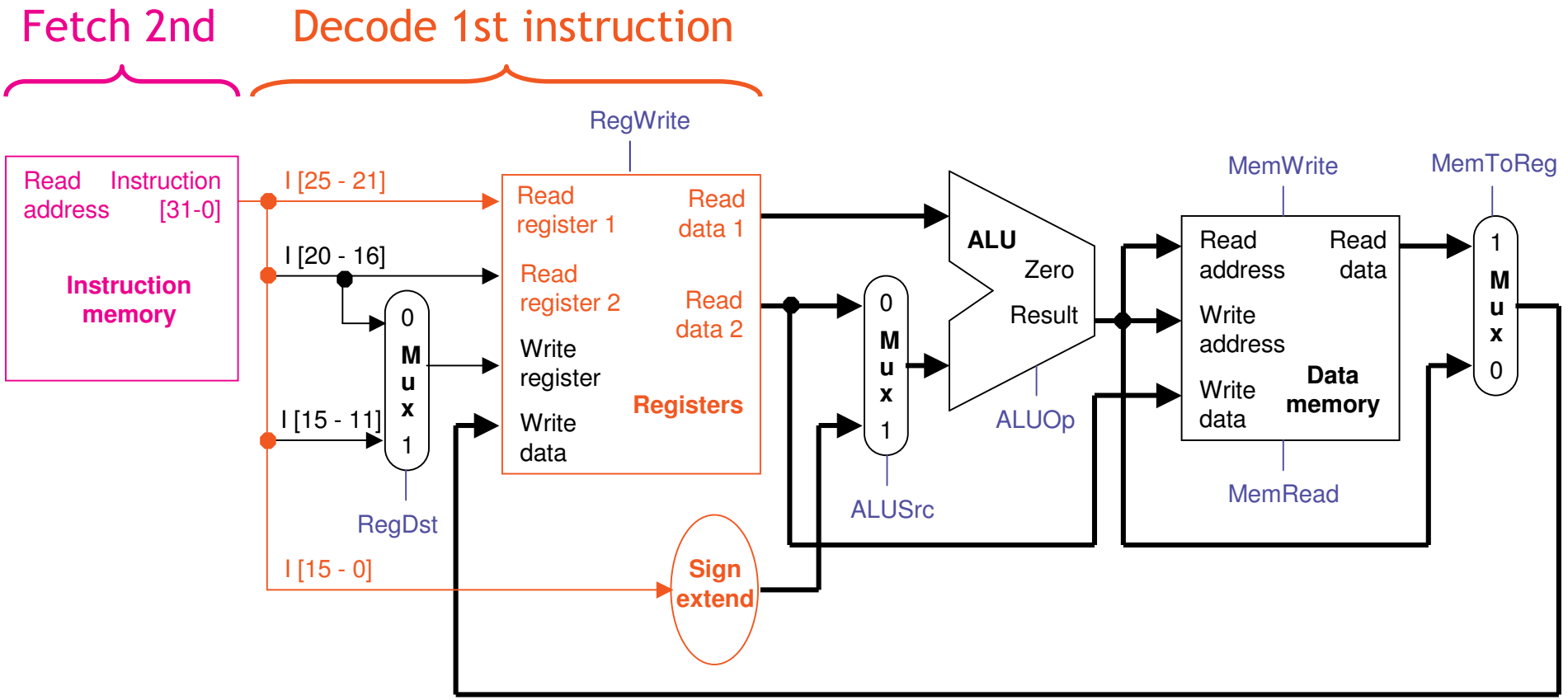
- …and the WB portion.
  - what about the "clash" with the IF stage over the register file?

  Answer: Register file is *written* on the positive edge, but *read* later in the clock cycle. Hence, there is no clash.



