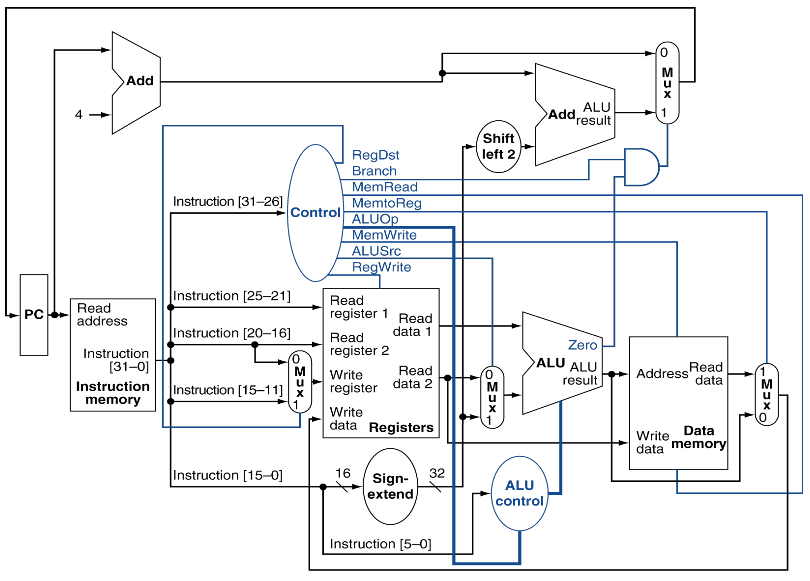
CSCE 5610
Computer System Architecture

Instruction Level Parallelism

Single-Cycle implementation



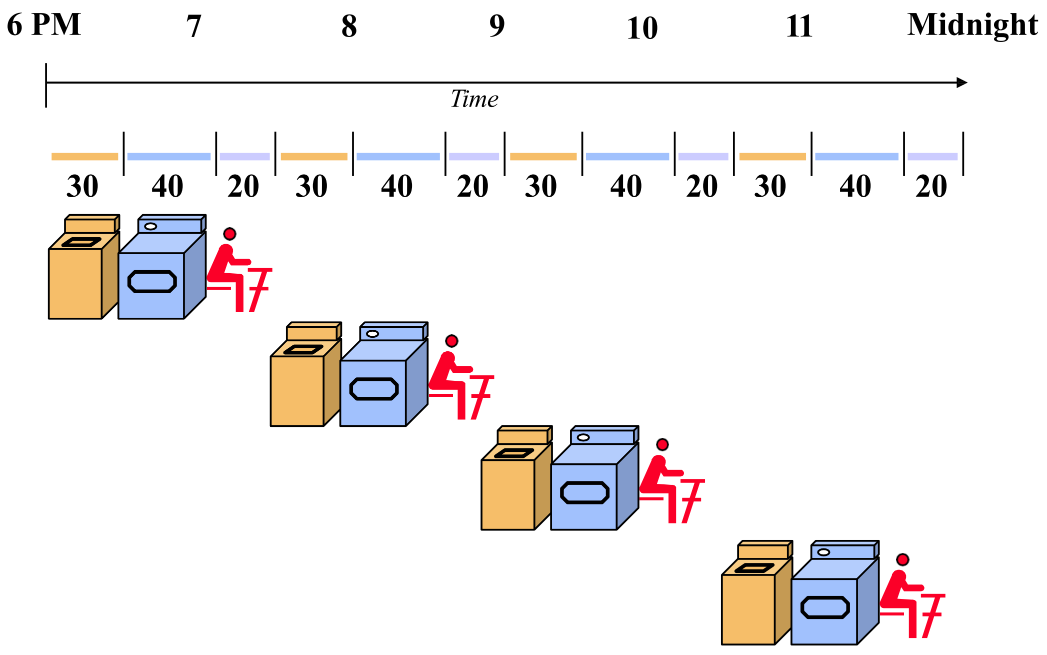
Why Single-Cycle implementation is not used today?

- Requires the same length for every instruction
- The clock cycle is determined by the longest possible path (worst case scenario!)
  - The load instruction (lw) that uses five functional units in series:
    Inst. Mem → Reg. File → ALU→ Data Mem. → Reg. File

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?

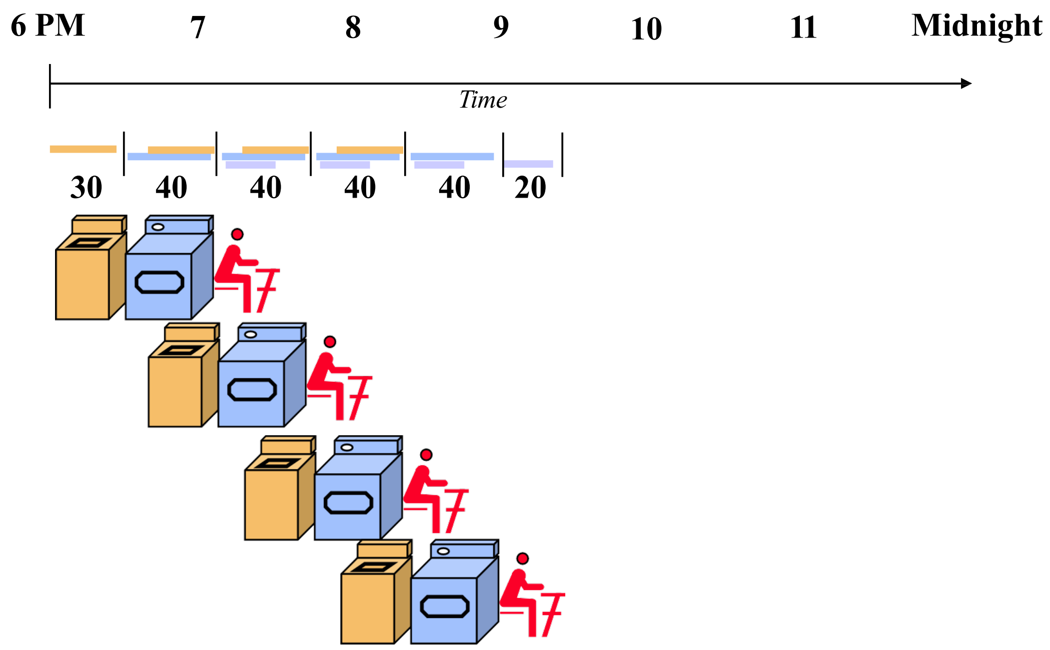
The Slow Way



- 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

Instruction Execution Review

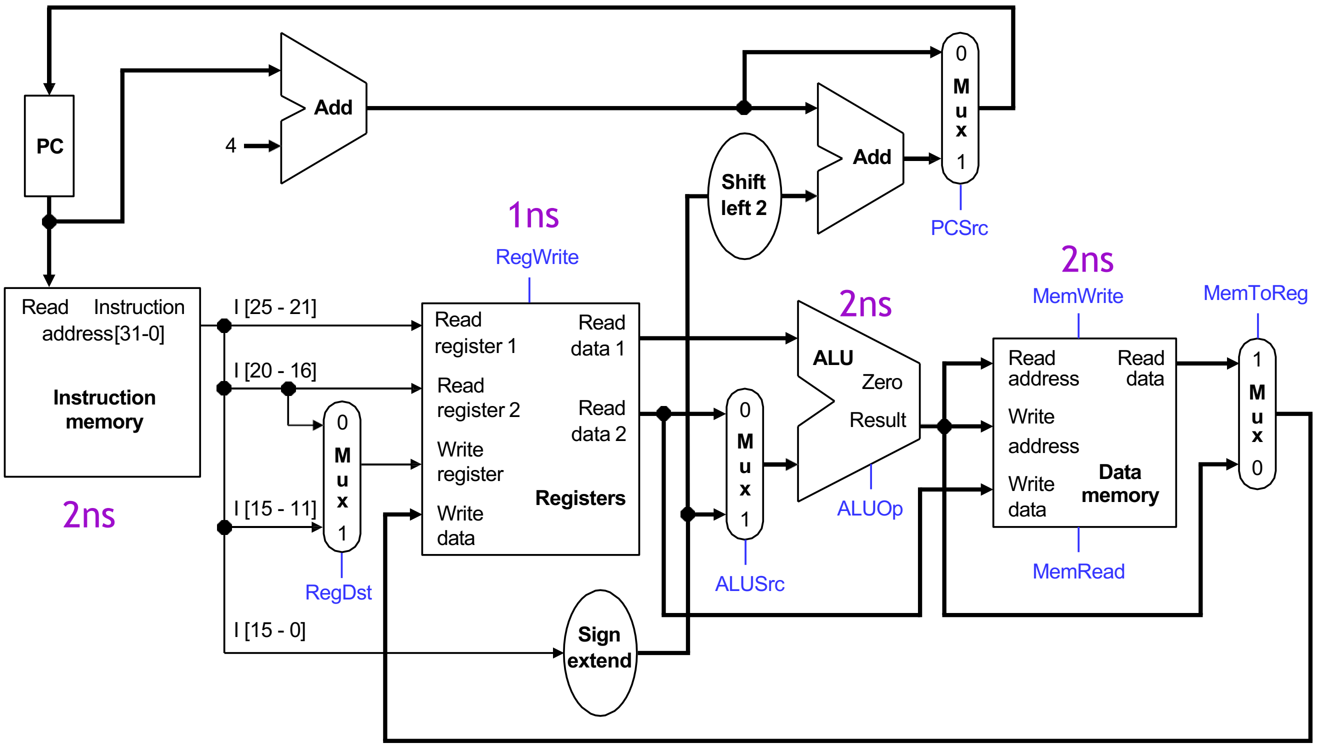
- Executing a MIPS instruction can take up to five steps.

