Lecture 8

Simple & Pipelined Processor Designs

Christos Kozyrakis
Stanford University
http://eeclass.stanford.edu/ee108b

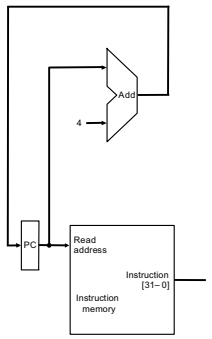
Announcements

- Upcoming deadlines
  - HW2 due today
  - PA1 due on Thursday 2/8
- Quiz 1 review session on Friday
- Quiz 1
  - Tue 2/6, 7pm–9pm, location TBD
  - Local SCPD students must come to Stanford for the midterm
  - Covers lectures 1-7
  - Closed book, 1 page of notes + green card, calculator

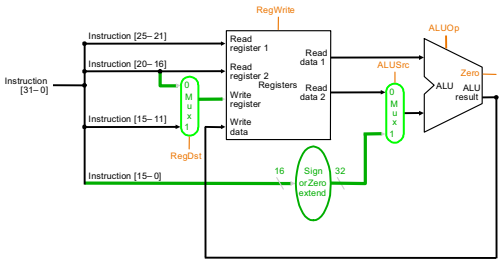
Review: How to Execute Instructions

- First we need to:
  - Fetch the instruction
- Then we need to:
  - Decode instruction / fetch register
- Then we need to:
  - Do the operation
- Then we need to:
  - Write the result into register-file
- Finally we need to:
  - Calculate the next instruction address

Review: Datapath for Instruction Fetch Unit

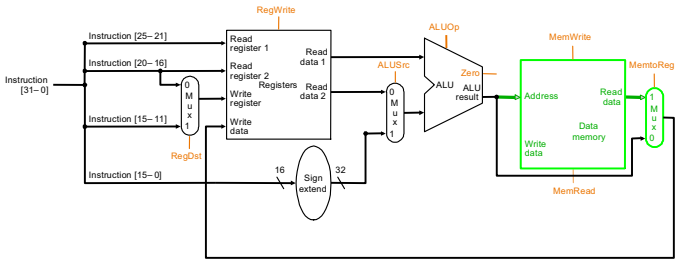


Review: Datapath for Arithmetic & Logical Instructions



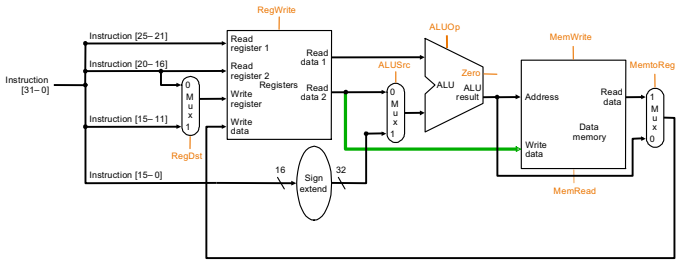
Review: Load Datapath

- Extend datapath to support other immediate operations
- Extender handles either sign or zero extension
- MUX selects between ALU result and Memory output

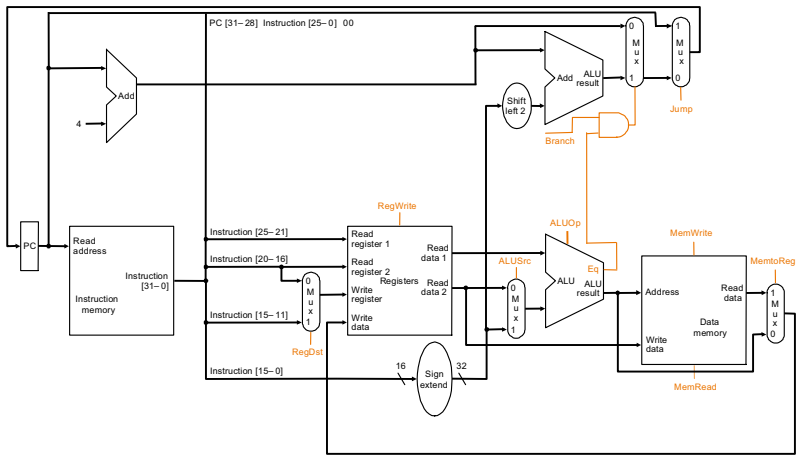


Review: Store Datapath

- Read Register 2 is passed on to Memory
- Memory address calculated just as in lw case



