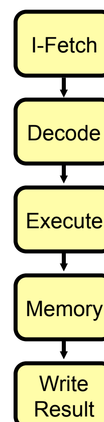
Lecture 11: A Simple Datapath & Pipelining

- Last time
  - Exam discussion (average 73 before regrade)
  - Broke down execution & state (IF,ID,EX,MEM,WB)
    - PC state
    - By instruction type: Control, Register, Memory
- Today
  - Take QUIZ 7 over P&H 4.5-6, before 11:59pm today
  - Homework 4 due Thursday March 4
  - Putting the parts together
  - Logic & control
  - How can we execute instructions faster?
    - multicycle execution
    - pipelining

5 Stages for Multicycle Execution

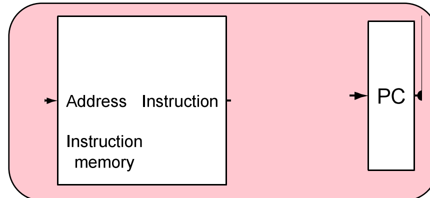
5 logical and distinct steps

IF: fetch instruction
ID/R: decode instruction and read registers
EX: execute (add, sub, …)
MEM: access memory
WB: store result (write back)

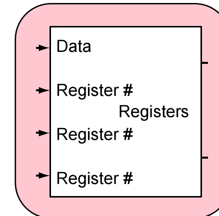


What do we need to execute instructions?

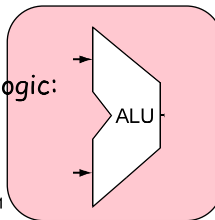
Which instruction?
• instruction memory, PC: `beq, j`

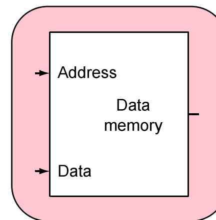


Which registers?
• register storage file



Which Math?
• combinational logic:
`add, sub, and, or, slt`



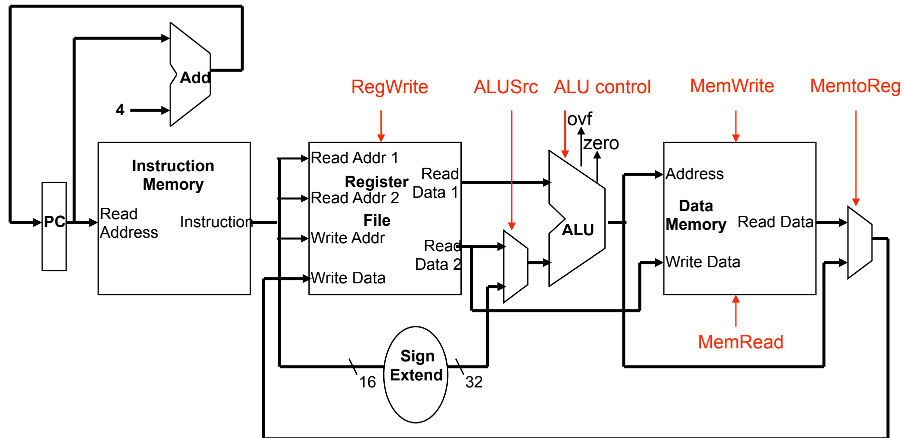


Which data?
• memory:
`lw, sw`

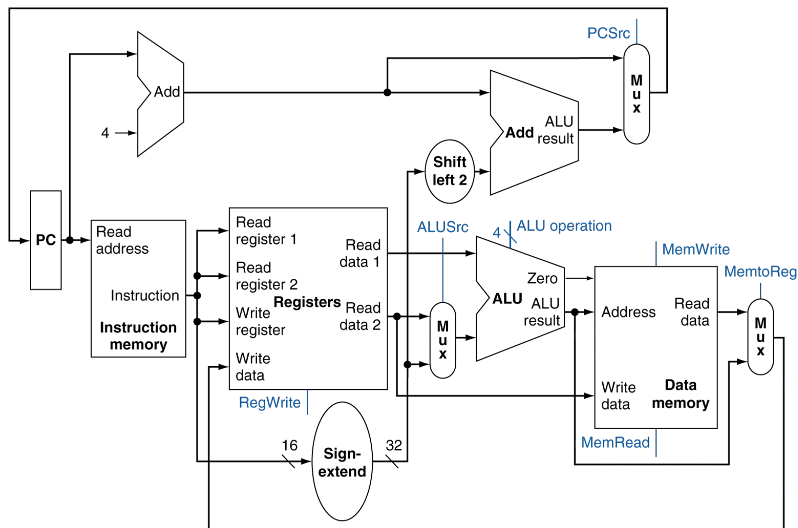
Creating a Datapath from the Parts

- Assemble the datapath segments, add control lines, and multiplexors
- Single cycle design – fetch, decode and execute each instructions in one clock cycle
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)
  - multiplexors (mux) needed at the input of shared elements with control lines to do the selection
  - write signals to control writing to the Register File and Data Memory
- Cycle time is determined by length of the longest path