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



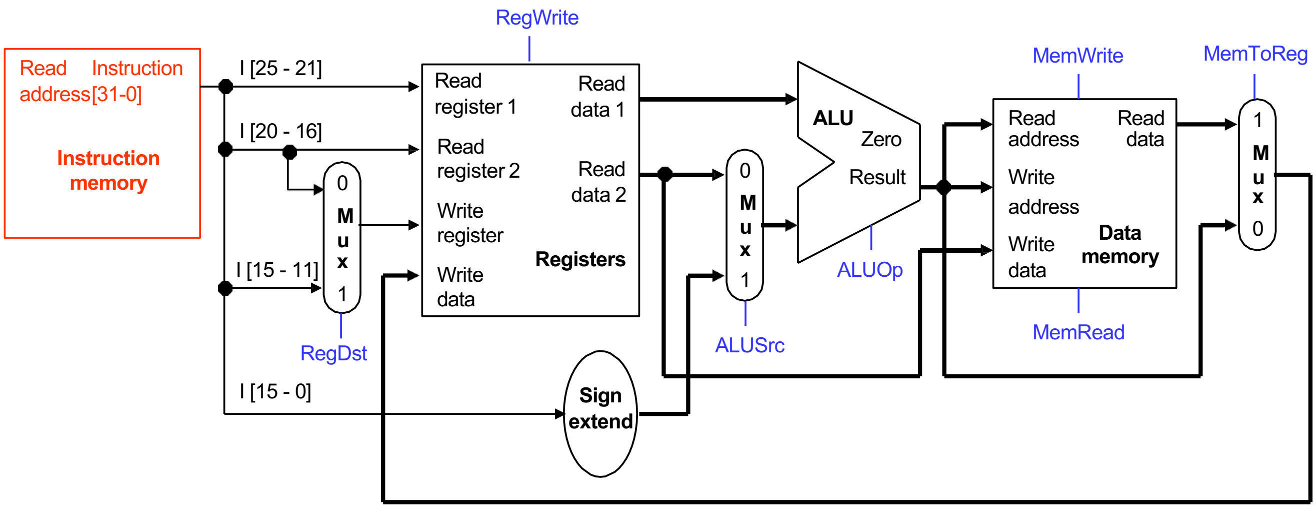
- How long does it take to execute each instruction?

Single-cycle Review

- All five execution steps occur in one clock cycle.
- 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.

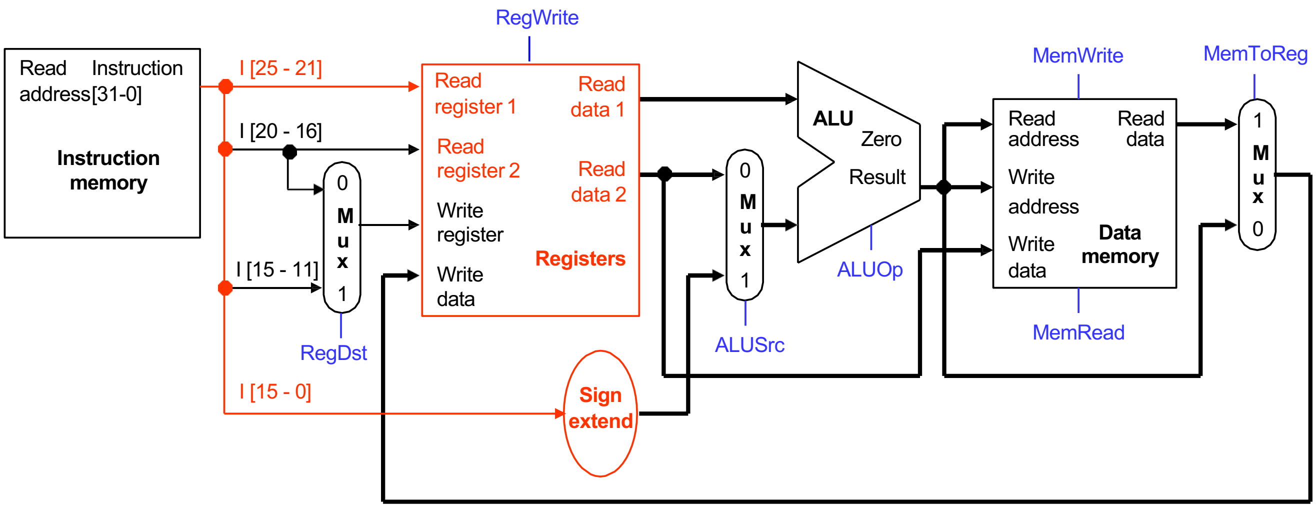
Example: Instruction Fetch (IF)

- 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.



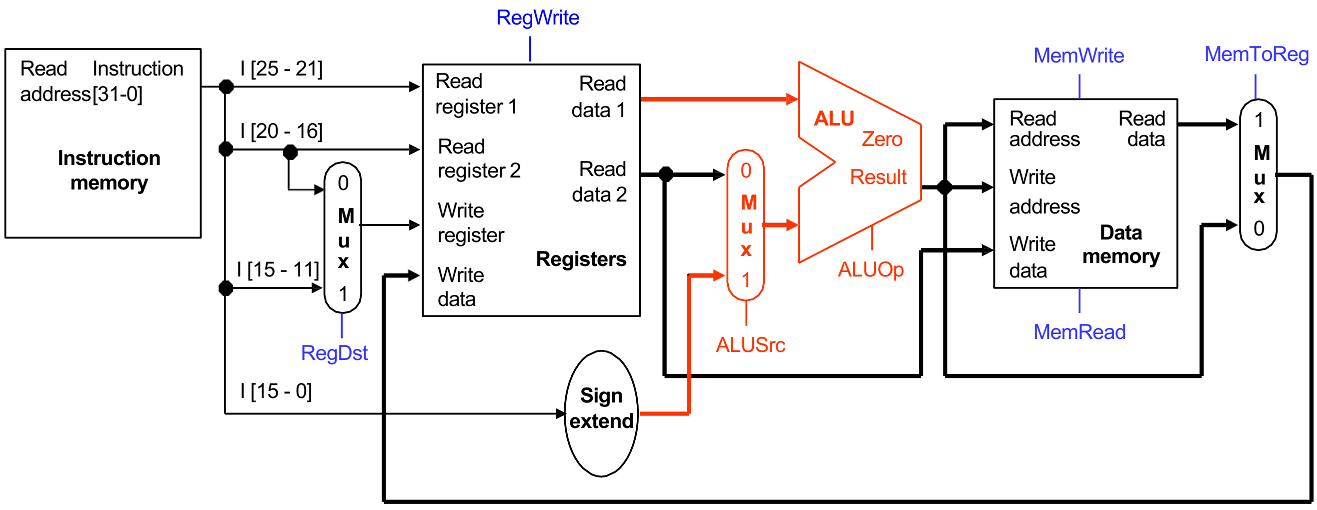
Instruction Decode (ID)

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



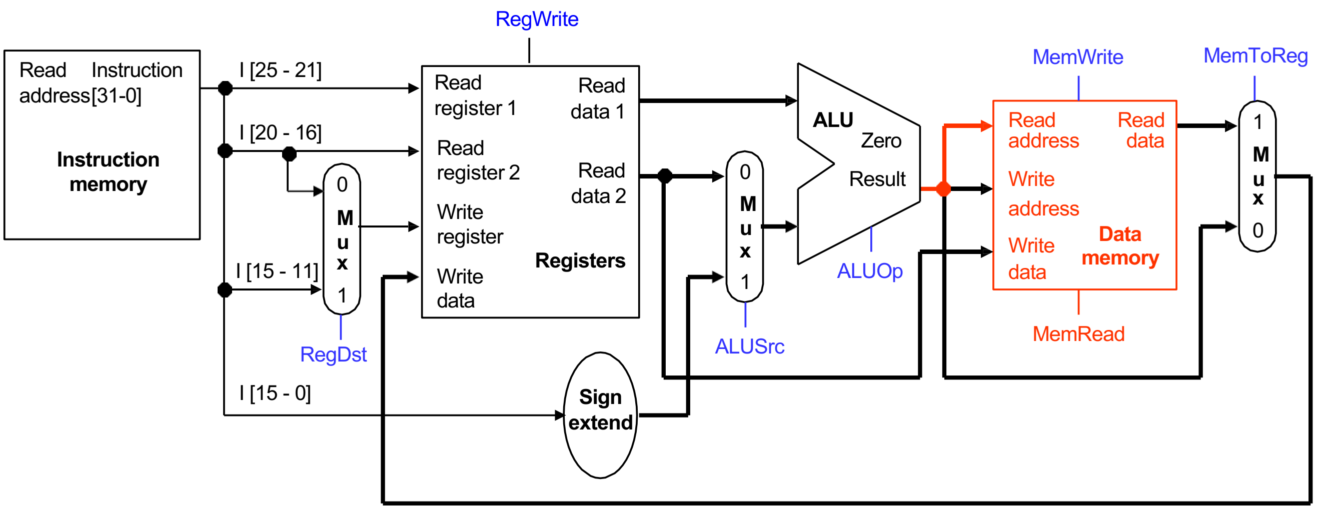
Execute (EX)

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



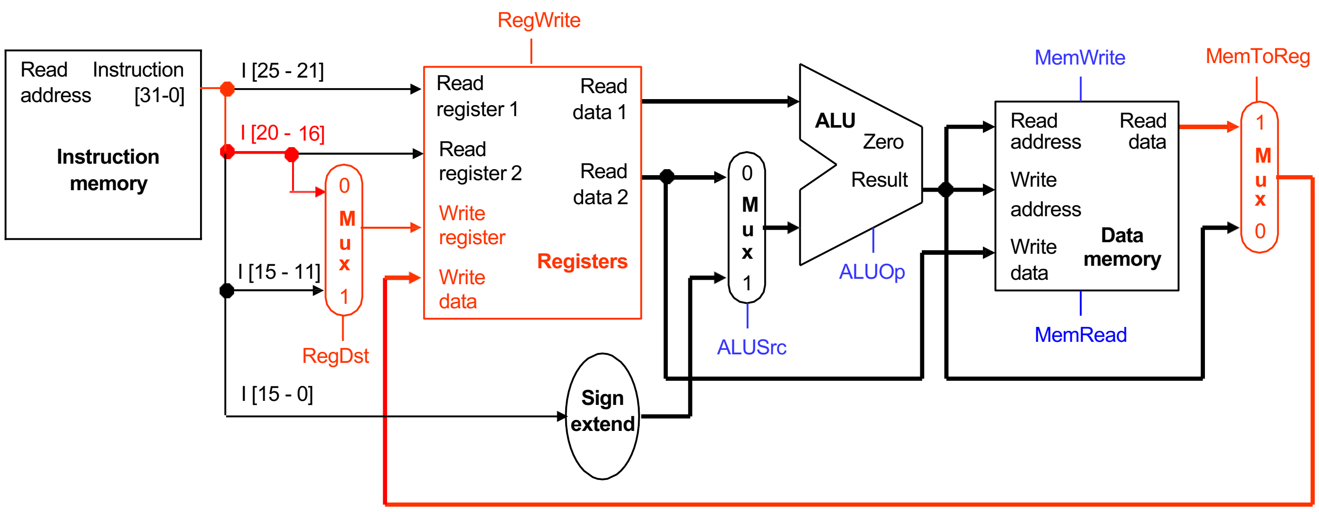
Memory (MEM)

- 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.

