inst.eecs.berkeley.edu/~cs61c

UC Berkeley CS61C : Machine Structures

Lecture 29
CPU Design : Pipelining to Improve Performance

2007-04-02

Lecturer SOE Dan Garcia

www.cs.berkeley.edu/~ddgarcia

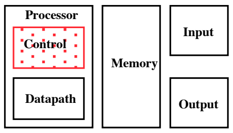
Wirelessly recharge batt ⇒

Powercast & Philips have developed a wireless power system. Pacemakers & defibrillators req surgery to replace dead batteries… not any more!

money.cnn.com/magazines/business2/business2_archive/2007/04/01/8403349/

Review: Single cycle datapath

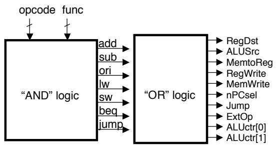
- 5 steps to design a processor
  - 1. Analyze instruction set ⇒ datapath requirements
  - 2. Select set of datapath components & establish clock methodology
  - 3. Assemble datapath meeting the requirements
  - 4. Analyze implementation of each instruction to determine setting of control points that 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,

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

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

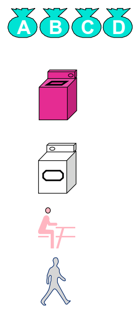
How do we implement this in gates?

Processor Performance

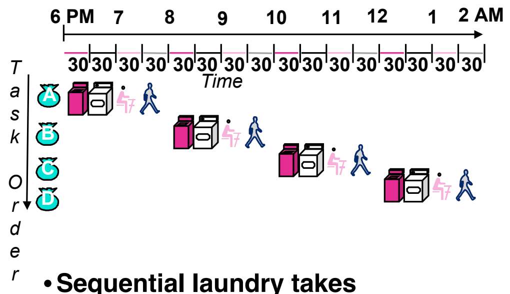
- Can we estimate the clock rate (frequency) of our single-cycle processor? We know:
  - 1 cycle per instruction
  - `lw` is the most demanding instruction.
  - Assume approximate delays for major pieces of the datapath:
    - Instr. Mem, ALU, Data Mem : 2ns each, regfile 1ns
    - Instruction execution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

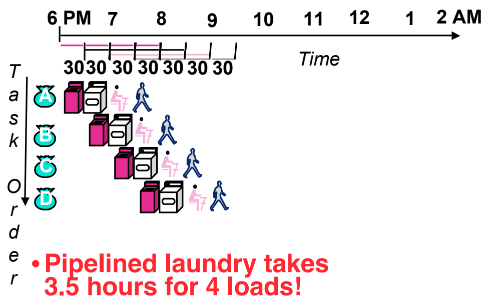


Sequential Laundry



- Sequential laundry takes 8 hours for 4 loads

Pipelined Laundry

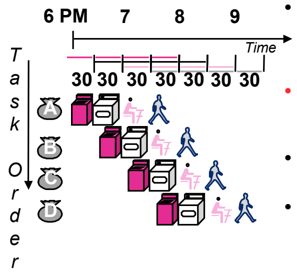


- Pipelined laundry takes 3.5 hours for 4 loads!

General Definitions

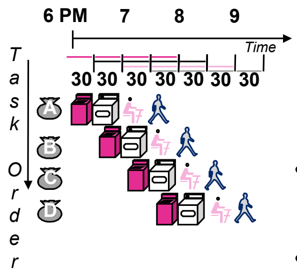
- Latency: time to completely execute a certain task
  - for example, time to read a sector from disk is disk access time or disk latency
- Throughput: amount of work that can be done over a period of time

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

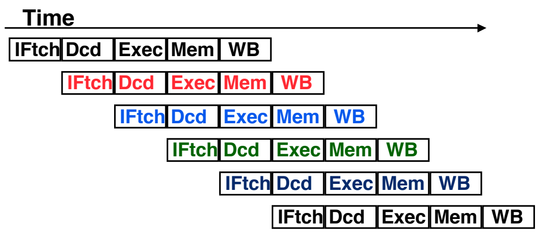
2) Dcd: Instruction Decode, Read Registers