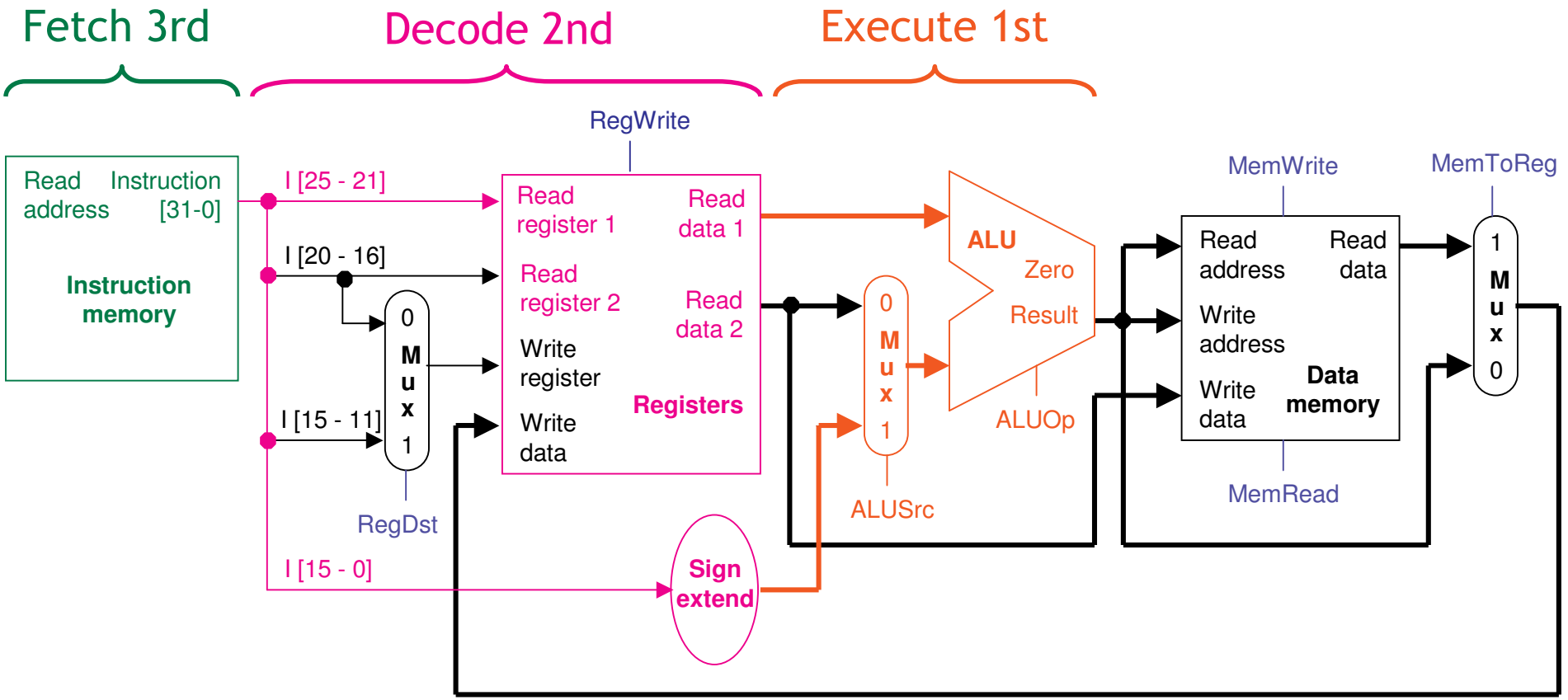
Decoding and fetching together

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



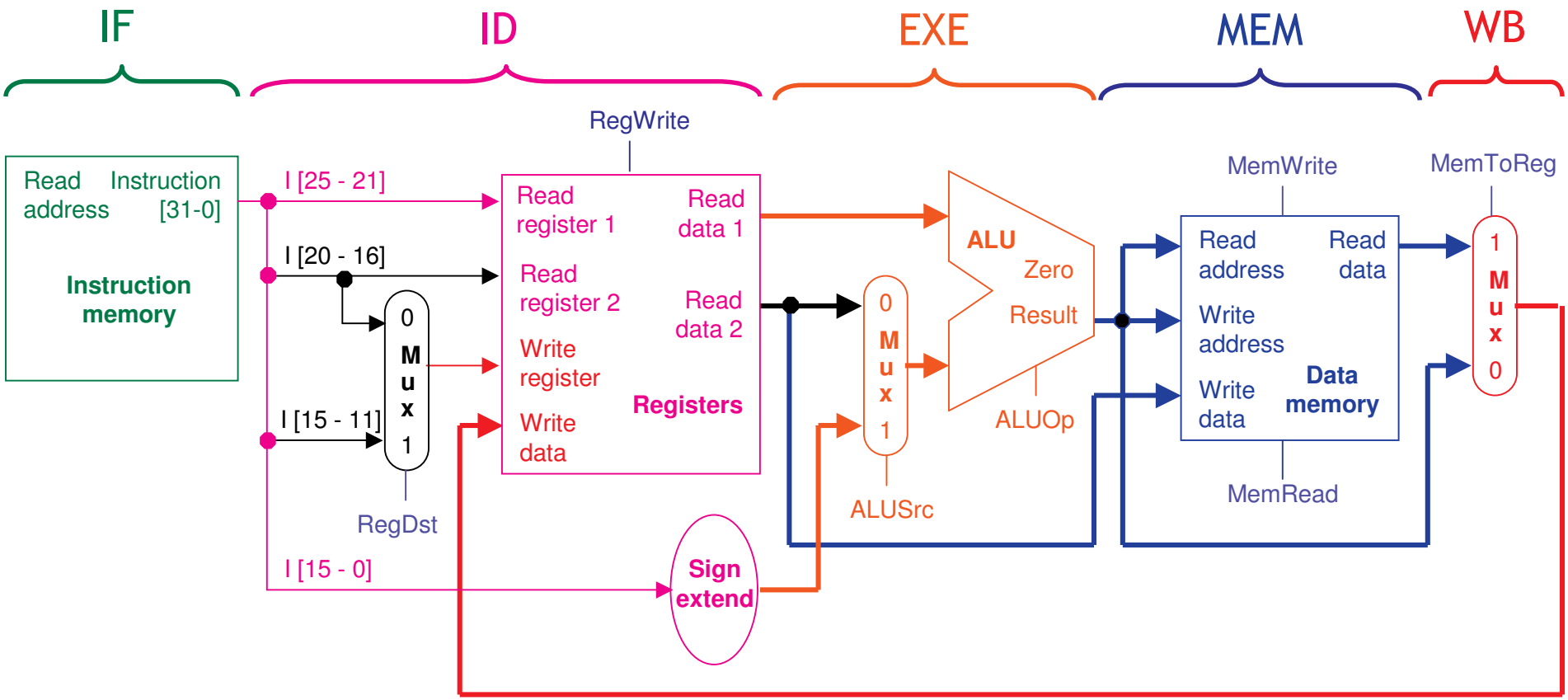
Executing, decoding and fetching

