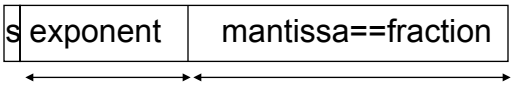
Review

- Floating point evaluates $(-1)^{S} \cdot F \cdot 2^{E}$ using values from a 32-bit word:

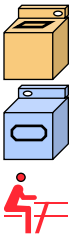


- What are the values for double precision
- If the field is x bits, what is the zero in biased notation?
- How can you tell if a float is negative?

A relevant question

Assuming you've got:

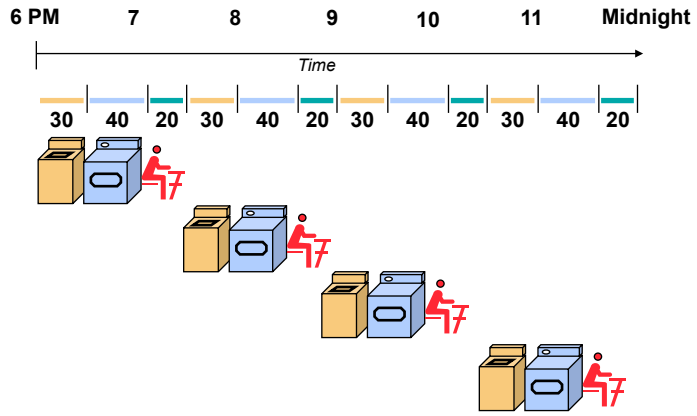
- One washer (takes 30 minutes)
- One drier (takes 40 minutes)
- One "folder" (takes 20 minutes)



It takes 30 + 40 + 20 = 90 minutes to wash, dry, and fold 1 load of laundry.

- How long do 4 loads take?

The slow way



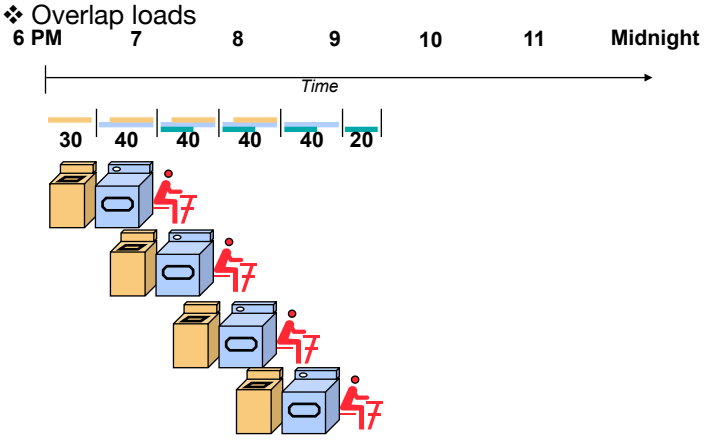
If each load is done sequentially it takes 6 hours

3

Laundry Pipelining

Start each load as soon as possible

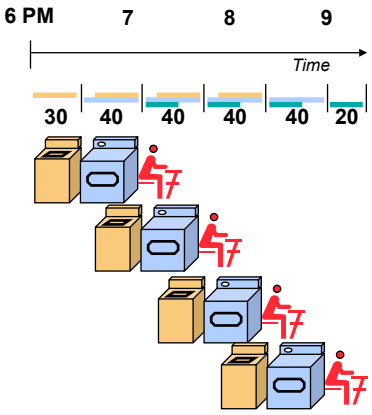
- Overlap loads



Pipelined laundry takes 3.5 hours

4

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

5

Pipelining Processors

We've seen 2 possible implementations of the MIPS architecture

- A single-cycle datapath executes each instruction in just one clock cycle, but the cycle time may be very long
- A multicycle datapath has much shorter cycle times, but each instruction requires many cycles to execute

Pipelining gives the best of both worlds and is used in just about every modern processor

- Cycle times are short so clock rates are high
- But we can still execute an instruction in about 1 clock cycle!

