

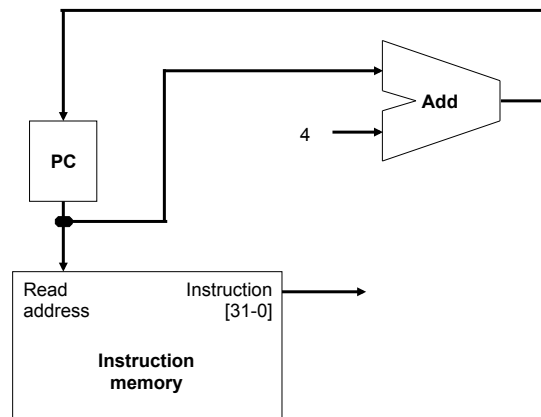
MIPS Pipelining

CS 250

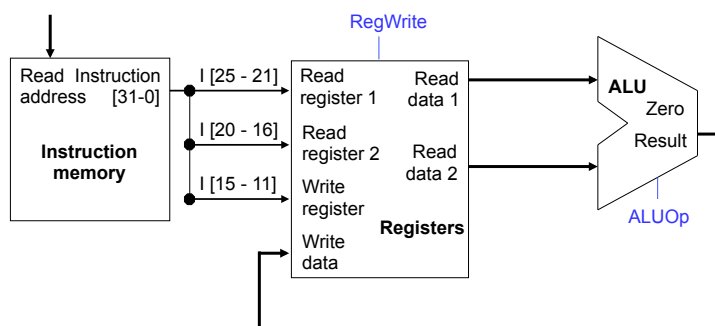
Single Cycle Processor Recap

1. Instruction Fetching and Incrementing the PC
2. R type instruction
3. I Type – MemToReg instruction (lw)
4. I Type branch instruction – beq

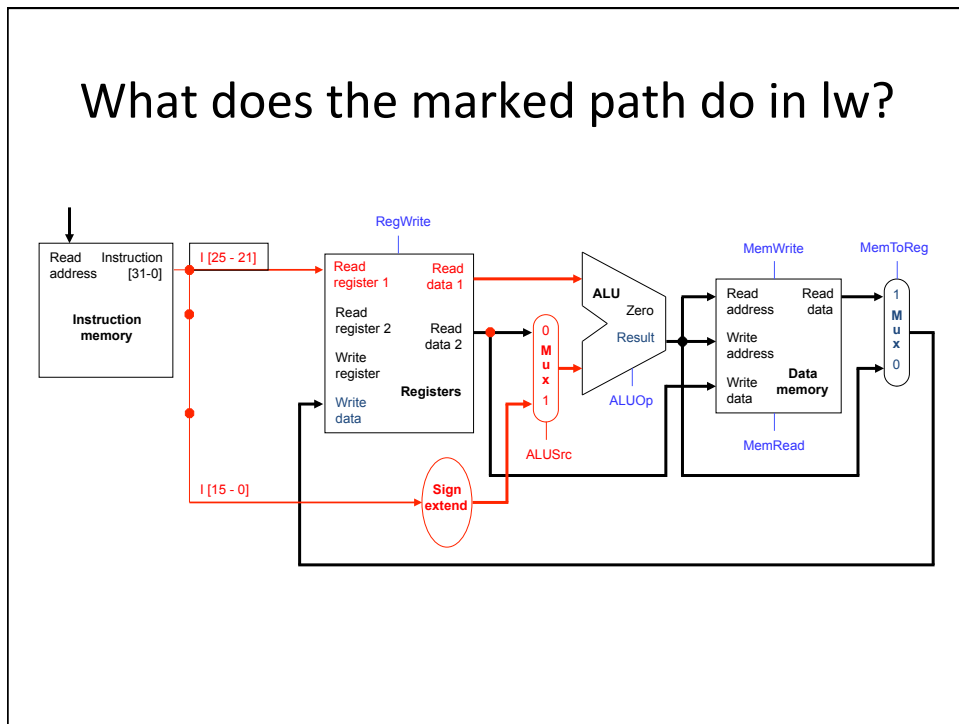
Instruction Fetch and PC Increment



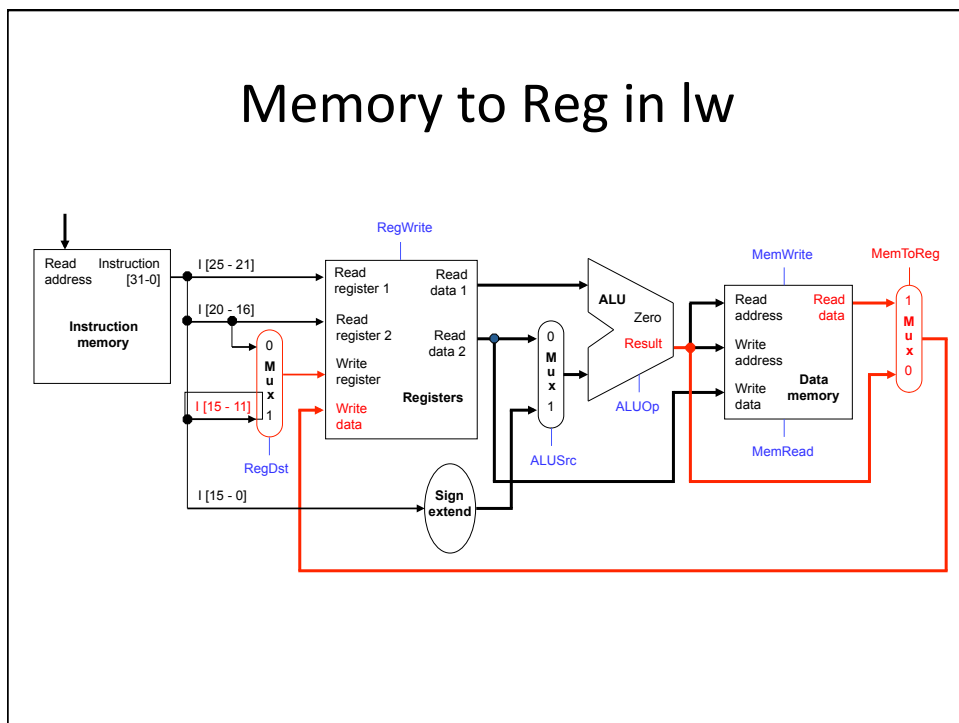
?? Type Instruction Data path



What does the marked path do in lw?



Memory to Reg in lw



The steps in executing a beq

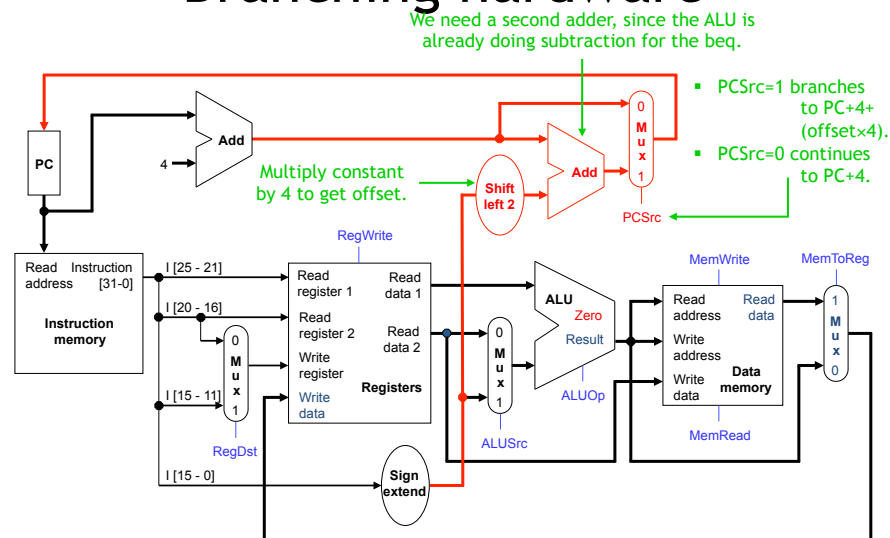
1. Fetch the instruction, like `beq $t0, $s0, offset`, from memory.
2. Read the source registers, `$t0` and `$s0`, from the register file.
3. Compare the values by subtracting them in the ALU.
4. If the subtraction result is 0, the source operands were equal and the PC should be loaded with the target address, $PC + 4 + (\text{offset} \times 4)$.
5. Otherwise the branch should not be taken, and the PC should just be incremented to $PC + 4$ to fetch the next instruction sequentially.