Simplified Datapath
Fetch, R, and Memory Access Portions



More Datapath with Multiplexors



What Do We Control and How?

Branch
M u x
4
Add
Add
M u x
ALU operation
PC
Address Instruction
Instruction memory
Data
Register #
Registers
Register #
Register # RegWrite
M u x
ALU
Zero
MemWrite
Address
Data memory
Data MemRead
Control

The Main Control Unit

- Control signals derived from instruction

R-type	0	rs	rt	rd	shamt	funct
	31:26	25:21	20:16	15:11	10:6	5:0
Load/ Store	35 or 43	rs	rt	address		
	31:26	25:21	20:16	15:0		
Branch	4	rs	rt	address		
	31:26	25:21	20:16	15:0		

opcode

always read

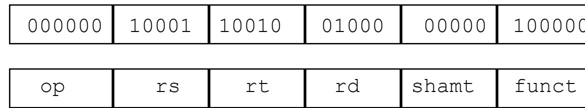
read, except for load

write for R-type and load

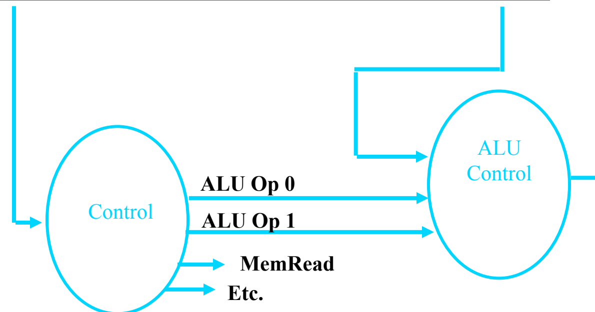
sign-extend and add

How do we convert instruction bits to ALU control bits?