Single Cycle Datapath	CPI = 1	Long Cycle Time
Multi-cycle Datapath	CPI = ~4	Short Cycle Time
Pipelined Datapath	CPI = ~1	Short Cycle Time

6

Instruction execution review

Executing a MIPS instruction can take up to five

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

But as we saw, 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

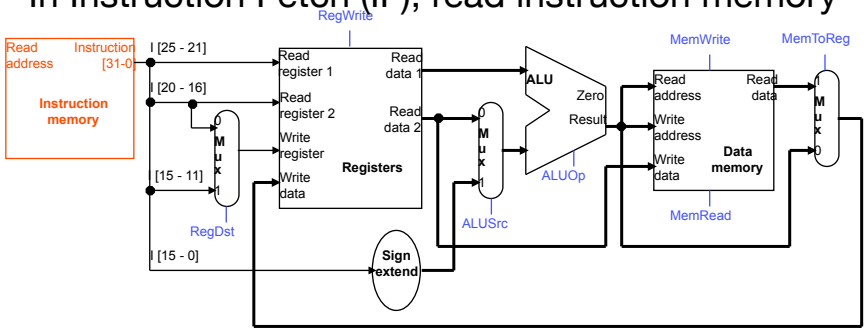
7

Example: Instruction Fetch (IF)

Review how lw executes in the 1-cycle datapath

Ignore PC incrementing and branching for now

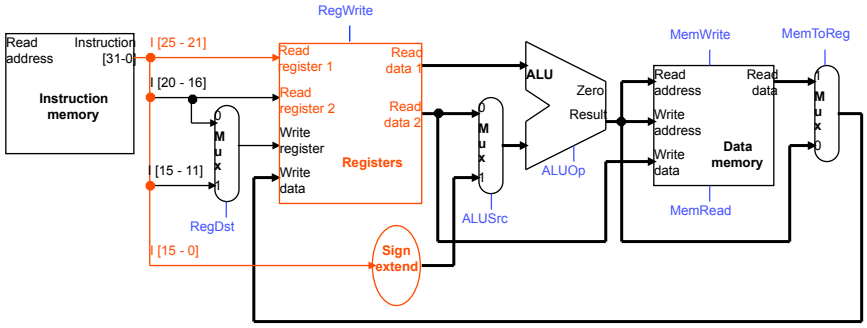
In Instruction Fetch (IF), read instruction memory



8

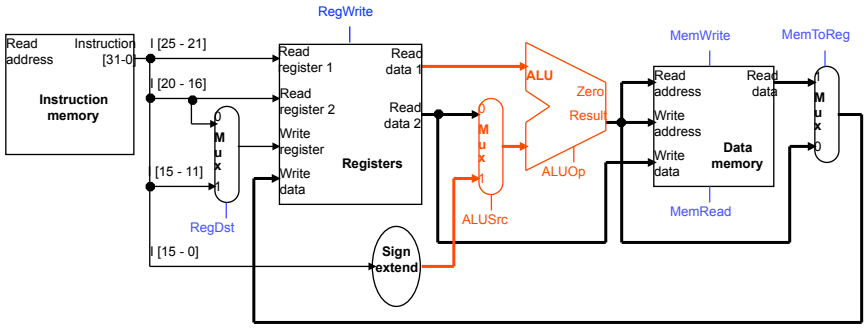
Instruction Decode (ID)

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



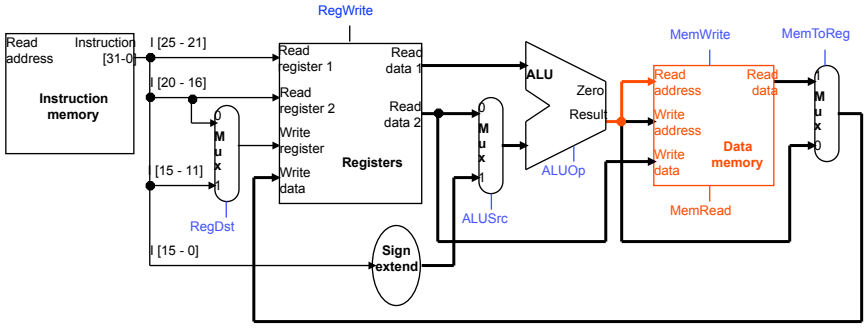
Execute (EX)

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



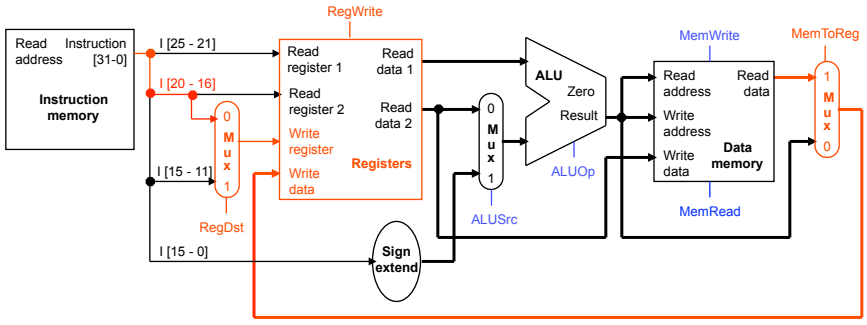
Memory (MEM)

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



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!

