Lecture 12: Pipelining to Improve Performance

(CPEG323: Intro. to Computer System Engineering)

Performance of Single-cycle Design

CPU time = Instructions executed × CPI × Clock cycle time

CPI = 1 for a single-cycle design

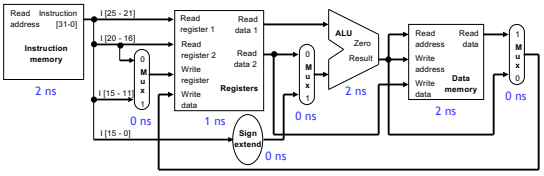
- At the start of the cycle, PC is updated (PC + 4, or PC + 4 + offset × 4)
- New instruction loaded from memory, control unit sets the datapath signals appropriately so that
  - registers are read,
  - ALU output is generated,
  - data memory is accessed,
  - branch target addresses are computed, and
  - 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?

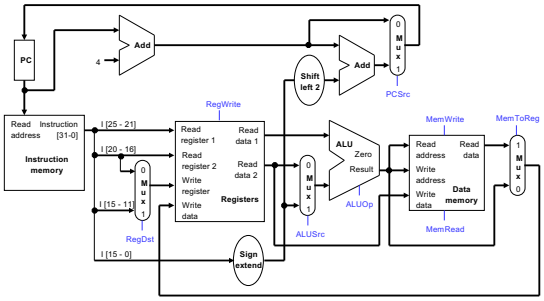
Components of the data-path

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

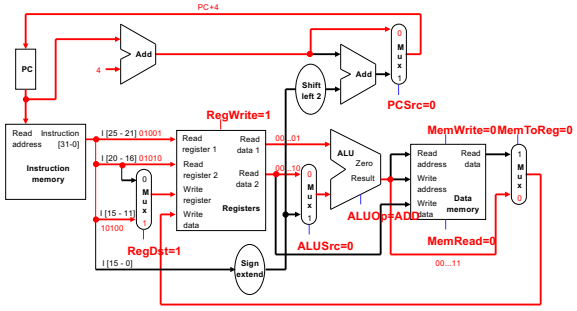
reading the instruction memory	2ns
reading the register file	1ns
ALU computation	2ns
accessing data memory	2ns
writing to the register file	1ns



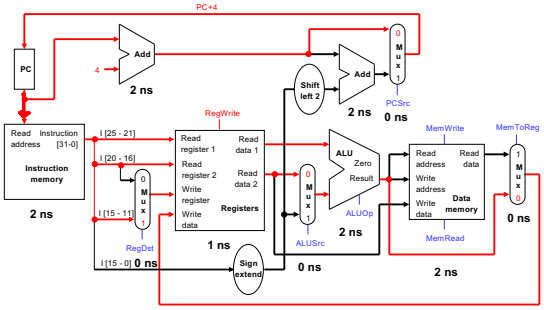
How the add goes through the datapath



How the add goes through the datapath

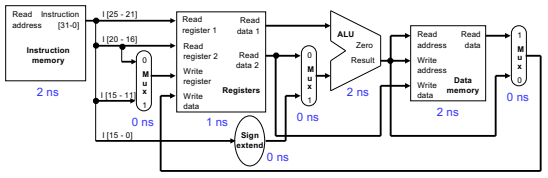


Compute the longest path in the add instruction



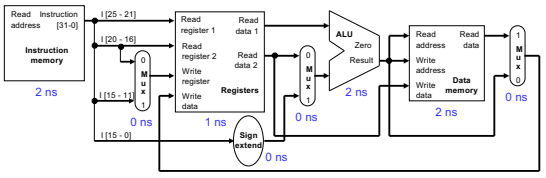
The slowest instruction...

- If all instructions must complete within one clock cycle, then the cycle time has to be large enough to accommodate the *slowest* instruction.
- For example, lw $t0, –4($sp) is the slowest instruction needing __ns.
  - Assuming the circuit latencies below.



The slowest instruction...

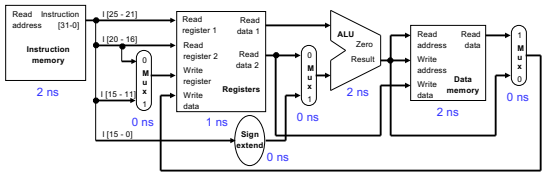
- If all instructions must complete within one clock cycle, then the cycle time has to be large enough to accommodate the *slowest* instruction.
- For example, lw $t0, –4($sp) is the slowest instruction needing _8_ns.
  - Assuming the circuit latencies below.



