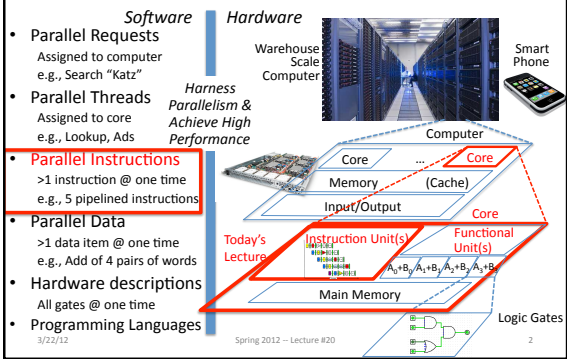
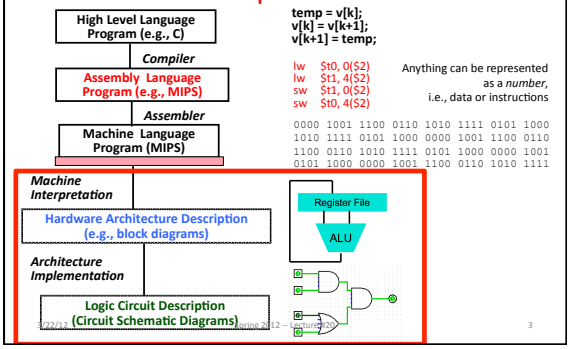
CS 61C:
Great Ideas in Computer Architecture
Control and Pipelining

Instructor:
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/sp12

You Are Here!



Levels of Representation/ Interpretation

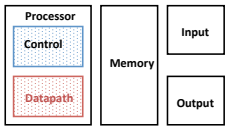


Agenda

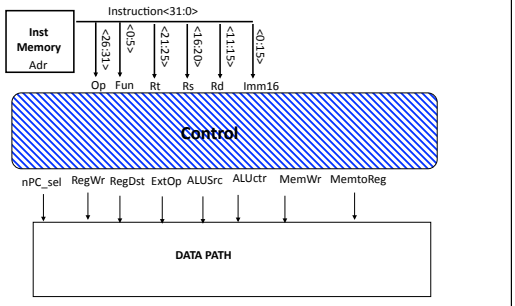
- Control Implementation
- Administrivia
- Vtune Demo (Rimas Avizenis)
- Pipelined Execution
- Pipelined Datapath
- (Pipeline Hazards)
- Summary

Review: Single-Cycle Processor

- Five 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: re-examine for pipelining
  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



Given Datapath: RTL → Control



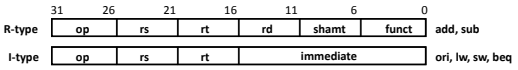
Summary of the Control Signals (1/2)

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"

Summary of the Control Signals (2/2)

See Appendix A → func / op

	10 0000 / 00 0000 add	10 0010 / 00 0000 sub	We Don't Care :-) 00 1101 ori	10 0011 lw	10 1011 sw	00 0100 beq
RegDst	1	1	0	0	x	x
ALUSrc	0	0	1	1	1	0
MemtoReg	0	0	0	1	x	x
RegWrite	1	1	1	1	0	0
MemWrite	0	0	0	0	1	0
nPCsel	0	0	0	0	0	1
Jump	0	0	0	0	0	0
ExtOp	x	x	0	1	1	x
ALUctr<2:0>	Add	Subtract	Or	Add	Add	Subtract