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

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

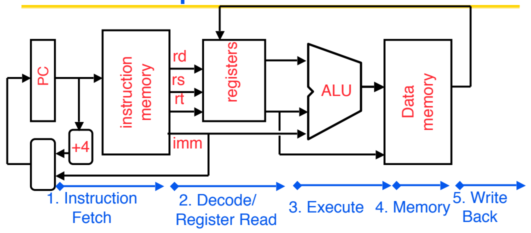
5) WB: Write Data Back to Register

Pipelined Execution Representation

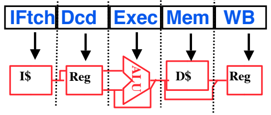


- Every instruction must take same number of steps, also called pipeline "stages", so some will go idle sometimes

Review: Datapath for MIPS

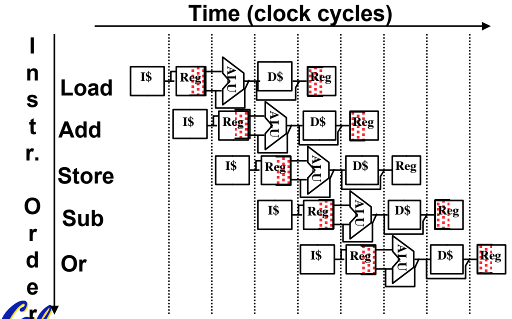


- Use datapath figure to represent pipeline



Graphical Pipeline Representation

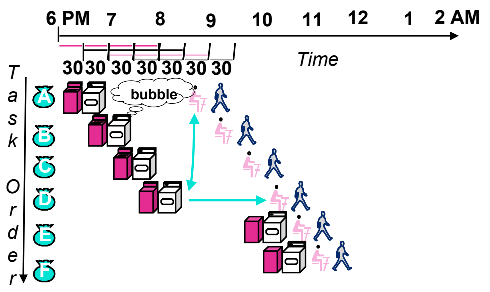
(In Reg, right half highlight read, left half write)



Example

- Suppose 2 ns for memory access, 2 ns for ALU operation, and 1 ns for register file read or write; compute instr 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

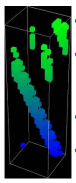
Pipeline Hazard: Matching socks in later load



A depends on D; stall since folder tied up

Administrivia

- Want to redo your autograded assignments for more credit?
  - We may have an opportunity for you…
- Performance Competition Up!

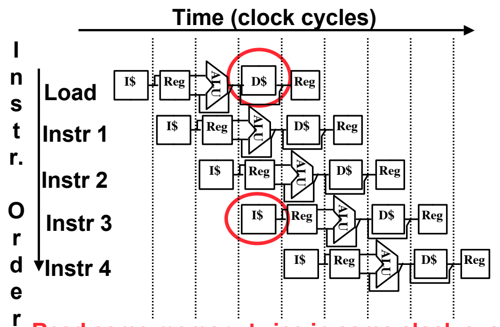


  - Rewrite HW2 to be as fast as possible
  - It'll be run on real MIPS machine (PS2)
    - You can optimize C or MIPS or BOTH!!
  - Do it for pride, fame (& EPA points)
  - Two competitions
    - Traditional (same spec as H2)
    - Unbounded (same H2 Extra for Experts spec)

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)

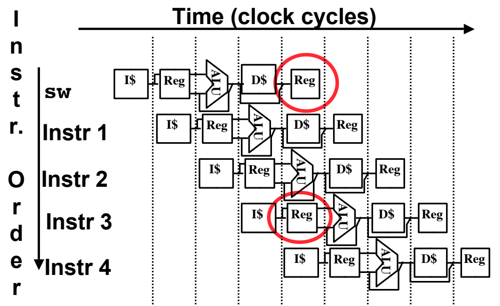


Read same memory twice in same clock cycle

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

Peer Instruction

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

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

C. We can rely on compilers to help us avoid data hazards by reordering instrs.

	ABC
0:	FFF
1:	FFT
2:	FTF
3:	FTT
4:	TFF
5:	TFT
6:	TTF
7:	TTT

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.