CS61C : Machine Structures

Lecture 20
CPU Design: Control II & Pipelining I

TA Noah Johnson

2010-07-26

AN x64 PROCESSOR IS SCREAMING ALONG AT BILLIONS OF CYCLES PER SECOND TO RUN THE XNU KERNEL, WHICH IS FRANTICALLY WORKING THROUGH ALL THE POSIX-SPECIFIED ABSTRACTION TO CREATE THE DARWIN SYSTEM UNDERLYING OS X, WHICH IN TURN IS STRAINING ITSELF TO RUN FIREFOX AND ITS GECKO RENDERER, WHICH CREATES A FLASH OBJECT WHICH RENDERS DOZENS OF VIDEO FRAMES EVERY SECOND

BECAUSE I WANTED TO SEE A CAT JUMP INTO A BOX AND FALL OVER.

http://xkcd.com/676/

Johnson, Summer 2010 © UCB

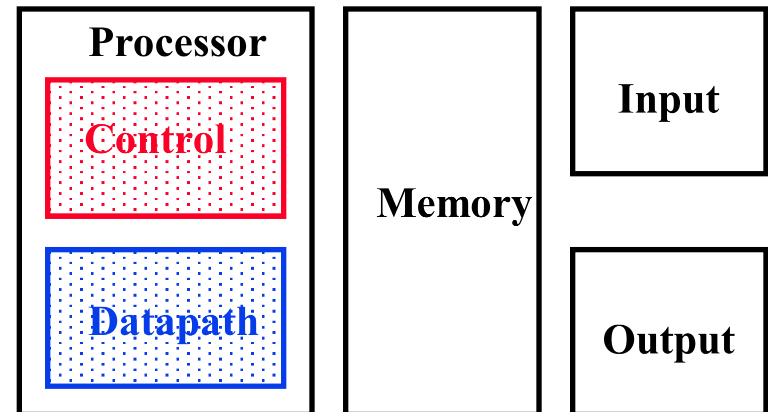
In Review: A Single Cycle Datapath

- We have everything! Now we just need to know how to BUILD **CONTROL**

nPC_sel
clk
instr fetch unit
Instruction<31:0>
<21:25> Rs
<16:20> Rt
<11:15> Rd
<0:15> Imm16
RegDst
Rd Rt
1 0
RegWr
Rs Rt
5 5 5
Rw Ra Rb
RegFile
busW 32
clk
busA 32
busB 32
zero
ALUctr
ALU
32
MemWr
MemtoReg
0 1
0 1
32
Data In
WrEn Adr
Data Memory
clk
imm16 16
Extender
32
ExtOp
ALUSrc

Summary: Single-cycle Processor

- 5 steps to design a processor
  - 1. Analyze instruction set → datapath requirements
  - 2. Select set of datapath components & establish clock methodology
  - 3. Assemble datapath meeting the requirements
  - 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  - 5. Assemble the control logic
    - Formulate Logic Equations
    - Design Circuits



Step 4: Given Datapath: RTL → Control

Instruction<31:0>

Inst Memory

Adr

<26:31> Op

<0:5> Fun

<21:25> Rt

<16:20> Rs

<11:15> Rd

<0:15> Imm16

Control

nPC_sel RegWr RegDst ExtOp ALUSrc ALUctr MemWr MemtoReg

DATA PATH

A Summary of the Control Signals (1/2)

inst	Register Transfer	
add	R[rd] ← R[rs] + R[rt];	PC ← PC + 4
	ALUsrc = RegB, ALUctr = "ADD", RegDst = rd, RegWr, nPC_sel = "+4"	
sub	R[rd] ← R[rs] – R[rt];	PC ← PC + 4
	ALUsrc = RegB, ALUctr = "SUB", RegDst = rd, RegWr, nPC_sel = "+4"	
ori	R[rt] ← R[rs] + zero_ext(Imm16);	PC ← PC + 4
	ALUsrc = Im, Extop = "Z",ALUctr = "OR", RegDst = rt,RegWr, nPC_sel ="+4"	
lw	R[rt] ← MEM[R[rs] + sign_ext(Imm16)];	PC ← PC + 4
	ALUsrc = Im, Extop = "sn", ALUctr = "ADD", MemtoReg, RegDst = rt, RegWr, nPC_sel = "+4"	
sw	MEM[R[rs] + sign_ext(Imm16)] ← R[rs];	PC ← PC + 4
	ALUsrc = Im, Extop = "sn", ALUctr = "ADD", MemWr, nPC_sel = "+4"	
beq	if (R[rs] == R[rt]) then PC ← PC + sign_ext(Imm16)] || 00 else PC ← PC + 4	
	nPC_sel = "br", ALUctr = "SUB"	