Example: add $8, $17, $18

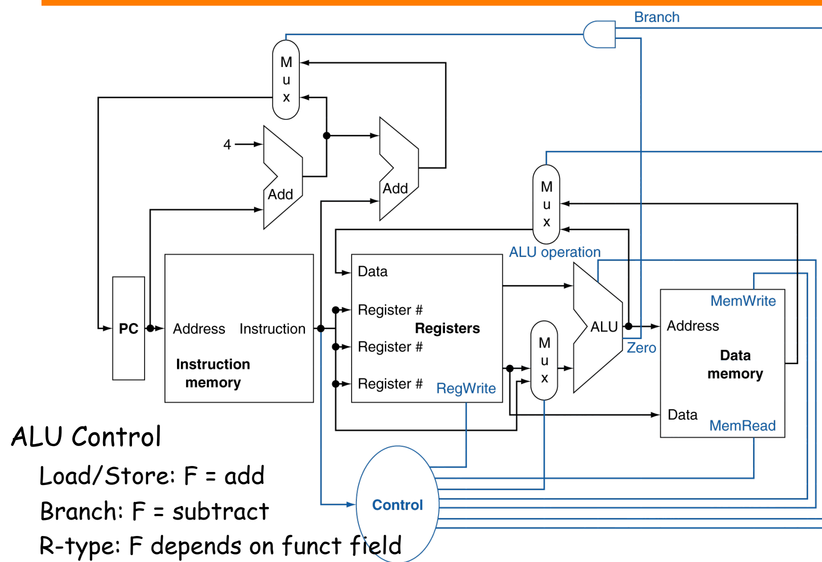


000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct



0000 AND
0001 OR
0010 add
0110 subtract
0111 set-on-less-than
1100 NOR

ALU Active on All Instructions



ALU Control

Load/Store: F = add

Branch: F = subtract

R-type: F depends on funct field

ALU Control

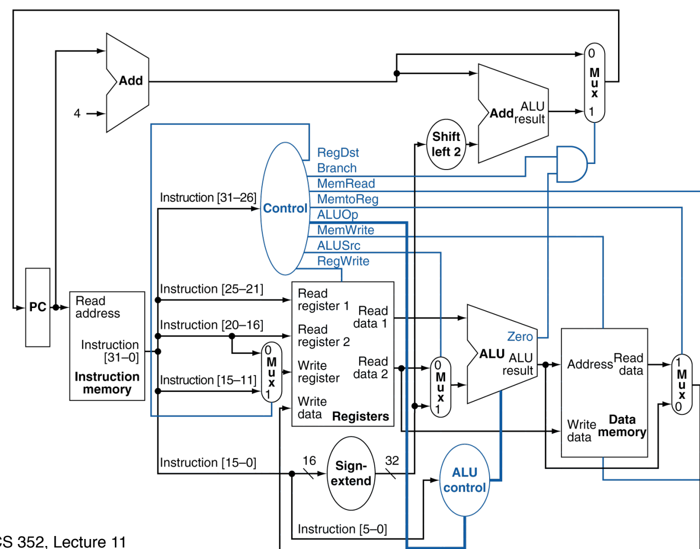
2-bit ALUOp **derived** from opcode

- Combinational logic derives ALU control

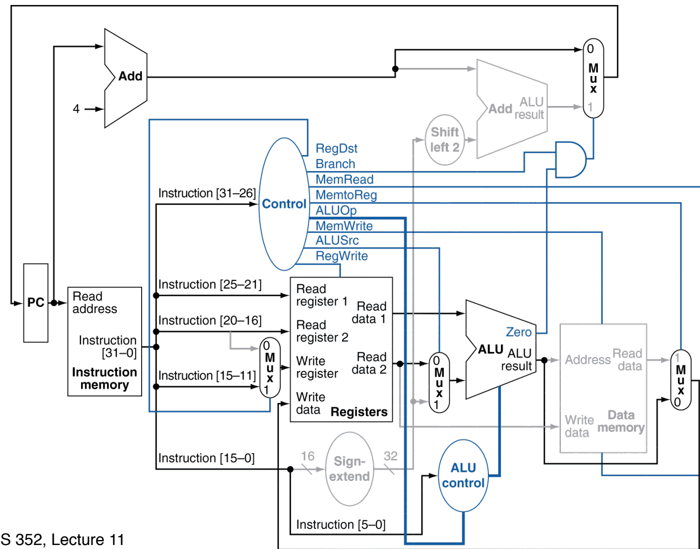
4-bit ALU control **derived** from opcode

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

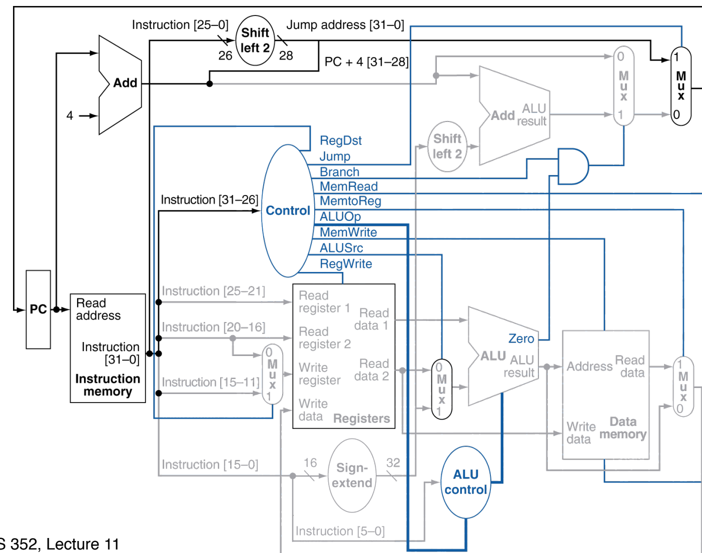
Datapath With Control



R-Type Instruction



Datapath With Jumps Added

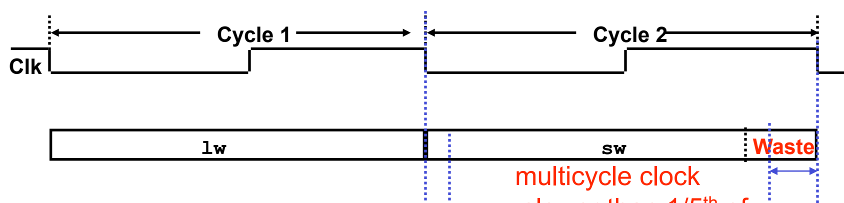


Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period based on instruction
- Violates design principle
  - Making the common case fast
- **Motivates multiple (5) steps**