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
```

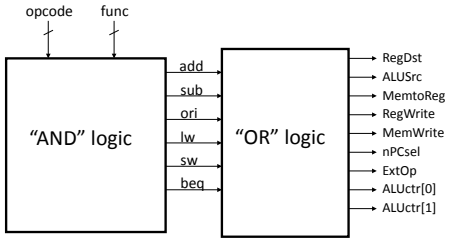
Where:

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

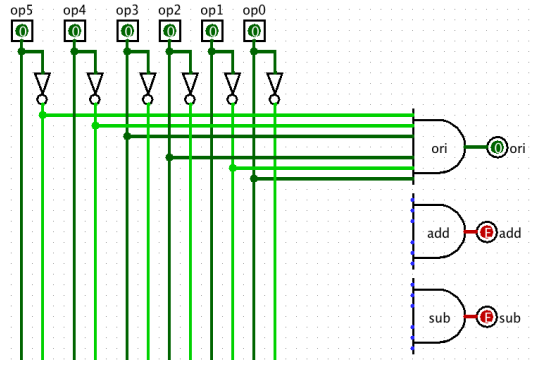
How do we implement this in gates?

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

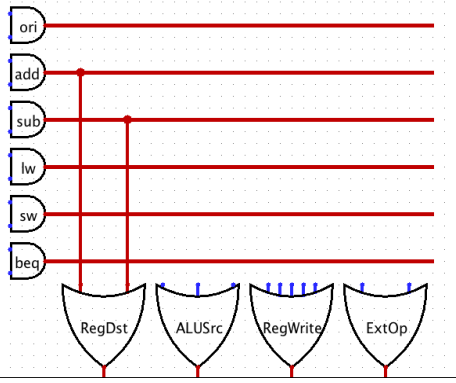
Controller Implementation



AND Control in Logisim



OR Control Logic in Logisim



Single Cycle Performance

- Assume time for actions are
  - 100ps for register read or write; 200ps for other events
- Clock rate is?

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

- What can we do to improve clock rate?
- Will this improve performance as well?
  Want increased clock rate to mean faster programs

Student Roulette?

Administrivia

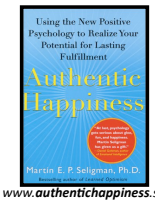
- Project 4: Pipelined Cycle Processor in Logicsim
  - Due Part 1, datapath, due 4/8, Part 2 due 4/15
- Extra Credit: Fastest Version of Project 3
  - Due 4/22 11:59 PM
- Final Review: April 29, 2PM-5PM, 2050 VLSB
- Final: Wednesday May 9 11:30-2:30 (TBD)

Getting To Know Your Prof

- Decided when got PhD to try to maximize happiness vs. wealth

✓Family
✓A job you enjoy
✓Friends
✓Helping others

- Playing?



*www.**authentichappiness**.sas.upenn.edu*

Getting to Know Your Prof

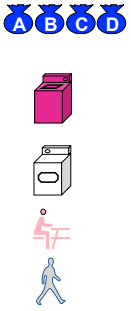
- Play soccer on Sunday mornings for >30 years
- After college, can forget it's important to find time to play (if want to be happy)

Agenda

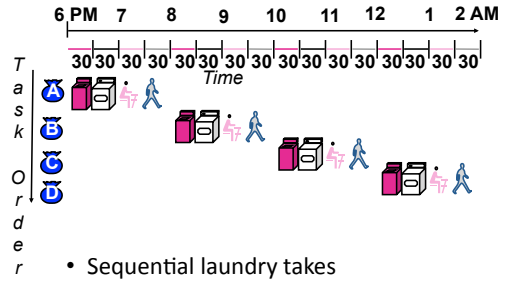
- Control Implementation
- Administrivia
- Vtune Demo (Rimas Avizenis)
- Pipelined Execution
- Pipelined Datapath
- (Pipeline Hazards)
- Summary

Pipeline Analogy: Doing 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

