

Lecture 9

- Today's objectives:
 - Intro to Pipelining

- office hours?

M 1:30 - 2:30 ? ~12

F 2:30 - 3:30

Tu

A relevant question

- Assuming you've got:

- One washer (takes 30 minutes)



- One drier (takes 40 minutes)



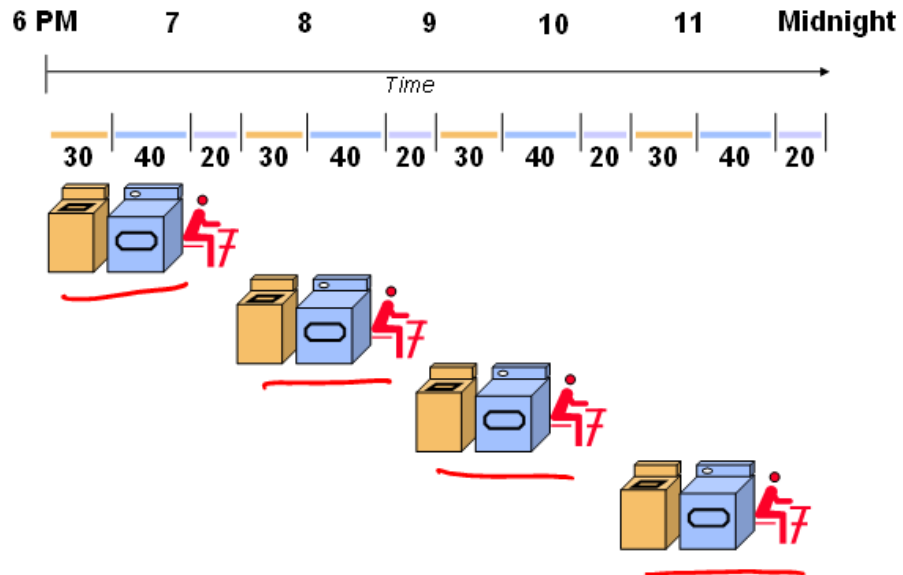
- One "folder" (takes 20 minutes)



- It takes 90 minutes to wash, dry, and fold 1 load of laundry.
 - How long does 4 loads take?