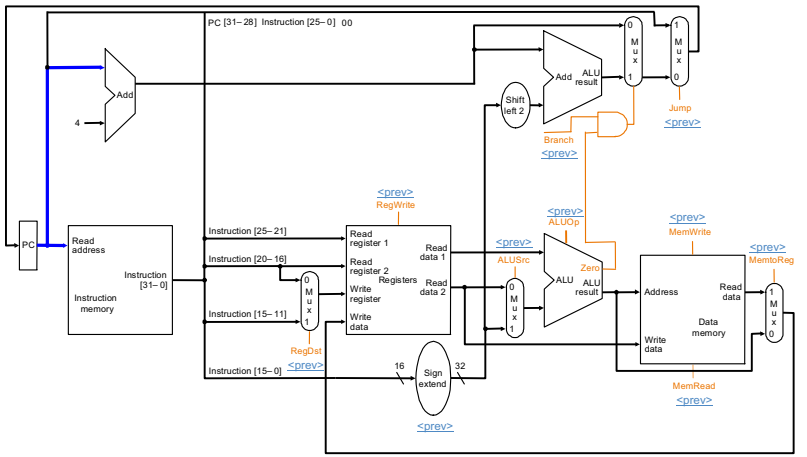
Putting it All Together



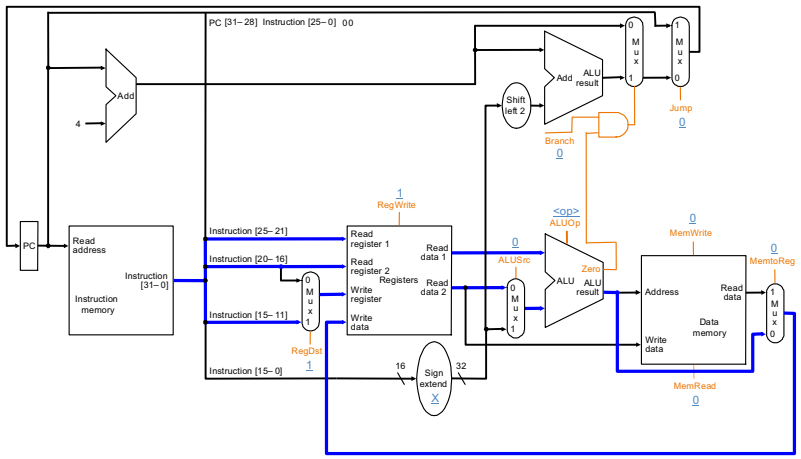
Control

- Since every instruction takes one cycle, control is state free!
  - It is just decoded instruction bits
- There are also few control points
  - Control on the multiplexers
  - Operation type for the ALU
  - Write control on the Instruction & Data memories
- First part of cycle does not have any control
  - Which is good, since we don't have instruction yet
- Look at setting of the control points for different instructions