...determines the clock cycle time

- If we make the cycle time 8ns then *every* instruction will take 8ns, even if they don't need that much time.
- For example, the instruction add $s4, $t1, $t2 really needs just 6ns.

reading the instruction memory	2 ns	6 ns
reading registers $t1 and $t2	1 ns	
computing $t1 + $t2	2 ns	
storing the result into $s0	1 ns	

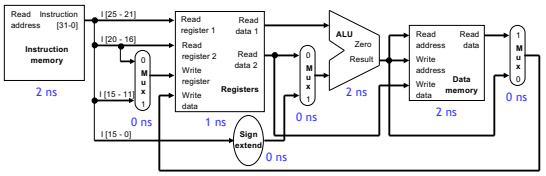


Components of the data-path

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

reading the instruction memory	2ns	8ns -> 125MHz
reading the register file	1ns	
ALU computation	2ns	
accessing data memory	2ns	
writing to the register file	1ns	

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



10

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 following **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, etc.), multiple loads of laundry (washer + dryer), …

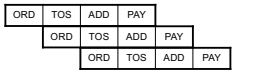
12

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:



13

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 ***increase*** *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

14

Pipelining is not just Multiprocessing

- Both multiprocessing and pipelining relate to the processing of multiple "things" using multiple "functional units"
  - Multiprocessing implies 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.
    - Supermarket analogy?
- Pipelining and multiprocessing are not mutually exclusive
  - Modern processors do both, with multiple pipelines (e.g., superscalar)

Pipelining MIPS

- 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.

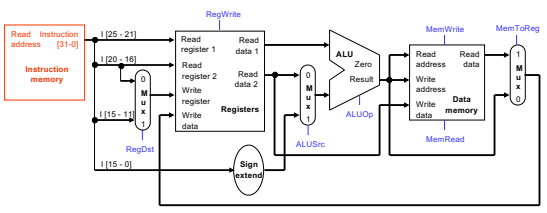
- Not all instructions need all give stages and stages have different lengt

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)

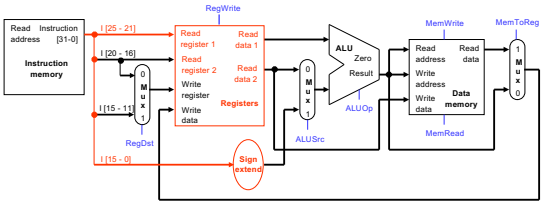
Instruction Fetch (IF)

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



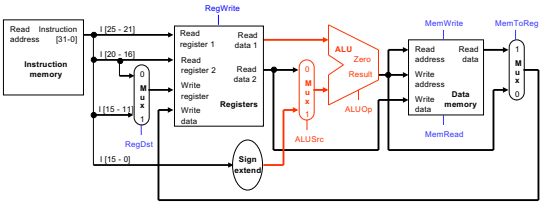
Instruction Decode (ID)

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



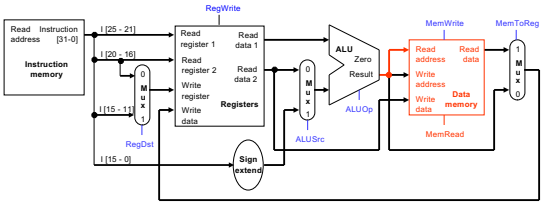
Execute (EX)

- … and so on for the EX portion…



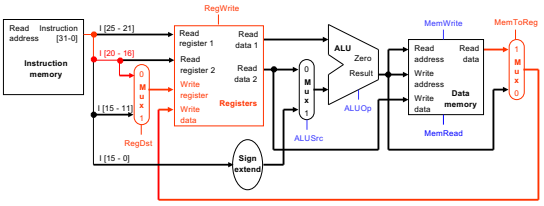
Memory (MEM)

- … MEM portion…



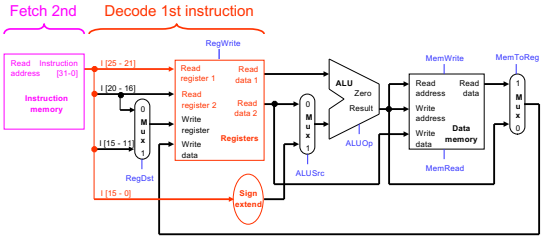
Writeback (WB)

- … and the WB portion.
- What about the "clash" with the IF stage over the register file?
- Answer: register file is read at the end of the clock bycle, but written at the start of the clock cycle. Hence, there is no clash