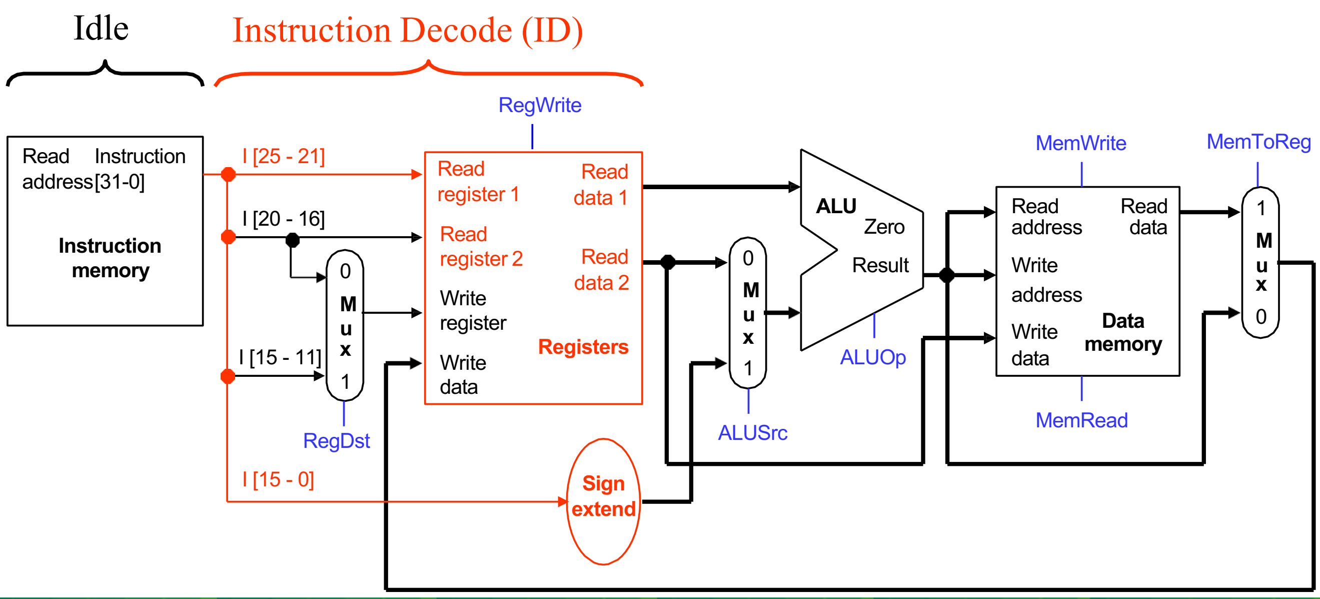


A Bunch of Lazy Functional Units

- 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.

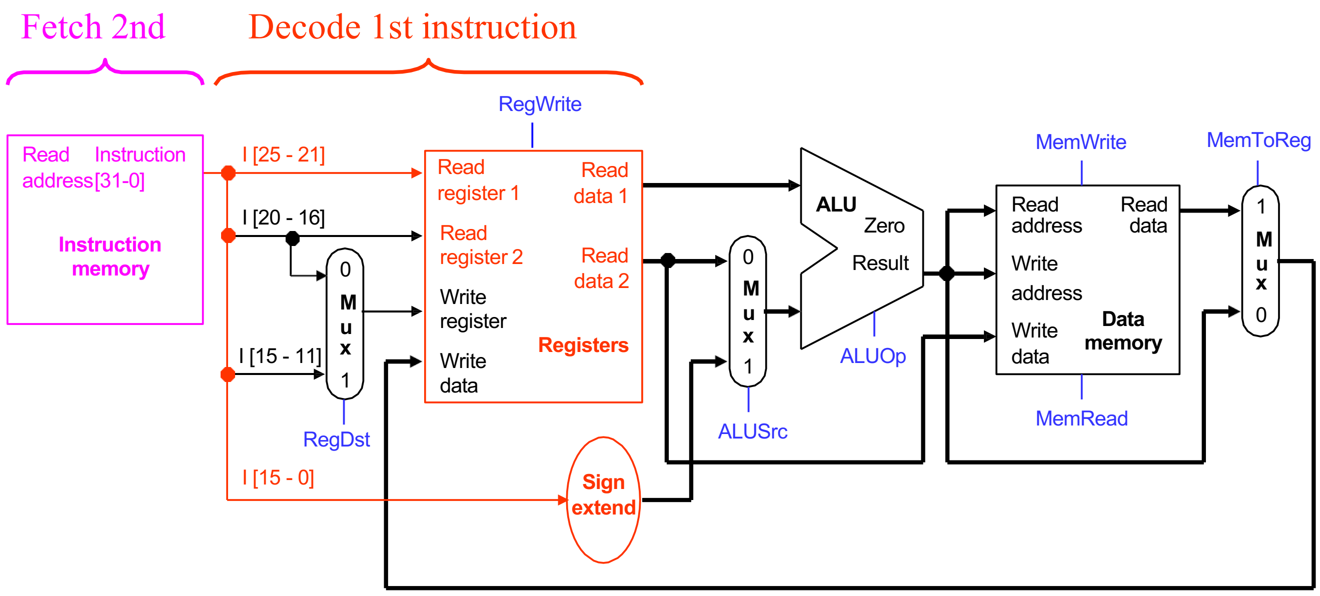
Putting Those Slackers to Work

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



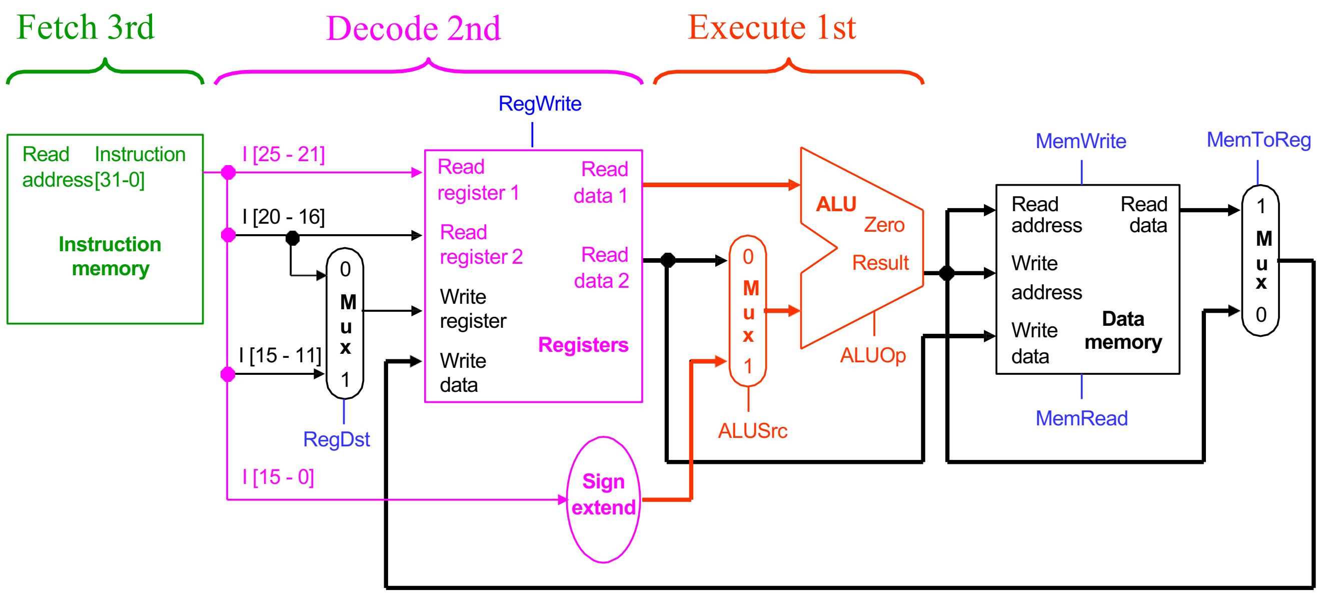
Decoding and Fetching Together

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



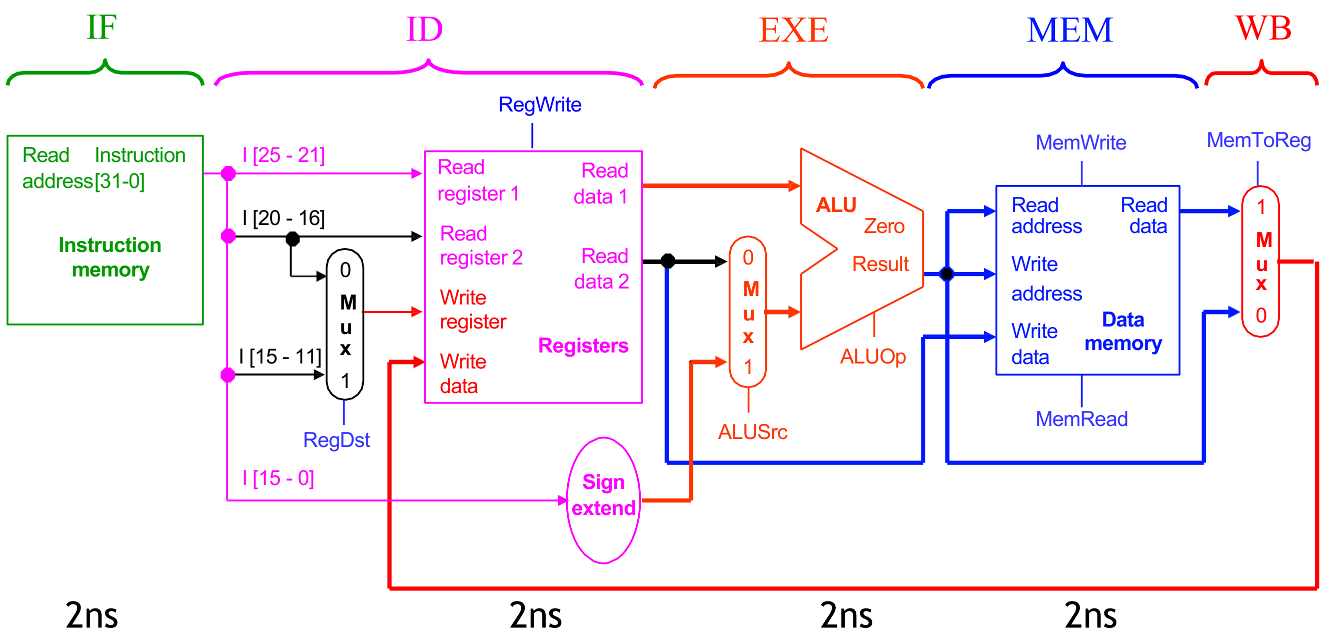
Executing, Decoding and Fetching

- 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!