A Summary of the Control Signals (2/2)

See Appendix A → func, op

func	10 0000	10 0010	We Don't Care :-)				
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
nPCsel	0	0	0	0	0	1	?
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x
ALUctr<2:0>	Add	Subtract	Or	Add	Add	Subtract	x

	31–26	26–21	21–16	16–11	11–6	6–0	
R-type	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**	**add, sub**
I-type	**op**	**rs**	**rt**	**immediate**			**ori, lw, sw, beq**
J-type	**op**	**target address**					**jump**

Boolean Expressions for Controller

RegDst = add + sub
ALUSrc = ori + lw + sw
MemtoReg = lw
RegWrite = add + sub + ori + lw
MemWrite = sw
nPCsel = beq
Jump = jump
ExtOp = lw + sw
ALUctr[0] = sub + beq (assume ALUctr is 00 ADD, 01: SUB, 10: OR)
ALUctr[1] = or

where,

rtype = $\sim op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet \sim op_2 \bullet \sim op_1 \bullet \sim op_0$,
ori = $\sim op_5 \bullet \sim op_4 \bullet op_3 \bullet op_2 \bullet \sim op_1 \bullet op_0$
lw = $op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet \sim op_2 \bullet op_1 \bullet op_0$
sw = $op_5 \bullet \sim op_4 \bullet op_3 \bullet \sim op_2 \bullet op_1 \bullet op_0$
beq = $\sim op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet op_2 \bullet \sim op_1 \bullet \sim op_0$
jump = $\sim op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet \sim op_2 \bullet op_1 \bullet \sim op_0$

add = rtype $\bullet func_5 \bullet \sim func_4 \bullet \sim func_3 \bullet \sim func_2 \bullet \sim func_1 \bullet \sim func_0$
sub = rtype $\bullet func_5 \bullet \sim func_4 \bullet \sim func_3 \bullet \sim func_2 \bullet func_1 \bullet \sim func_0$

How do we implement this in gates?