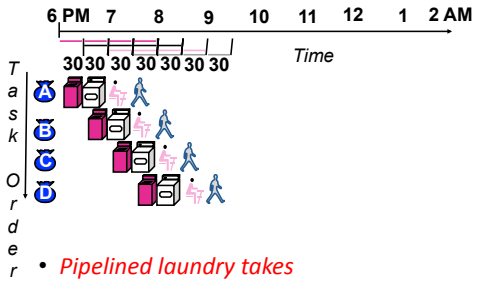


Sequential Laundry



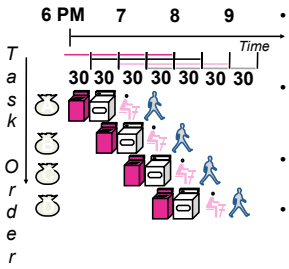
- Sequential laundry takes 8 hours for 4 loads

Pipelined Laundry



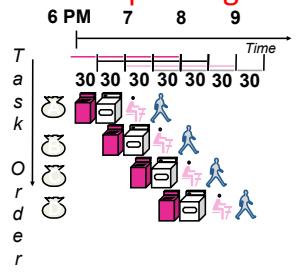
- *Pipelined laundry takes 3.5 hours for 4 loads!*

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 (4 in this case)
- Time to fill pipeline and time to drain it reduces speedup: 8 hours/3.5 hours or 2.3X v. potential 4X in this example

Pipelining Lessons (2/2)

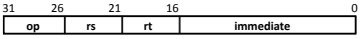


- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup

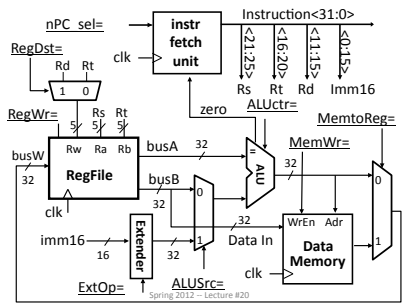
Review: RISC Design Principles

- "A simpler core is a faster core"
- Reduction in the number and complexity of instructions in the ISA → simplifies pipelined implementation
- Common RISC strategies:
  - *Fixed* instruction length, generally a single word (MIPS = 32b); Simplifies process of fetching instructions from memory
  - *Simplified* addressing modes; (MIPS just register + offset) Simplifies process of fetching operands from memory
  - *Fewer* and *simpler* instructions in the instruction set; Simplifies process of executing instructions
  - *Simplified memory access:* only load and store instructions access memory;
  - *Let the compiler do it.* Use a good compiler to break complex high-level language statements into a number of simple assembly language statements

Review: Single Cycle Datapath



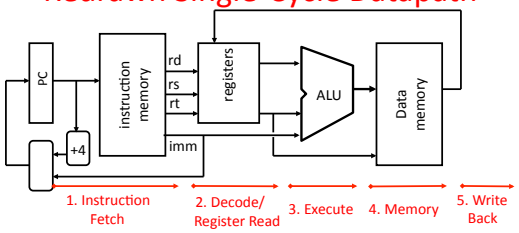
- Data Memory {R[rs] + SignExt[imm16]} = R[rt]



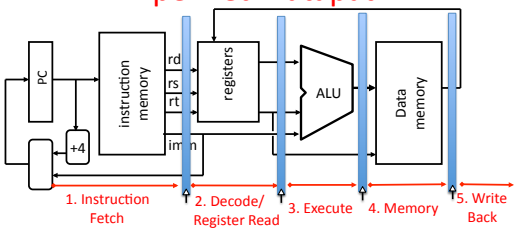
Steps in Executing MIPS

1) IF: Instruction Fetch, Increment PC
2) ID: Instruction Decode, Read Registers
3) EX: Execution
   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