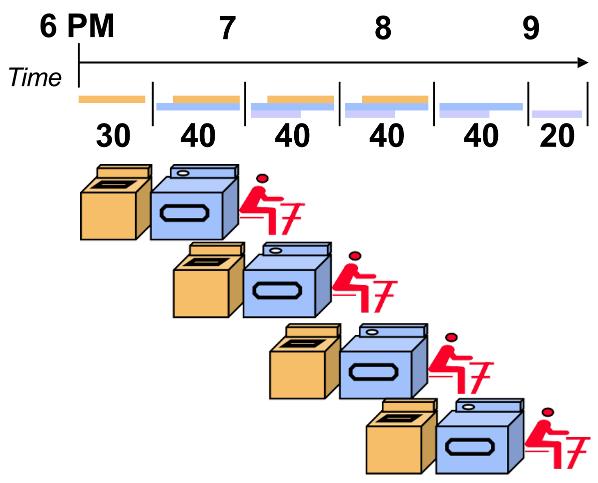
Break Datapath into 5 Stages

- Each stage has its own functional units.
- Each stage can execute in 2ns



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

	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

- 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

		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

filling full emptying

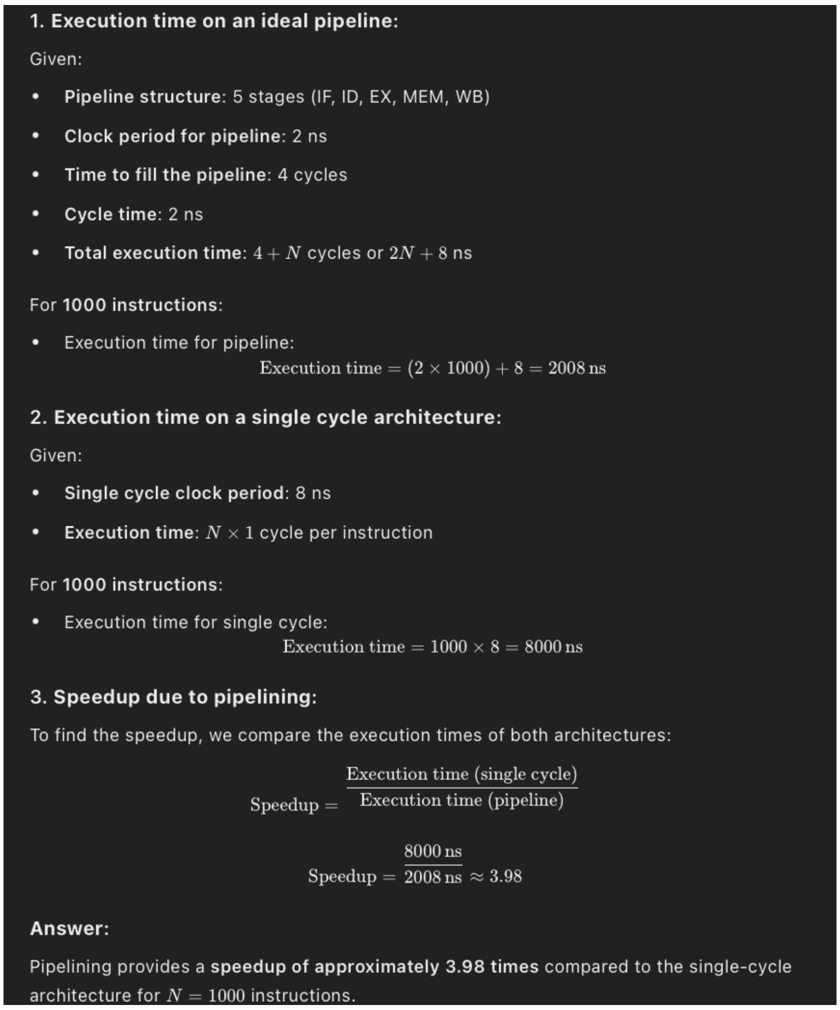
- The pipeline depth is the number of stages—in this case, five.
- In the first four cycles here, the pipeline is filling, since there are unused functional units.
- In cycle 5, the pipeline is full. Five instructions are being executed simultaneously, so all hardware units are in use.
- In cycles 6-9,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

Pipelining Performance

		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

filling

- 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
- Compare with other implementations:
  - —Single Cycle: N cycles or 8N ns for 8ns clock period
- How much faster is pipelining for N=1000 ?



1. Execution time on an ideal pipeline:

Given:

- **Pipeline structure**: 5 stages (IF, ID, EX, MEM, WB)
- **Clock period for pipeline**: 2 ns
- **Time to fill the pipeline**: 4 cycles
- **Cycle time**: 2 ns
- **Total execution time**: $4 + N$ cycles or $2N + 8$ ns

For **1000 instructions**:

- Execution time for pipeline:

$$\text{Execution time} = (2 \times 1000) + 8 = 2008\,\text{ns}$$

2. Execution time on a single cycle architecture:

Given:

- **Single cycle clock period**: 8 ns
- **Execution time**: $N \times 1$ cycle per instruction

For **1000 instructions**:

- Execution time for single cycle:

$$\text{Execution time} = 1000 \times 8 = 8000\,\text{ns}$$

3. Speedup due to pipelining:

To find the speedup, we compare the execution times of both architectures:

$$\text{Speedup} = \frac{\text{Execution time (single cycle)}}{\text{Execution time (pipeline)}}$$

$$\text{Speedup} = \frac{8000\,\text{ns}}{2008\,\text{ns}} \approx 3.98$$

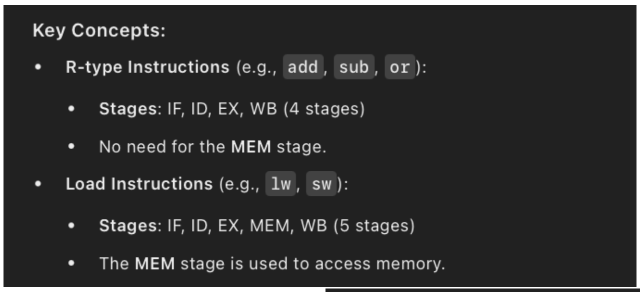
Answer:

Pipelining provides a **speedup of approximately 3.98 times** compared to the single-cycle architecture for $N = 1000$ instructions.

Pipelining Other Instruction Types

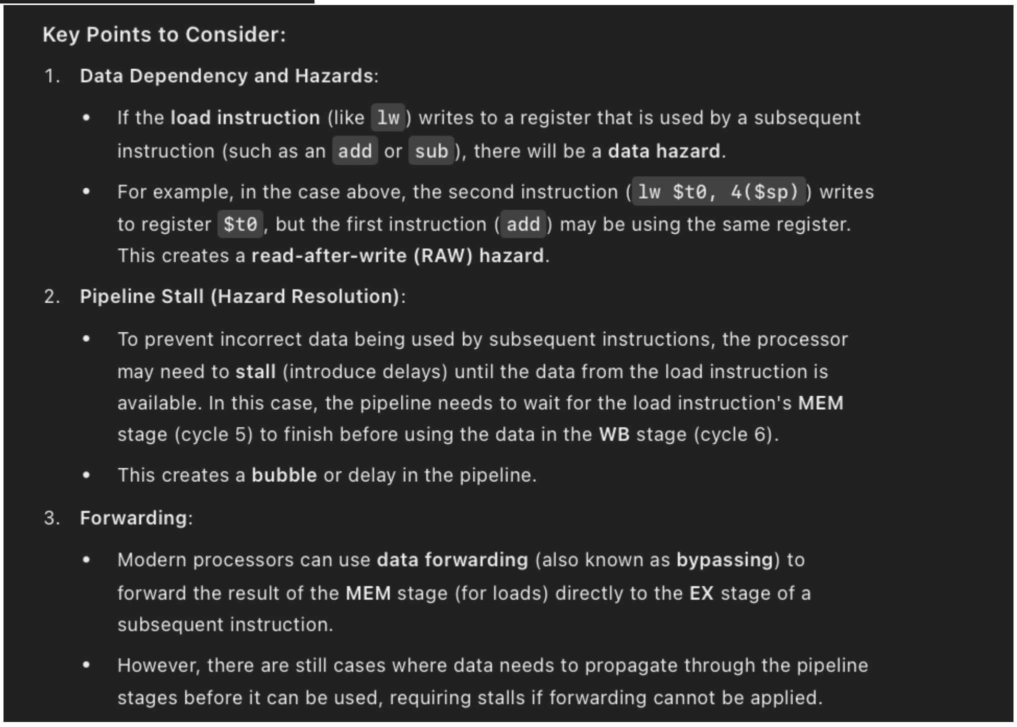
- R-type instructions only require 4 stages: IF, ID, EX, and WB
  — We don't need the MEM stage
- What happens if we try to pipeline loads with R-type instructions?