September 19, 2016

A single-cycle MIPS processor

7

Branching hardware

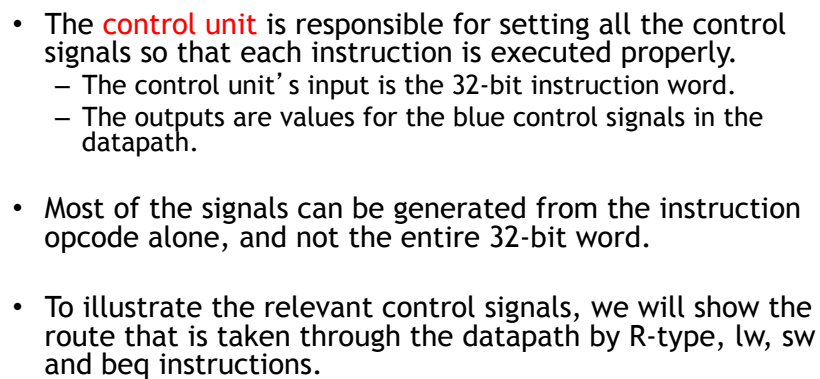


September 19, 2016

A single-cycle MIPS processor

8

Control



Control signal table

Operation	RegDst	RegWrite	ALUSrc	ALUOp	MemWrite	MemRead	MemToReg
add	1	1	0	010	0	0	0
sub	1	1	0	110	0	0	0
and	1	1	0	000	0	0	0
or	1	1	0	001	0	0	0
slt	1	1	0	111	0	0	0
lw	0	1	1	010	0	1	1
sw	X	0	1	010	1	0	X
beq	X	0	0	110	0	0	X

- **sw and beq are the only instructions that do not write any registers.**
- **lw and sw are the only instructions that use the constant field. They also depend on the ALU to compute the effective memory address.**
- **ALUOp for R-type instructions depends on the instructions' func field.**
- The PCSrc control signal (not listed) should be set if the instruction is beq and the ALU's Zero output is true.

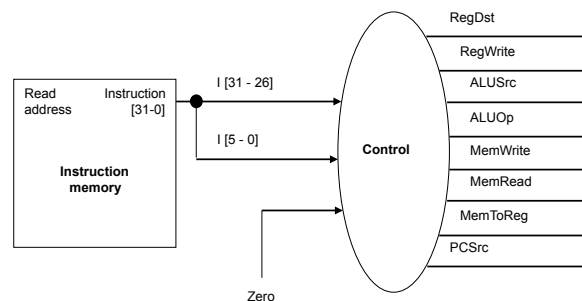
September 19, 2016

A single-cycle MIPS processor

11

Generating control signals

- The control unit needs 13 bits of inputs.
 - Six bits make up the instruction's opcode.
 - Six bits come from the instruction's func field.
 - It also needs the Zero output of the ALU.
- The control unit generates 10 bits of output, corresponding to the signals mentioned on the previous page.
- You can build the actual circuit by using big K-maps or Boolean algebra.

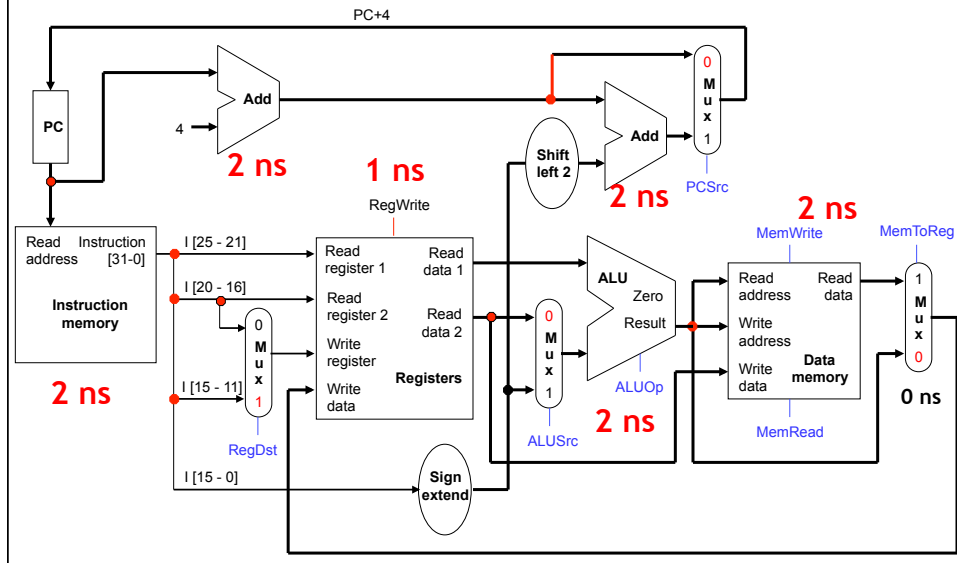


September 19, 2016

12




Latency of a datapath

- What is the latency of **BEQ**?

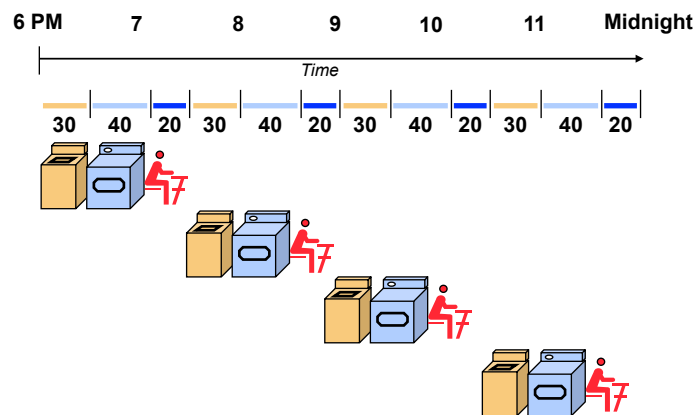


Pipelining

Time to do work

- 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?

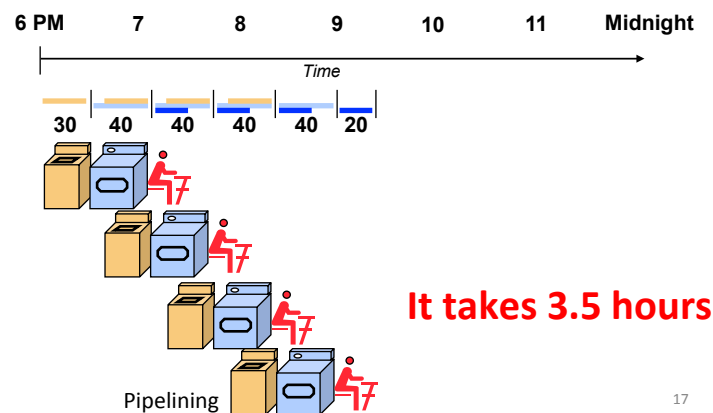
The slow way



- If each load is done **sequentially** it takes **6 hours**

Laundry Pipelining

- Start each load as soon as possible - Overlap loads



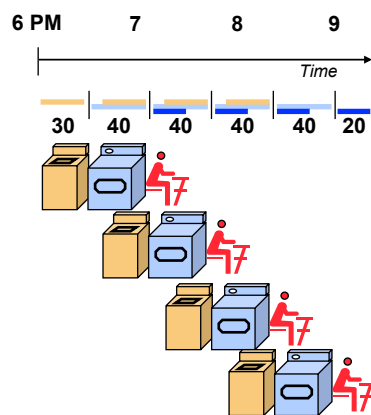
September 19, 2016

17

Questions

1. What can you say about **latency** and **throughput** of a pipelined process?
2. **Max speedup** (of all jobs put together) in a pipeline? Under what conditions?
3. What about the **bottleneck** of a pipelined process at steady state?
4. If we had a limited number of laundry loads, can we get the same throughput as in steady state with unlimited jobs?