- 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

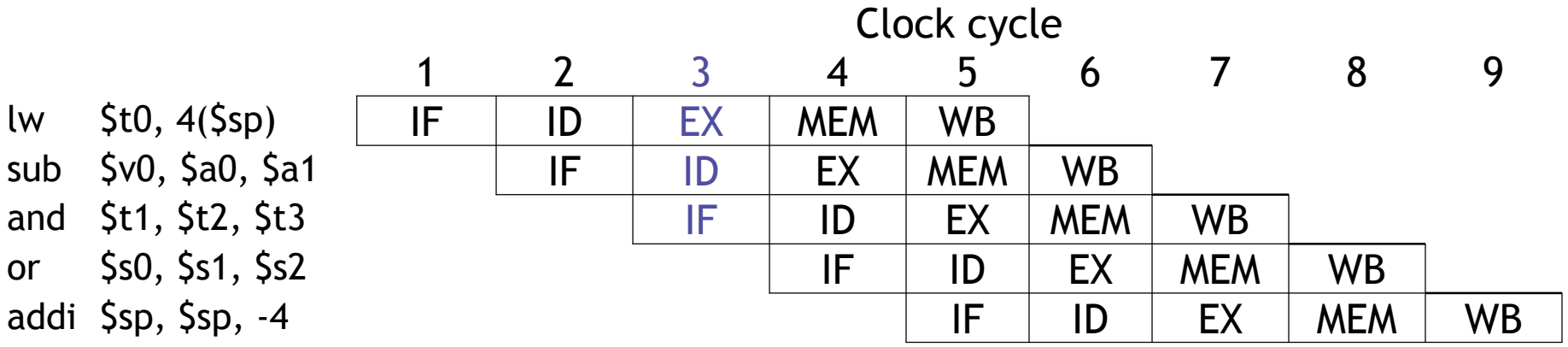
- Each stage has its own functional units
- *Full* pipeline ⇒ the datapath is simultaneously working on 5 instructions!