$$90 \times 4 = 360 \text{ min} \quad \underline{\sim 6 \text{ hrs}}$$

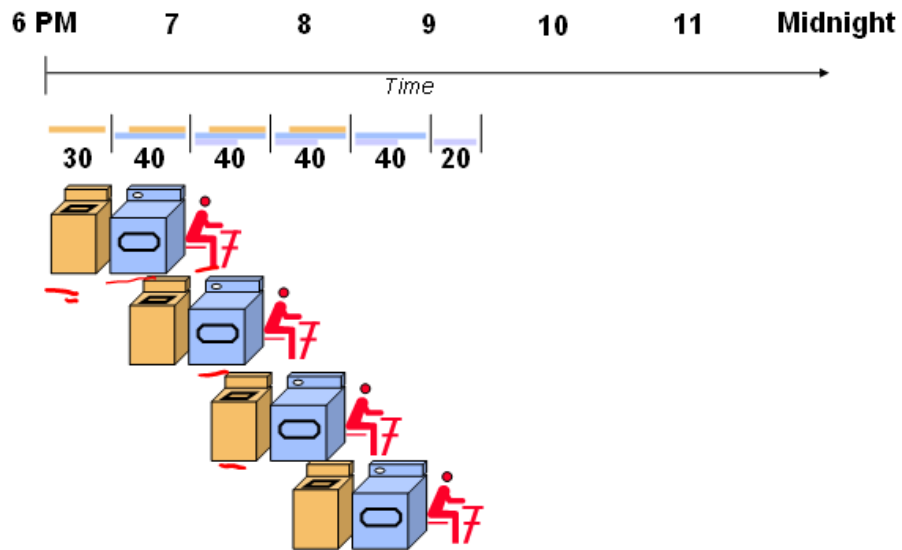
The slow way



- If each load is done sequentially it takes 6 hours

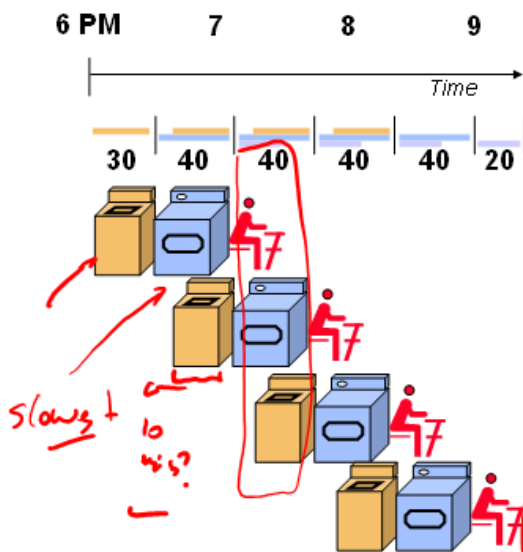
Laundry Pipelining

- Start each load as soon as possible
 - Overlap loads



- Pipelined laundry takes 3.5 hours 210 min

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

→ - Balanced stages to
- infinite amount of clocks⁵

Pipelining

- Pipelining is a general-purpose efficiency technique
 - It is not specific to processors
- Pipelining is used in:
 - Assembly lines ✓
 - Bucket brigades ✓
 - Fast food restaurants ✓
- Pipelining is used in other CS disciplines:
 - Networking ✓
 - Server software architecture
- Useful to increase throughput in the presence of long latency
 - More on that later...

Instruction execution review

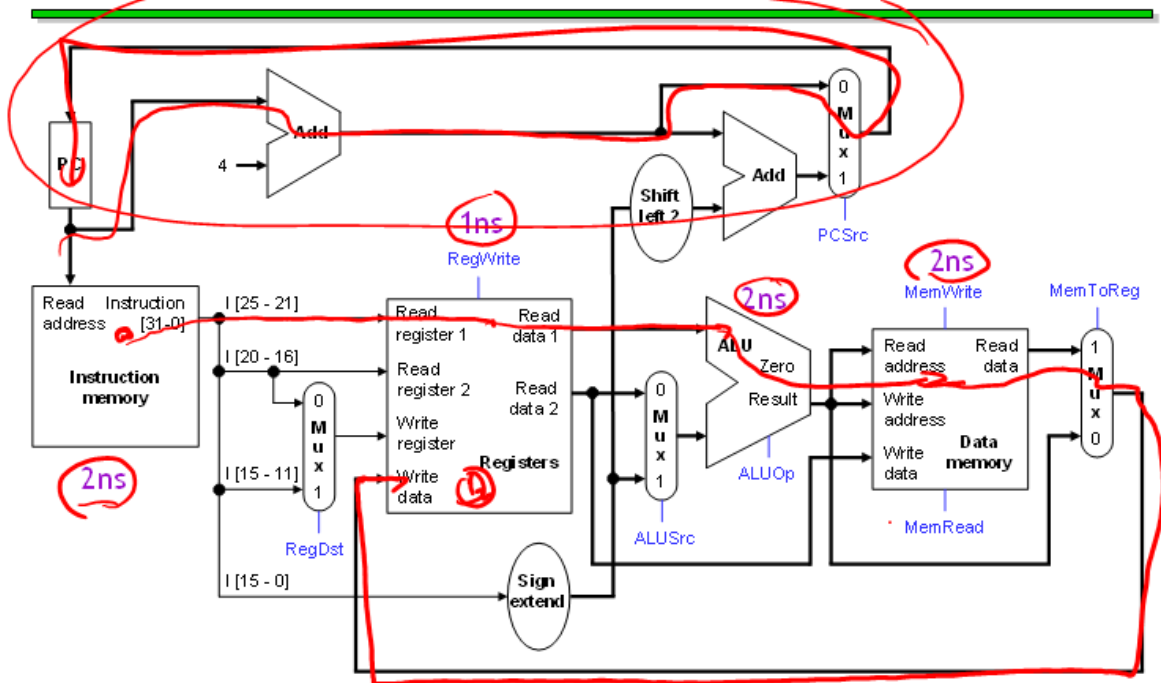
- Executing a MIPS instruction can take up to five steps.

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

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

Instruction	Steps required				
	IF	ID	EX	MEM	WB
→ 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