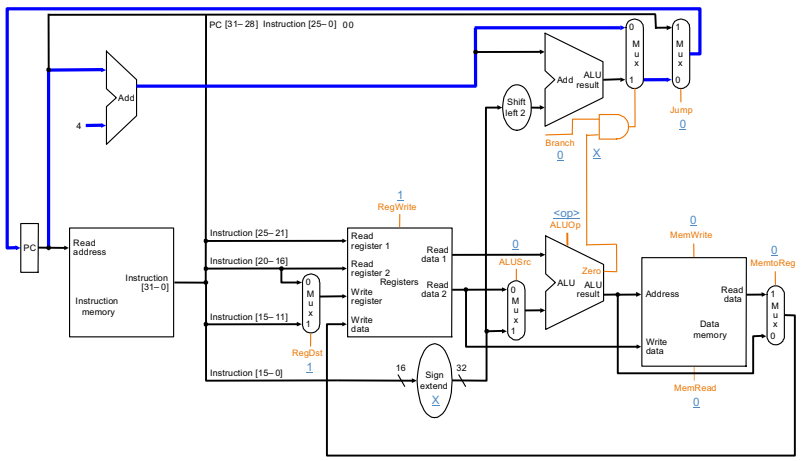
At Beginning Of Clock Cycle



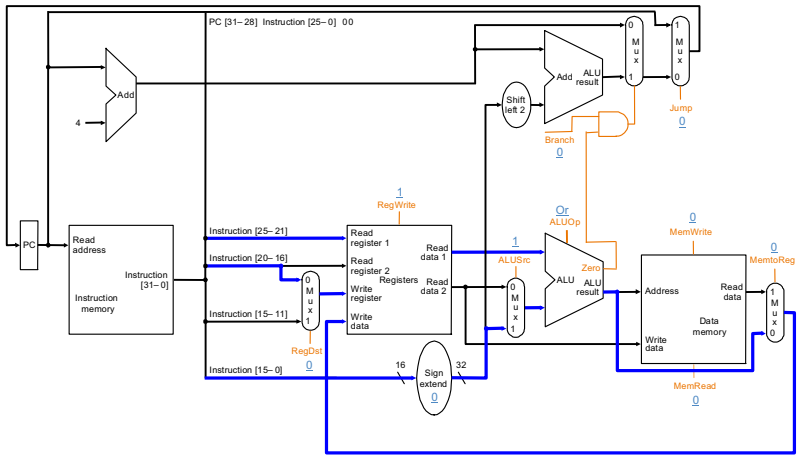
Control for Arithmetic



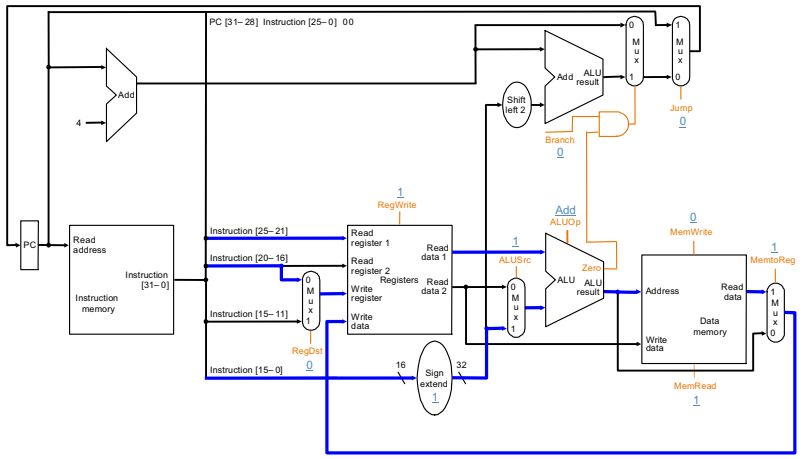
Instruction Fetch at End



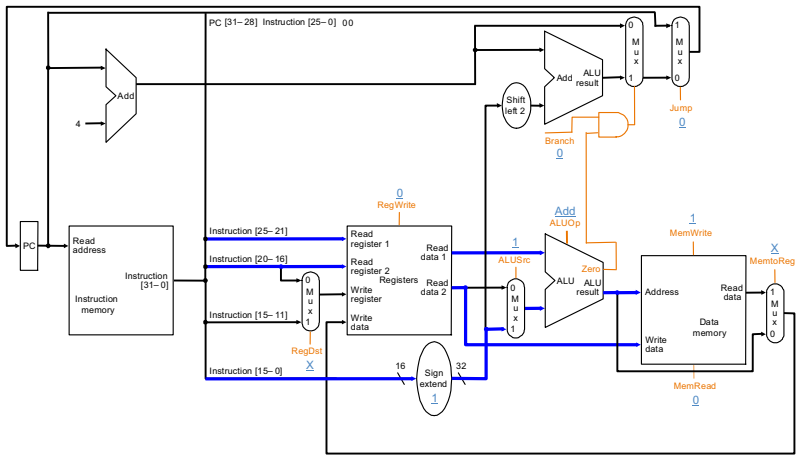
Arithmetic Immediate (`ori`)



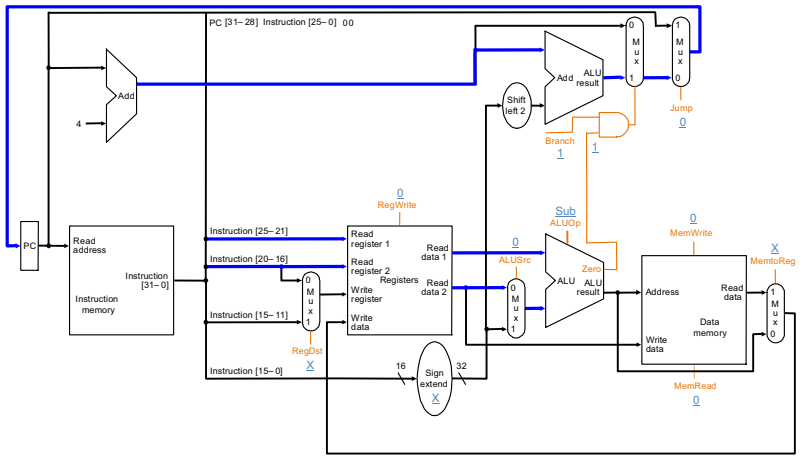
Control for Load



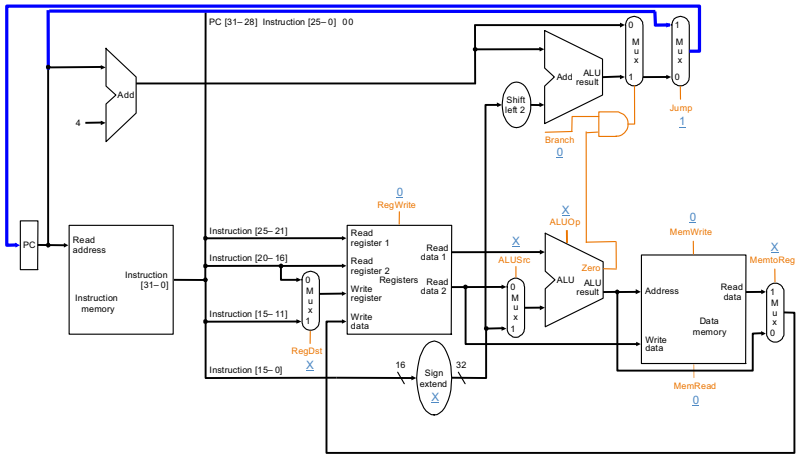
Control for Store



Control for Branch (`beq`)



Control for Jump (j)

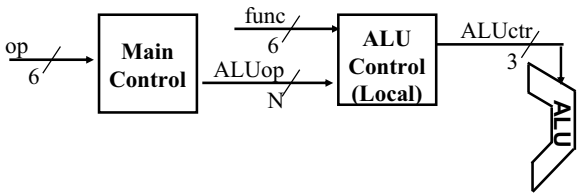


Summary of Control Signals

func	10 0000	10 0010	Not Important				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	**sub**	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
Branch	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x
ALUctr<2:0>	Add	Sub	Or	Add	Add	Sub	xxx

Multilevel Decoding

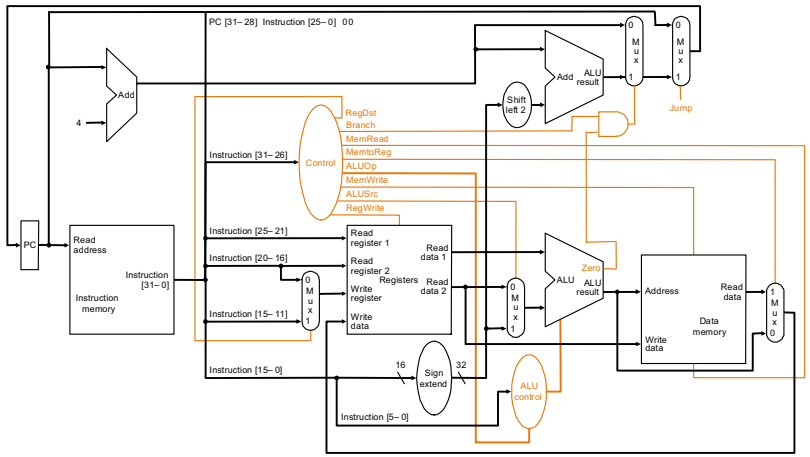
- Since only the ALU needs the func field
  - Pass it to the ALU unit, and have a local decoder there



Multilevel Decoding (cont)

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<N:0>	"R-type"	Or	Add	Add	Subtract	xxx

Putting It All Together



Single Cycle Processor

- Advantages
  - Single cycle per instruction makes logic and clock simple
- Disadvantages
  - Inefficient utilization of memory and functional units since different instructions take different lengths of time
    - ALU only computes values a small amount of the time
  - Cycle time is the worst case path → long cycle times
    - Load instruction
  - All machines would have a CPI of 1

Single Cycle Processor Performance

- Functional unit delay
  - Memory: 2ns
  - ALU and adders: 1 ns
  - Register file: 0.5 ns

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	2	0.5	1		0.5	4
load	2	0.5	1	2	0.5	6
store	2	0.5	1	2		5.5
branch	2	0.5	1			3.5
jump	2					2

- CPU clock cycle = 6 ns

Variable Clock Single Cycle Processor Performance

- Instruction Mix
  - 45% ALU
  - 25% loads
  - 10% stores
  - 15% branches
  - 5% jumps

Instr. class	Instr. memory	Reg. read	ALU op	Data memory	Register write	Total
R-type	2	0.5	1		0.5	4
load	2	0.5	1	2	0.5	6
store	2	0.5	1	2		5.5
branch	2	0.5	1			3.5
jump	2					2

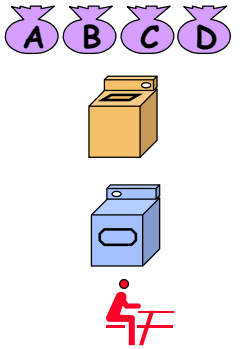
- CPU clock cycle = 4 x 45% + 6 x 25% + 5.5 x 10% + 3.5 x15% + 2 x 5%
  = 4.5 ns

Increasing Parallelism

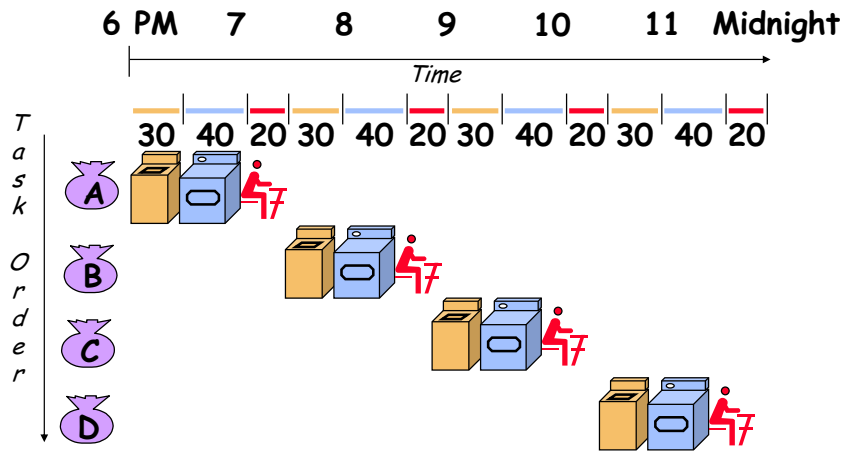
- Problem:
  - Each functional unit used once per cycle
  - Most of the time it is sitting waiting for its turn
    - Well it is calculating all the time, but it is waiting for valid data
  - There is no parallelism in this arrangement
- Making instructions take more cycles can make machine faster!
  - Each instruction takes roughly the same time
    - While the CPI is much worse, the clock freq is much higher
  - Overlap execution of multiple instructions at the same time
    - Different instructions will be active at the same time
  - This is called "Pipelining"
  - We will look at a 5 stage pipeline
    - Modern machines (Pentium 4) have order 20 cycles/instruction

Pipelining: It's Natural and You Do It All the Time!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- "Folding bench" takes 20 minutes

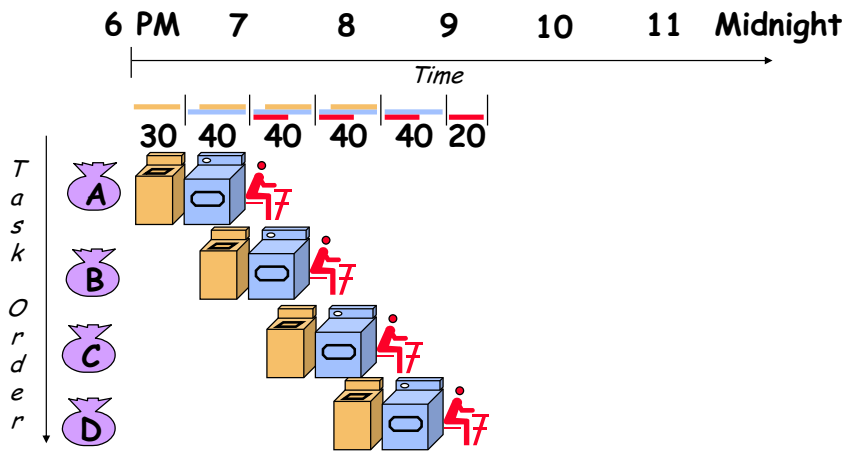


Sequential Laundry



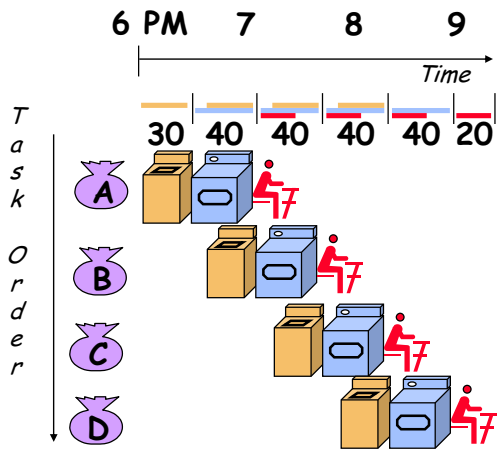
Sequential laundry takes 6 hours for 4 loads

Pipelined Laundry: Start work ASAP



Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons



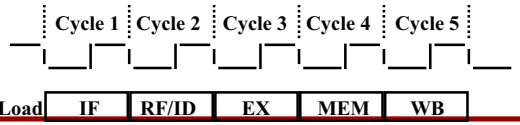
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and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

Dividing Up The Execution

- Since we are going to break execution into different clock cycles
  - We want to balance the work done in each clock
  - Break into a "reasonable" number of cycles
- Imagine we are building a system and know the following:
  - Register file – 1 ns
  - ALU operation – 2 ns
  - Memory access – 2 ns
- How to divide up the cycle?
  - Make memory access one cycle, that is largest factor

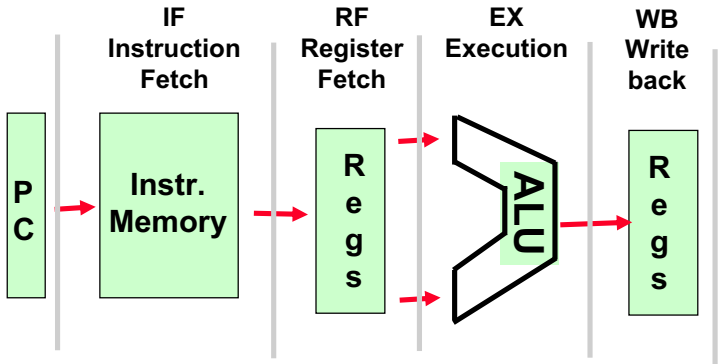
5 Stage Execution

- IF: Instruction Fetch
  - Fetch the instruction from memory
  - Increment the PC
- RF/ID: Register Fetch and Instruction Decode
  - Fetch base register
- EX: Execute
  - Calculate base + sign-extended offset
- MEM: Memory
  - Read the data from the data memory
- WB: Write back
  - Write the results back to the register file



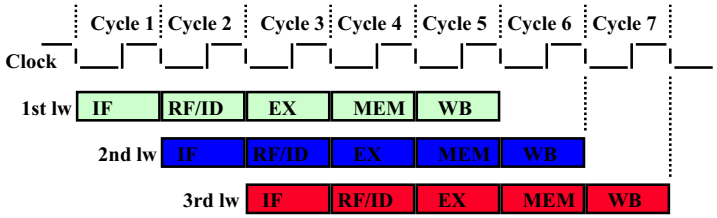
Processor Pipeline

- Fetch a new instruction each cycle
  - Each stage of the pipeline is working on a different instruction



Pipelining Load

- Load instruction takes 5 stages
  - Five independent functional units work on each stage
    - Each functional unit used only once
  - Another load can start as soon as 1st finishes IF stage
  - Each load still takes 5 cycles to complete
  - The *throughput*, however, is much higher

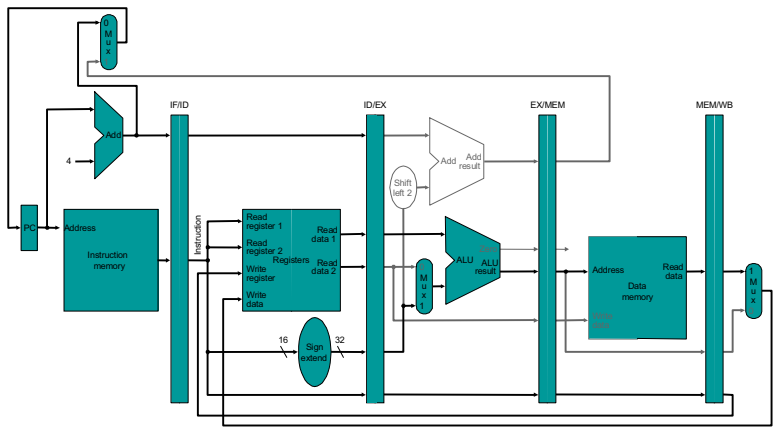


Functional Units Are Busy

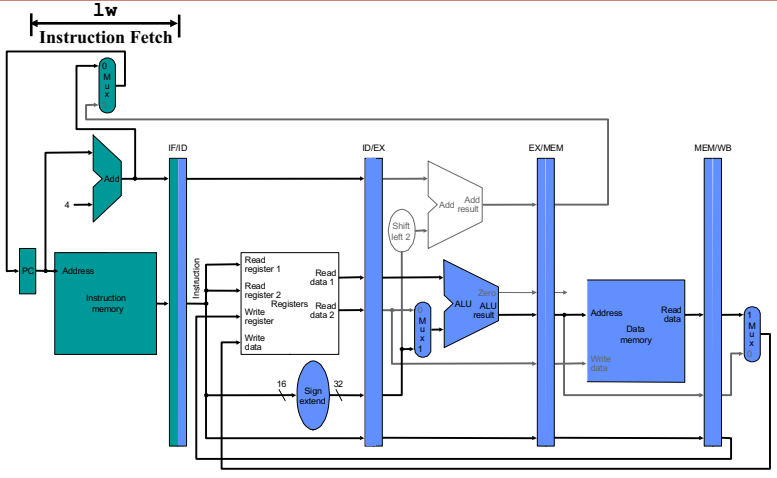
- Pipelining now keeps all the functional units busy
  - Fetch a new instruction each cycle
  - Fetch registers every cycle
  - Use the ALU almost every cycle
  - Use the Data Memory many cycles
- Instructions still take 10ns to complete
  - But start a new instruction every 2ns
  - Looks like CPI is 1 cycle

	1	2	3	4	5	6	7	8	9
I1	IF	ID	EX	MEM	WB				
I2		IF	ID	EX	MEM	WB			
I3			IF	ID	EX	MEM	WB		
I4				IF	ID	EX	MEM	WB	
I5					IF	ID	EX	MEM	WB

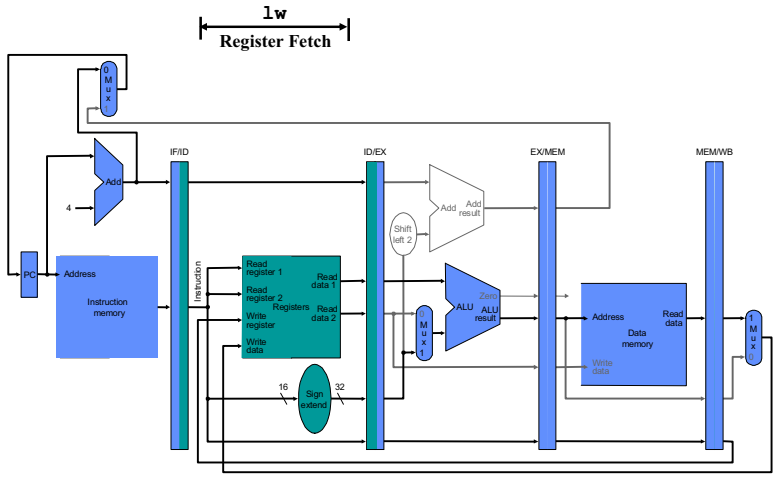
Pipeline Datapath



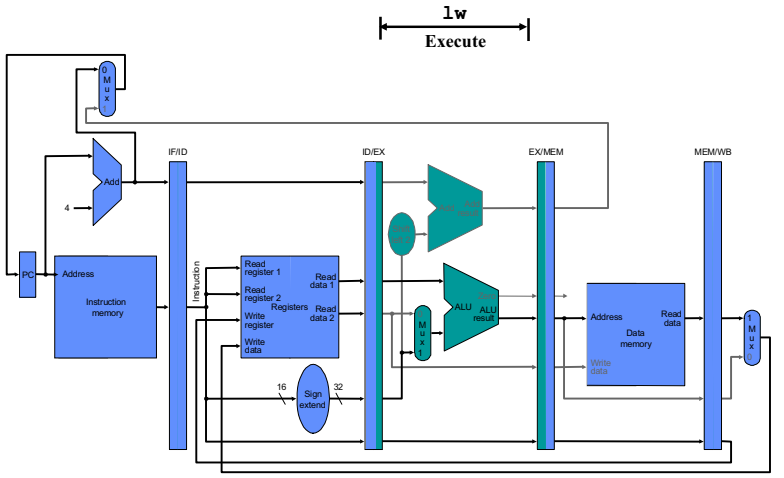
Load Datapath: Stage 1



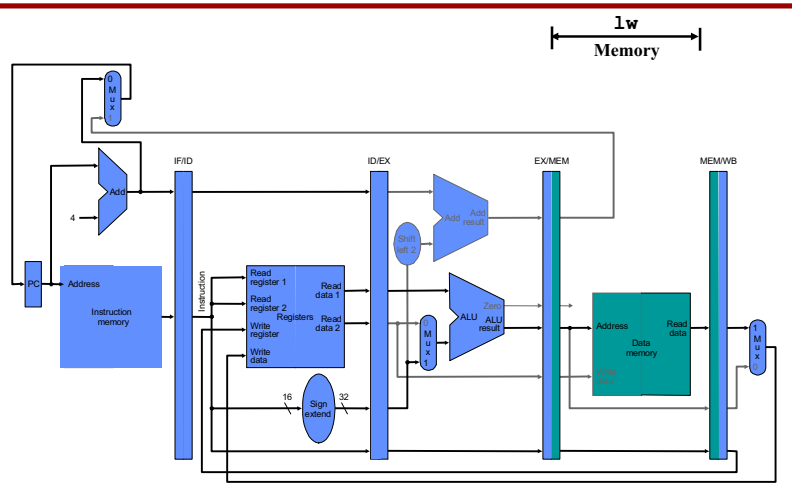
Load Datapath: Stage 2



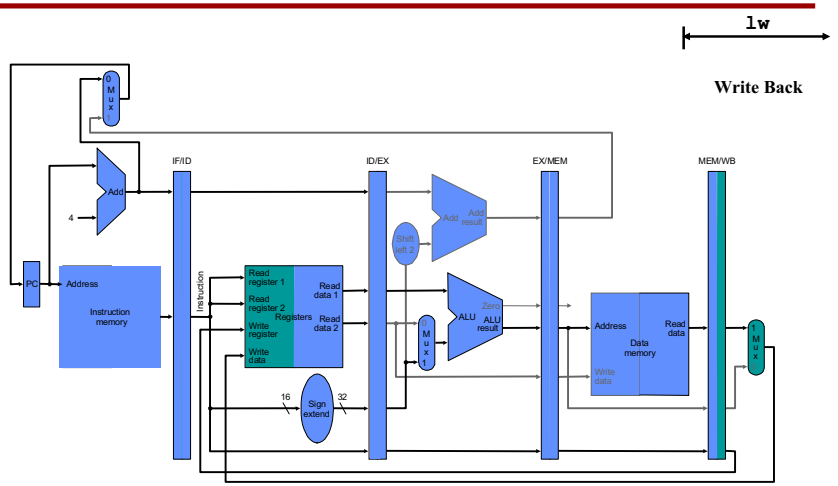
Load Datapath: Stage 3



Load Datapath: Stage 4



Load Datapath: Stage 5



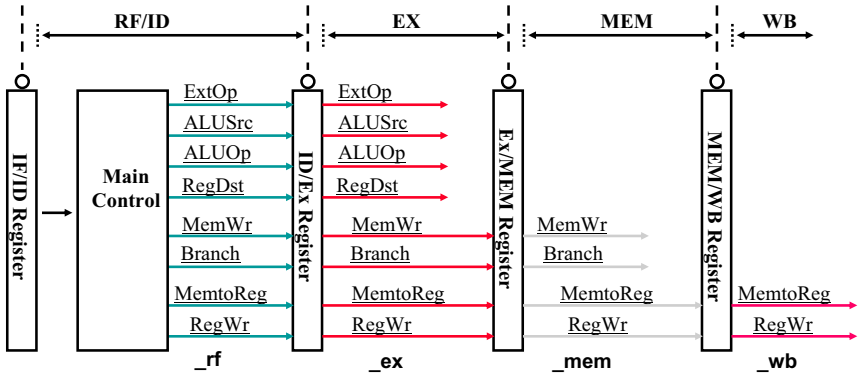
Pipeline Control

- Need to control functional units
  - But they are from working on different instructions!
- Not a problem
  - Just pipeline the control signals along with the data
  - Make sure they line up
- Using labeling conventions often helps
  - Instruction_rf – means this instruction is in RF
  - Every time it gets flopped, changes pipestage
    - Make sure right signals go to the right places

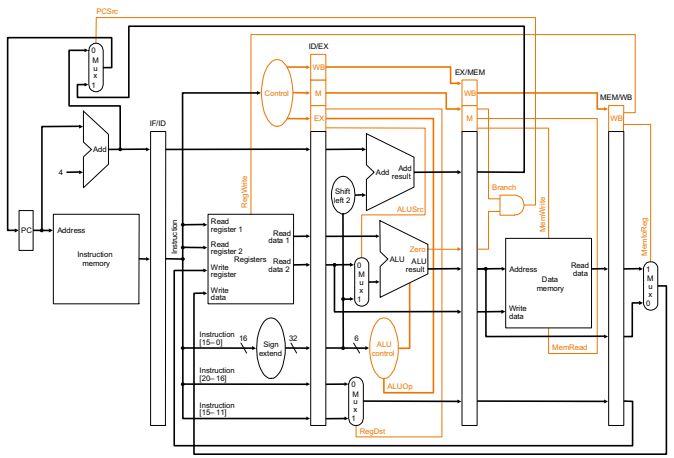
Control Signals