Johnson, Summer 2010 © UCB

Controller Implementation

opcode func

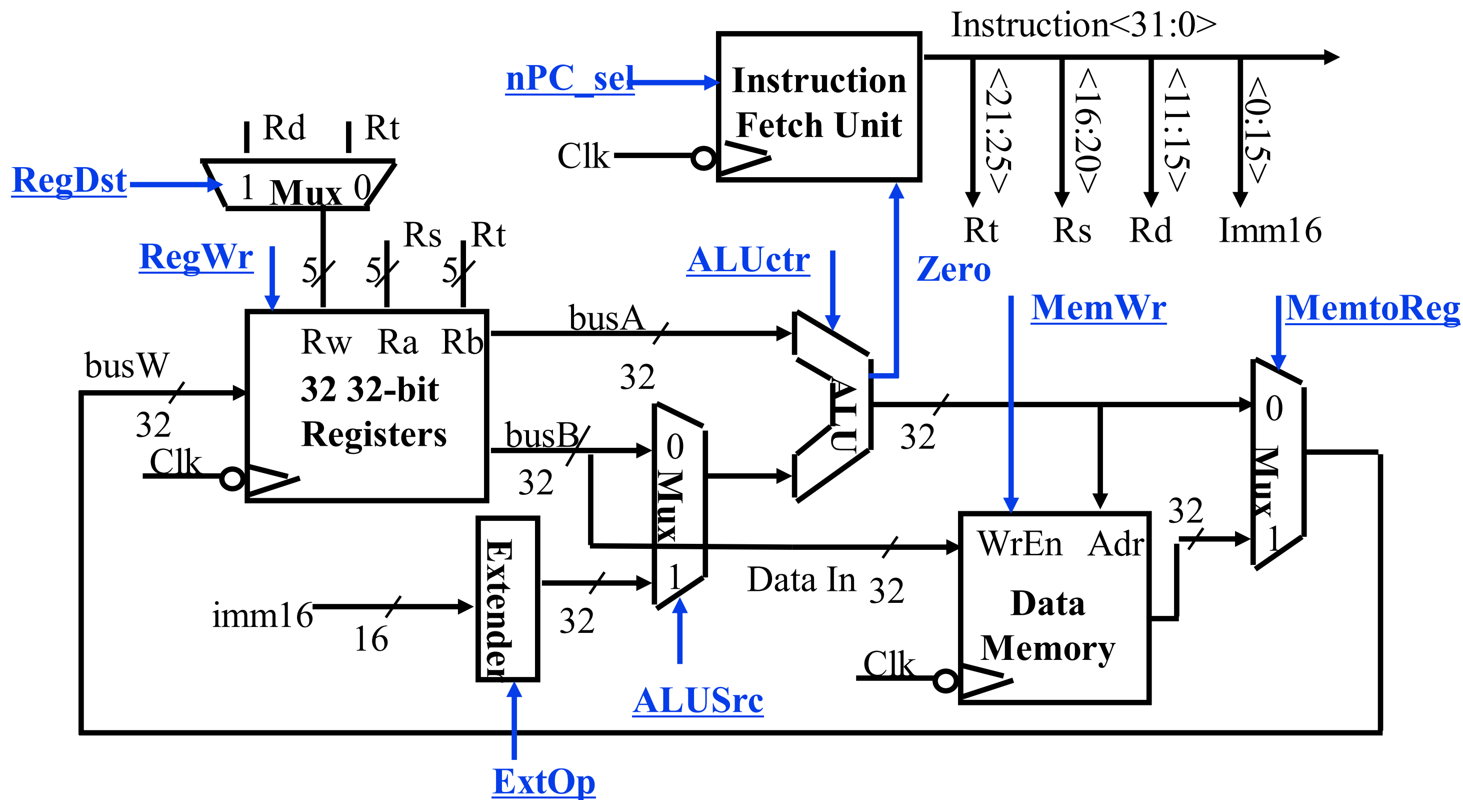
"AND" logic

add
sub
ori
lw
sw
beq
jump

"OR" logic

RegDst
ALUSrc
MemtoReg
RegWrite
MemWrite
nPCsel
Jump
ExtOp
ALUctr[0]
ALUctr[1]

Peer Instruction



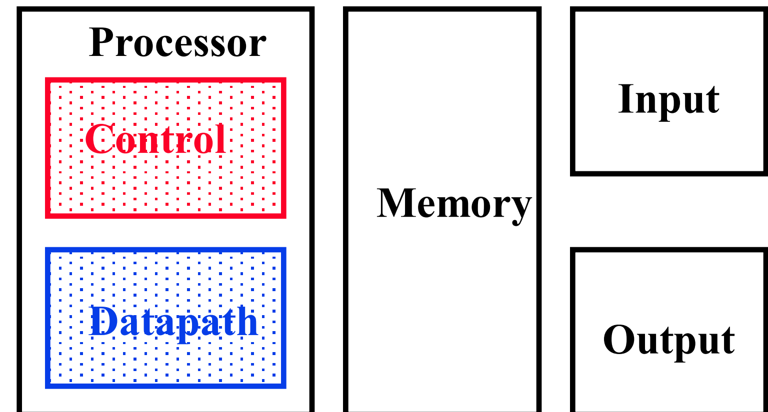
1) MemToReg=‘x’ & ALUctr=‘sub’. SUB or BEQ?

2) ALUctr=‘add’. Which 1 signal is different for all 3 of: ADD, LW, & SW? RegDst or ExtOp?

	12
a)	SR
b)	SE
c)	BR
d)	BE

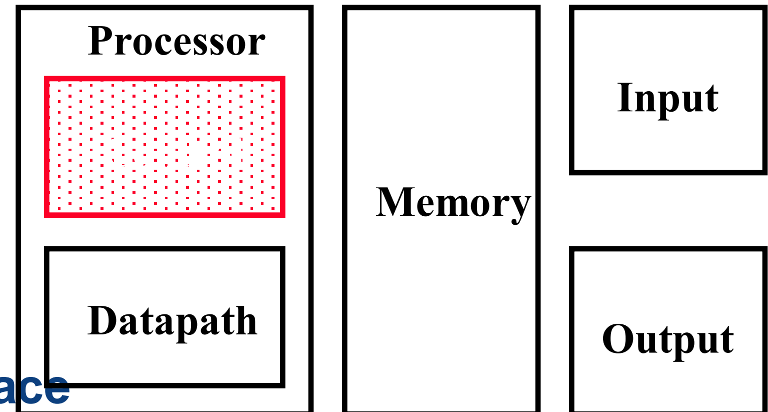
Summary: Single-cycle Processor

- 5 steps to design a processor
  - 1. Analyze instruction set → datapath requirements
  - 2. Select set of datapath components & establish clock methodology
  - 3. Assemble datapath meeting the requirements
  - 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  - 5. Assemble the control logic
    - Formulate Logic Equations
    - Design Circuits