Redrawn Single-Cycle Datapath



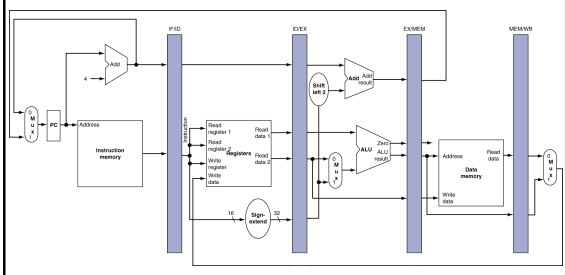
Pipelined Datapath



- Add registers between stages
  - Hold information produced in previous cycle
- 5 stage pipeline; clock rate potential 5X faster

More Detailed Pipeline

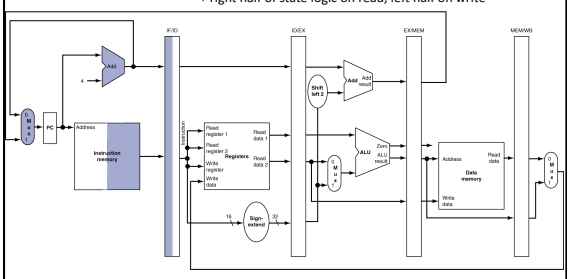
Registers named for adjacent stages, e.g., IF/ID



IF for Load, Store, …

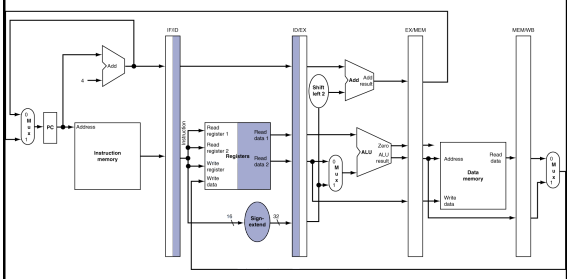
lw
Instruction fetch

Highlight combinational logic components used
+ right half of state logic on read, left half on write

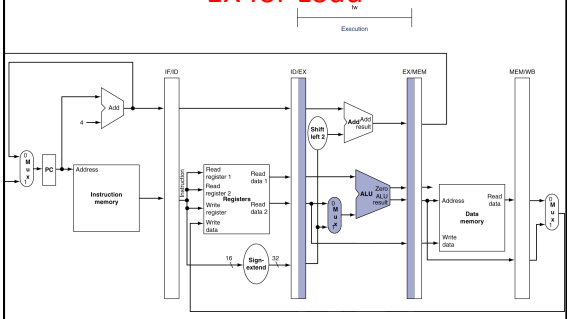


ID for Load, Store, …

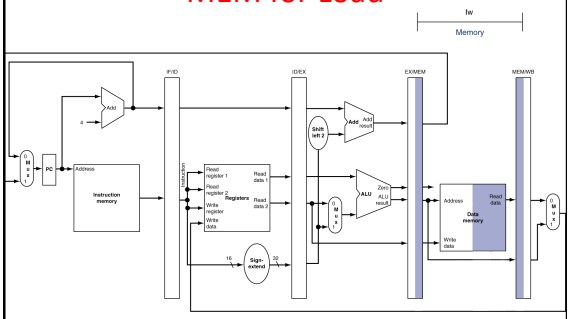
lw
Instruction decode



EX for Load

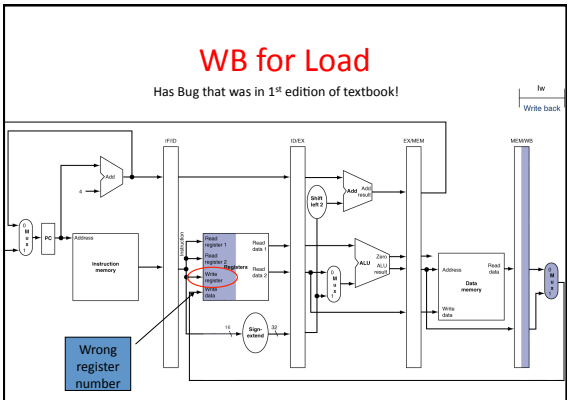


MEM for Load

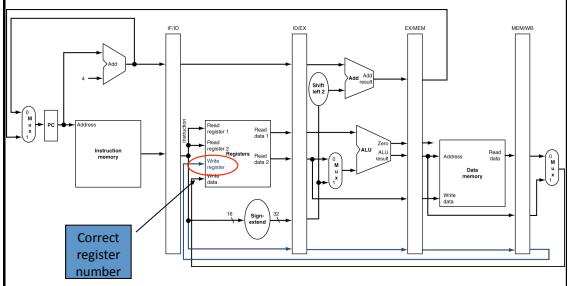


WB for Load

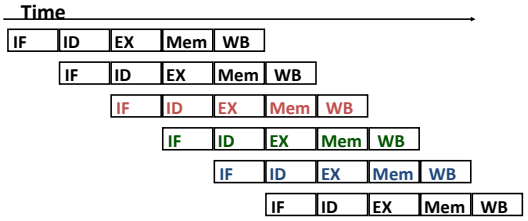
Has Bug that was in 1st edition of textbook!



Corrected Datapath for Load

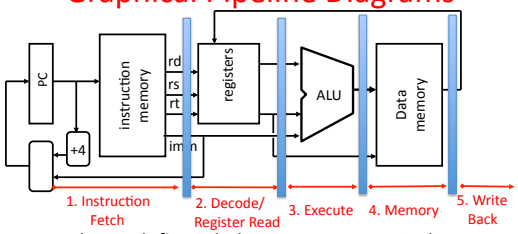


Pipelined Execution Representation

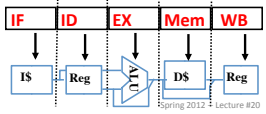


- Every instruction must take same number of steps, also called pipeline stages, so some will go idle sometimes

Graphical Pipeline Diagrams

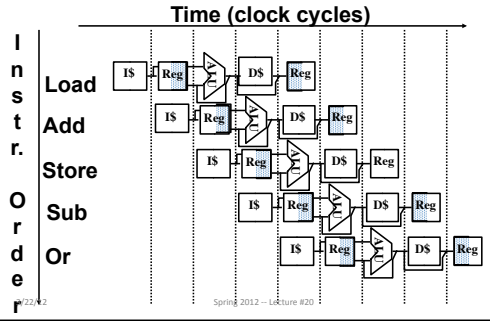


- Use datapath figure below to represent pipeline



Graphical Pipeline Representation

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



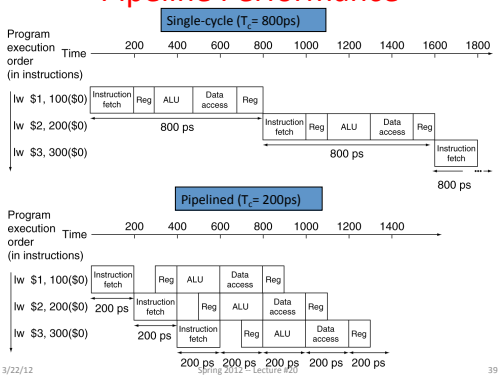
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Student Roulette?

Pipeline Performance



Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$ = $\frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Instruction Level Parallelism (ILP)

- Another parallelism form to go with Request Level Parallelism and Data Level Parallelism