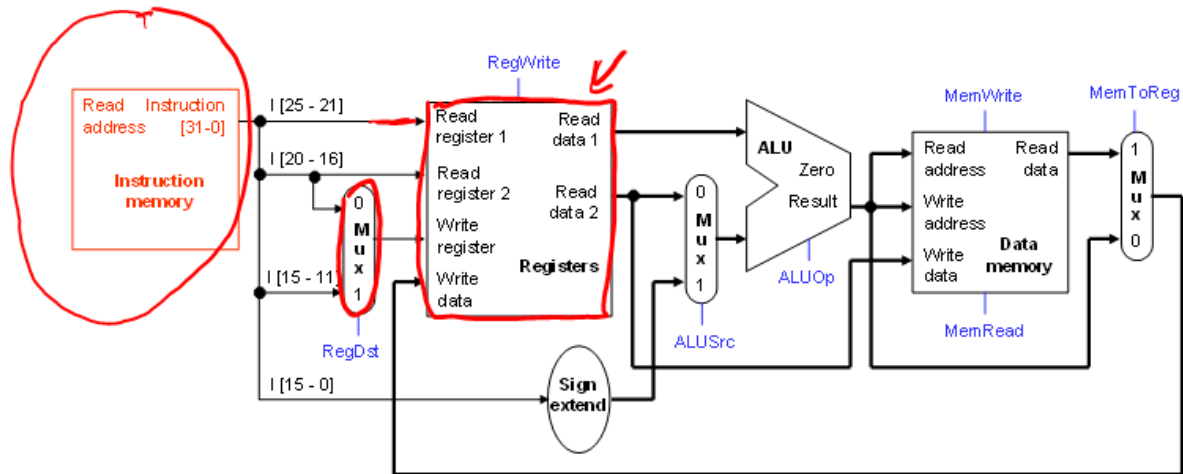
- How long does it take to execute each instruction? 8ns

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.

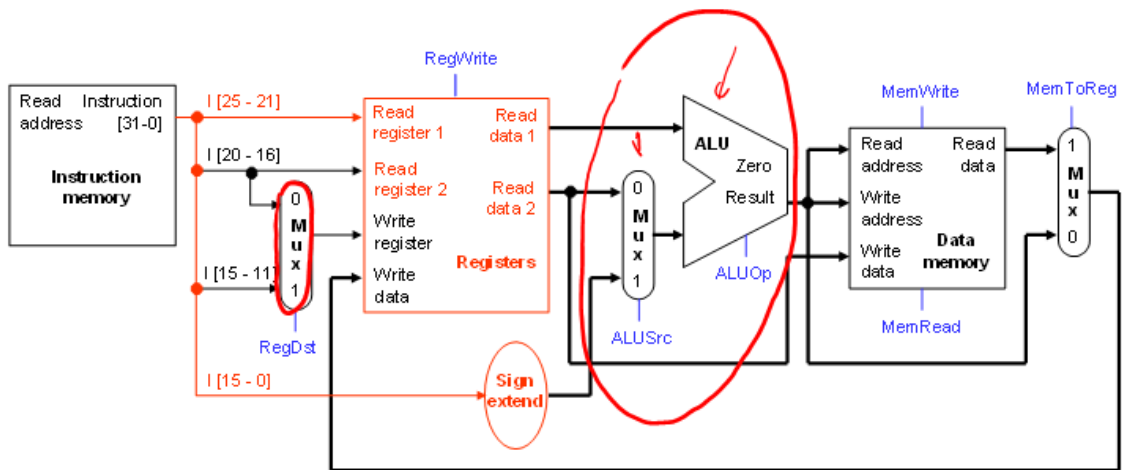
Example: Instruction Fetch (IF)

- 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.



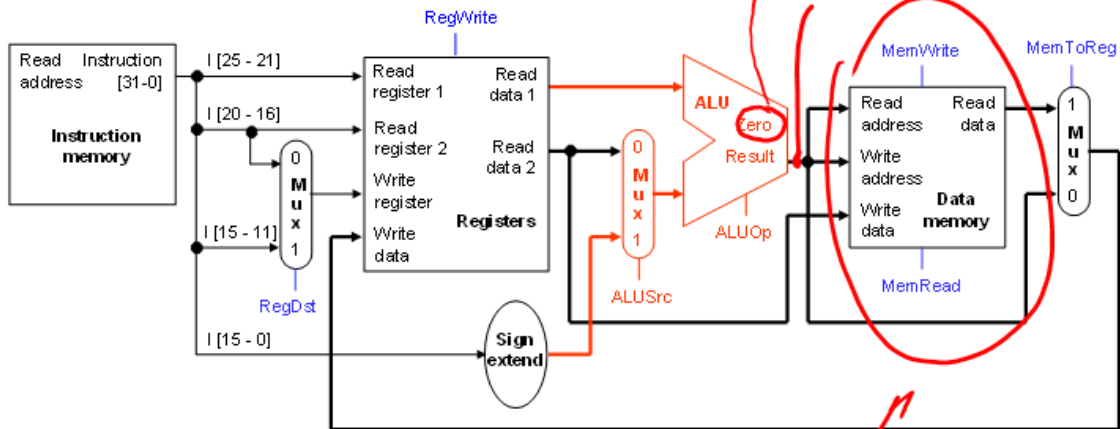
Instruction Decode (ID)