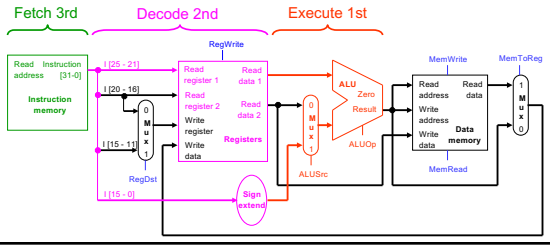
Decoding and fetching together

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



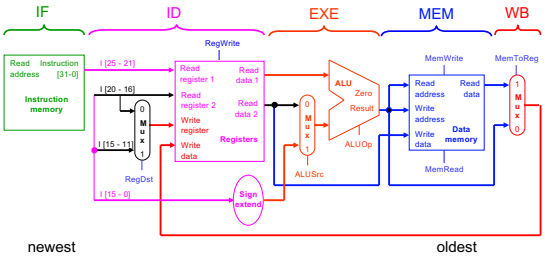
Executing, decoding and fetching

- 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!

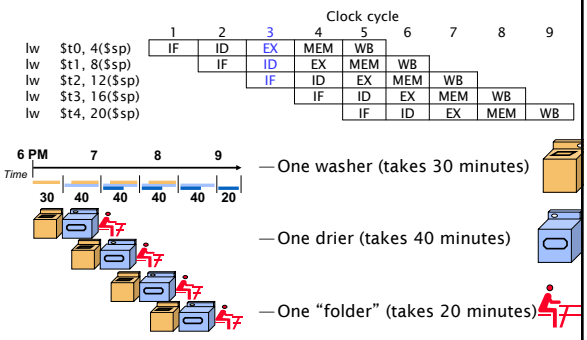


Break datapath into 5 stages

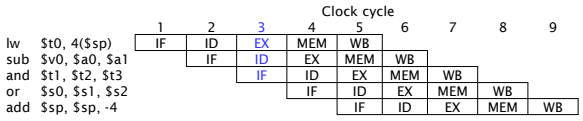
- Insert pipeline registers
- Each stage has its own functional units.
- Each stage can execute in 2ns



Pipelining Loads

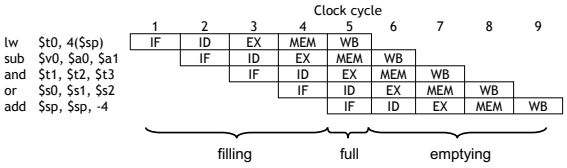


A pipeline diagram



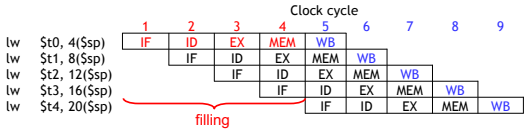
- A pipeline diagram shows the execution of a series of instructions.
  - The instruction sequence is shown vertically, from top to bottom.
  - Clock cycles are shown horizontally, from left to right.
  - Each instruction is divided into its component stages.
- For example, in cycle 3, there are three 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



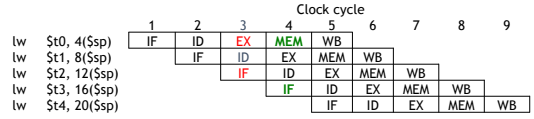
- The pipeline depth is the number of stages—in this case, five.
- The pipeline is filling in the first four cycles (unused functional units).
- In cycle 5, the pipeline is full. Five instructions are being executed simultaneously, so all hardware units are in use.
- In cycles 6-9, the pipeline is emptying.

Pipelining Performance



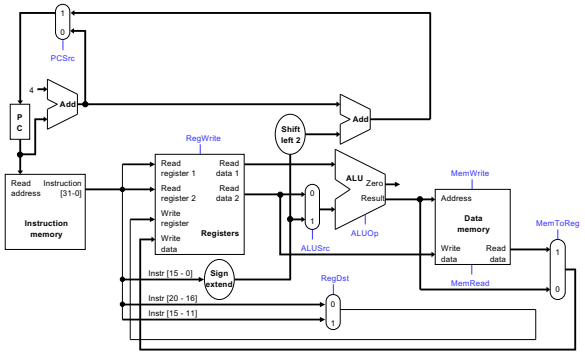
- How many cycles to execute N instructions on a k stage pipeline?
  - Solution 1: $k - 1$ cycles to fill the pipeline + one cycle per instruction = $k - 1 + N$ cycles
  - Solution 2: k cycles for the first instruction + one cycle for each of the remaining $N - 1$ instructions
- When $N = 1000$, how much faster is a 5-stage pipeline (2ns clock cycle) vs. a single cycle implementation (8ns clock cycle)?