Pipelining Lessons



- Pipelining doesn't help **latency** of single load, it helps **throughput** of entire workload
- Pipeline rate limited by **slowest** pipeline stage
- Potential speedup = **Number pipe stages**
- Unbalanced lengths of pipe stages reduces speedup
- Time to “**fill**” pipeline and time to “**drain**” it reduces throughput

September 19, 2016

Pipelining

19

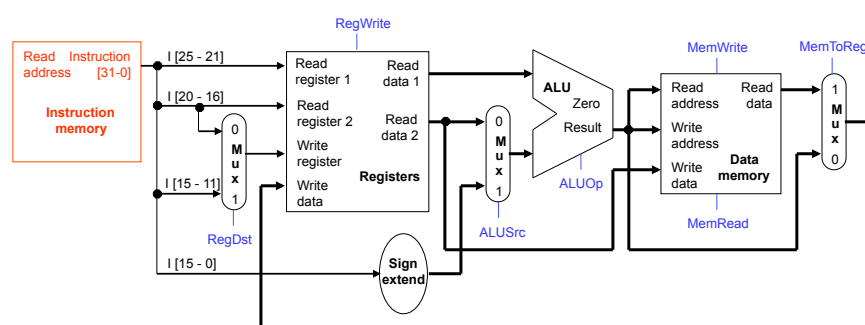
Pipelining Vs Multiprocessing

- **Multiprocessing** – one job is done entirely by one processor and multiple jobs are spread out over many processors.
- **Pipelining** – one job needs many processors but at different times and so we overlap the execution of these jobs with one another.
- Which one of the above is **checkout line at a supermarket?**

Coming back to Instruction processing – steps of instruction execution

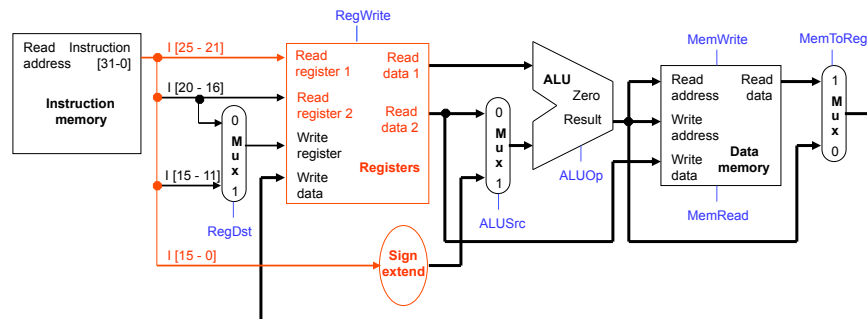
Step	Name	Description
Instruction Fetch	IF	Read an instruction from memory.
Instruction Decode	ID	Read source registers and generate control signals.
Execute	EX	Compute an R-type result or a branch outcome.
Memory	MEM	Read or write the data memory.
Writeback	WB	Store a result in the destination register.

Instruction Fetch (IF)



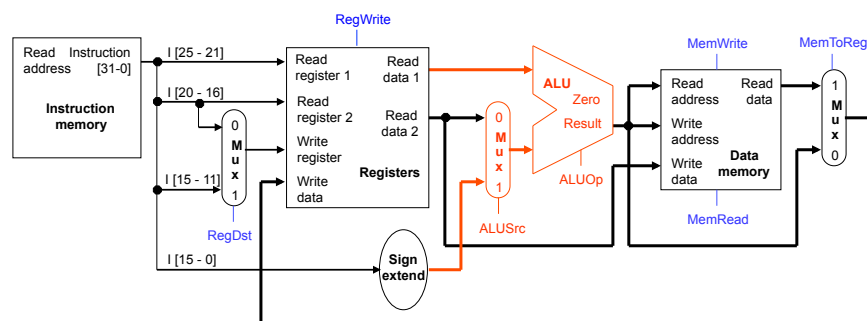
Instruction Decode (ID)

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



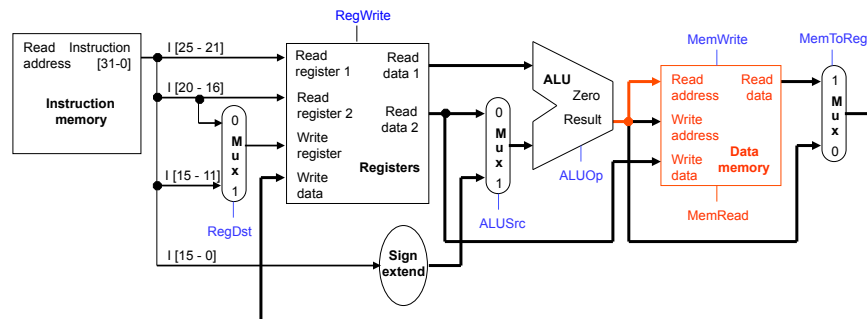
Execute (EX)

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



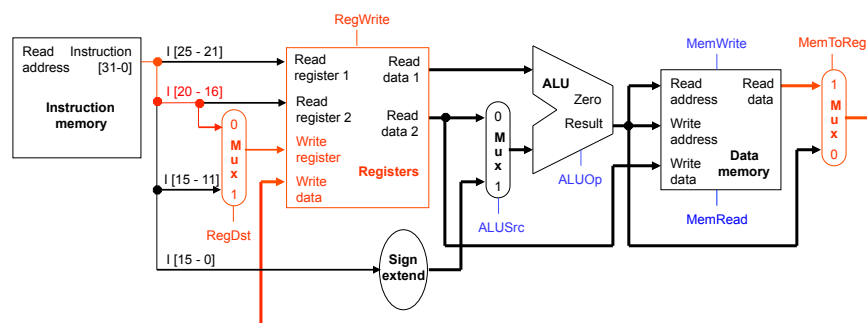
Memory (MEM)

- 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.



Not all instructions need all steps

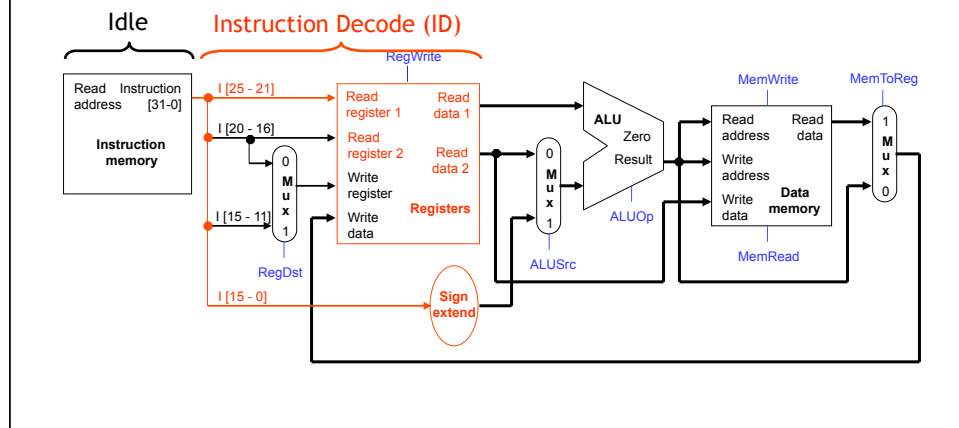
Instruction	Steps required				
beq	IF	ID	EX		
R-type	IF	ID	EX		WB
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