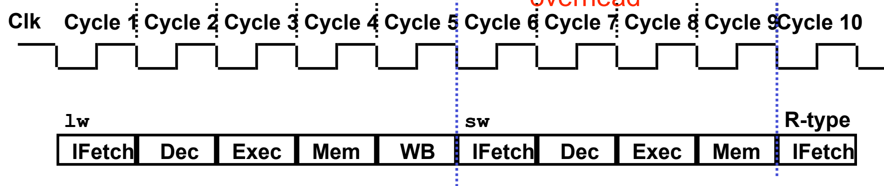
Single Cycle vs. Multiple Cycle Timing

Single Cycle Implementation:

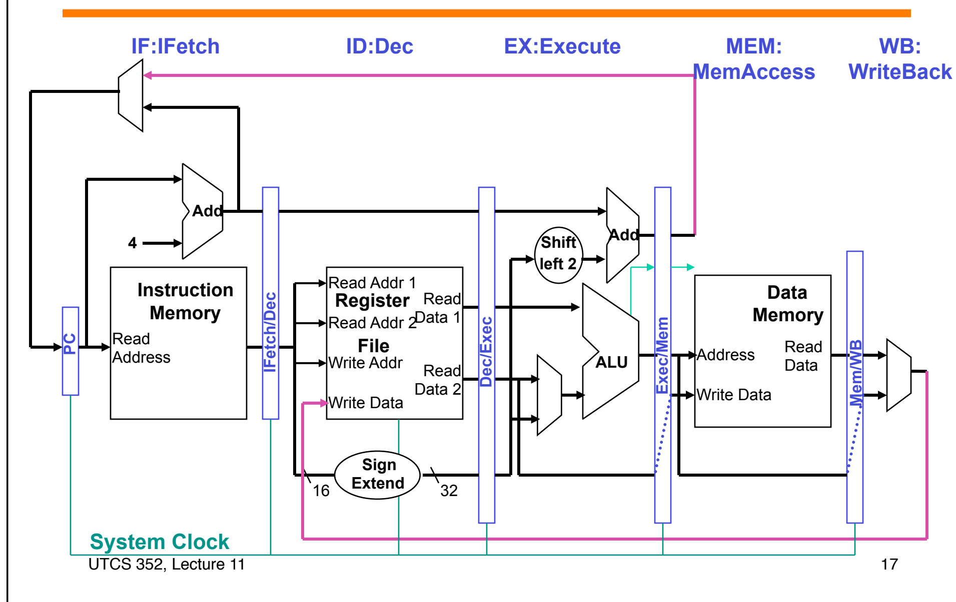


multicycle clock slower than 1/5th of single cycle clock due to state register overhead

Multiple Cycle Implementation:



MIPS Multicycle Datapath
logical division of states

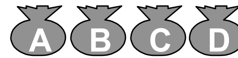


How Can We Make it Faster?

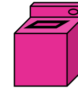
- Split the multiple instruction cycle into smaller and smaller steps
  - Point of diminishing returns where as much time is spent loading the state registers as doing the work
- Start fetching and executing the next instruction before the current one has completed
  - Pipelining – (all?) modern processors are pipelined for performance
  - Remember *the* performance equation:
    CPU time = IC * CPI * CCT
- Fetch & execute more than one instruction at a time!
  - Superscalar processing – stay tuned

Pipelining is Natural!

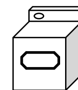
Laundry Example



- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes



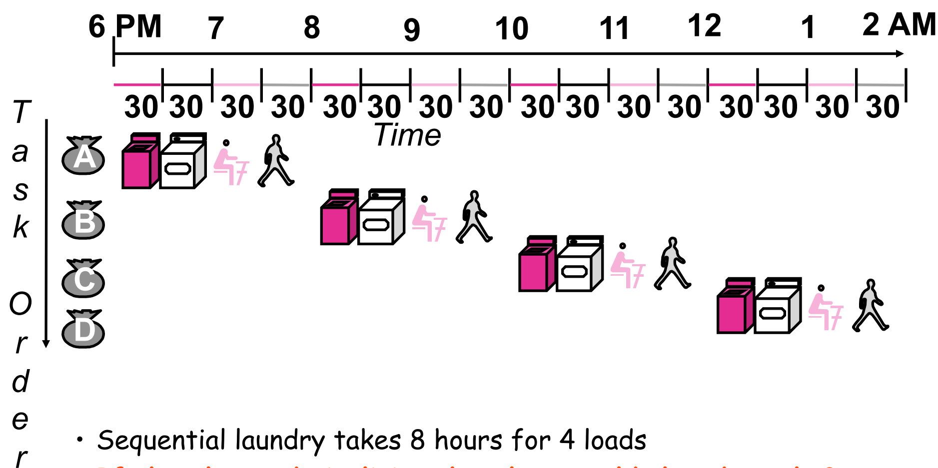
- Dryer takes 30 minutes



- "Folder" takes 30 minutes

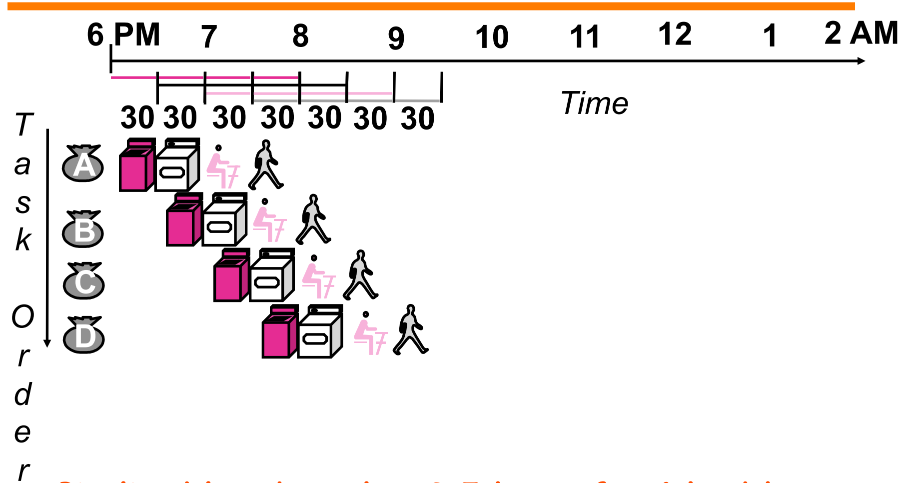
- "Stasher" takes 30 minutes to put clothes into drawers

Sequential Laundry



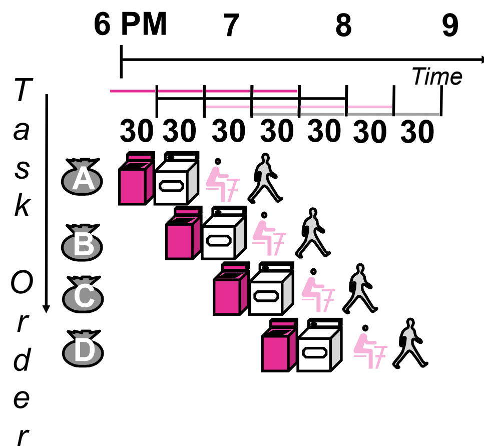
- Sequential laundry takes 8 hours for 4 loads
- **If they learned pipelining, how long would laundry take?**

Pipelined Laundry: Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads!

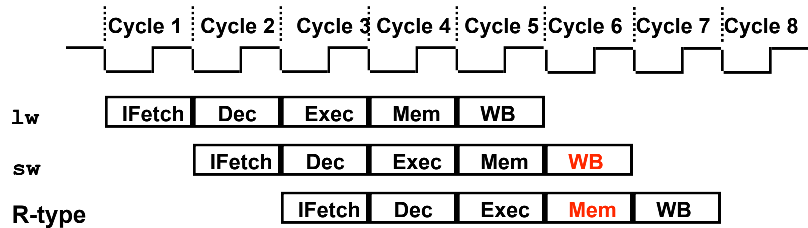
Pipelining Lessons



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependences

A Pipelined MIPS Processor

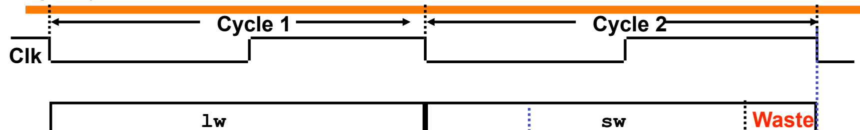
- Start the next instruction before the current one has completed
  - improves throughput - total amount of work done in a given time
  - instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced



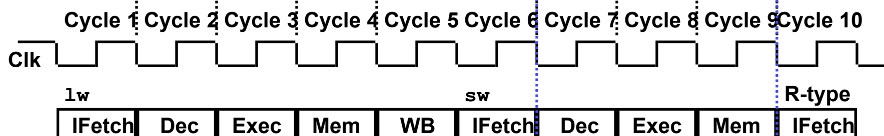
- clock cycle (pipeline stage time) is limited by the slowest stage
- for some instructions, some stages are wasted cycles

Single Cycle, Multiple Cycle, vs. Pipeline

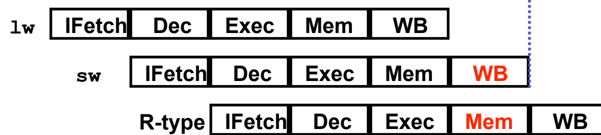
Single Cycle Implementation:

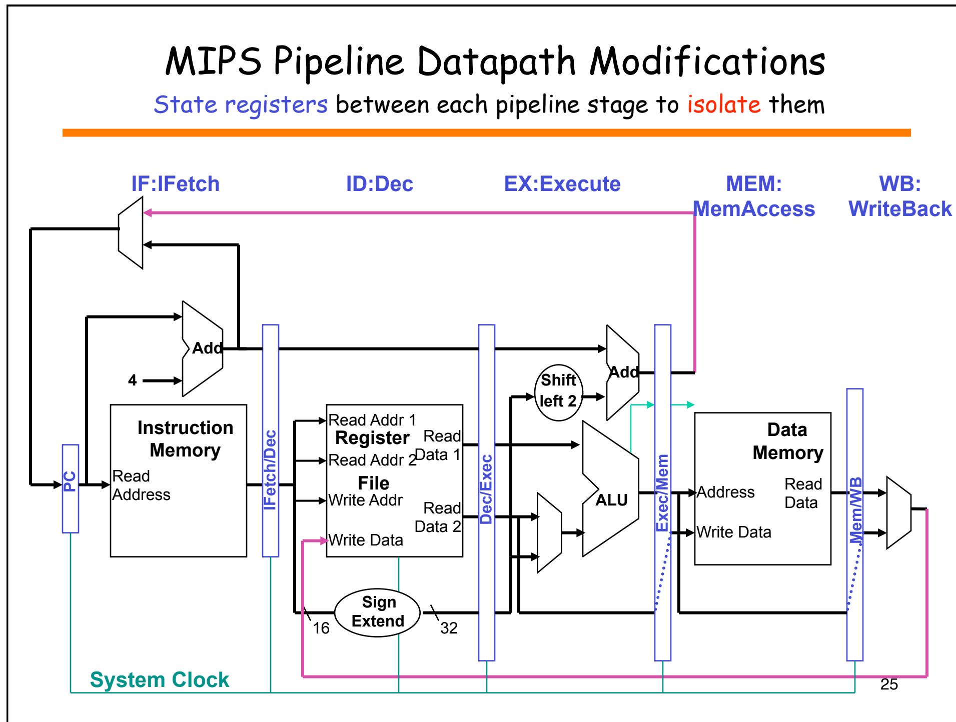


Multiple Cycle Implementation:



Pipeline Implementation:





Pipelining the MIPS ISA

- What makes it easy
  - all instructions are the same length (32 bits)
    - can fetch in the 1st stage and decode in the 2nd stage
  - few instruction formats (three) with symmetry across formats
    - can begin reading register file in 2nd stage
  - memory operations can occur only in loads and stores
    - can use the execute stage to calculate memory addresses
  - each MIPS instruction writes at most one result (i.e., changes the machine state) and does so near the end of the pipeline (MEM and WB)
- What makes it hard
  - structural hazards: what if we had only one memory?
  - control hazards: what about branches?
  - data hazards: what if an instruction's input operands depend on the output of a previous instruction?

How Much Performance?

- If all stages are balanced (all take the same time)
- Ideal Speedup = Instructions/Number of Stages
- Why it is never ideal:
  - Stages are never perfectly balanced
  - Pipeline fill and drain
  - Breaking down of instructions into stages adds time to each stage:
    - Time between $\text{stages}_{\text{pipelined}}$ > Time between $\text{instructions}_{\text{nonpipelined}}$
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Summary

- Simplistic version of pipelining
  - Pipelining improves instruction throughput
  - Longest stage determines latency
- Next Time
  - Realistic version of Pipelining
    - Hazards & forwarding
  - Homework 4 is due Thursday March 4, 2010
- Reading: P&H 4.7-10