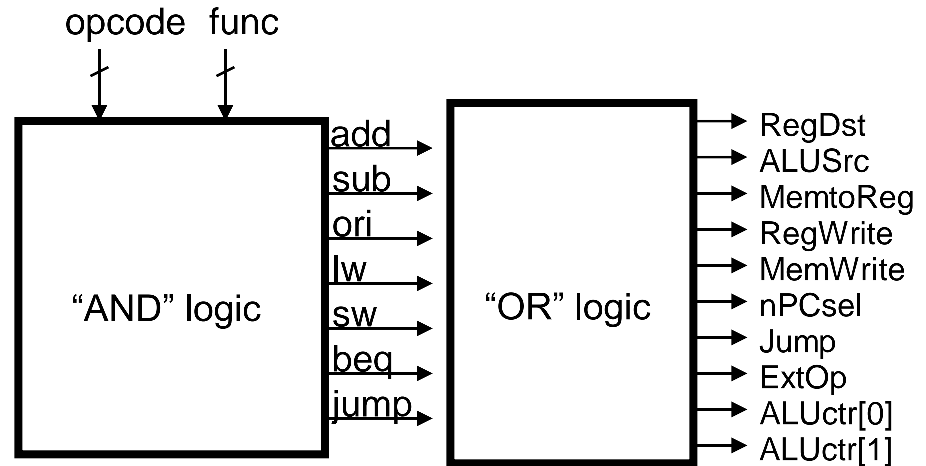


Review: Single cycle datapath

- 5 steps to design a processor
  1. Analyze instruction set datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements
  4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  5. Assemble the control logic
- Control is the hard part
- MIPS makes that easier
  - Instructions same size
  - Source registers always in same place
  - Immediates same size, location
  - Operations always on registers/immediates



How We Build The Controller



RegDst = add + sub
ALUSrc = ori + lw + sw
MemtoReg = lw
RegWrite = add + sub + ori + lw
MemWrite = sw
nPCsel = beq
Jump = jump
ExtOp = lw + sw
ALUctr[0] = sub + beq (assume ALUctr is 0 ADD, 01: SUB, 10: OR)
ALUctr[1] = or

where,

$rtype = \sim op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet \sim op_2 \bullet \sim op_1 \bullet \sim op_0,$
$ori = \sim op_5 \bullet \sim op_4 \bullet op_3 \bullet op_2 \bullet \sim op_1 \bullet op_0$
$lw = op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet \sim op_2 \bullet op_1 \bullet op_0$
$sw = op_5 \bullet \sim op_4 \bullet op_3 \bullet \sim op_2 \bullet op_1 \bullet op_0$
$beq = \sim op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet op_2 \bullet \sim op_1 \bullet \sim op_0$
$jump = \sim op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet \sim op_2 \bullet op_1 \bullet \sim op_0$

$add = rtype \bullet func_5 \bullet \sim func_4 \bullet \sim func_3 \bullet \sim func_2 \bullet \sim func_1 \bullet \sim func_0$
$sub = rtype \bullet func_5 \bullet \sim func_4 \bullet \sim func_3 \bullet \sim func_2 \bullet func_1 \bullet \sim func_0$

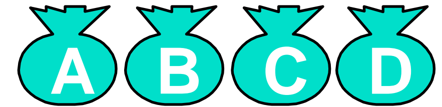
Omigosh omigosh, do you know what this means?

Processor Performance

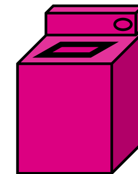
- Can we estimate the clock rate (frequency) of our single-cycle processor? We know:
  - 1 cycle per instruction
  - `lw` is the most demanding instruction.
  - Assume these delays for major pieces of the datapath:
    - Instr. Mem, ALU, Data Mem : 2ns each, regfile 1ns
    - Instruction execution requires: 2 + 1 + 2 + 2 + 1 = 8ns
    - $\Rightarrow$ 125 MHz
- What can we do to improve clock rate?
- Will this improve performance as well?
  - We want increases in clock rate to result in programs executing quicker.

Gotta Do Laundry

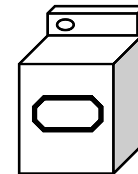
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, fold, and put away



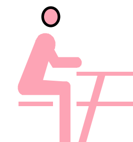
  - Washer takes 30 minutes



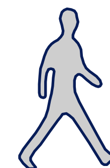
  - Dryer takes 30 minutes



  - "Folder" takes 30 minutes



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



Johnson, Summer 2010 © UCB

Sequential Laundry

6 PM 7 8 9 10 11 12 1 2 AM

30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30

Time

Task Order

A

B

C

D

- **Sequential laundry takes 8 hours for 4 loads**

Johnson, Summer 2010 © UCB

Pipelined Laundry

6 PM 7 8 9 10 11 12 1 2 AM

Time

30 30 30 30 30 30 30

Task Order

A

B

C

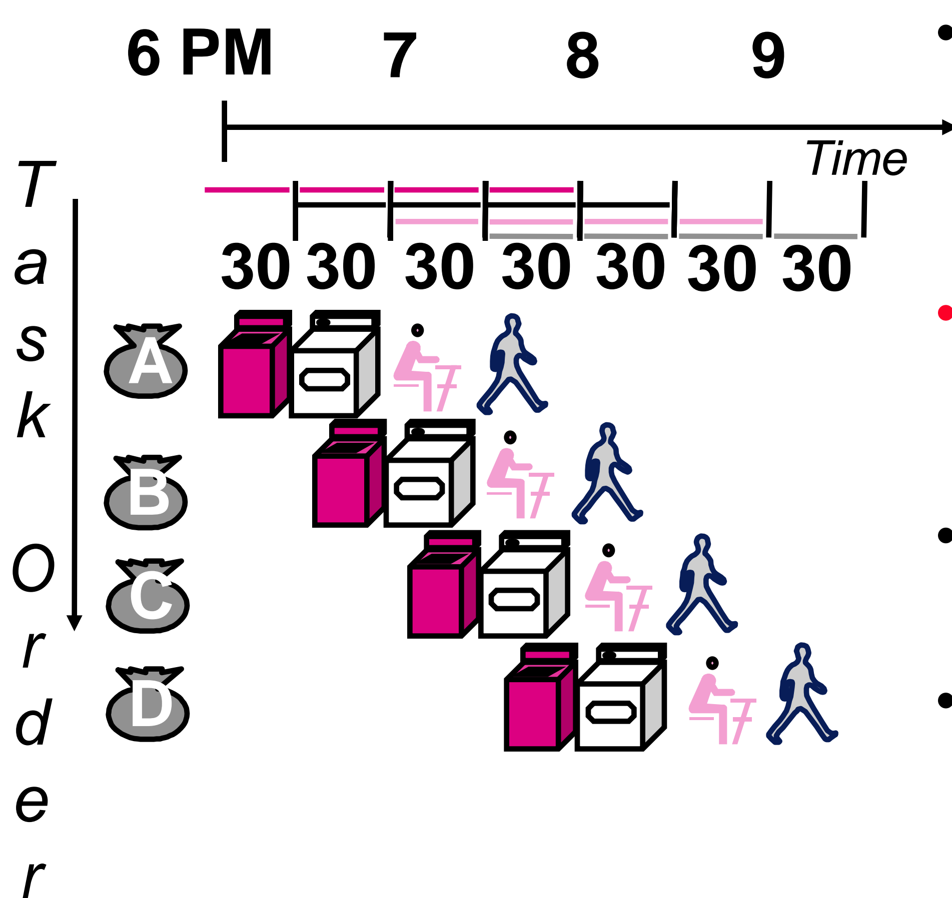
D

- **Pipelined laundry takes 3.5 hours for 4 loads!**

General Definitions

- **Latency**: time to completely execute a certain task
  - for example, time to read a sector from disk is disk access time or disk latency
- **Throughput**: amount of work that can be done over a period of time

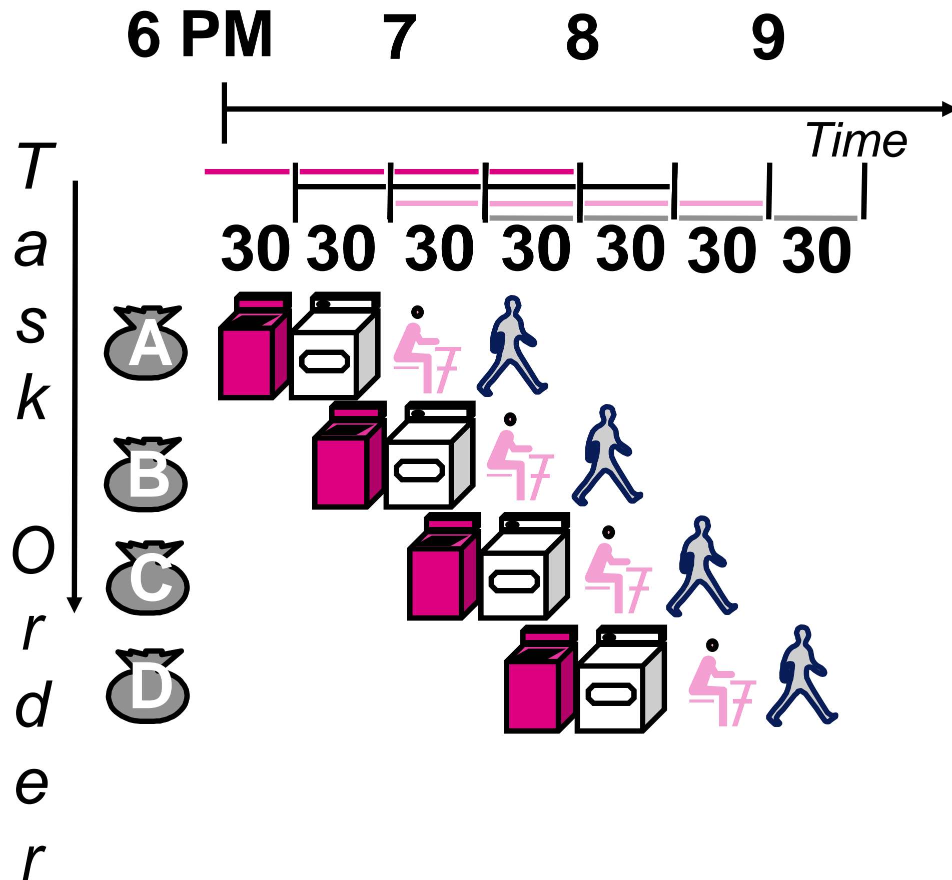
Pipelining Lessons (1/2)



- 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**
- Time to "**fill**" pipeline and time to "**drain**" it reduces speedup: 2.3X v. 4X in this example

Johnson, Summer 2010 © UCB

Pipelining Lessons (2/2)



- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup

Johnson, Summer 2010 © UCB

Steps in Executing MIPS

1) **IFtch**: Instruction Fetch, Increment PC

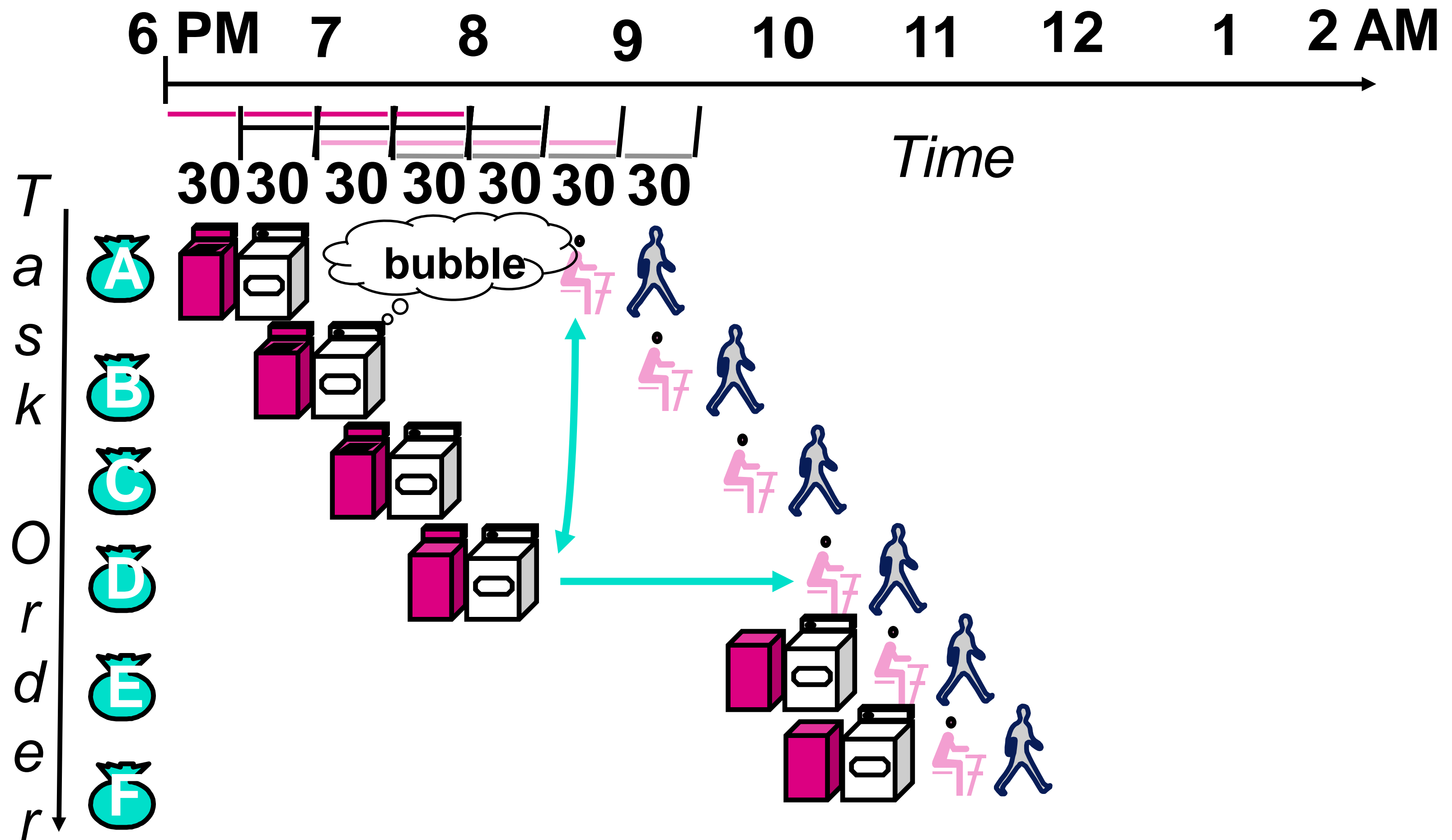
2) **Dcd**: Instruction Decode, Read Registers