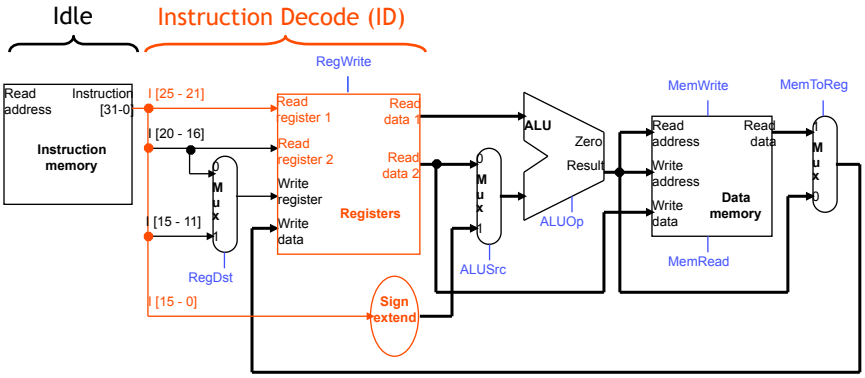
- The instruction RAM is used for just 2ns at the start of the cycle.
- Registers are read once in ID (1ns), & written once in WB (1ns)
- The ALU is used for 2ns near the middle of the cycle
- Reading the data memory only takes 2ns as well

That's a lot of hardware sitting around doing nothing

Putting those slackers to work

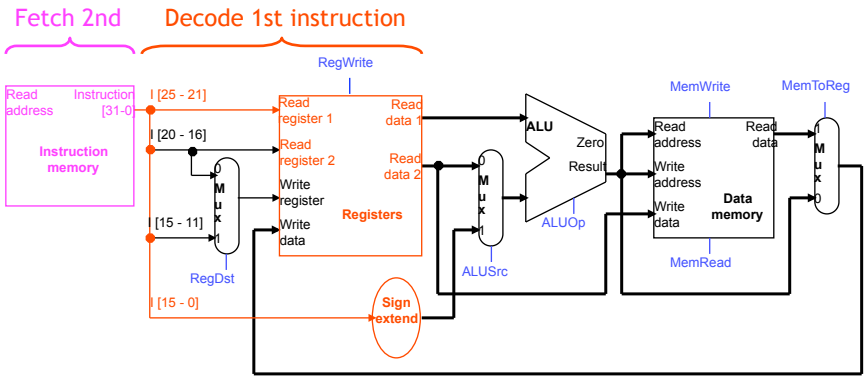
Let's don't wait to complete the instruction before re-using the functional units

Eg., the instruction memory is free in the ID step



Decoding and fetching together

Why not go ahead and fetch the *next* instruction while decoding the first one?