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



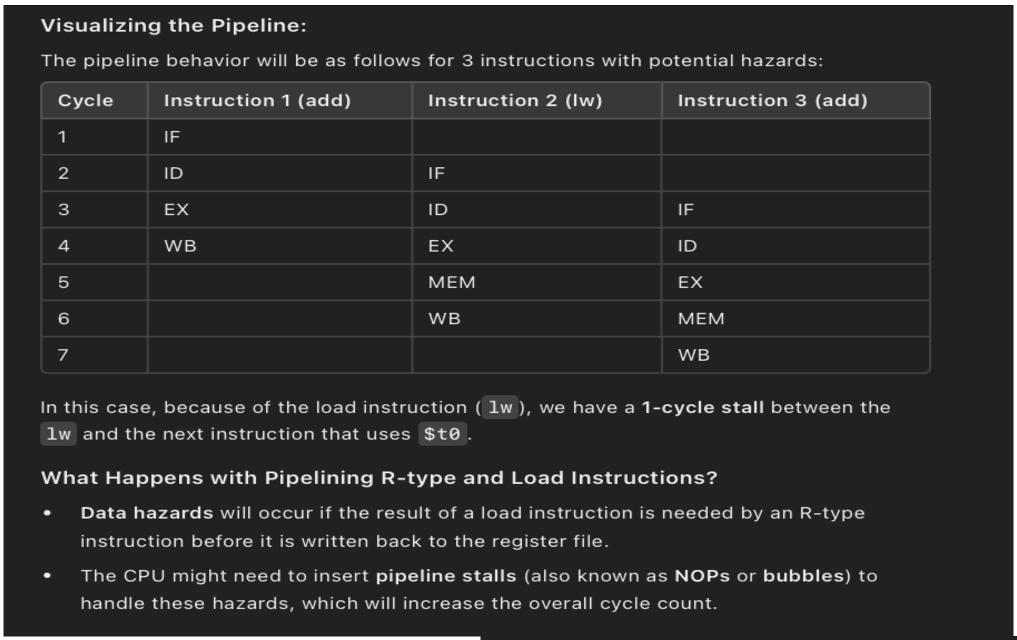
Key Concepts:

- **R-type Instructions** (e.g., `add`, `sub`, `or`):
  - **Stages**: IF, ID, EX, WB (4 stages)
  - No need for the **MEM** stage.
- **Load Instructions** (e.g., `lw`, `sw`):
  - **Stages**: IF, ID, EX, MEM, WB (5 stages)
  - The **MEM** stage is used to access memory.



Key Points to Consider:

1. **Data Dependency and Hazards**:
   - If the **load instruction** (like `lw`) writes to a register that is used by a subsequent instruction (such as an `add` or `sub`), there will be a **data hazard**.
   - For example, in the case above, the second instruction (`lw $t0, 4($sp)`) writes to register `$t0`, but the first instruction (`add`) may be using the same register. This creates a **read-after-write (RAW) hazard**.
2. **Pipeline Stall (Hazard Resolution)**:
   - To prevent incorrect data being used by subsequent instructions, the processor may need to **stall** (introduce delays) until the data from the load instruction is available. In this case, the pipeline needs to wait for the load instruction's **MEM** stage (cycle 5) to finish before using the data in the **WB** stage (cycle 6).
   - This creates a **bubble** or delay in the pipeline.
3. **Forwarding**:
   - Modern processors can use **data forwarding** (also known as **bypassing**) to forward the result of the **MEM** stage (for loads) directly to the **EX** stage of a subsequent instruction.
   - However, there are still cases where data needs to propagate through the pipeline stages before it can be used, requiring stalls if forwarding cannot be applied.



Visualizing the Pipeline:

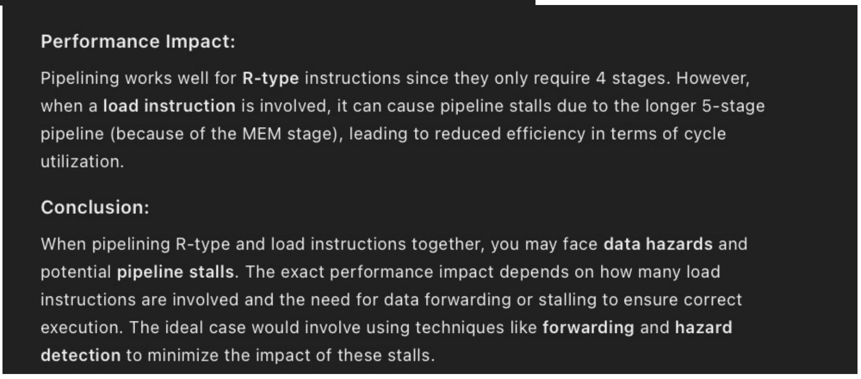
The pipeline behavior will be as follows for 3 instructions with potential hazards:

Cycle	Instruction 1 (add)	Instruction 2 (lw)	Instruction 3 (add)
1	IF		
2	ID	IF	
3	EX	ID	IF
4	WB	EX	ID
5		MEM	EX
6		WB	MEM
7			WB

In this case, because of the load instruction (`lw`), we have a **1-cycle stall** between the `lw` and the next instruction that uses `$t0`.

What Happens with Pipelining R-type and Load Instructions?

- **Data hazards** will occur if the result of a load instruction is needed by an R-type instruction before it is written back to the register file.
- The CPU might need to insert **pipeline stalls** (also known as **NOPs** or **bubbles**) to handle these hazards, which will increase the overall cycle count.



Performance Impact:

Pipelining works well for **R-type** instructions since they only require 4 stages. However, when a **load instruction** is involved, it can cause pipeline stalls due to the longer 5-stage pipeline (because of the MEM stage), leading to reduced efficiency in terms of cycle utilization.

Conclusion:

When pipelining R-type and load instructions together, you may face **data hazards** and potential **pipeline stalls**. The exact performance impact depends on how many load instructions are involved and the need for data forwarding or stalling to ensure correct execution. The ideal case would involve using techniques like **forwarding** and **hazard detection** to minimize the impact of these stalls.

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

Clock cycle

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

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

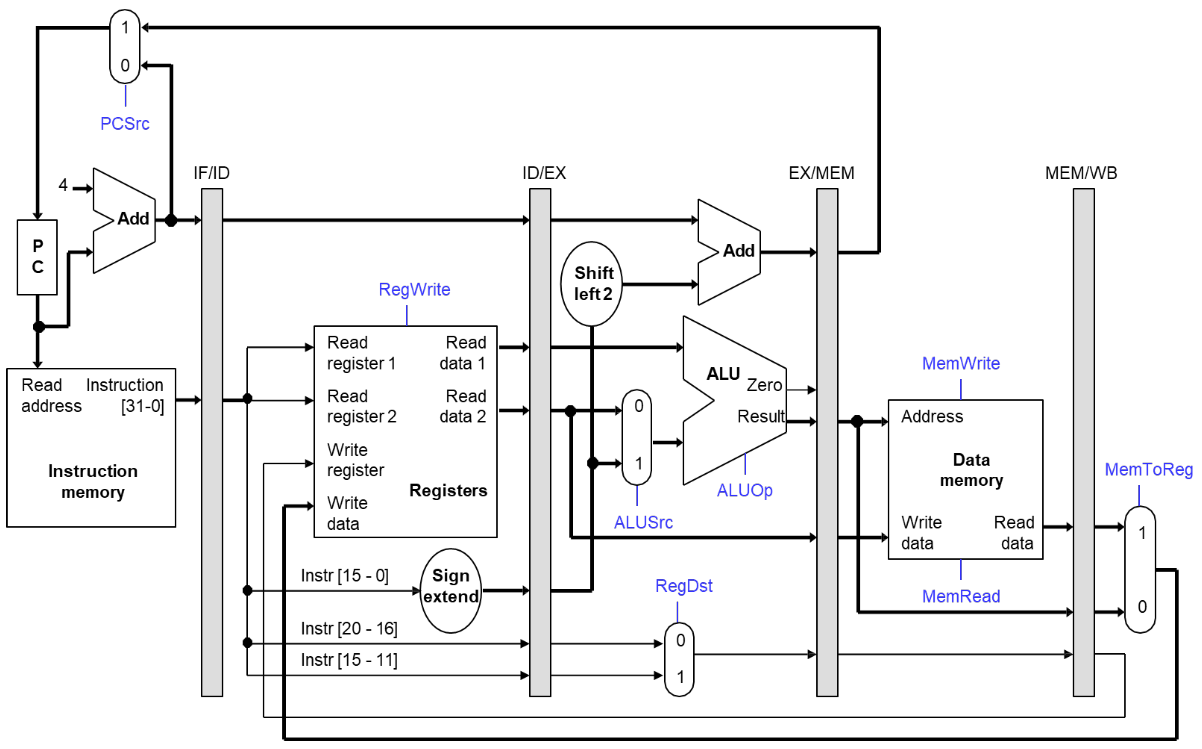
Pipeline Registers

- 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.