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



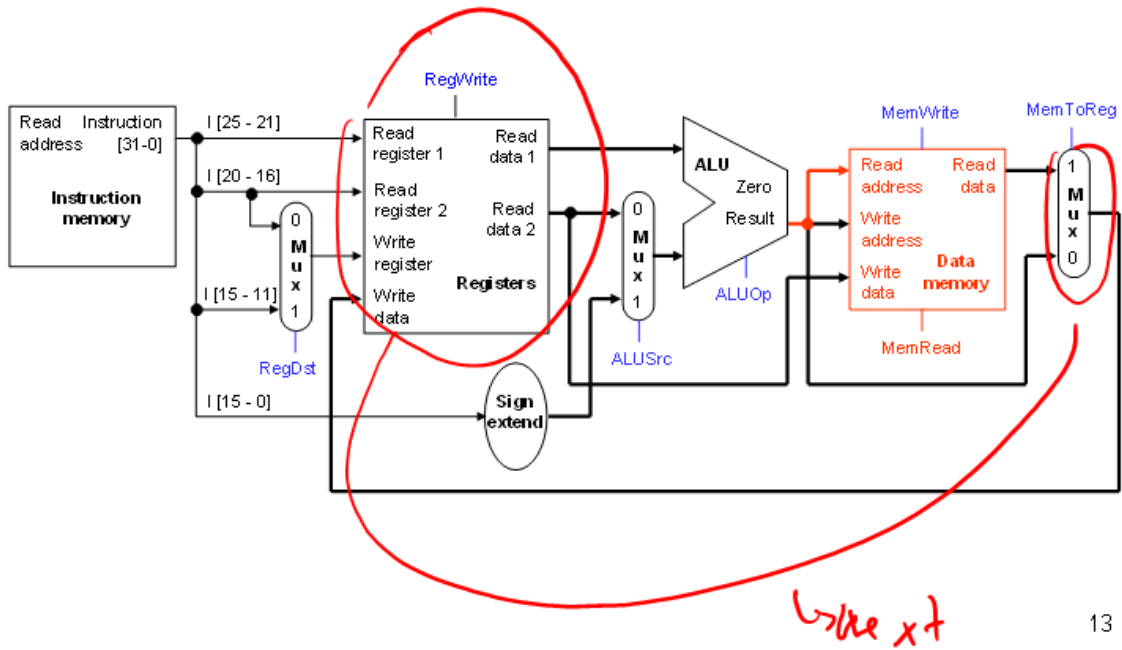
Execute (EX)

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



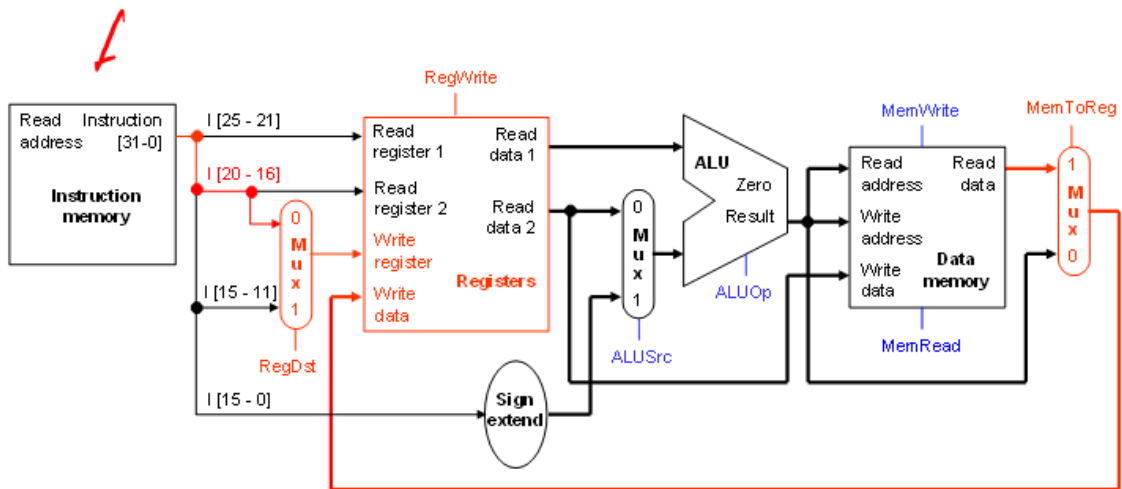
Memory (MEM)

- 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.

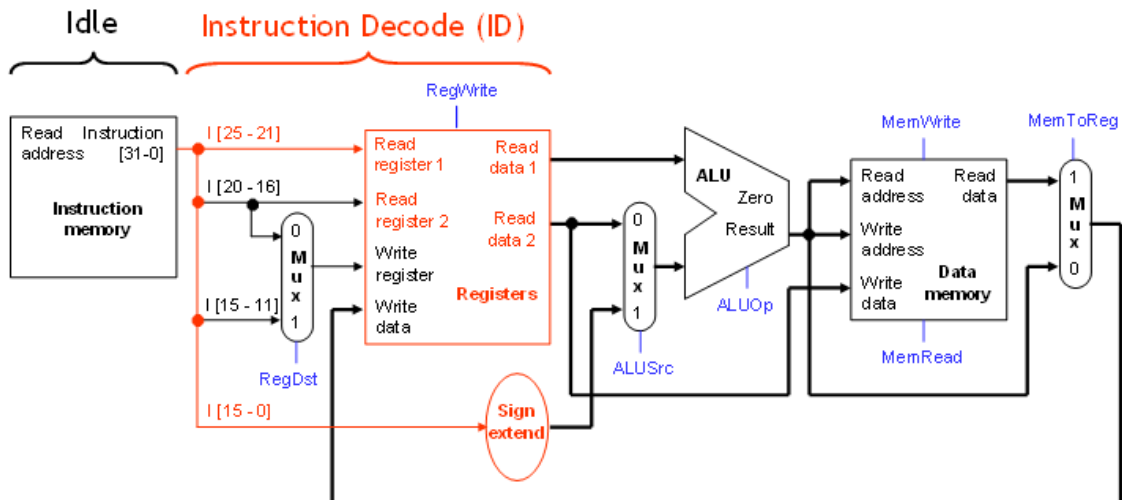


A bunch of lazy functional units

- 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.

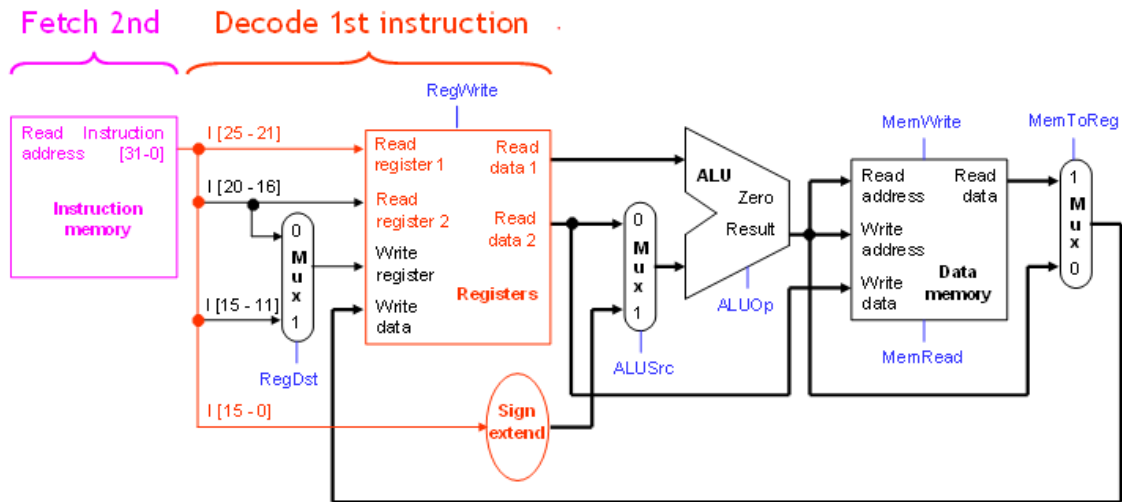
Putting those slackers to work

- 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...



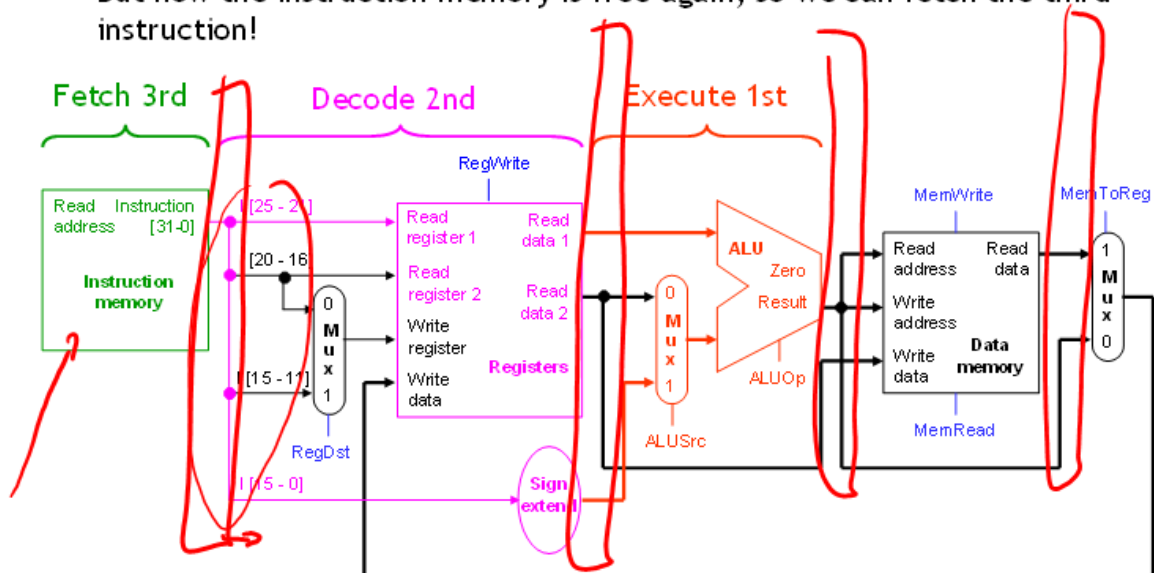
Decoding and fetching together

- 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!



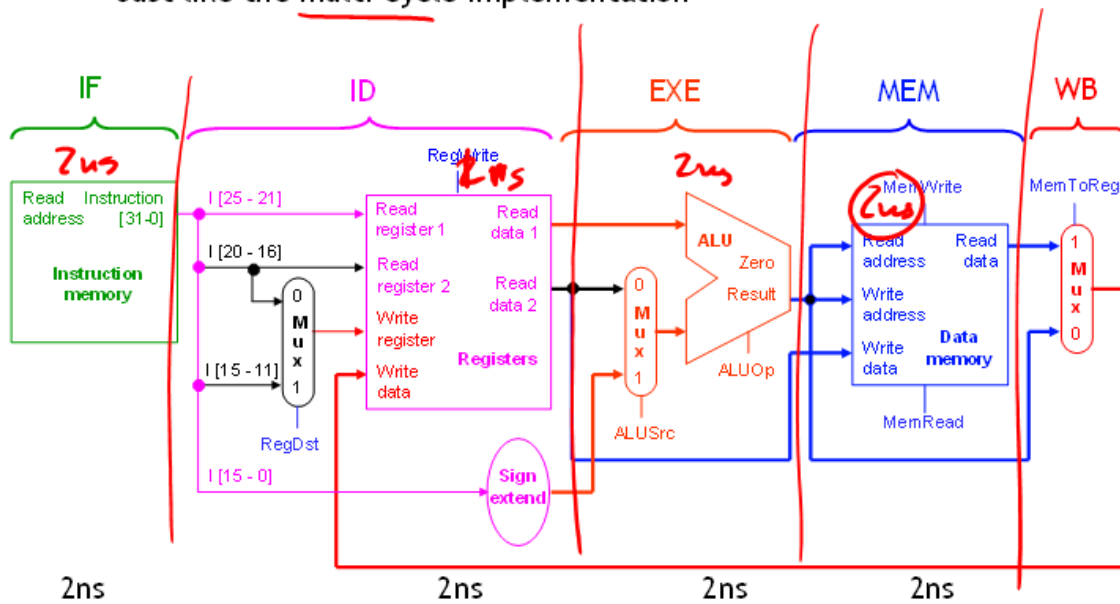
Making Pipelining Work

- 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.



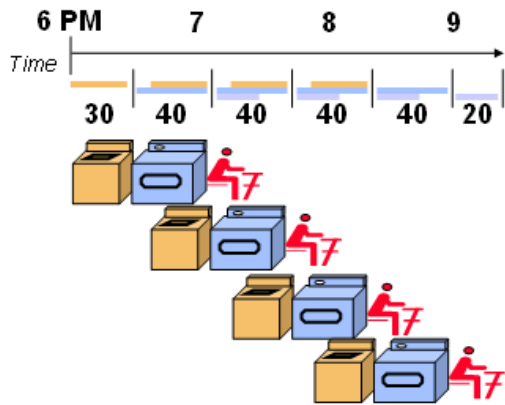
Break datapath into 5 stages

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



Pipelining Loads

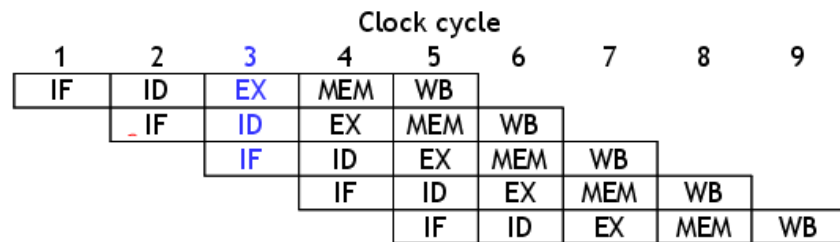
	Clock cycle								
	1	2	3	4	5	6	7	8	9
lw \$t0, 4(\$sp)	IF	ID	EX	MEM	WB				
lw \$t1, 8(\$sp)		IF	ID	EX	MEM	WB			
lw \$t2, 12(\$sp)			IF	ID	EX	MEM	WB		
lw \$t3, 16(\$sp)				IF	ID	EX	MEM	WB	
lw \$t4, 20(\$sp)					IF	ID	EX	MEM	WB



A pipeline diagram

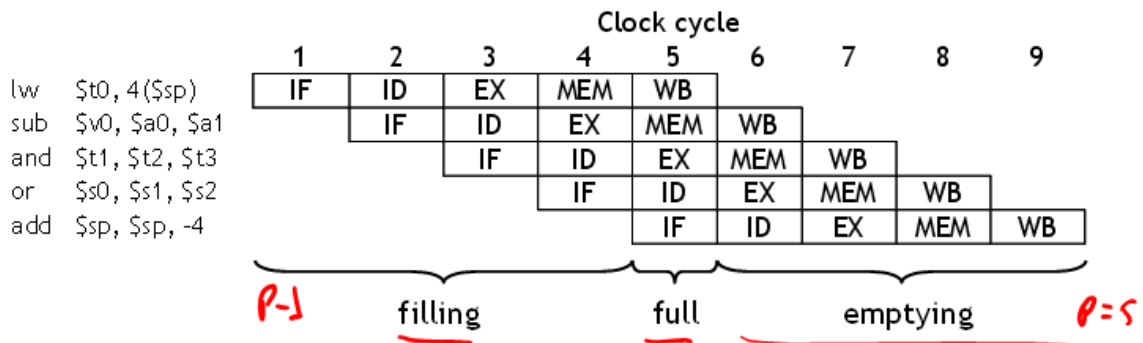
t = 10 cycles

1 lw \$t0, 4(\$sp)
 2 sub \$v0, \$a0, \$a1
 3 and \$t1, \$t2, \$t3
 4 or \$s0, \$s1, \$s2
 5 add \$sp, \$sp, -4



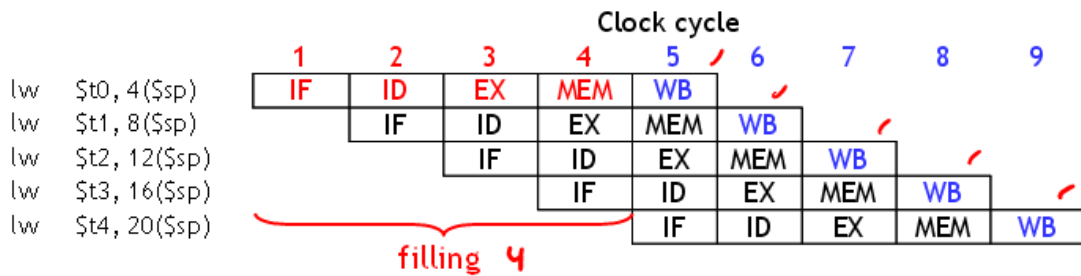
- A **pipeline diagram** shows the execution of a series of instructions.
 - The instruction sequence is shown vertically, from top to bottom.
 - Clock cycles are shown horizontally, from left to right.
 - Each instruction is divided into its component stages. (We show five stages for every instruction, 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



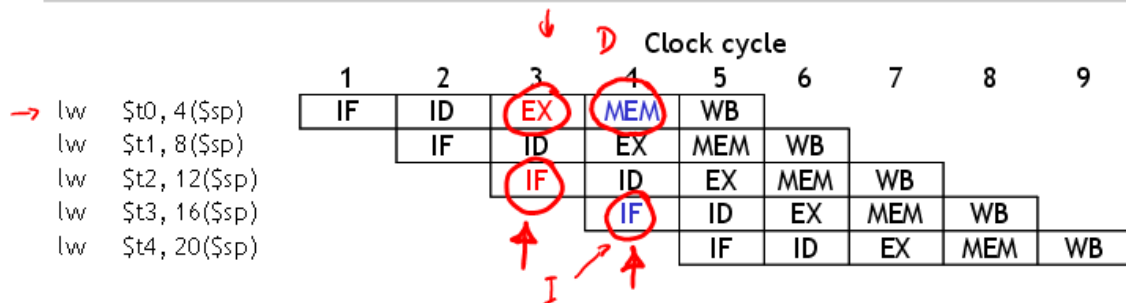
- The **pipeline depth** is the number of stages—in this case, five.
- In the first four cycles here, the pipeline is **filling**, since there are unused functional units.
- In cycle 5, the pipeline is **full**. Five instructions are being executed simultaneously, so all hardware units are in use.
-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

Pipelining Performance



- Execution time on ideal pipeline:
 - time to fill the pipeline + one cycle per instruction
 - N instructions -> 4 cycles + N cycles or $(2N + 8) ns$ for 2ns clock period
 $(N+4) \times 2ns = 2N+8ns$
- Compare with other implementations:
 - Single Cycle: N cycles or $8N ns$ for 8ns clock period
 - Multicycle: CPI * N cycles or $\sim 8N ns$ for 2ns clock period and CPI = ~ 4
- How much faster is pipelining for N=1000 ?
 - $\rightarrow \frac{t_{old}}{t_{new}} = \frac{8000}{2008} \approx 4$

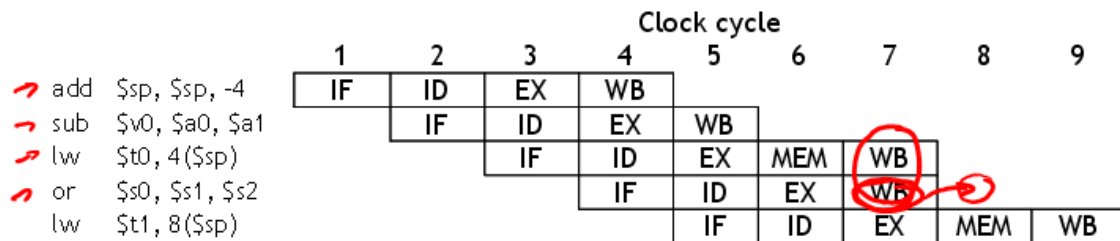
Pipeline Datapath: Resource Requirements



- 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 this mean for our HW?

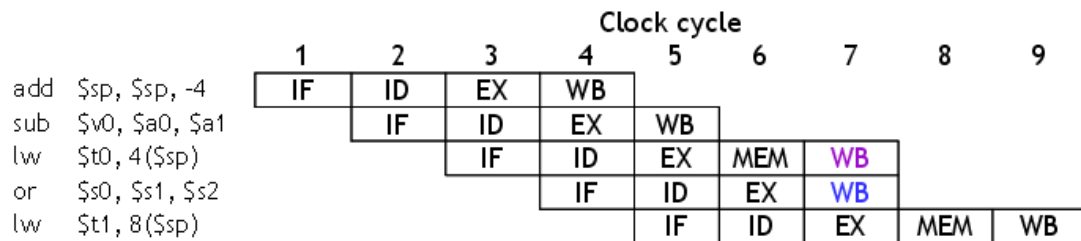
Pipelining other instruction types

- 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.
See the problem if:
 - Load uses Register File's Write Port during its **5th** stage
 - R-type uses Register File's Write Port during its **4th** stage

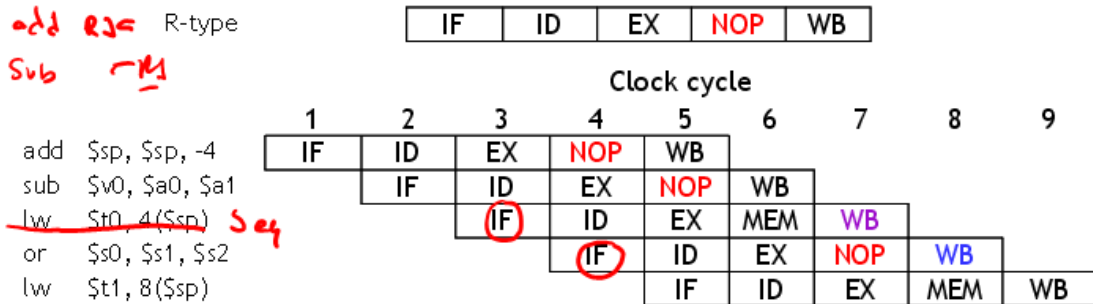


A solution: Insert NOP stages

- 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

sw A ←
lw ← A



- 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 its own functional units
- Next time we'll see the datapath and control, and walk through an example execution.