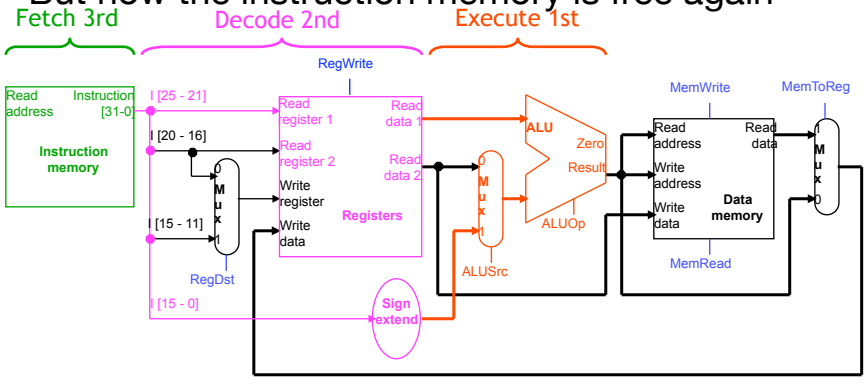


15

Executing, decoding and fetching

Similarly, once the 1st instruction enters EX, decode the second instruction

But now the instruction memory is free again



16

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

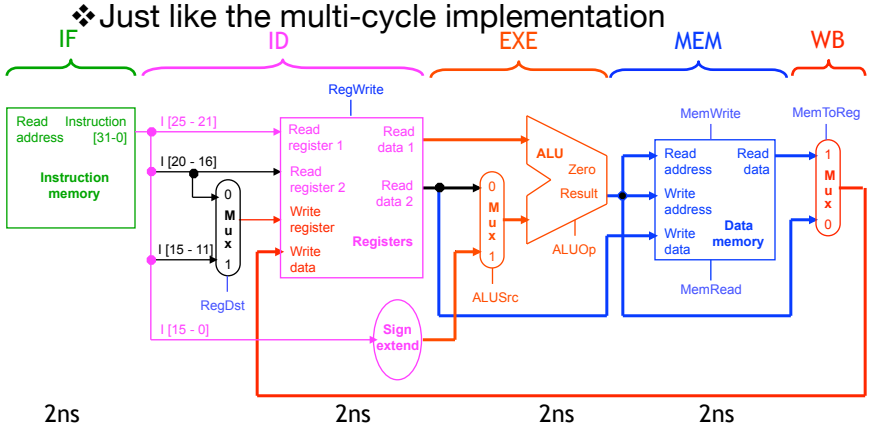
17

Break datapath into 5 stages

Each stage has its own functional units

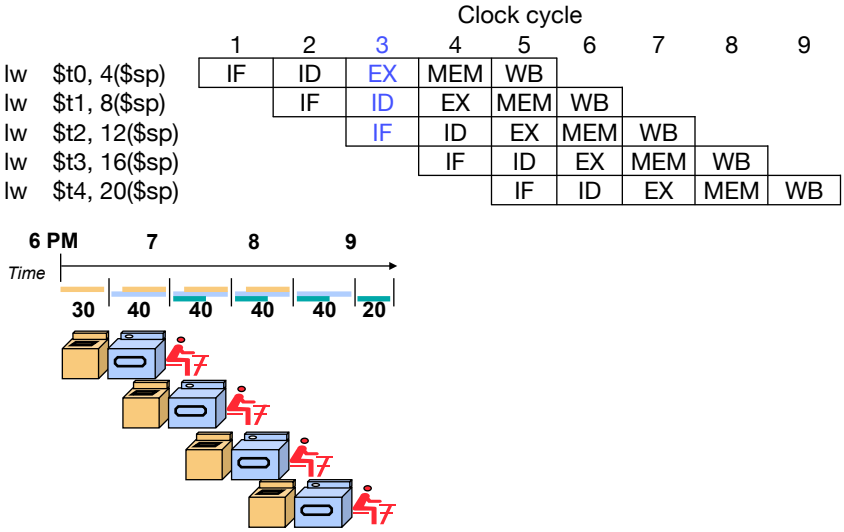
Each stage can execute in 2ns

- Just like the multi-cycle implementation

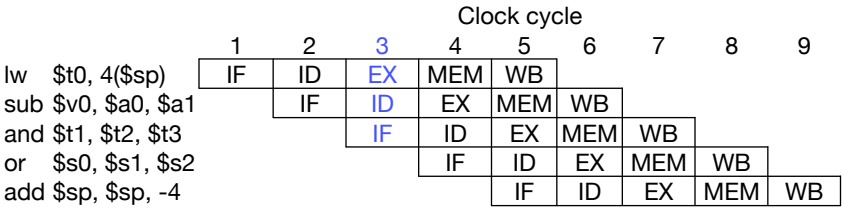


18

Pipelining Loads



A pipeline diagram



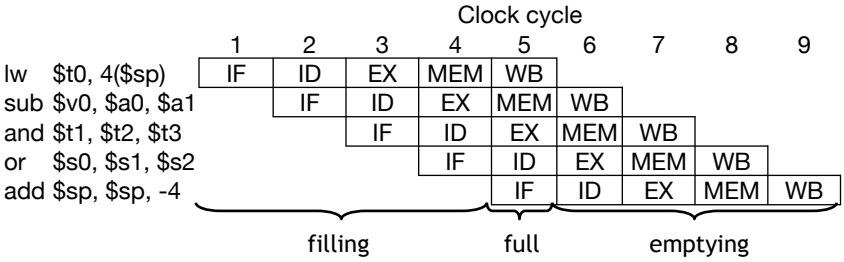
A pipeline diagram shows execution of an series insts.