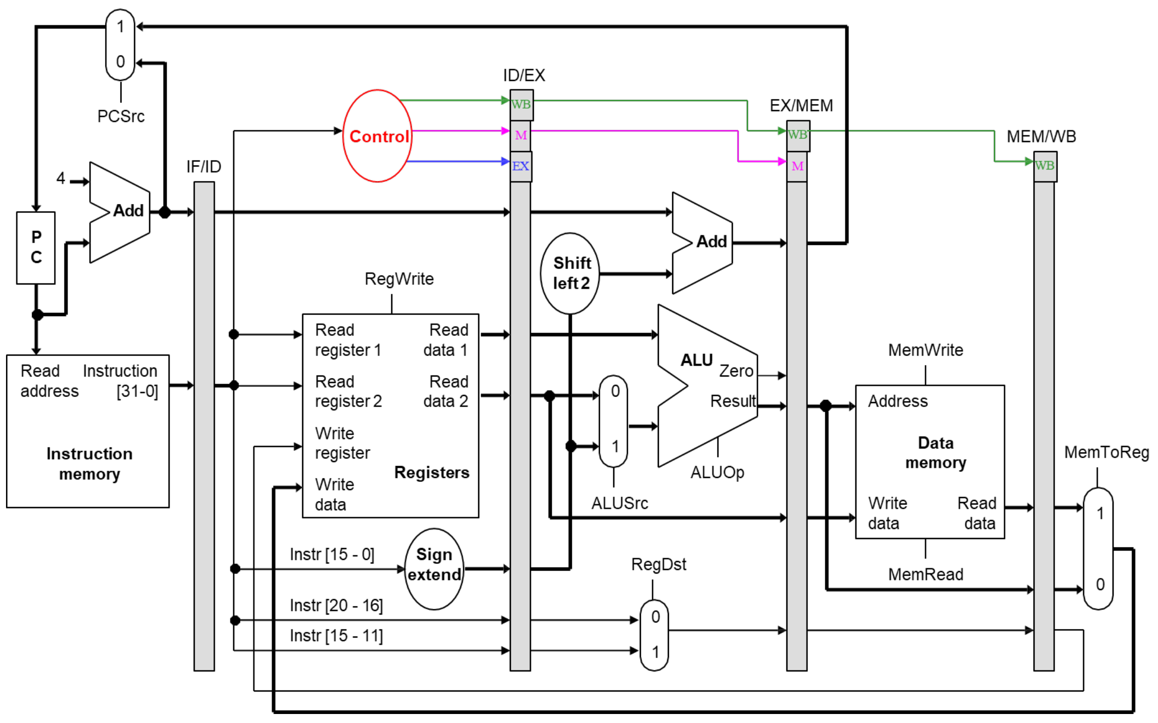
Pipelined Datapath



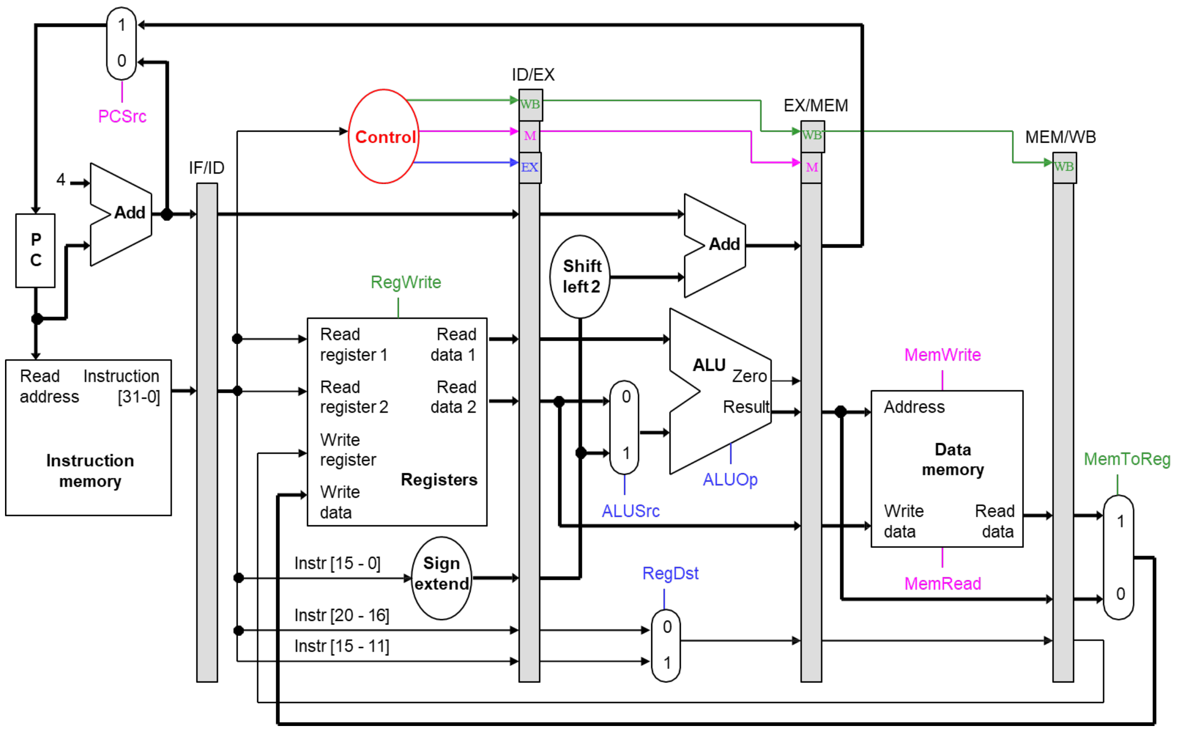
What about Control Signals?

- The control signals are generated in the same way as in the single-cycle processor—after an instruction is fetched, the processor decodes it and produces the appropriate control values.
- Control signals can be categorized by the pipeline stage that uses them.

Pipelined Datapath and Control



Pipelined Datapath and Control



An Example Execution Sequence

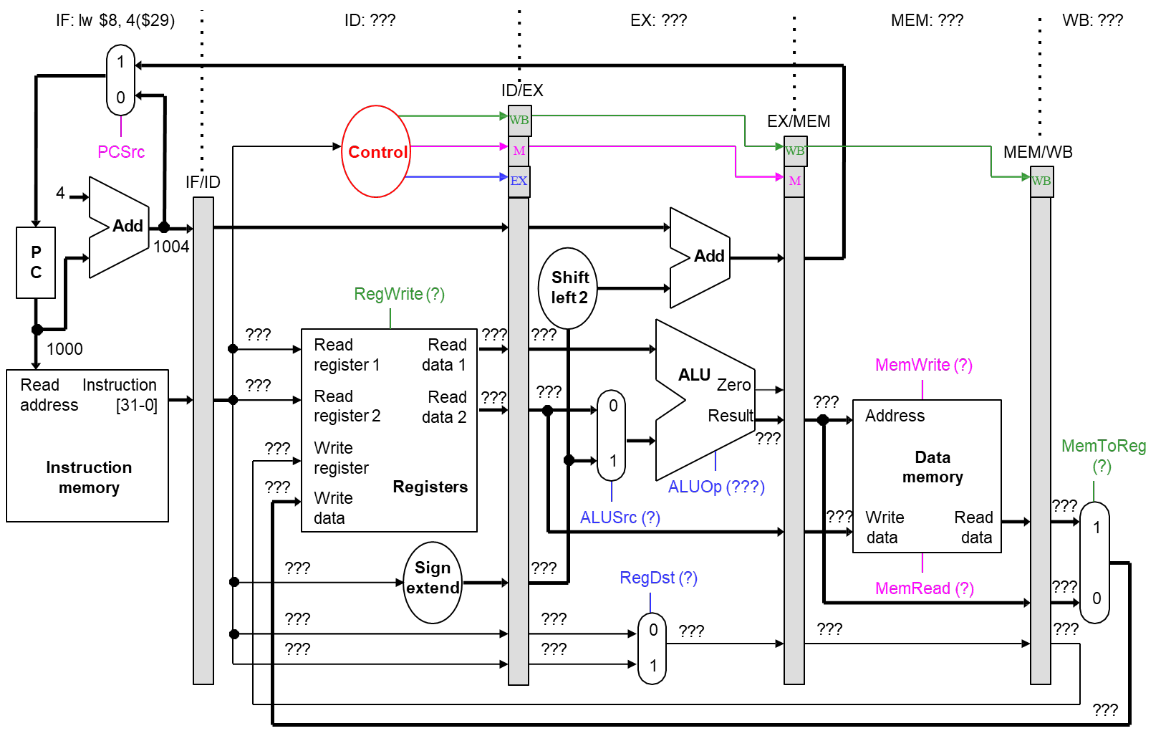
- Here's a sample sequence of instructions to execute.

addresses in decimal

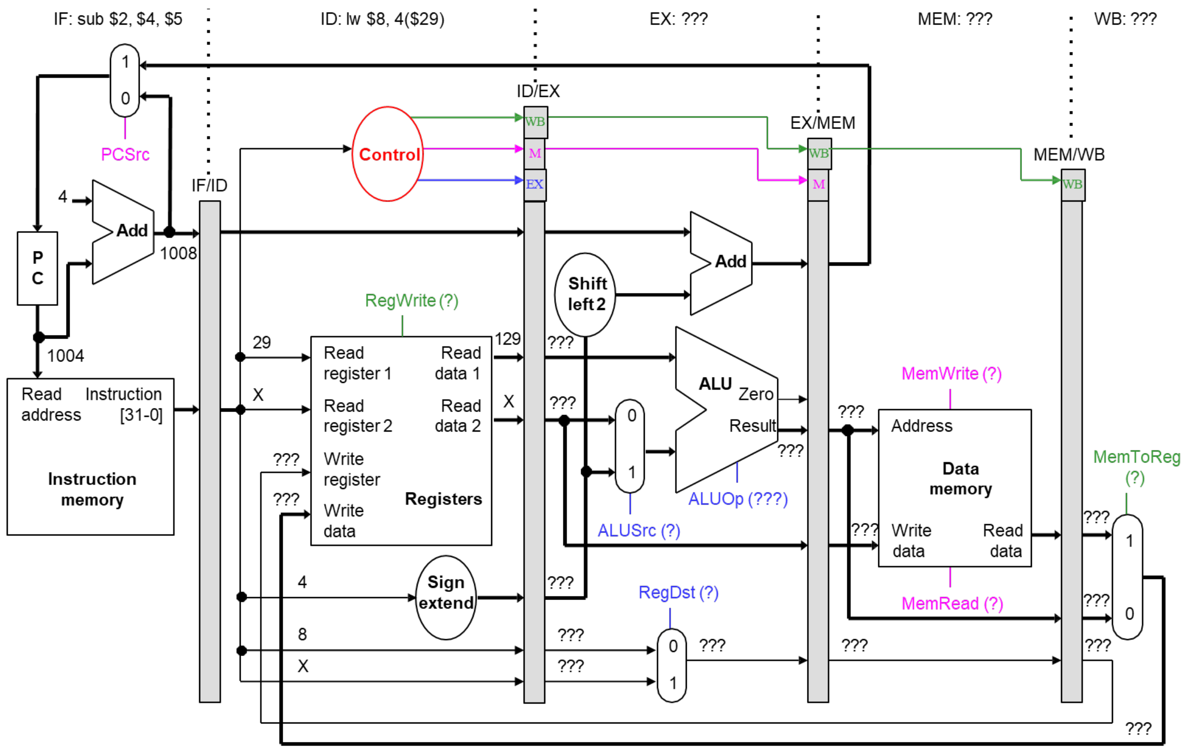
```
1000:  lw   $8, 4($29)
1004:  sub  $2, $4, $5
1008:  and  $9, $10, $11
1012:  or   $16, $17, $18
1016:  add  $13, $14, $0
```

- We'll make some assumptions, just so we can show actual data values.
  - Each register contains its number plus 100. For instance, register $8 contains 108, register $29 contains 129, and so forth.
  - Every data memory location contains 99.
- Our pipeline diagrams will follow some conventions.
  - An X indicates values that aren't important, like the constant field of an R-type instruction.
  - Question marks ??? indicate values we don't know, usually resulting from instructions coming before and after the ones in our example.