A bunch of lazy functional units

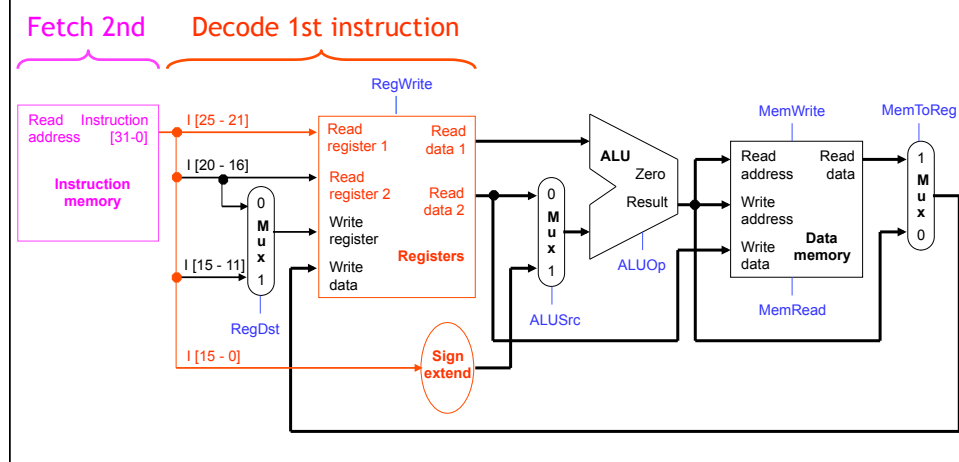
- Notice that each execution step uses one **different** functional unit.
- In other words, the main units are idle for most of the (long) cycle!
 - The instruction RAM is used for just 2ns at the start of the cycle.
 - Registers are read once in ID (1ns), and written once in WB (1ns).
 - The ALU is 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

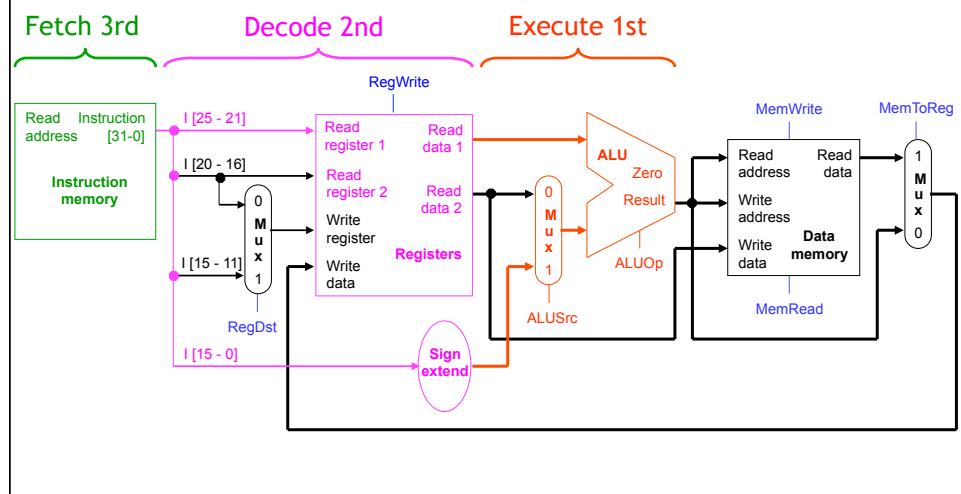
- We shouldn't have to wait for the entire instruction to be complete before we can re-use the functional units.
- For example, the instruction memory is free in the Instruction Decode step as shown below, so...



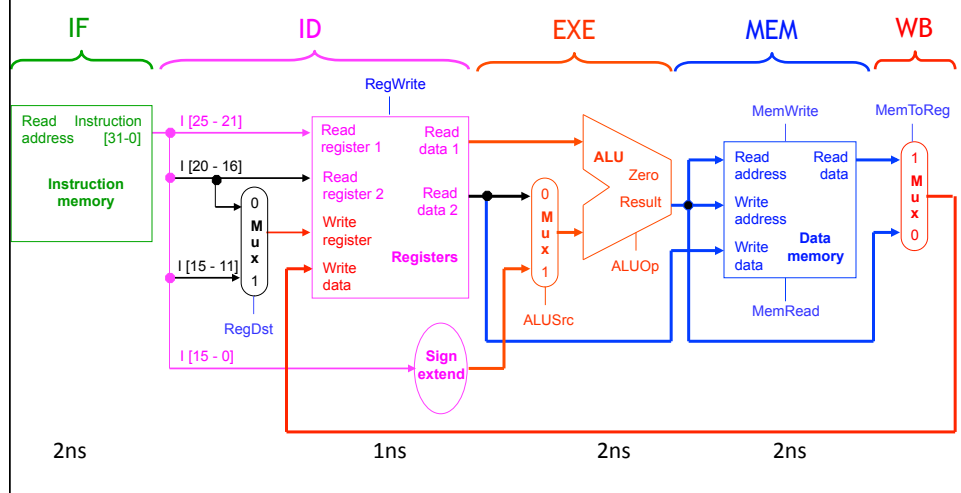
Overlapping two instructions



Continuing the process



Creating a pipelined datapath



Pipelined execution

	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

Pipelining Performance

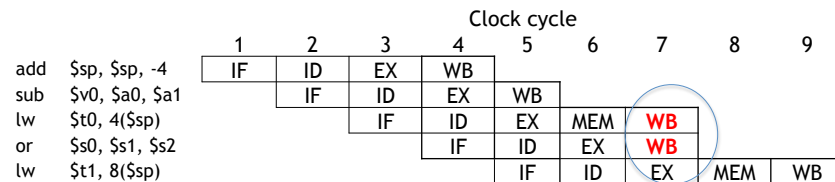
	Clock cycle								
	1	2	3	4	5	6	7	8	9
lw \$t0, 4(\$sp)	IF	ID	EX	MEM	WB				
lw \$t1, 8(\$sp)		IF	ID	EX	MEM	WB			
lw \$t2, 12(\$sp)			IF	ID	EX	MEM	WB		
lw \$t3, 16(\$sp)				IF	ID	EX	MEM	WB	
lw \$t4, 20(\$sp)					IF	ID	EX	MEM	WB

filling

- Execution time on ideal pipeline:
 - time to fill the pipeline + one cycle per instruction
 - How long for N instructions?
- Compare with other implementations:
 - Single Cycle: (8ns clock period)
- How much faster is pipelining for N=1000 ?

Pipelining other instruction types

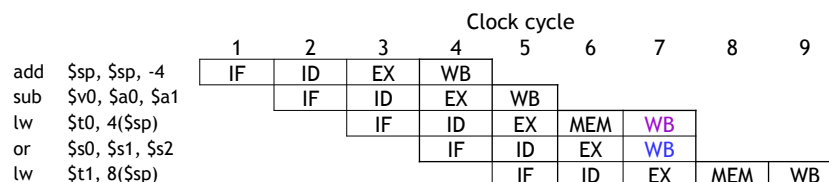
- R-type instructions only require 4 stages: IF, ID, EX, and WB
 - We don't need the MEM stage
- What happens if we try to pipeline loads with R-type instructions?



35

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



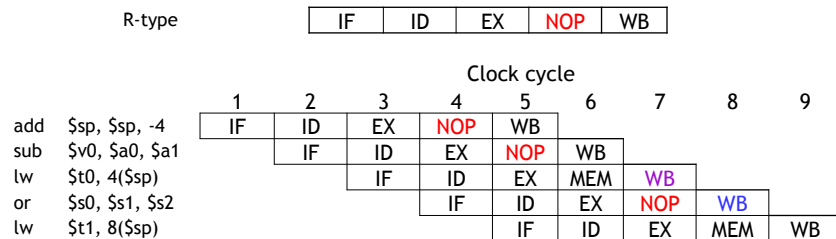
This doesn't work!

September 19, 2016

36

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



- Stores and Branches have **NOP** stages, too...

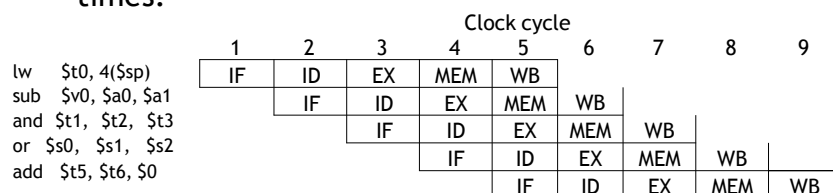
store	IF	ID	EX	MEM	NOP
branch	IF	ID	EX	NOP	NOP

September 19, 2016

37

Pipelining Summary

- A pipelined processor allows multiple instructions to execute at once, and each instruction uses a different functional unit in the datapath.
- This increases throughput, so programs can run faster.
 - One instruction can finish executing on every clock cycle, and simpler stages also lead to shorter cycle times.



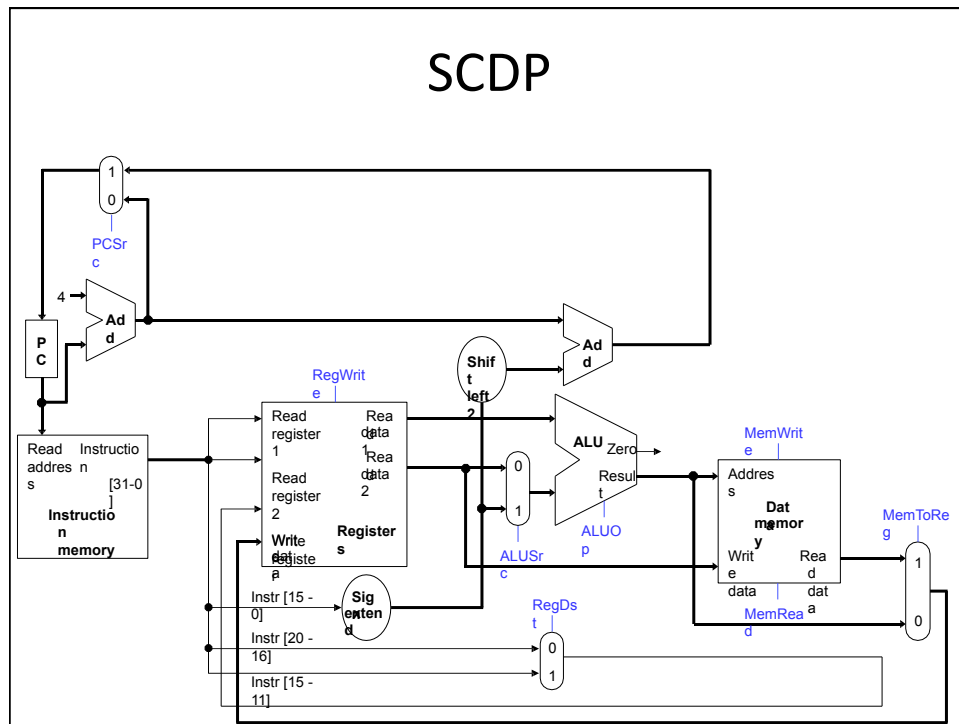
Pipelined Datapath

- We may need to perform *multiple 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.
- Thus, like the single-cycle datapath, a pipelined processor will need to duplicate hardware elements that are needed several times in the same clock cycle.
 - What about the register file?

How about if **WB** and **ID** are in one cycle?

lw \$t0, 4(\$sp)	IF	ID	EX	MEM	WB				
sub \$v0, \$a0, \$a1		IF	ID	EX	NOP	WB			
and \$t1, \$t2, \$t3			IF	ID	EX	NOP	WB		
or \$s0, \$s1, \$s2				IF	ID	EX	NOP	WB	
add \$t5, \$t6, \$0					IF	ID	EX	NOP	WB

Read register 1	Read data 1
Read register 2	Read data 2
Write register	Registers
Write data	



The need for pipelined registers

- We'll add intermediate registers to our pipelined datapath.
- 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.

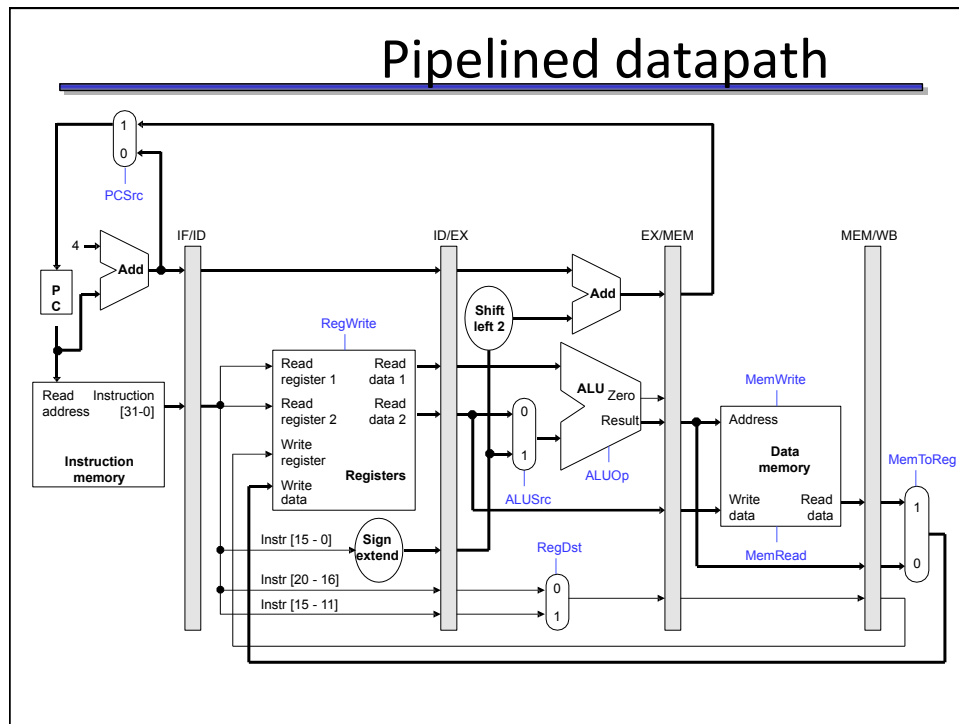
IF/ID

ID/EX

EX/MEM

MEM/WB

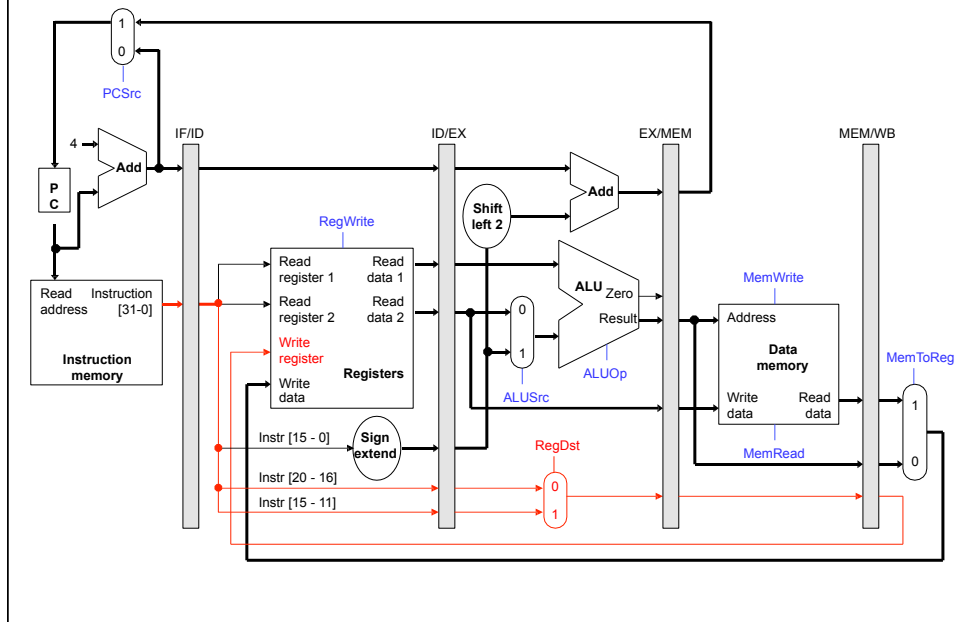
- No register is needed after the WB stage, because after WB the instruction is done.



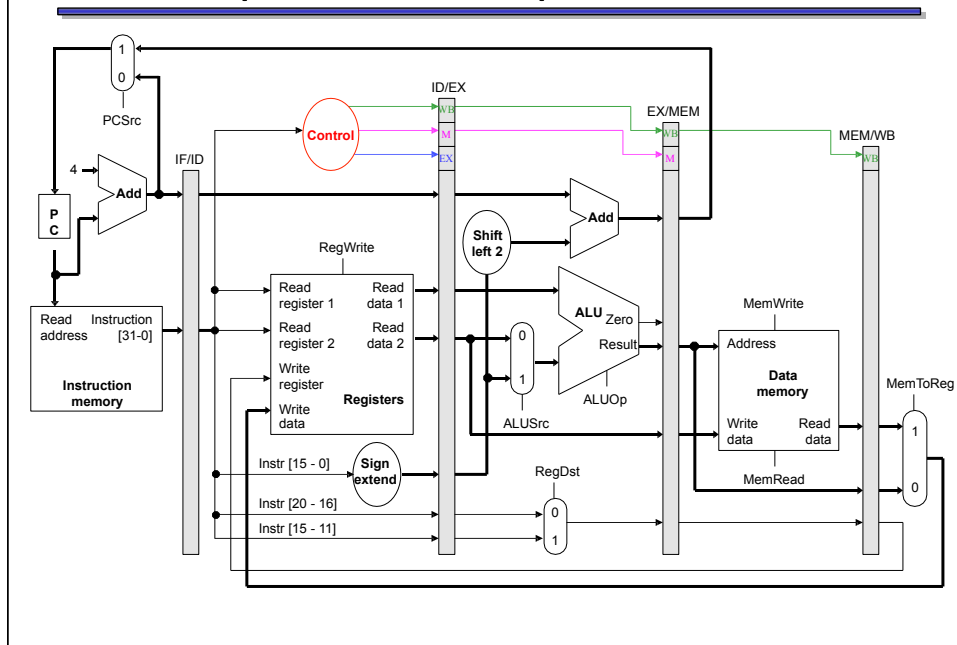
Propagating values forward

- Any data values required in later stages must be propagated through the pipeline registers.
- The most extreme example is the destination register.
 - The rd field of the instruction word, retrieved in the first stage (IF), determines the destination register. But that register isn't updated until the *fifth* stage (WB).
 - Thus, the **rd** field must be passed through all of the pipeline stages, as shown in red on the next slide.

The destination register



Pipelined datapath and control



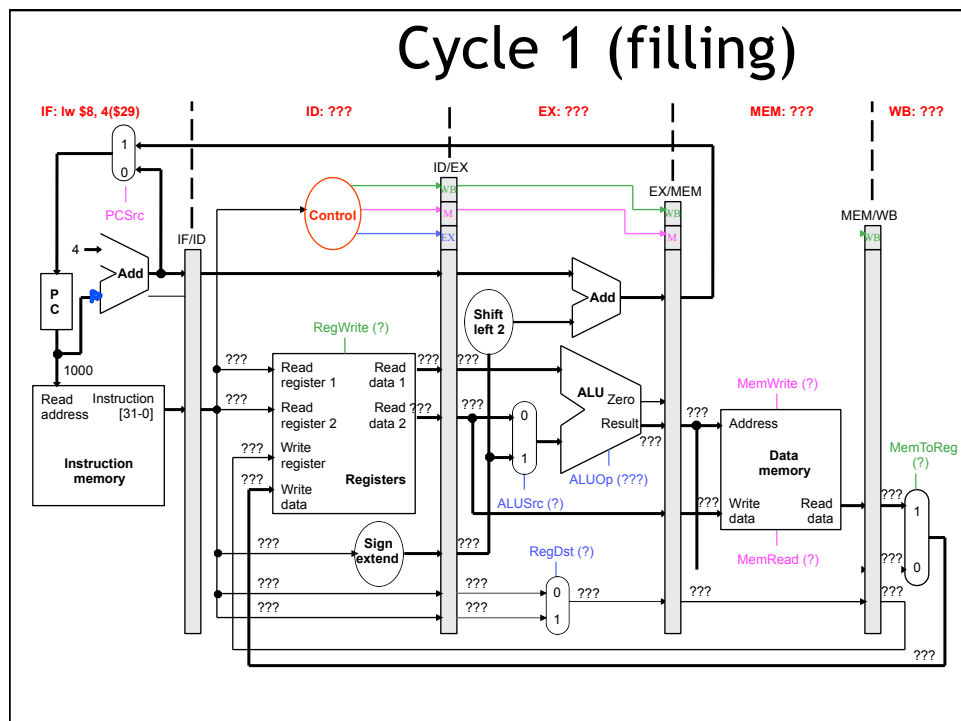
An example execution sequence

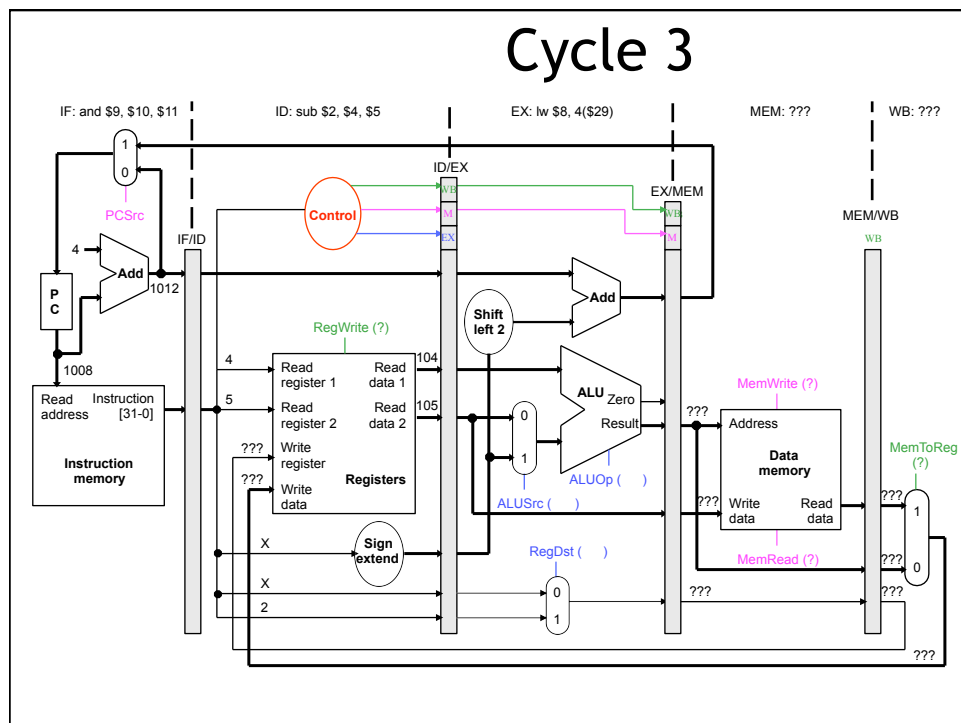
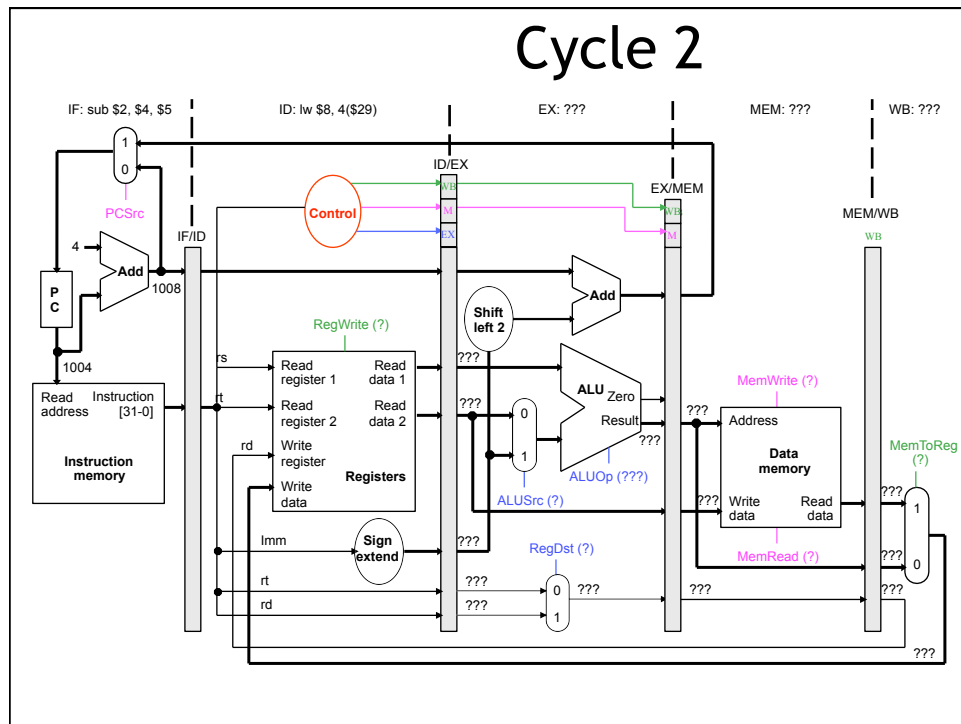
- Here's a sample sequence of instructions to execute.

addresses
in decimal

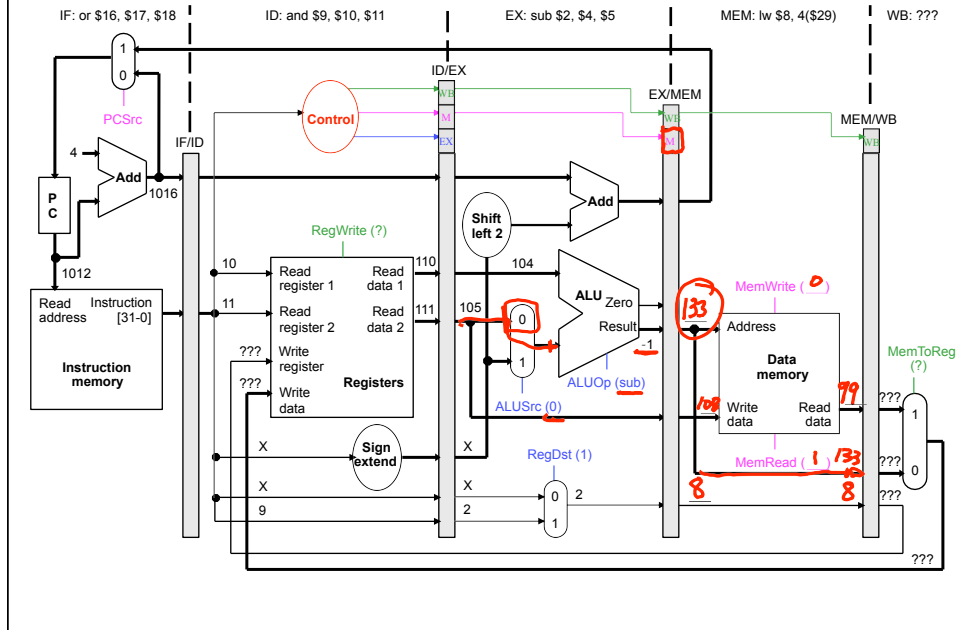
```
1000: lw $8, 4($29)
1004: sub $2, $4, $5
1008: and $9, $10, $11
1012: or $16, $17, $18
1016: add $13, $14, $0
```

- 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.

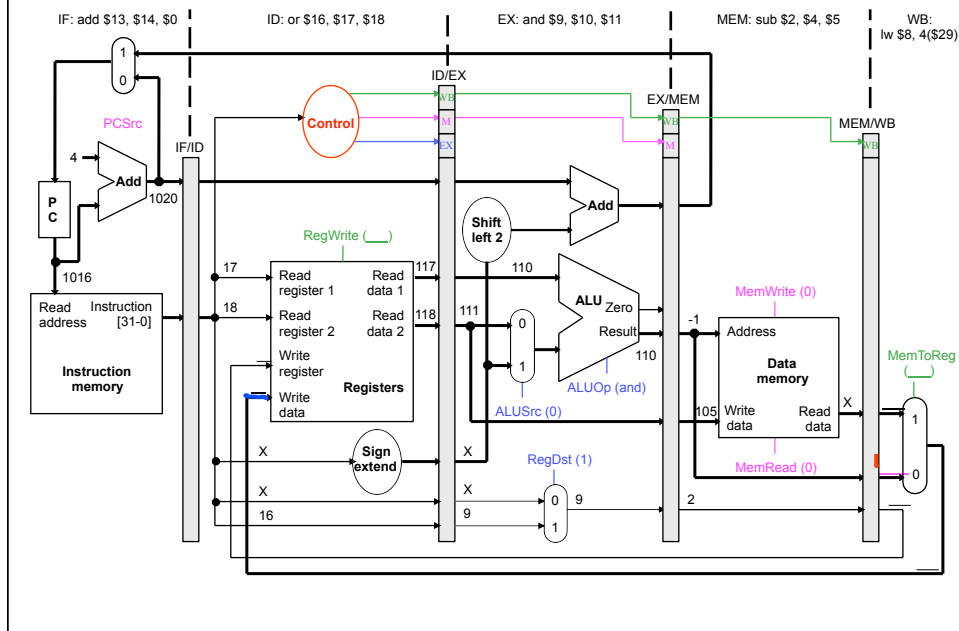


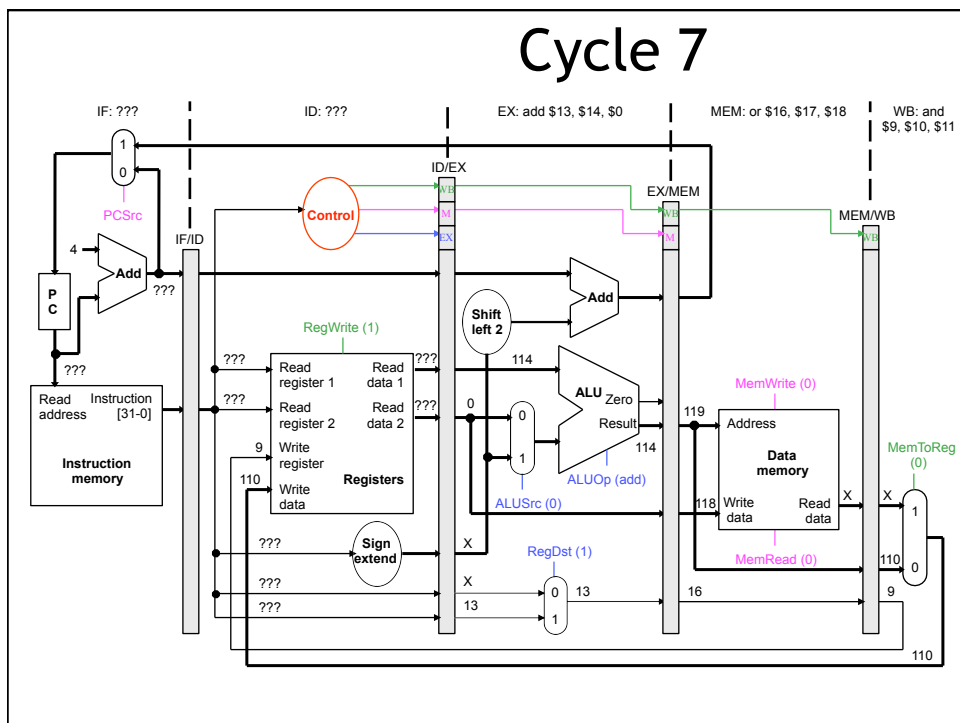
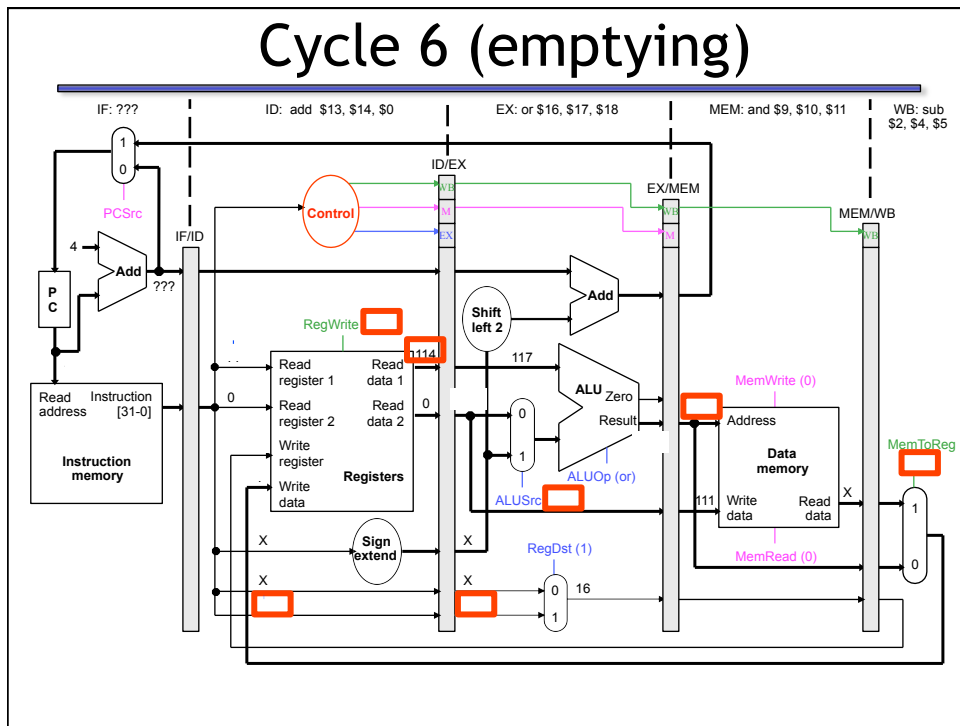


Cycle 4

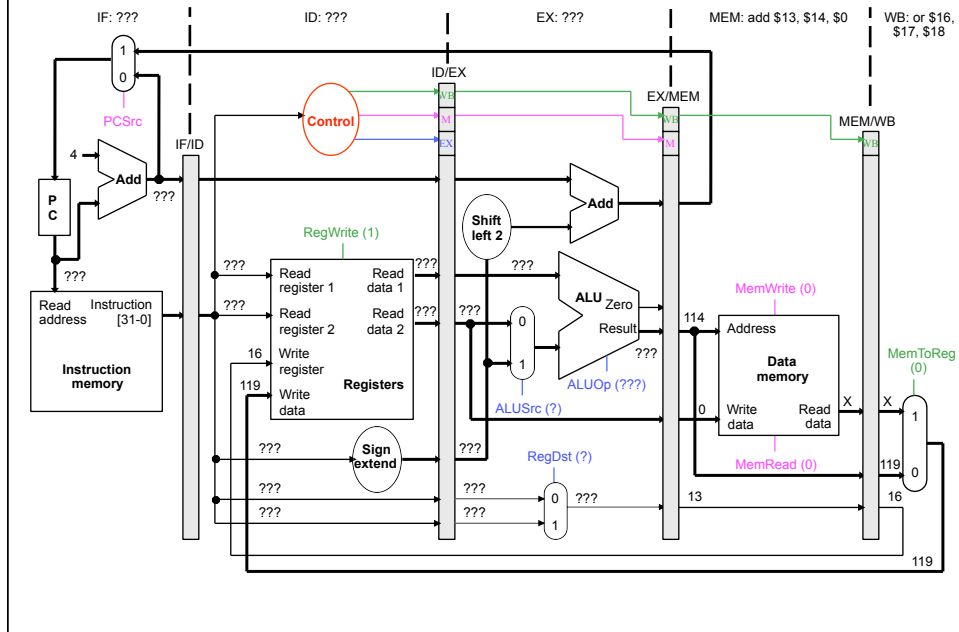


Cycle 5 (full)

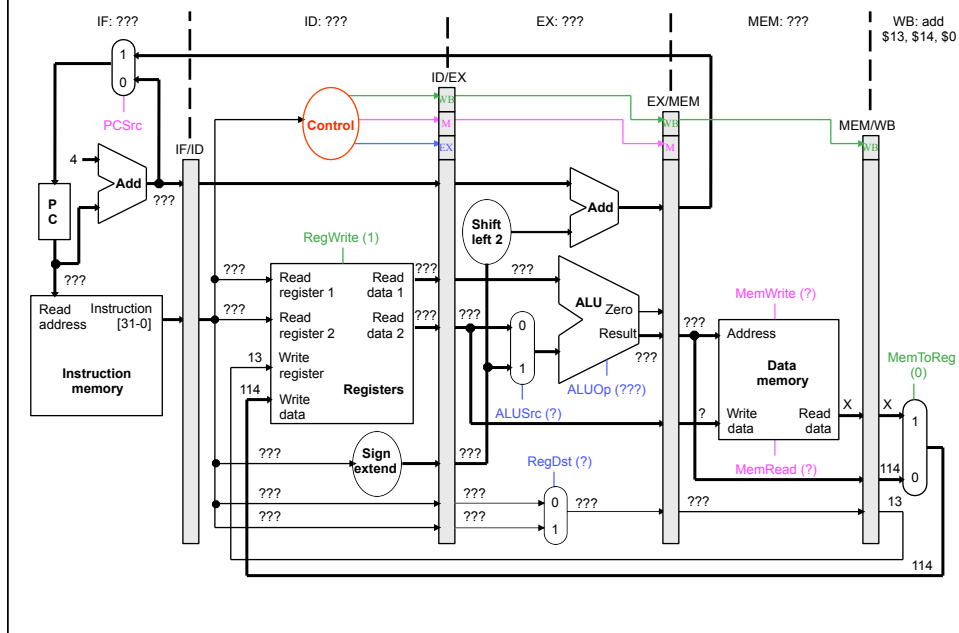




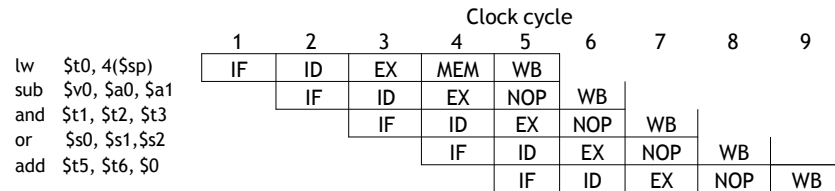
Cycle 8



Cycle 9

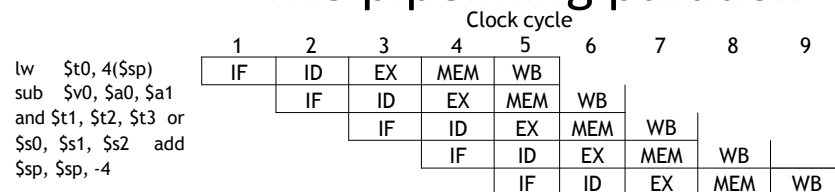


That's a lot of diagrams here



- 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.

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 (Why?!)
- 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.

Instruction set architectures and pipelining

- The MIPS instruction set was designed especially for easy pipelining.
 - All instructions are 32-bits long, so the instruction fetch stage just needs to read one word on every clock cycle.
 - Fields are in the same position in different instruction formats—the opcode is always the first six bits, rs is the next five bits, etc. This makes things easy for the ID stage.
 - MIPS is a register-to-register architecture, so arithmetic operations cannot contain memory references. This keeps the pipeline shorter and simpler.
- Pipelining is harder for older, more complex instruction sets.
 - If different instructions had different lengths or formats, the fetch and decode stages would need extra time to determine the actual length of each instruction and the position of the fields.
 - With memory-to-memory instructions, additional pipeline stages may be needed to compute effective addresses and read memory *before* the EX stage.