- The instruction sequence is shown vertically, 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.
Eg., 3 instructions are 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—here it's 5

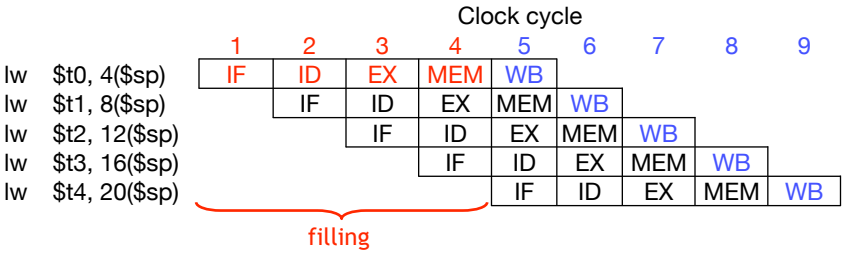
In the first 4 cycles here, the pipeline is filling, since there are unused functional units

In cycle 5, the pipeline is full. Five instructions are being executed simultaneously, so all hardware is in use

In cycles 6-9, the pipeline is emptying

21

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

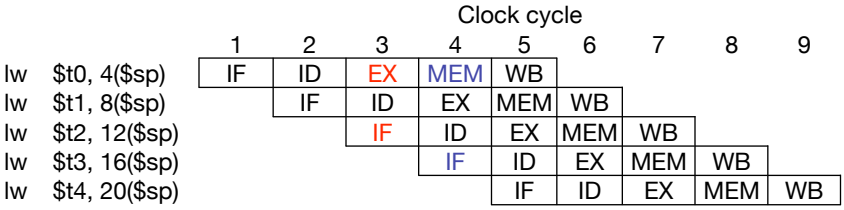
Compare with other implementations:

- Single Cycle: N cycles or 8N ns for 8ns clock period
- Multicycle: CPI * N cycles or ~8N ns for 2ns clock period and CPI = ~4

How much faster is pipelining for N=1000 ?

22

Pipeline Datapath: Resource Needs



We need to perform several operations in each cycle

- Increment the PC and add registers at the same time
- Fetch one instruction while another reads or writes data

Thus, like the single-cycle datapath, a pipelined processor duplicates hardware elements that are needed several times in the same clock cycle.

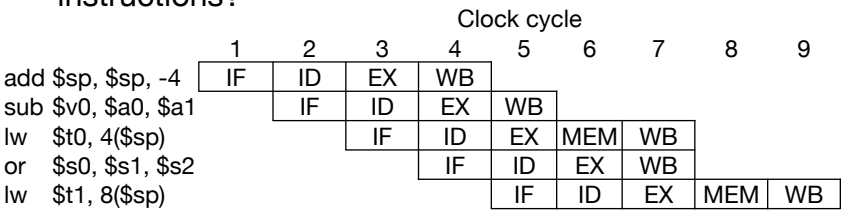
23

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?



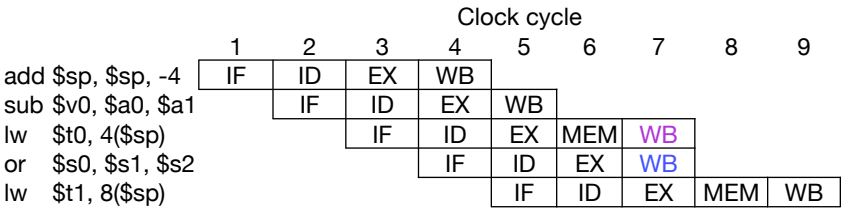
24

Important Observation

Each functional unit can be used only once per instr

Each functional unit must be used at the same stage for all instructions. See the problem if:

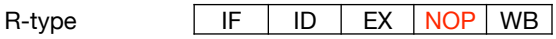
- Load uses Register File's Write Port during its 5th stage
- R-type uses Register File's Write Port during its 4th stage

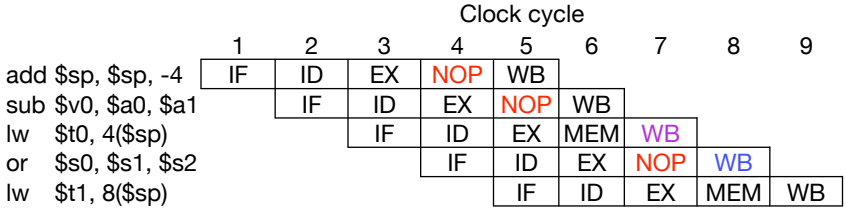


A solution: Insert NOP stages

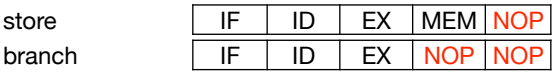
Enforce uniformity

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





  - Stores and Branches have NOP stages, too…



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

27