Pipeline Datapath: Resource Requirements



- 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)

Single-cycle datapath, slightly rearranged



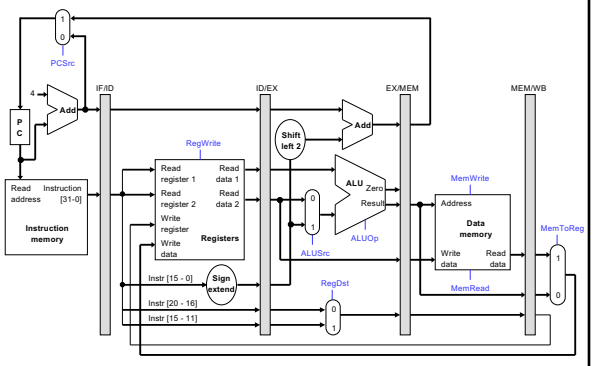
Pipeline registers

- In pipelining, we divide instruction execution into multiple cycles
  - IF ID EX MEM WB
- Information computed during one cycle may be needed in a later cycle:
  - Instruction read in IF stage determines which registers are fetched in ID stage, what immediate is used for EX stage, and what destination register is for WB
  - Register values read in ID are used in EX and/or MEM stages
  - ALU output produced in EX is an effective address for MEM or a result for WB
- A lot of information to save!
  - Saved in intermediate registers called pipeline registers
- 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

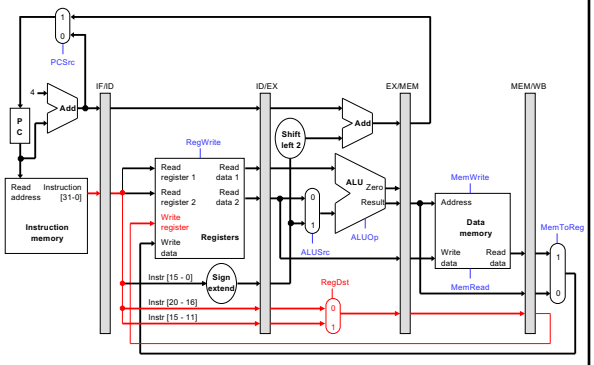
Pipelined datapath



Propagating values forward

- Data values required later propagated through the pipeline registers.
- The most extreme example is the destination register (rd or rt).
  - It is retrieved in IF, but isn't updated until WB.
  - Thus, it must be passed through all pipeline stages, as shown in red on the next slide.
- Notice that we can't keep a single "instruction register," because the pipelined machine needs to fetch a new instruction every clock cycle.

The destination register

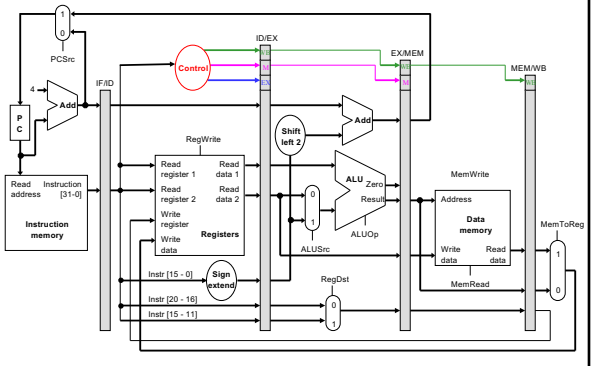


What about control signals?

- Control signals generated similar to the single-cycle processor
  - in the ID stage, the processor decodes the instruction fetched in IF and produces the appropriate control values
- Some of the control signals will not be needed until later stages
  - These signals must be propagated through the pipeline until they reach the appropriate stage
  - We just pass them in the pipeline registers, along with the data
- Control signals can be categorized by the pipeline stage that uses them

Stage	Control signals needed		
EX	ALUSrc	ALUOp	RegDst
MEM	MemRead	MemWrite	PCSrc
WB	RegWrite	MemToReg	

Pipelined datapath and control



An example execution sequence

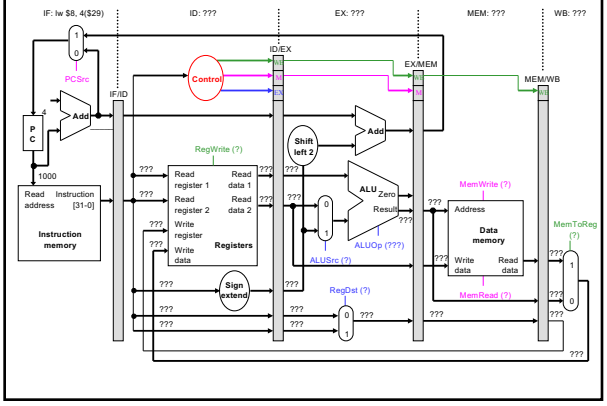
- Here's a sample sequence of instructions to execute.

addresses in decimal

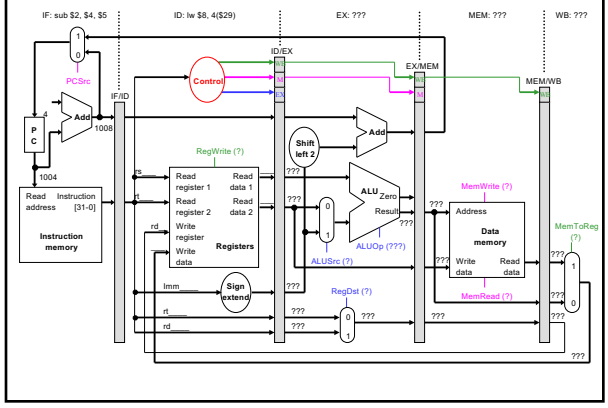
```
1000:  lw   $8, 4($29)
1004:  sub  $2, $4, $5
1008:  and  $9, $10, $11
1012:  or   $16, $17, $18
1016:  add  $13, $14, $0
```

- 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.

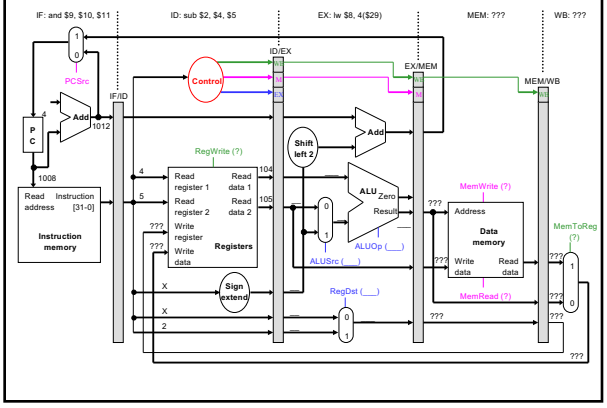
Cycle 1 (filling)



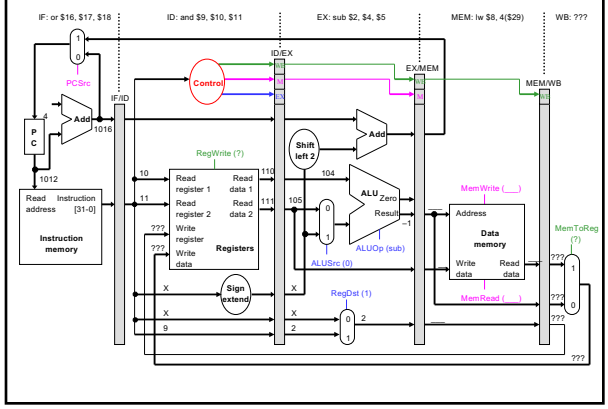
Cycle 2



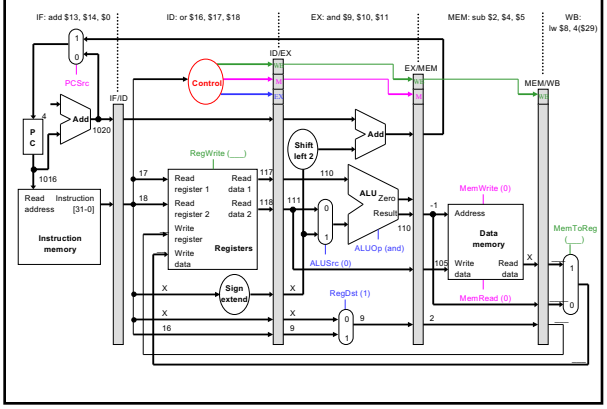
Cycle 3



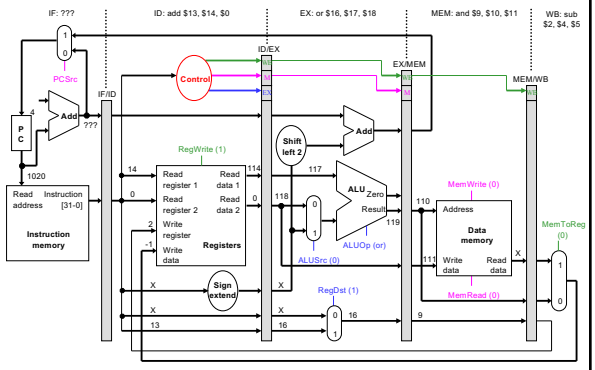
Cycle 4



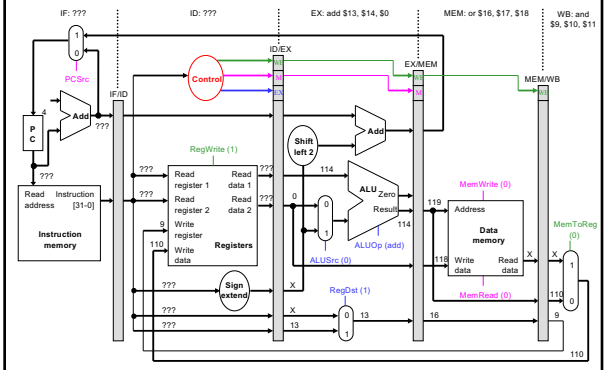
Cycle 5 (full)



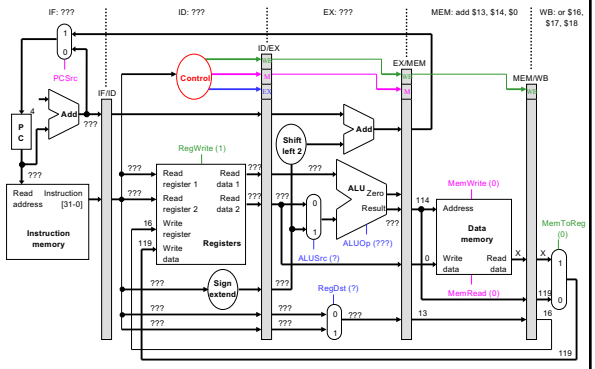
Cycle 6 (emptying)



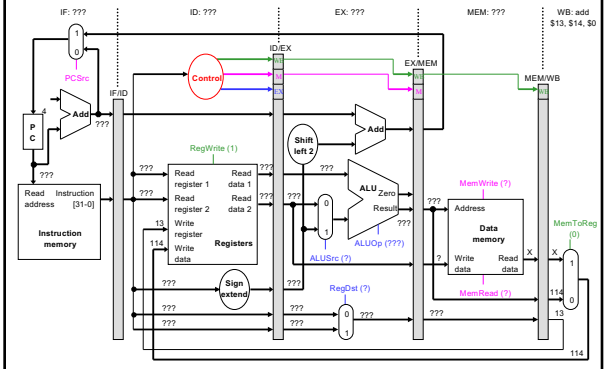
Cycle 7



Cycle 8



Cycle 9



Pipeline diagrams provide a quick snapshot

		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

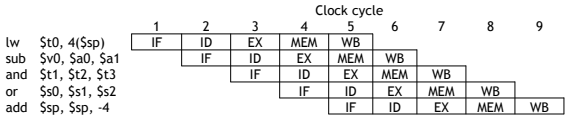
- Compare the last few slides with the pipeline diagram above.
  - You can see how instruction executions are overlapped.
  - Each functional unit is used by a *different* instruction in each cycle.
  - The pipeline registers save control and data values generated in previous clock cycles for later use.
  - 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.

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 can be done in 1ns, but we must stretch that out to 2ns to keep the ID and WB stages synchronized with IF, EX and MEM.
  - Balancing the stages is one of the many hard parts in designing a pipelined processor.

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 (10ns vs. 8ns)!
- Instead, pipelining increases the throughput, or the amount of work done per unit time. Here, several instructions are executed together in each clock cycle.

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.

Summary

- The pipelined datapath extends the single-cycle processor that we saw earlier to improve instruction throughput.
  - Instruction execution is split into several stages.
  - Multiple instructions flow through the pipeline simultaneously.
- Pipeline registers propagate data and control values to later stages.
- The MIPS instruction set architecture supports pipelining with uniform instruction formats and simple addressing modes.

54

Reading

- **5th Edition: 4.5,4.6**

55