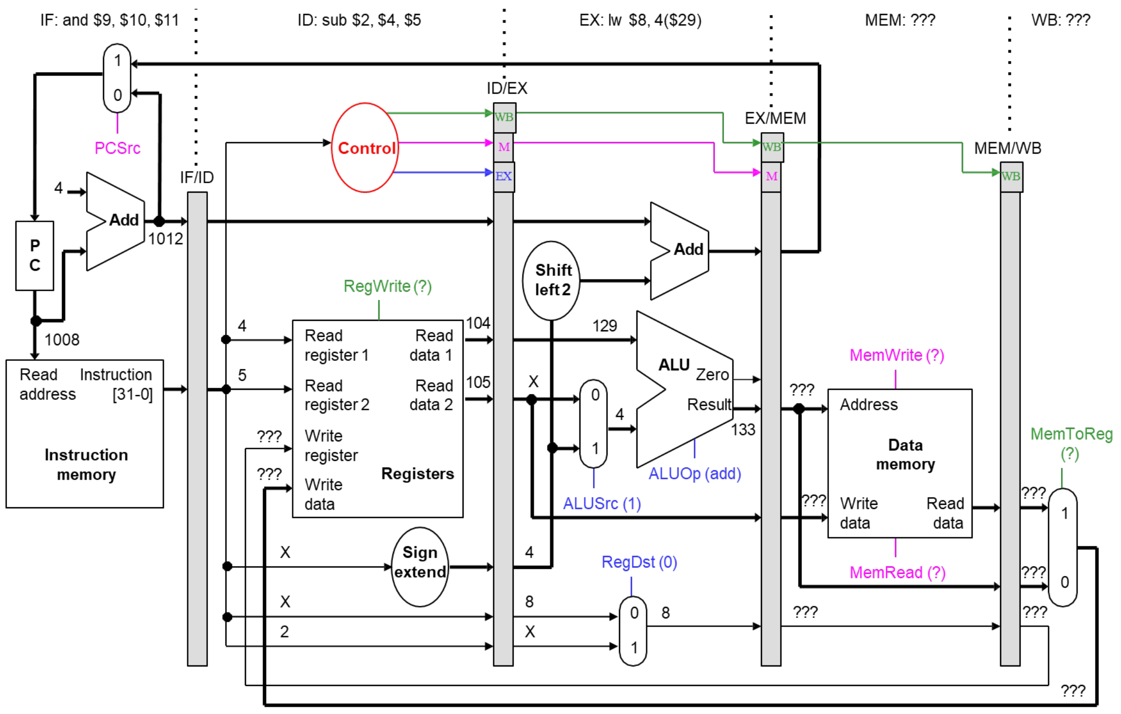
Cycle 1 (Filling)



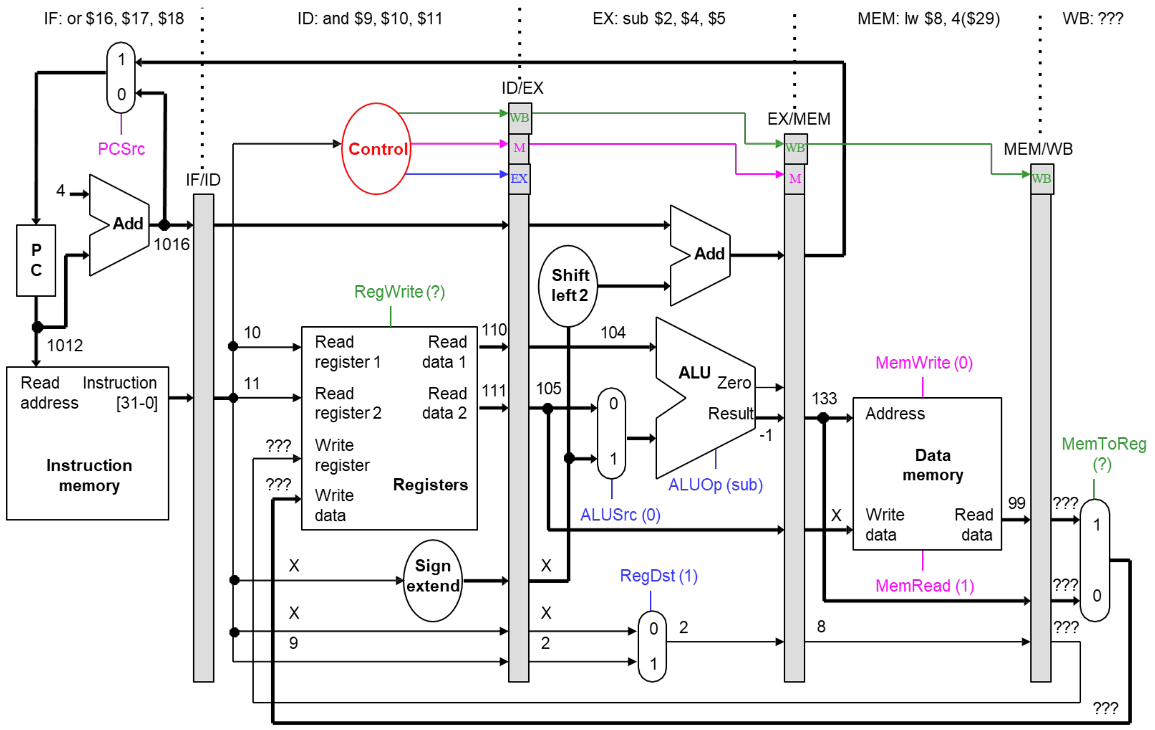
Cycle 2



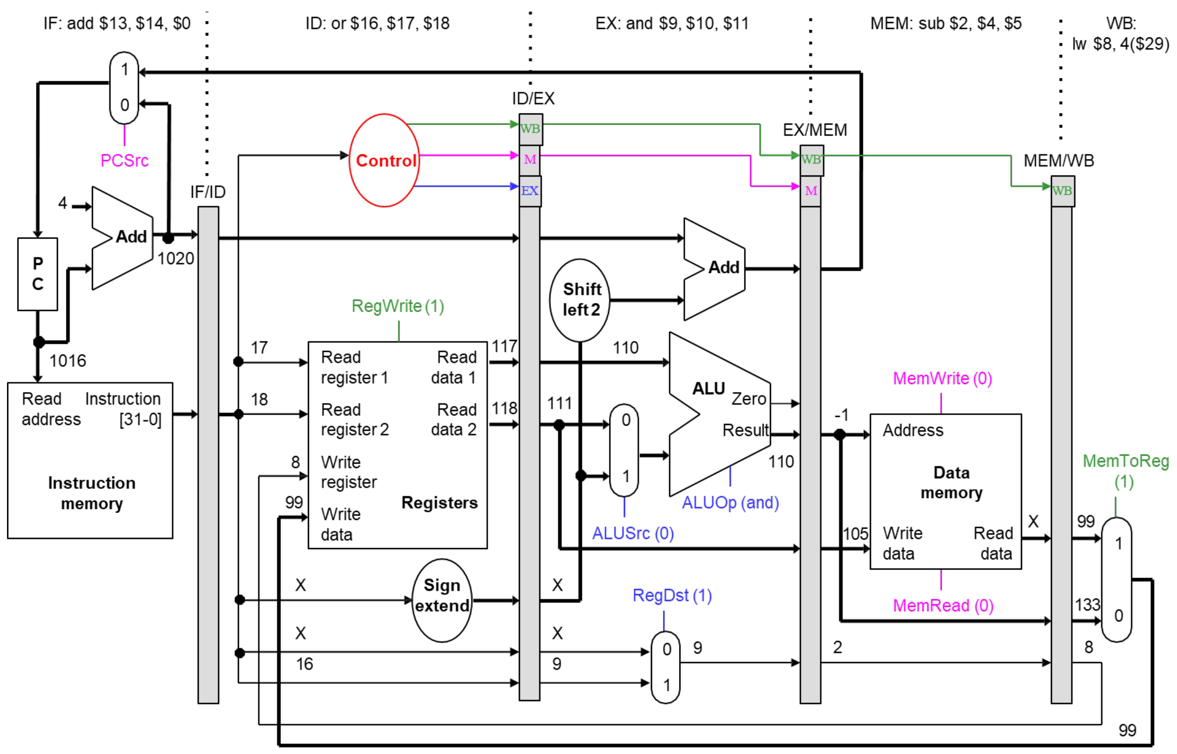
Cycle 3



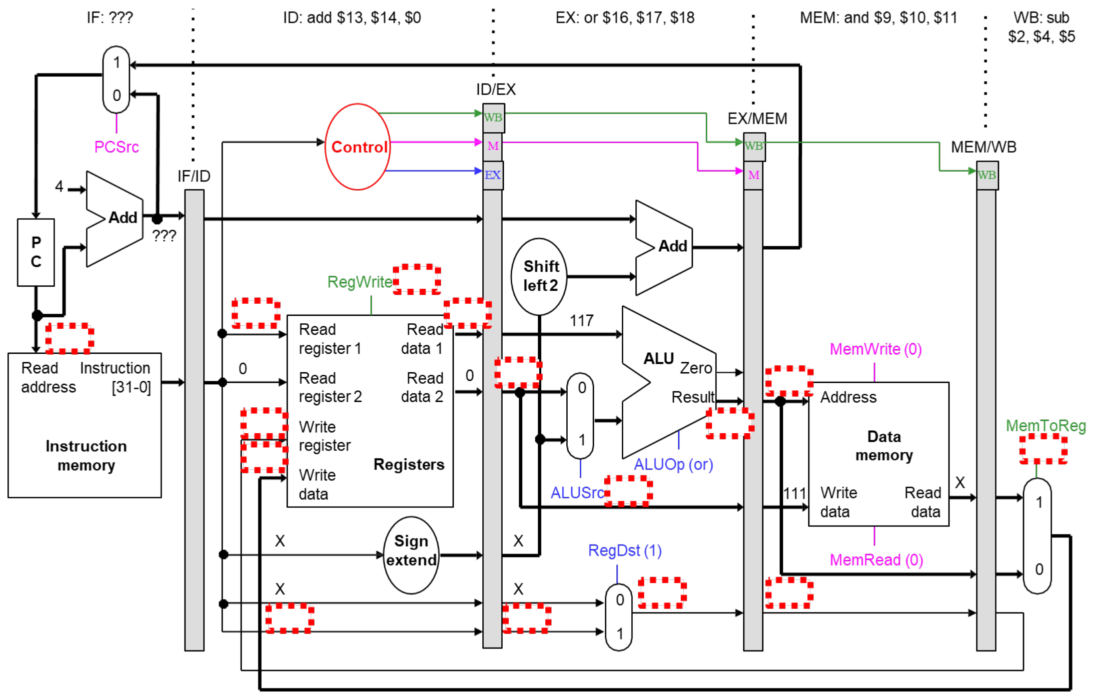
Cycle 4



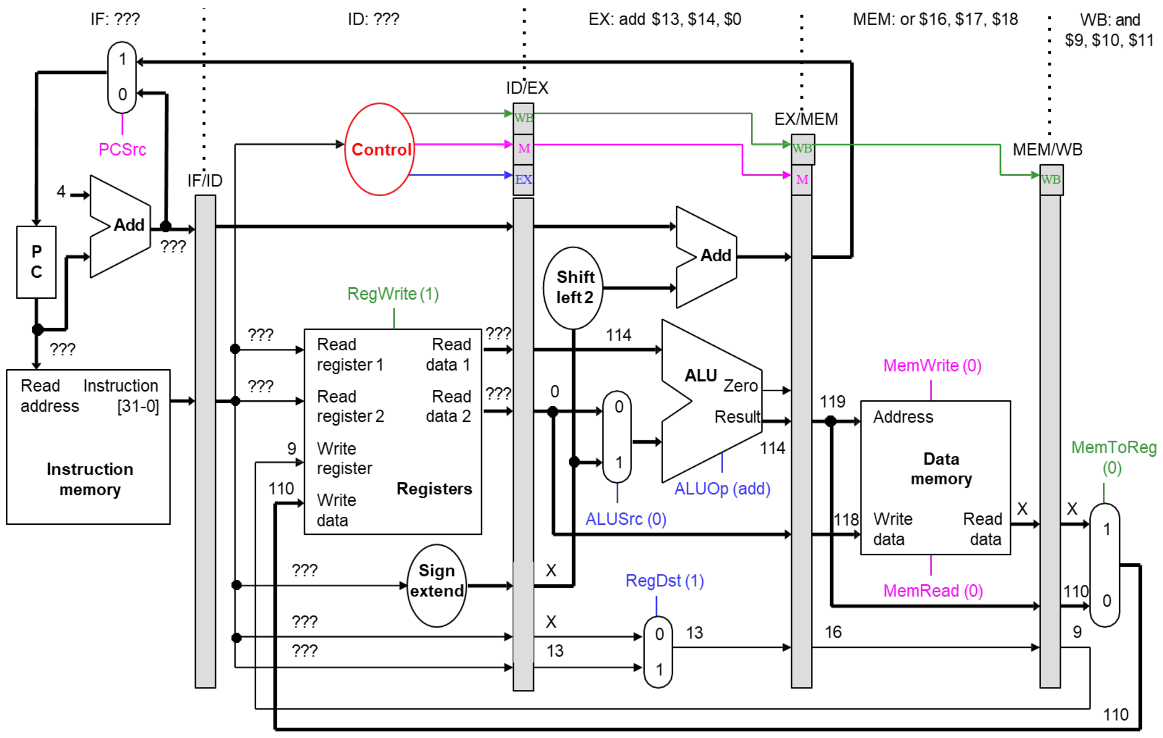
Cycle 5 (Full)



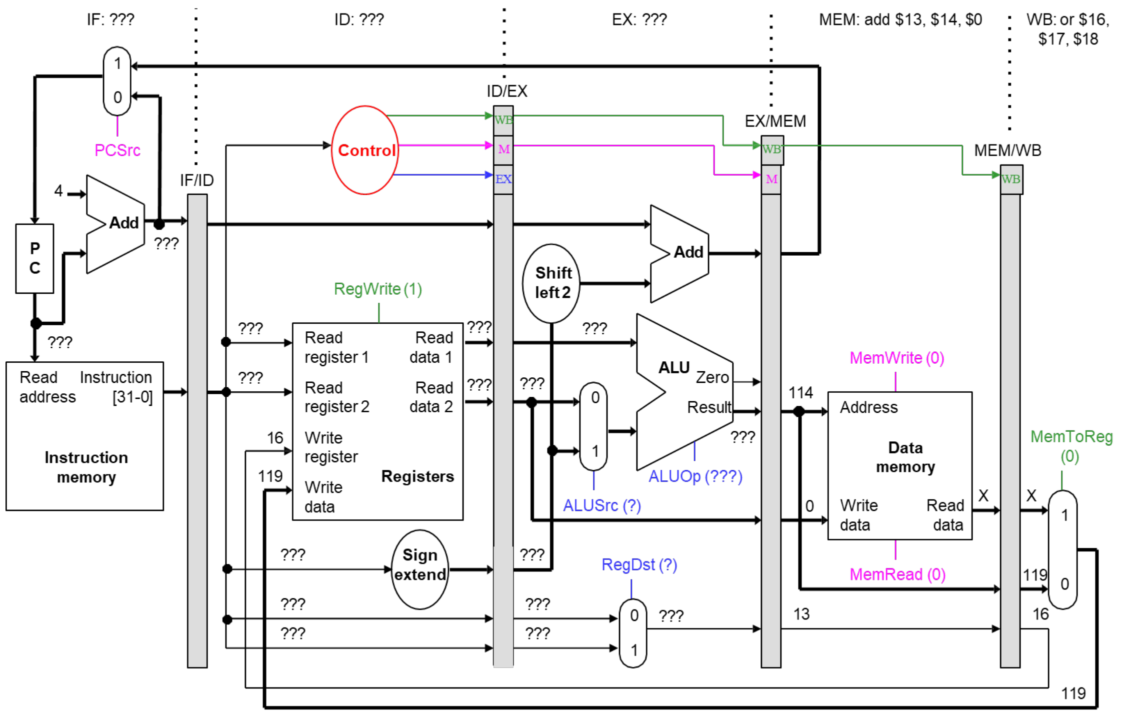
Cycle 6 (Emptying)



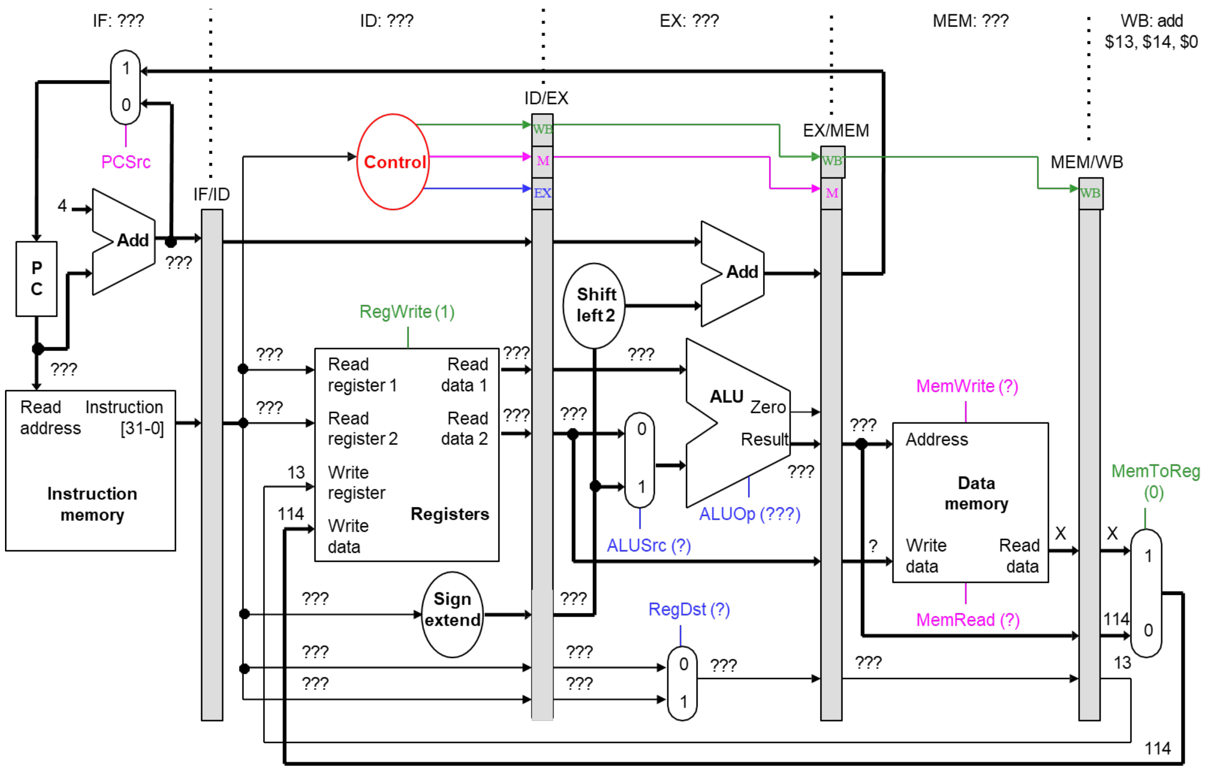
Cycle 7



Cycle 8



Cycle 9



That's a lot of Diagrams There

	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 $t5, $t6, $0					IF	ID	EX	MEM	WB

- Compare the last nine slides with the pipeline diagram above.
  - — You can see how instruction executions are overlapped.
  - — Each functional unit is used by a *different* instruction in each cycle.
  - — The pipeline registers save control and data values generated in previous clock cycles for later use.
  - — When the pipeline is full in clock cycle 5, all of the hardware units are utilized. This is the ideal situation, and what makes pipelined processors so fast.

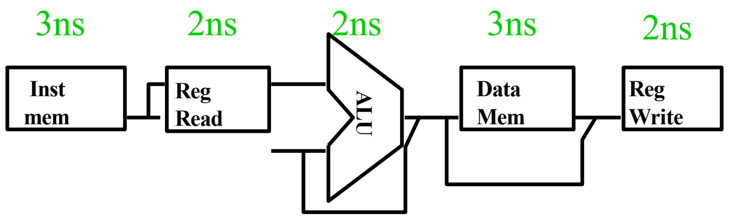
$N = 100000$

$T(S) = 1000 * 10$

$T(P) = (4 + 1000) * 2$

Performance Revisited

- Assuming the following functional unit latencies:



- What is the cycle time of a single-cycle implementation?
  - What is its throughput (how many works/instr. finished in a unit of time)?

$Single: \frac{5instr.}{(5*12)ns} = \frac{1}{12}$ $Pipeline: \frac{5instr.}{(9*3)ns} = \frac{5}{27}$

- What is the cycle time of a ideal pipelined implementation?
  - What is its steady-state throughput?

$Pipeline: \frac{1000instr.}{((4+1000)*3)ns} = 0.33$

$Single: \frac{1000instr.}{(1000*12)ns} = \frac{1}{12}$

- How much faster is pipelining?

Ideal Speedup

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

- 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 "4" times?
  - The pipeline stages are imbalanced: a register file and ALU operations can be done in 2ns, but we must stretch that out to 3ns to keep the ID, EX, and WB stages synchronized with IF and MEM.
  - Balancing the stages is one of the many hard parts in designing a pipelined processor.

The Pipelining Paradox

		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

- 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 (15ns vs. 12ns)!
- 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.