- Use a Main Control unit to generate signals during RF/ID Stage
  - Control signals for EX
    - (ExtOp, ALUSrc, …) used 1 cycle later
  - Control signals for Mem
    - (MemWr, Branch) used 2 cycles later
  - Control signals for WB
    - (MemtoReg, MemWr) used 3 cycles later

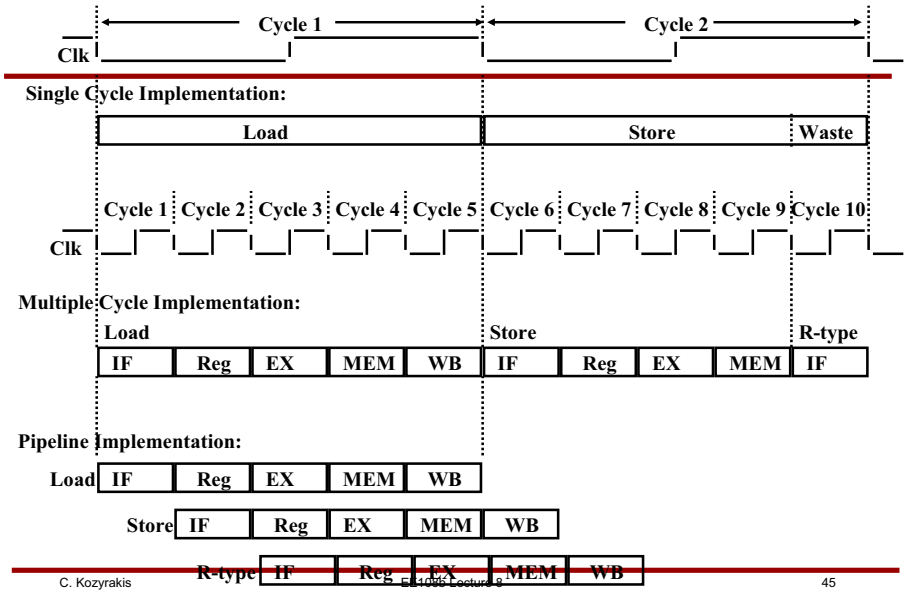
Implementing Control



Putting it All Together



Comparison



But Something Is Fishy Here

- If dividing it into 5 parts made the clock faster
  - And the effective CPI is still one
- Then dividing it into 10 parts would make the clock even faster
  - And wouldn't the CPI still be one?
- Then why not go to twenty cycles?
- Really two issues
  - Some things really have to complete in a cycle
    - Find next PC from current PC
  - CPI is not really one
    - Sometimes you need the results from the previous instruction