  - RLP – e.g., Warehouse Scale Computing
  - DLP – e.g., SIMD, Map-Reduce
- *ILP – e.g., Pipelined Instruction Execution*
  - 5 stage pipeline => 5 instructions executing simultaneously, one at each pipeline stage

Agenda

- Control Implementation
- Administrivia
- Vtune Demo (Rimas Avizenis)
- Pipelined Execution
- Pipelined Datapath
- (Pipeline Hazards)
- Summary

Hazards

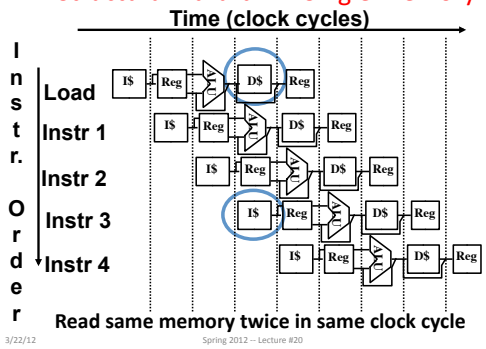
Situations that prevent starting the next logical instruction in the next clock cycle

1. Structural hazards
   – Required resource is busy (e.g., stasher is studying)
2. Data hazard
   – Need to wait for previous instruction to complete its data read/write (e.g., pair of socks in different loads)
3. Control hazard
   – Deciding on control action depends on previous instruction (e.g., how much detergent based on how clean prior load turns out)

1. Structural Hazards

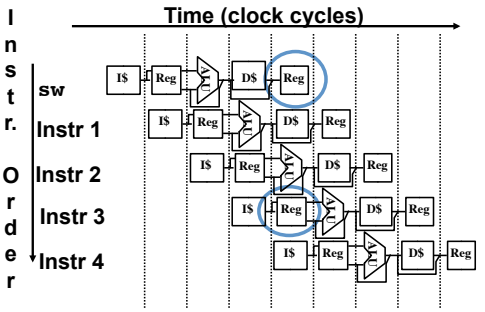
- Conflict for use of a resource
- In MIPS pipeline with a single memory
  – Load/Store requires memory access for data
  – Instruction fetch would have to *stall* for that cycle
    - Causes a pipeline "*bubble*"
- Hence, pipelined datapaths require separate instruction/data memories
  – In reality, provide separate L1 instruction cache and L1 data cache

1. Structural Hazard #1: Single Memory



Read same memory twice in same clock cycle

1. Structural Hazard #2: Registers (1/2)



Can we read and write to registers simultaneously?

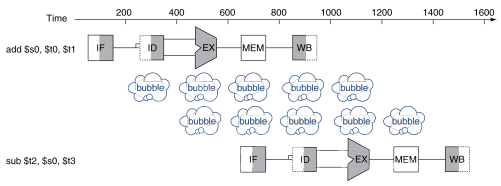
1. Structural Hazard #2: Registers (2/2)

- 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

2. Data Hazards

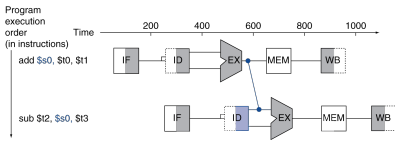
- An instruction depends on completion of data access by a previous instruction