3) **Exec**:
Mem-ref: Calculate Address
Arith-log: Perform Operation

4) **Mem**:
Load: Read Data from Memory
Store: Write Data to Memory

5) **WB**: Write Data Back to Register

Pipeline Hazard: Matching socks in later load



- A depends on D; stall since folder tied up

Administrivia

- HW8 due tomorrow
- Project 2 due next Monday
- Newsgroup problems
- Reminder: Midterm regrades due today

Problems for Pipelining CPUs

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support some combination of instructions (single person to fold and put clothes away)
  - Control hazards: Pipelining of branches causes later instruction fetches to wait for the result of the branch
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
- These might result in pipeline stalls or "bubbles" in the pipeline.

Structural Hazard #1: Single Memory (1/2)

Time (clock cycles)

I n s t r. O r d e r

Load

Instr 1

Instr 2

Instr 3

Instr 4

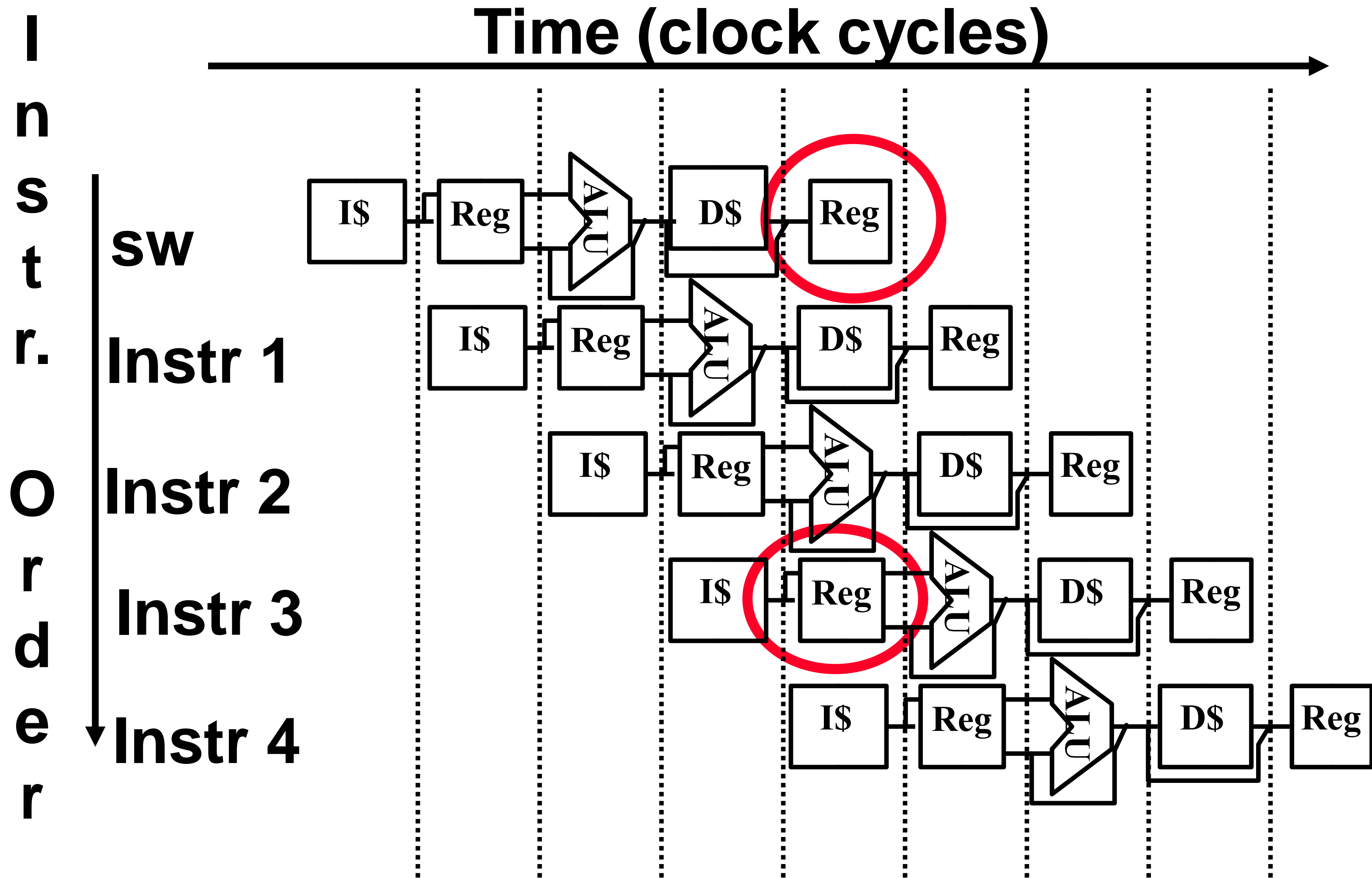
Load	I$	Reg	ALU	D$	Reg			
Instr 1		I$	Reg	ALU	D$	Reg		
Instr 2			I$	Reg	ALU	D$	Reg	
Instr 3				I$	Reg	ALU	D$	Reg
Instr 4					I$	Reg	ALU	D$	Reg

Read same memory twice in same clock cycle

Structural Hazard #1: Single Memory (2/2)

- Solution:
  - infeasible and inefficient to create second memory
  - (We'll learn about this more next week)
  - so simulate this by having two Level 1 Caches (a temporary smaller [of usually most recently used] copy of memory)
  - have both an L1 Instruction Cache and an L1 Data Cache
  - need more complex hardware to control when both caches miss

Structural Hazard #2: Registers (1/2)



Can we read and write to registers simultaneously?

Structural Hazard #2: Registers (2/2)

- Two different solutions have been used:
  1) RegFile access is *VERY* fast: takes less than half the time of ALU stage
     - Write to Registers during first half of each clock cycle
     - Read from Registers during second half of each clock cycle
  2) Build RegFile with independent read and write ports
- Result: can perform Read and Write during same clock cycle

Johnson, Summer 2010 © UCB

Peer Instruction

1) Thanks to pipelining, I have reduced the time it took me to wash my one shirt.

2) Longer pipelines are always a win (since less work per stage & a faster clock).

	12
a)	FF
b)	FT
c)	TF
d)	TT

Things to Remember

- Optimal Pipeline
  - Each stage is executing part of an instruction each clock cycle.
  - One instruction finishes during each clock cycle.
  - On average, execute far more quickly.
- What makes this work?
  - Similarities between instructions allow us to use same stages for all instructions (generally).
  - Each stage takes about the same amount of time as all others: little wasted time.

Bonus slides

- These are extra slides that used to be included in lecture notes, but have been moved to this, the "bonus" area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

Bonus

The Single Cycle Datapath during Jump

- New PC = { PC[31..28], target address, 00 }

Jump=1
nPC_sel=?
Instruction Fetch Unit
Clk
Instruction<31:0>
<21:25> <16:20> <11:15> <0:15> <0:25>
Rt Rs Rd Imm16 TA26
RegDst = x
Rd Rt
1 Mux 0
RegWr = 0
Rs Rt
5 5 5
Rw Ra Rb
32 32-bit Registers
busW
32
Clk
busA
32
busB
32
ALUctr =x
ALU
Zero
MemWr = 0
MemtoReg = x
0 Mux 1
32
32
Extender
imm16
16
32
0 Mux 1
ALUSrc = x
ExtOp = x
Data In 32
WrEn Adr
Data Memory
Clk

Instruction Fetch Unit at the End of Jump

- **New PC = { PC[31..28], target address, 00 }**

Jump

nPC_sel

Zero

Inst Memory

Adr

Instruction<31:0>

nPC_MUX_sel

4

Adder

Adder

0

Mux

1

00

PC

Clk

imm16

How do we modify this to account for jumps?

Instruction Fetch Unit at the End of Jump

- **New PC = { PC[31..28], target address, 00 }**

Jump
nPC_sel
Zero
nPC_MUX_sel
Inst
Memory
Adr
Instruction<31:0>
4
Adder
Adder
imm16
0
Mux
1
TA
26
00
4 (MSBs)
1
Mux
0
00
PC
Clk

Query

- Can Zero still get asserted?
- Does nPC_sel need to be 0?
  - If not, what?