newest

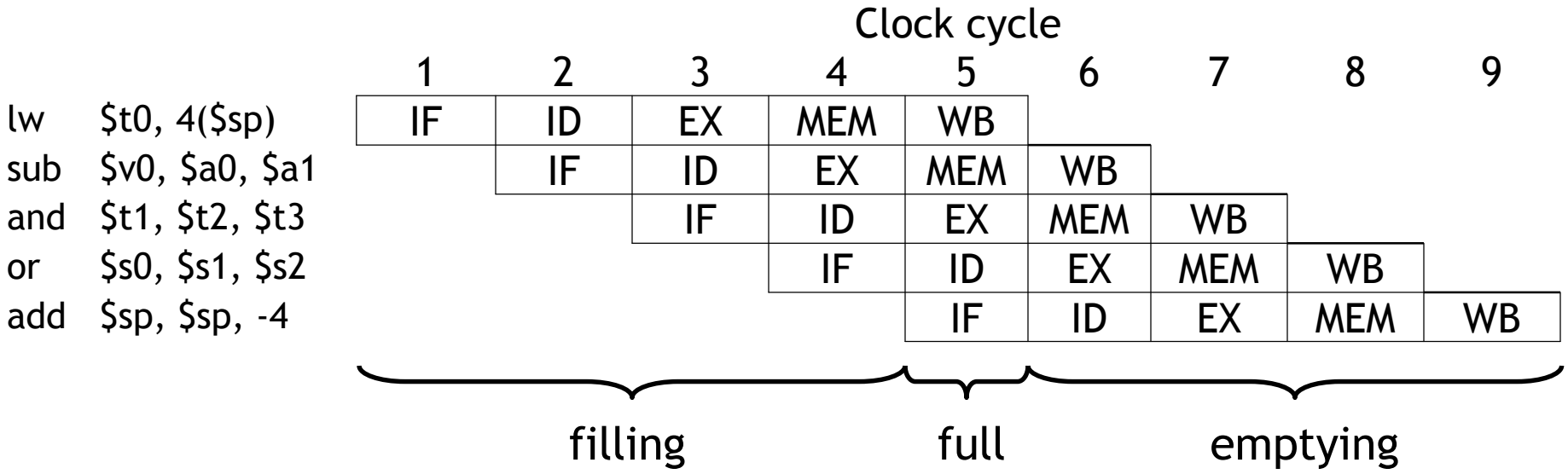
oldest

A pipeline diagram



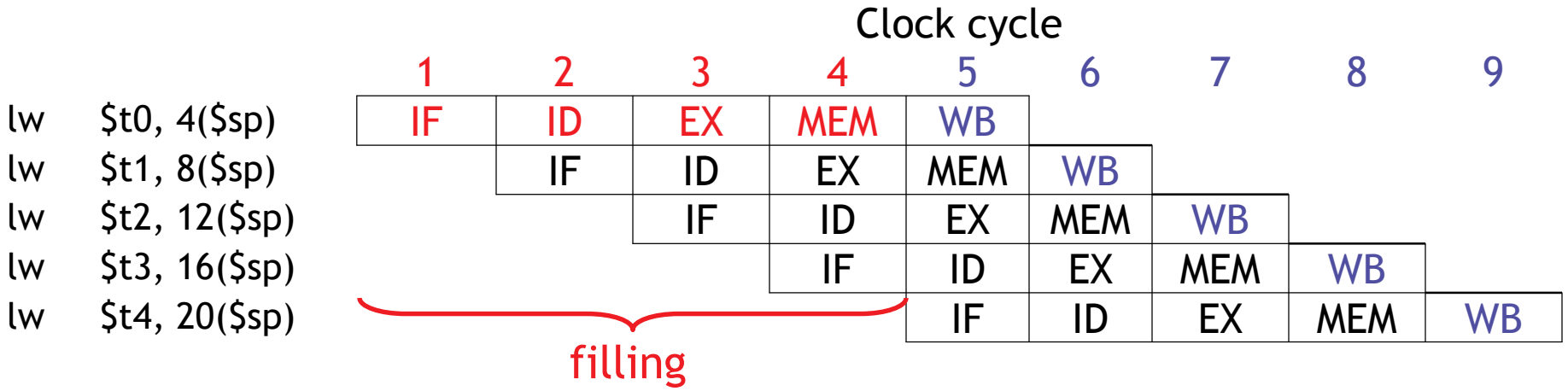
- A pipeline diagram shows the execution of a series of instructions
  - The instruction sequence is shown vertically, from top to bottom
  - Clock cycles are shown horizontally, from left to right
  - Each instruction is divided into its component stages
- This clearly indicates the overlapping of instructions. 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

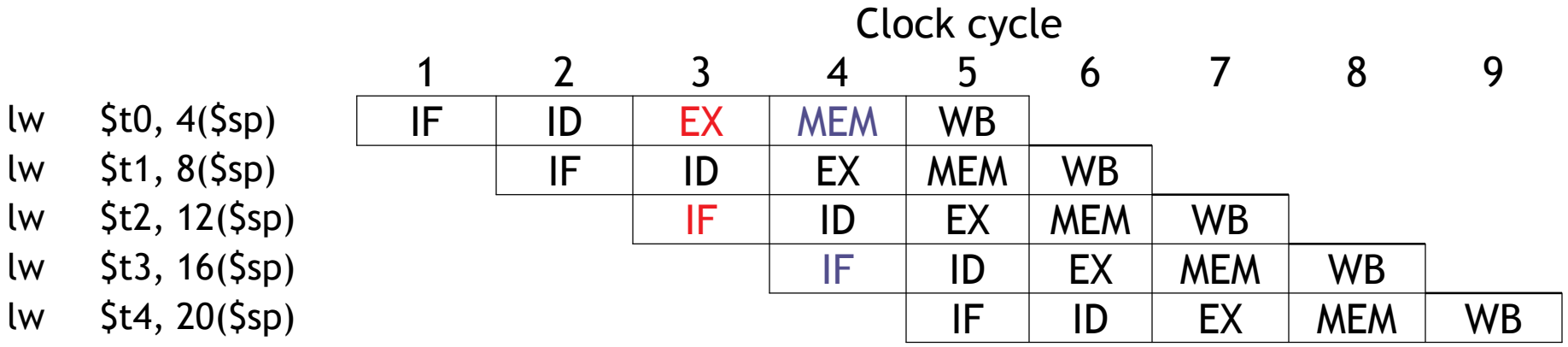
Pipelining Performance



- Execution time on ideal pipeline:
  - time to fill the pipeline + one cycle per instruction
  - How long for N instructions? $k - 1 + N$, where k = pipeline depth
- Alternate way of arriving at this formula: k cycles for the first instruction, plus 1 for each of the remaining $N - 1$ instructions.

- Compare this pipelined implementation (2ns clock period) vs. a single cycle implementation (8ns clock period). How much faster is pipelining for N=1000 ?
  - See Section 5 for an approach to solving such problems.