```
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
```

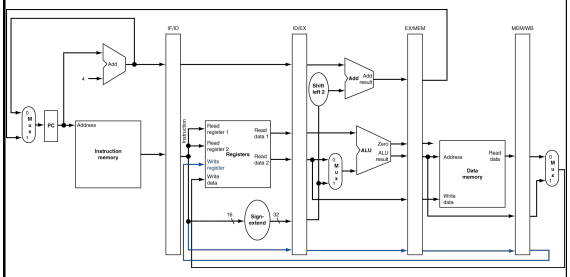


Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath

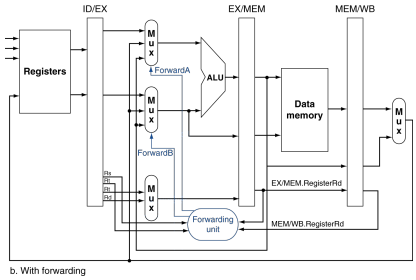


Corrected Datapath for Forwarding?



Student Roulette?

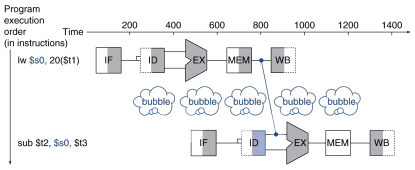
Forwarding Paths



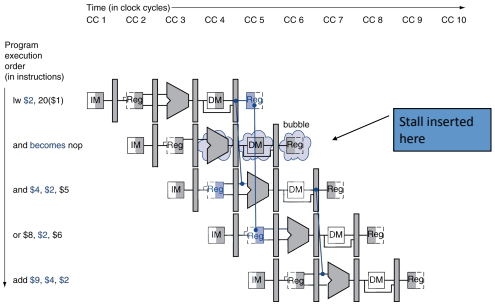
b. With forwarding

Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



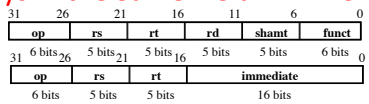
Stall/Bubble in the Pipeline



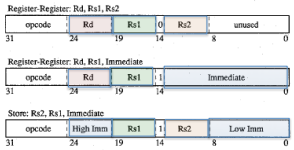
Pipelining and ISA Design

- MIPS Instruction Set designed for pipelining
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
  - x86: 1- to 17-byte instructions
  (x86 HW actually translates to internal RISC instructions!)
- Few and regular instruction formats, 2 source register fields always in same place
  - Can decode and read registers in one step
- Memory operands only in Loads and Stores
  - Can calculate address 3rd stage, access memory 4th stage
- Alignment of memory operands
  - Memory access takes only one cycle

Why Isn't the Destination Register Always in the Same Field in MIPS ISA?



- Need to have 2 part immediate if 2 sources and 1 destination always in same place

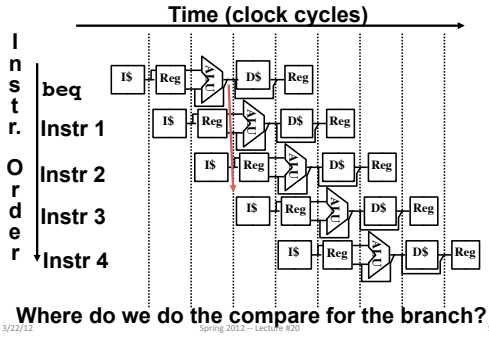


SPUR processor (A project Dave and Randy Katz worked on together)

3. Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- BEQ, BNE in MIPS pipeline
- Simple solution Option 1: *Stall* on every branch until have new PC value
  - Would add 2 bubbles/clock cycles for every Branch! (~ 20% of instructions executed)

Stall => 2 Bubbles/Clocks

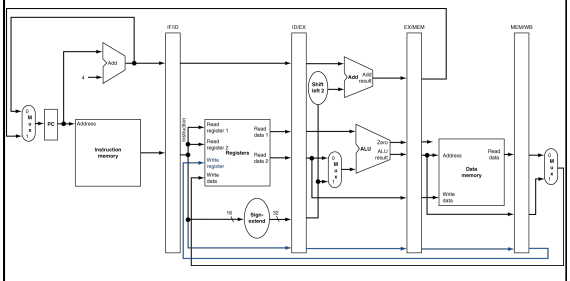


Where do we do the compare for the branch?

3. Control Hazard: Branching

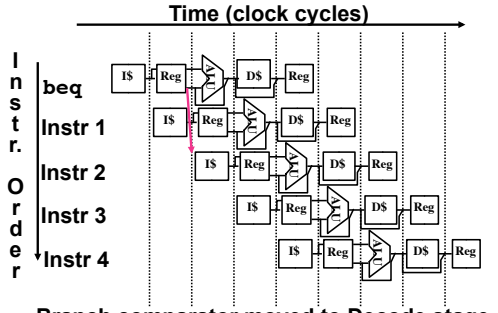
- Optimization #1:
  - Insert special branch comparator in Stage 2
  - As soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
  - Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only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

Corrected Datapath for BEQ/BNE?



Student Roulette?

One Clock Cycle Stall



Branch comparator moved to Decode stage.

3. Control Hazards

- Option 2: *Predict* outcome of a branch, fix up if guess wrong
  - Must cancel all instructions in pipeline that depended on guess that was wrong
- Simplest hardware if we predict that all branches are NOT taken
  - Why?

Student Roulette?

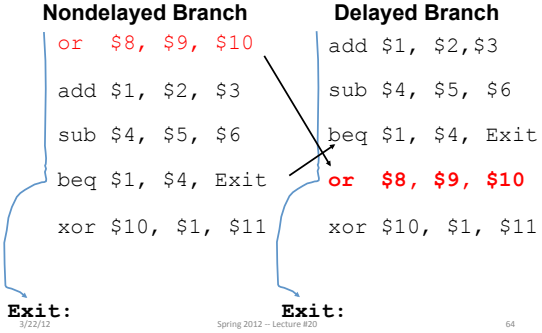
3. Control Hazard: Branching

- Option #3: 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 (the *branch-delay slot*)
- *Delayed Branch* means *we always execute inst after branch*
- This optimization is used with MIPS

3. Control Hazard: Branching

- Notes on Branch-Delay Slot
  - Worst-Case Scenario: put a no-op in the branch-delay slot
  - Better Case: place some instruction preceding the branch in the branch-delay slot—as long as the changed doesn't affect the logic of program
    - Re-ordering instructions is common way to speed up programs
    - Compiler usually finds such an instruction 50% of time
    - Jumps also have a delay slot …

Example: Nondelayed vs. Delayed Branch



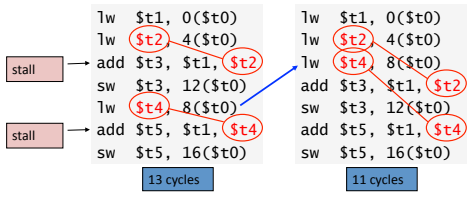
Delayed Branch/Jump and MIPS ISA?

- Why does JAL put PC+8 in register 31?

Student Roulette?

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `A = B + E; C = B + F;`



Peer Instruction

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

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

A)(orange) I is True and II is True

B)(green) I is False and II is True

C)(pink) I is True and II is False

D)(yellow) I is False and II is False

And in Conclusion, …

The BIG Picture

- Pipelining improves performance by increasing instruction throughput: exploits ILP
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure