inst.eecs.berkeley.edu/~cs61c

CS61CL : Machine Structures

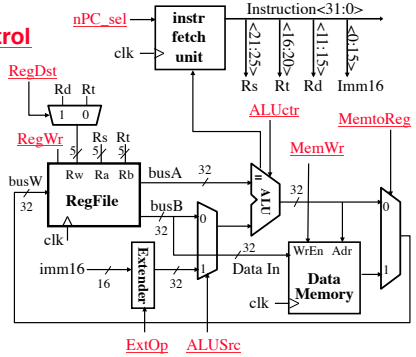
Lecture #9 – Single Cycle CPU Design

2009-07-22

Jeremy Huddleston

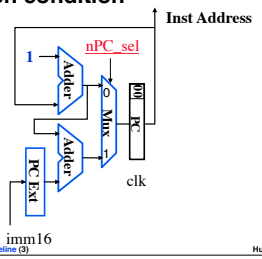
Review: A Single Cycle Datapath

- We have everything except control signals



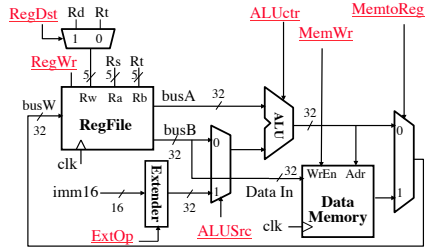
Recap: Meaning of the Control Signals

- nPC_sel: "n"=next
  - "+4" 0 ⇒ PC <– PC + 4
  - "br" 1 ⇒ PC <– PC + 4 + {SignExt(Im16) , 00 }
- Later in lecture: higher-level connection between mux and branch condition



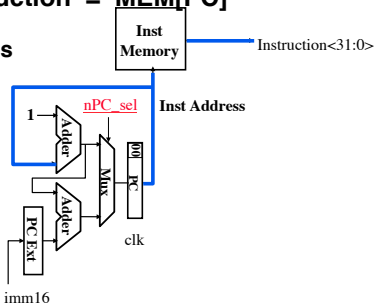
Recap: Meaning of the Control Signals

- ExtOp: "zero", "sign"
- ALUsrc: 0 ⇒ regB; 1 ⇒ immed
- ALUctr: "ADD", "SUB", "OR"
- MemWr: 1 ⇒ write memory
- MemtoReg: 0 ⇒ ALU; 1 ⇒ Mem
- RegDst: 0 ⇒ "rt"; 1 ⇒ "rd"
- RegWr: 1 ⇒ write register

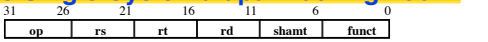


Instruction Fetch Unit at the Beginning of Add

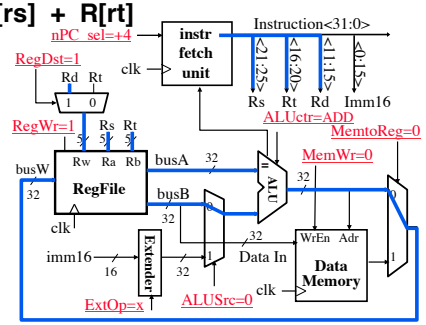
- Fetch the instruction from Instruction memory: Instruction = MEM[PC]
  - same for all instructions



The Single Cycle Datapath during Add

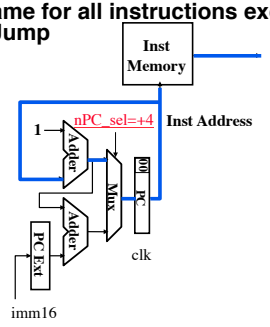


R[rd] = R[rs] + R[rt]

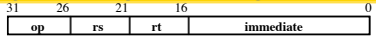


Instruction Fetch Unit at the End of Add

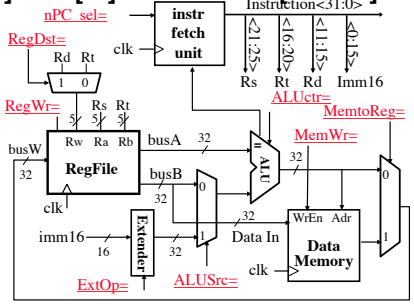
- PC = PC + 4
  - This is the same for all instructions except: Branch and Jump



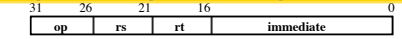
Single Cycle Datapath during Or Immediate?



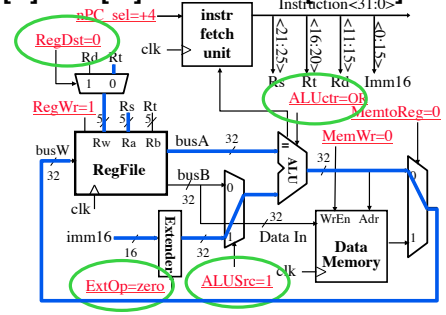
- R[rt] = R[rs] OR ZeroExt[Imm16]



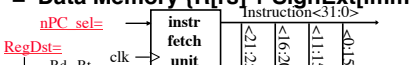
Single Cycle Datapath during Or Immediate?



- R[rt] = R[rs] OR ZeroExt[Imm16]

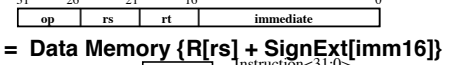


The Single Cycle Datapath during Load?

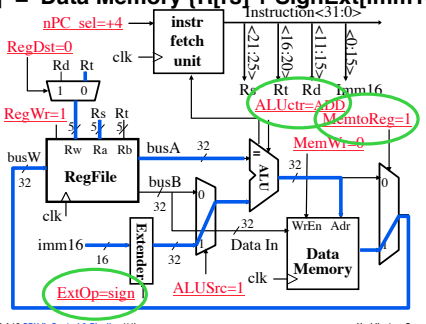


• R[rt] = Data Memory {R[rs] + SignExt[imm16]}

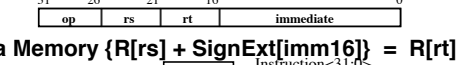
The Single Cycle Datapath during Load



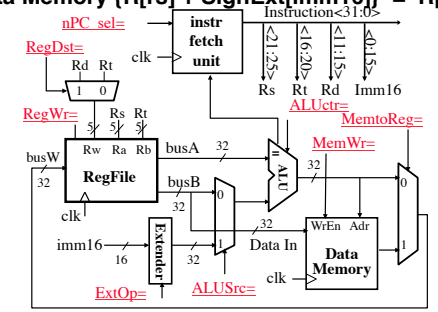
• R[rt] = Data Memory {R[rs] + SignExt[imm16]}



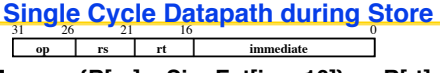
The Single Cycle Datapath during Store?



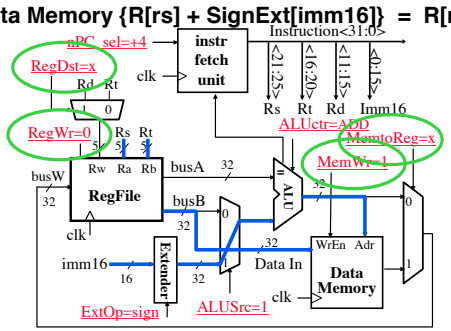
• Data Memory {R[rs] + SignExt[imm16]} = R[rt]



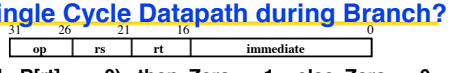
The Single Cycle Datapath during Store



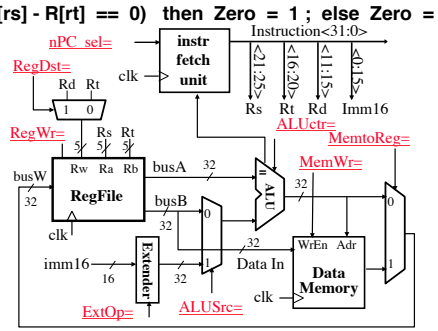
• Data Memory {R[rs] + SignExt[imm16]} = R[rt]



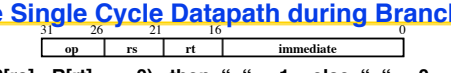
The Single Cycle Datapath during Branch?



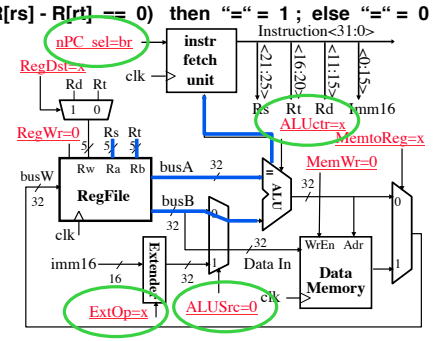
• if (R[rs] - R[rt] == 0) then Zero = 1 ; else Zero = 0



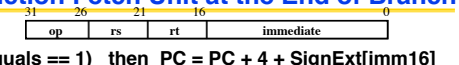
The Single Cycle Datapath during Branch



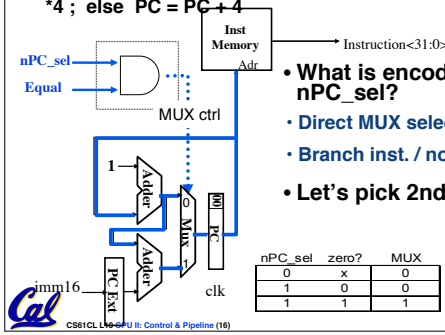
• if (R[rs] - R[rt] == 0) then "=" = 1 ; else "=" = 0



Instruction Fetch Unit at the End of Branch



• if (Equals == 1) then PC = PC + 4 + SignExt[imm16] *4 ; else PC = PC + 4



• What is encoding of nPC_sel?
  • Direct MUX select?
  • Branch inst. / not branch
• Let's pick 2nd option

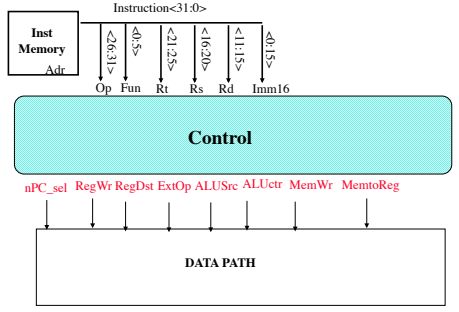
nPC_sel	zero?	MUX
0	x	0
1	0	0
1	1	1

Q: What logic gate?

How to Design a Processor: step-by-step

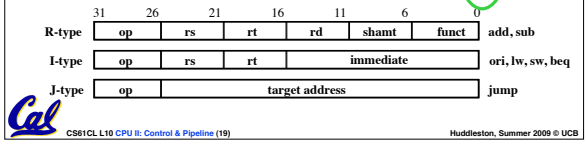
- 1. Analyze instruction set architecture (ISA) ⇒ datapath requirements
  - meaning of each instruction is given by the *register transfers*
  - datapath must include storage element for ISA registers
  - datapath must support each register transfer
- 2. Select set of datapath components and establish clocking methodology
- 3. Assemble datapath meeting requirements
- **4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.**
- **5. Assemble the control logic (hard part!)**

Step 4: Given Datapath: RTL → Control



A Summary of the Control Signals

See Appendix A	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



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 0 ADD, 01: SUB, 10: OR)
ALUctr[1] = or

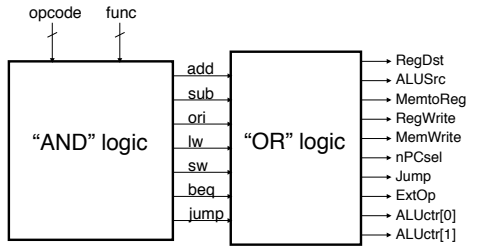
where,

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

How do we implement this in gates?

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

Controller Implementation

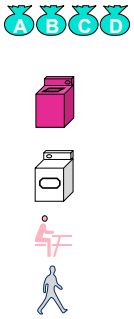


Processor Performance

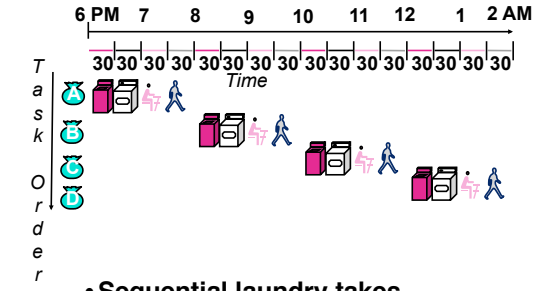
- Can we estimate the clock rate (frequency) of our single-cycle processor? We know:
  - 1 cycle per instruction
  - `lw` is the most demanding instruction.
  - Assume these delays for major pieces of the datapath:
    - Instr. Mem, ALU, Data Mem : 2 ns each, regfile 1 ns
    - Instruction execution requires: 2 + 1 + 2 + 2 + 1 = 8 ns
    - ⇒ 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

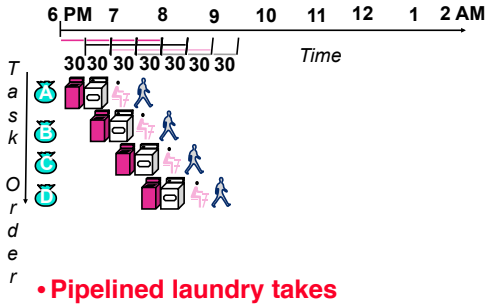


Sequential Laundry



- Sequential laundry takes 8 hours for 4 loads

Pipelined Laundry

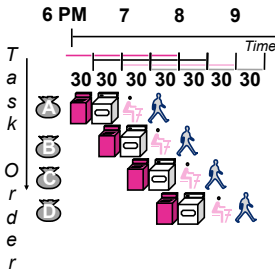


- Pipelined laundry takes 3.5 hours for 4 loads!

General Definitions

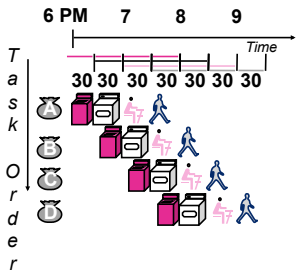
- Latency: time to completely execute a certain task
  - for example, time to read a sector from disk is disk access time or disk latency
- Throughput: amount of work that can be done over a period of time

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

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

Steps in Executing MIPS

1) IFtch: Instruction Fetch, Increment PC

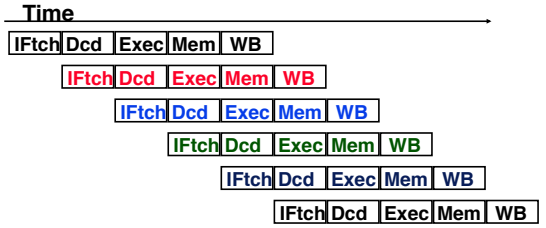
2) Dcd: Instruction Decode, Read Registers

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

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

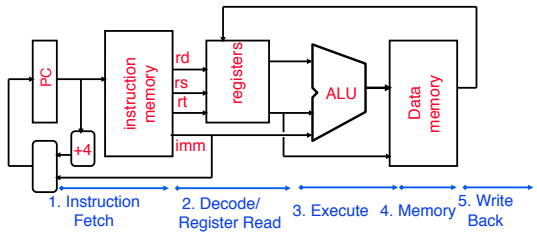
5) WB: Write Data Back to Register

Pipelined Execution Representation

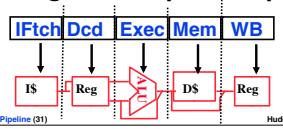


- Every instruction must take same number of steps, also called pipeline "stages", so some will go idle sometimes

Review: Datapath for MIPS

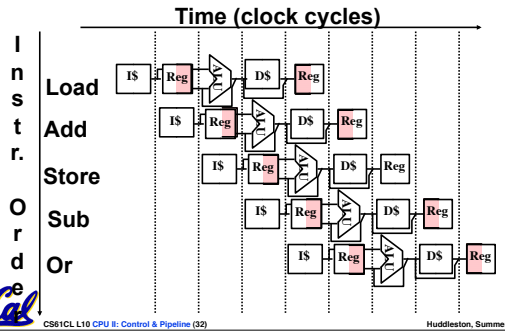


- Use datapath figure to represent pipeline



Graphical Pipeline Representation

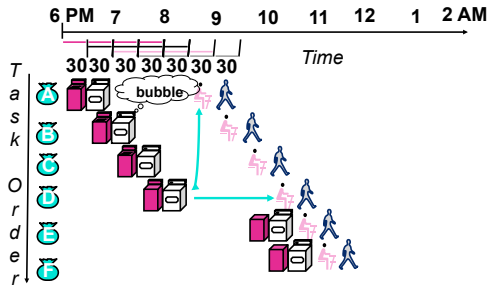
(In Reg, right half highlight read, left half write)



Example

- Suppose 2 ns for memory access, 2 ns for ALU operation, and 1 ns for register file read or write; compute instruction rate
- Nonpipelined Execution:
  - `lw` : IF + Read Reg + ALU + Memory + Write Reg = 2 + 1 + 2 + 2 + 1 = 8 ns
  - `add`: IF + Read Reg + ALU + Write Reg = 2 + 1 + 2 + 1 = 6 ns
    (recall 8ns for single-cycle processor)
- Pipelined Execution:
  - Max(IF,Read Reg,ALU,Memory,Write Reg) = 2 ns

Pipeline Hazard: Matching socks in later load



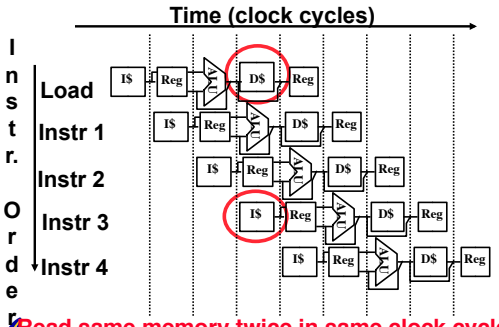
Administrivia

- Midterm Solutions
- Regrade Requests
- HW7 (Design Document)

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)

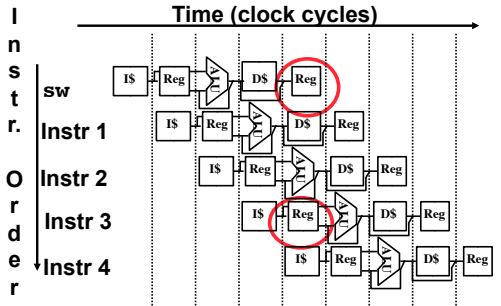


Read same memory twice in same clock cycle

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)

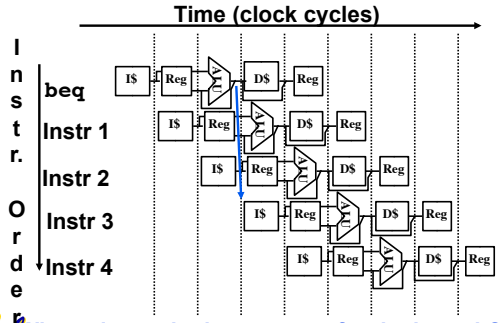


Can we read and write to registers simultaneously?

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

Control Hazard: Branching (1/8)



Where do we do the compare for the branch?

Control Hazard: Branching (2/8)

- We had put branch decision-making hardware in ALU stage
  - therefore two more instructions after the branch will always be fetched, whether or not the branch is taken
- Desired functionality of a branch
  - if we do not take the branch, don't waste any time and continue executing normally
  - if we take the branch, don't execute any instructions after the branch, just go to the desired label

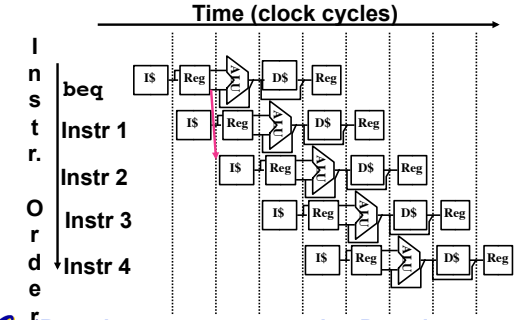
Control Hazard: Branching (3/8)

- Initial Solution: Stall until decision is made
  - insert "no-op" instructions (those that accomplish nothing, just take time) or hold up the fetch of the next instruction (for 2 cycles).
  - Drawback: branches take 3 clock cycles each (assuming comparator is put in ALU stage)

Control Hazard: Branching (4/8)

- Optimization #1:
  - insert special branch comparator in Stage 2
  - as soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
  - Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
  - Side Note: This means that branches are idle in Stages 3, 4 and 5.

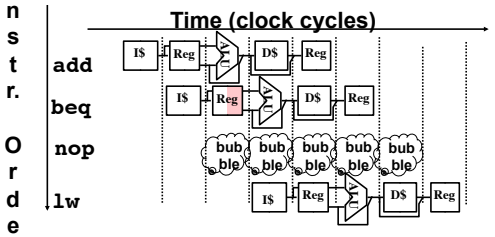
Control Hazard: Branching (5/8)



Branch comparator moved to Decode stage.

Control Hazard: Branching (6a/8)

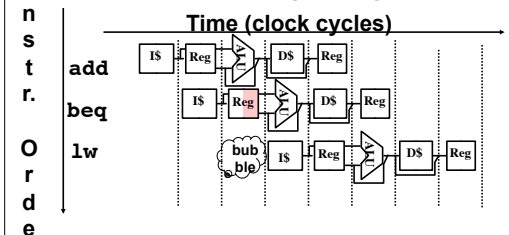
- User inserting no-op instruction



Impact: 2 clock cycles per branch instruction ⇒ slow

Control Hazard: Branching (6b/8)

- Controller inserting a single bubble



Impact: 2 clock cycles per branch instruction ⇒ slow

Control Hazard: Branching (7/8)

- Optimization #2: Redefine branches
  - Old definition: if we take the branch, none of the instructions after the branch get executed by accident
  - New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (called the branch-delay slot)
- The term "Delayed Branch" means we always execute inst after branch
- This optimization is used with MIPS

Control Hazard: Branching (8/8)

- Notes on Branch-Delay Slot
  - Worst-Case Scenario: can always put a no-op in the branch-delay slot
  - Better Case: can find an instruction preceding the branch which can be placed in the branch-delay slot without affecting flow of the program
    - re-ordering instructions is a common method of speeding up programs
    - compiler/assembler must be very smart in order to find instructions to do this
    - usually can find such an instruction at least 50% of the time
    - Jumps also have a delay slot…

Example: Nondelayed vs. Delayed Branc

MAL Nondelayed Branch	TAL Delayed Branch
`or $8, $9 ,$10`	`add $1 ,$2,$3`
`add $1 ,$2,$3`	`sub $4, $5,$6`
`sub $4, $5,$6`	`beq $1, $4, Exit`
`beq $1, $4, Exit`	`or $8, $9 ,$10`
`xor $10, $1,$11`	`xor $10, $1,$11`
`Exit:`	`Exit:`

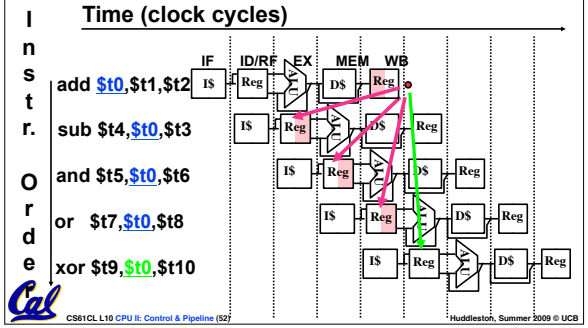
Data Hazards (1/2)

- Consider the following sequence of instructions

```
add $t0, $t1, $t2
sub $t4, $t0 ,$t3
and $t5, $t0 ,$t6
or  $t7, $t0 ,$t8
xor $t9, $t0 ,$t10
```

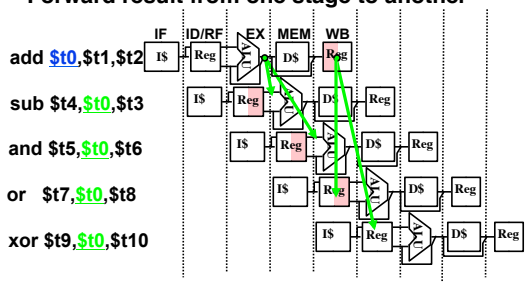

Data Hazards (2/2)

- Data-flow backward in time are hazards



Data Hazard Solution: Forwarding

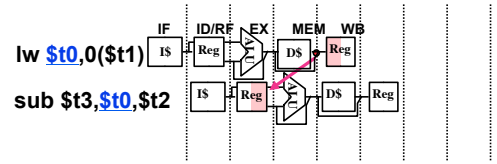
- Forward result from one stage to another



"or" hazard solved by register hardware

Data Hazard: Loads (1/4)

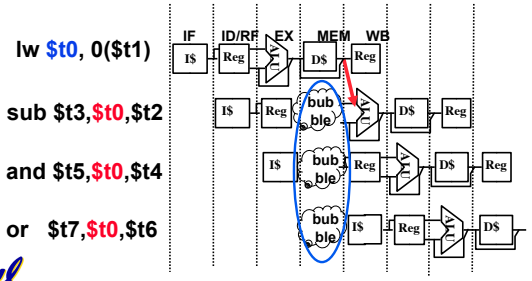
- Dataflow backwards in time are hazards



- Can't solve all cases with forwarding
- Must stall instruction dependent on load, then forward (more hardware)

Data Hazard: Loads (2/4)

- Hardware stalls pipeline
  - Called "interlock"

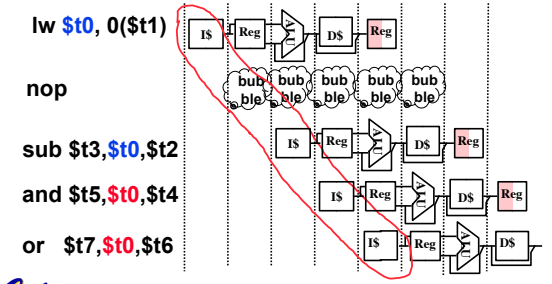


Data Hazard: Loads (3/4)

- Instruction slot after a load is called "load delay slot"
- If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle.
- If the compiler puts an unrelated instruction in that slot, then no stall
- Letting the hardware stall the instruction in the delay slot is equivalent to putting a nop in the slot (except the latter uses more code space)

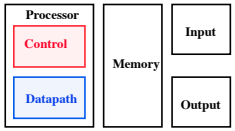
Data Hazard: Loads (4/4)

- Stall is equivalent to nop



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



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.

"And in Conclusion.."

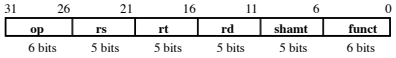
- Pipeline challenge is hazards
  - Forwarding helps w/many data hazards
  - Delayed branch helps with control hazard in 5 stage pipeline
  - Load delay slot / interlock necessary
- More aggressive performance:
  - Superscalar
  - Out-of-order execution

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

RTL: The Add Instruction

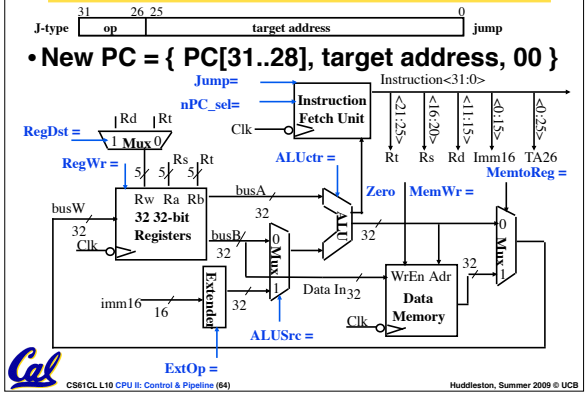


`add rd, rs, rt`
- MEM[PC] Fetch the instruction from memory
- R[rd] = R[rs] + R[rt] The actual operation
- PC = PC + 4 Calculate the next instruction's address

A Summary of the Control Signals (1/2)

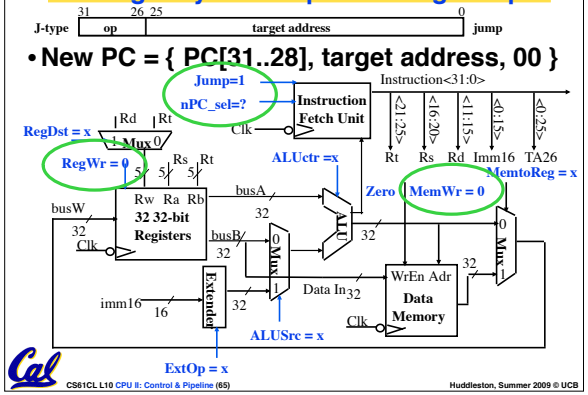
	inst	Register Transfer
	add	R[rd] ← R[rs] + R[rt]; PC ← PC + 4
		ALUsrc = RegB, ALUctr = "ADD", RegDst = rd, RegWr, nPC_sel = "+4"
	sub	R[rd] ← R[rs] – R[rt]; PC ← PC + 4
		ALUsrc = RegB, ALUctr = "SUB", RegDst = rd, RegWr, nPC_sel = "+4"
	ori	R[rt] ← R[rs] + zero_ext(Imm16); PC ← PC + 4
		ALUsrc = Im, 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"

The Single Cycle Datapath during Jump



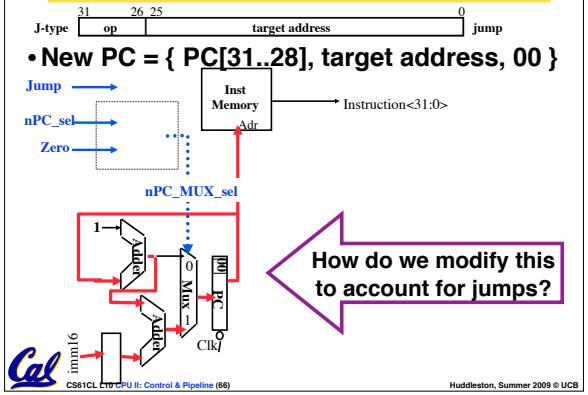
- New PC = { PC[31..28], target address, 00 }

The Single Cycle Datapath during Jump



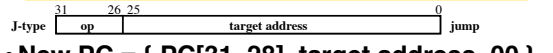
- New PC = { PC[31..28], target address, 00 }

Instruction Fetch Unit at the End of Jump

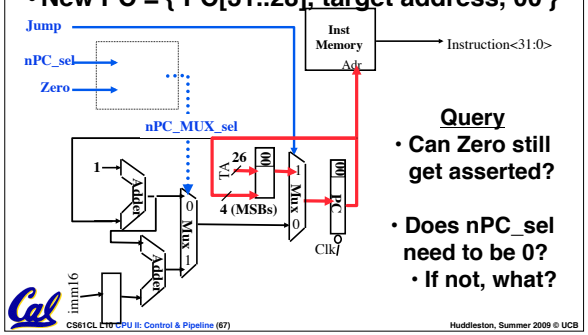


- New PC = { PC[31..28], target address, 00 }

Instruction Fetch Unit at the End of Jump



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



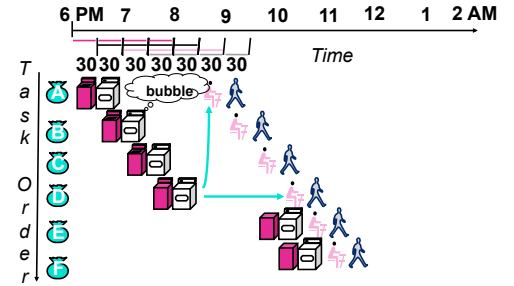
Query

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

Historical Trivia

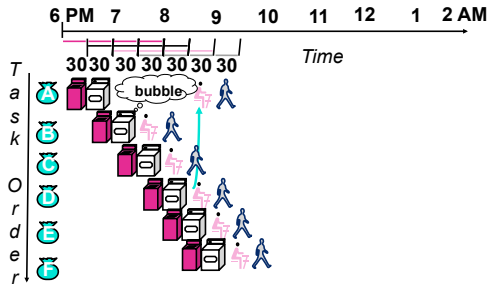
- First MIPS design did not interlock and stall on load-use data hazard
- Real reason for name behind MIPS:
  Microprocessor without
  Interlocked
  Pipeline
  Stages
  - Word Play on acronym for Millions of Instructions Per Second, also called MIPS

Pipeline Hazard: Matching socks in later load



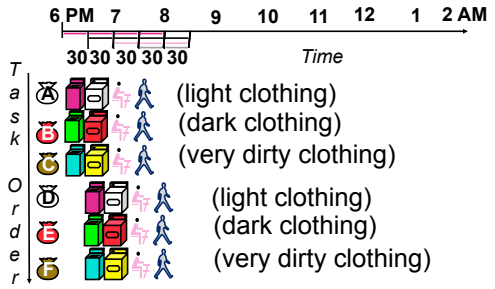
A depends on D; stall since folder tied up; Note this is much different from processor cases so far. We have not had a earlier instruction depend on a later one.

Out-of-Order Laundry: Don't Wait



- A depends on D; rest continue; need more

Superscalar Laundry: Parallel per stage



- More resources, HW to match mix of parallel

Superscalar Laundry: Mismatch Mix