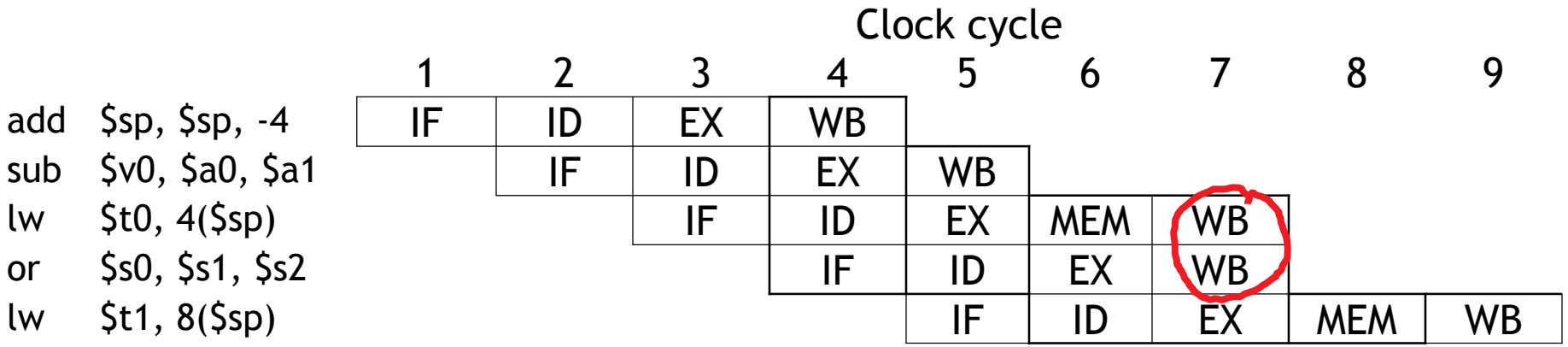
Pipeline Datapath: Resource Requirements



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

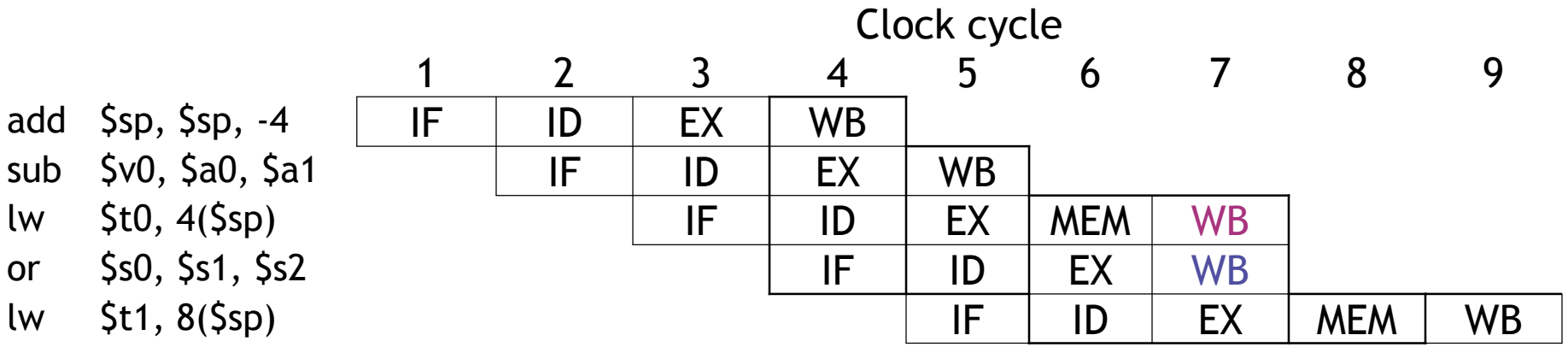
Pipelining other instruction types

- 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?



Important Observation

- Each functional unit can only be used once per instruction
- Each functional unit must be used at the same stage for all instructions:
  - Load uses Register File's Write Port during its 5th stage
  - R-type uses Register File's Write Port during its 4th stage



A solution: Insert NOP stages

- Enforce uniformity
  - Make all instructions take 5 cycles.
  - Make them have the same stages, in the same order
    - Some stages will do nothing for some instructions

R-type	IF	ID	EX	NOP	WB

Clock cycle

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

- Stores and Branches have NOP stages, too…

store	IF	ID	EX	MEM	NOP
branch	IF	ID	EX	NOP	NOP

Summary

- Pipelining attempts to maximize instruction throughput by overlapping the execution of multiple instructions.
- Pipelining offers amazing speedup.
  - In the best case, one instruction finishes on every cycle, and the speedup is equal to the pipeline depth.
- The pipeline datapath is much like the single-cycle one, but with added pipeline registers
  - Each stage needs is own functional units
- Next time we'll see the datapath and control, and walk through an example execution.