Slide 1

Administrivia

- Reminder: Homework 7 due Monday. (I revised this and Homework 8 to change the number of points. No other changes, except I fixed a glitch in the template answer for Homework 7, pointed out by an alert student!)
- Reminder: Quiz 5 Monday. Likely topic is design of processor, focusing on the circuit in Figure 4.24 (Figure 4.17 plus `j`).
- Quiz 4 graded. Almost everyone did well. Yay!
- Homework 5 graded. Almost everyone did well. Many did the extra-credit problems.

Slide 2

Homework 5 Essays

- Most common comment was probably about how much work it is to do conversions, but that working through an example helped with understanding. I hoped it would!
- Several said this one was easier than most; some others said the problems looked intimidating but then weren't so bad. A few said the video lectures were a help. Good to hear! And a few said they enjoyed the assignment.
- Several said it was interesting how the same bits can have multiple meanings.
- Several said it was interesting how non-integer values are actually stored.
- Some said working through the multiply and divide algorithms helped make sense of them. That was the goal!

Slide 3

Homework 7 Help — Tracing Operation of the Processor Circuit

- (I'll go through these slides quickly in class; they may be a useful summary when you start doing the assignment.)
- In this homework, you'll trace through what the circuit in Figure 4.17 is actually doing. Examples in video lecture(s) for April 3. Idea is for you to trace through what the circuit actually does rather than what you think it should do. But the two should match!
- So, you start with what you know — current saved value of the PC and what's at that address (in instruction memory) and contents of selected registers and data memory locations — and work from there. Taking the first few steps . . .

Slide 4

Homework 7 Help, Continued

- Right away you can write down output of PC and input/output of instruction memory. The problems give you the machine language for the instruction; it may be helpful to split it into fields before going on.
- Now you can write down all the control signals, the inputs and output of the top left adder, and the register-number inputs to the register file. You can get the control signals from the table in Figure 4.18.
- Once you have those, you can write down outputs of the register file and start figuring out what the main ALU is doing. You can also determine whether the top right adder and the data memory will be used (based on control signals).

Slide 5

Homework 7 Help, Continued

- Figuring out what the ALU does . . . You need to determine what operation it's doing (based on the `ALUop` control signal and the instruction function field, as shown in Figure 4.13). You also need to determine what the second operand is (contents of a register? sign-extended value from instruction?), again using control signals.
- "And so forth" . . .

Slide 6

Designing a Processor — Review/Recap

- So we've sketched the design of a processor that implements a supposedly representative set of instructions.
- A few more things to fill in . . .

Slide 7

Why Separate Instruction Memory and Data Memory?

- (Minute essay question for April 3. Sorry to spoil it for those who haven't watched the videos yet.)
- Design shows instruction and data memory separate.
- Why? isn't it all just ones and zeros? Yes, but . . .

Slide 8

Why Separate Instruction Memory and Data Memory? Continued

- Think about what has to happen on a `lw`. (Is this possible with a single memory?)
- (This is one of the textbook's "check yourself" questions.)

Slide 9

Implementing Jumps

- Discussion so far has omitted the `j` instruction. How should that work?
- We need to be able to get 26 bits from the instruction, shift them 2 bits left, combine with high-order bits of the current PC, and use that as the new PC. Figure 4.24 shows how . . .
- Is what's being added enough that the instruction can work?
- What should the values of the control signals be? (Think about this on your own. Potential quiz/exam question!)

Slide 10

Multi-Cycle Implementations

- So, we have a sketch for an implementation that executes one instruction per cycle. But clearly this isn't how all real systems work (if nothing else, most don't separate instruction memory from data memory).
- Why not? means cycle time is limited by length of longest path through the whole circuit, while many instructions can be done faster.
- What to do? break up work into multiple pieces . . .

Slide 11

Instruction Phases

- Work involved in fetching and executing a MIPS instruction can be split into phases:
  - Fetch instruction.
  - Read register operands and (at the same time) decode instruction. "At the same time" since inputs to the register file and inputs to the main control block all come from the instruction itself.
  - Do operation or address calculation.
  - Access data memory.
  - Write register result.
- How does this help? Two possibilities . . .

Slide 12

Simple Multi-Cycle Implementation

- One approach: Stick to the idea of executing one instruction at a time, but break things up so instructions potentially take multiple cycles.

  (This kind of implementation . . . Remember the discussion back in Chapter 1, in which different instructions took different numbers of cycles?)
- How's *that* going to help? Well . . .

Slide 13

Simple Multi-Cycle Implementation, Continued

- One potential payoff is skipping unused phases: E.g.., R-format (arithmetic/logic) instructions don't need to access data memory,
- Also, we don't need separate instruction/data memories.
- However, control logic becomes more complex: Must do everything we were doing before, plus keep track of which phase we're in. (Recall discussion of finite state machines from Appendix B.)
- Some previous editions of the textbook lay out a design for this (details later, maybe).

Slide 14

Pipelined Implementation

- Another approach is to use "pipelining": Modeled after assembly line; many real-world analogies possible. Textbook describes a laundry "assembly line", with stages corresponding to washing, drying, folding, and putting away.
- Could base a pipelined implementation of MIPS on the same phases used for a multi-cycle implementation, with one pipeline stage per phase.
- How does this help? well, doesn't make individual instructions faster, but means you can get more of them done in a given time.
- Like the simple multi-cycle implementation, it means added hardware complexity . . .

Slide 15

Pipelining — Implementation Overview

- First might observe that the five phases into which we've divided instruction processing seem to map onto the picture of our datapath: What we're doing is breaking up the flow of information through it into steps(!).
- So the idea will be: Somehow partition the datapath so each piece can work on a different instruction. For that to work, we have to add something ("pipeline registers") between pieces that saves results of one step for next step.
- Ignoring complications ("hazards", shortly next slides), this gives what's sketched in Figure 4.35.
- Textbook comments that MIPS ISA was designed for pipelining, and some aspects of the design reflect that (e.g., fixed-size instructions, fields common to all or at least many instruction formats). "Hm!"?

Slide 16

Pipelining — "Hazards"

- Another potential downside to pipelining (in addition to increased complexity): Have to worry about "hazards" — ways in which one instruction might interfere with another.
- Several ways in which things could go wrong . . .
- (Executive-level summary today; more next time.)

Slide 17

Pipelining Complications — "Structural Hazards"

- Idea is that two things we want to do at the same time conflict: E.g., read instruction from memory and read data from memory.
- Only solution is to avoid. For MIPS, we could just stick to separate instruction and data memories.

Slide 18

Pipelining Complications — "Control Hazards"

- Idea is that we need to make a decision but can't yet: E.g., can't know what instruction should logically follow a conditional branch until branch instruction is partly executed.
- Several possible solutions:
  - Stall: Just wait until we can be sure.
  - Predict: Make a guess, and if we guess wrong undo/redo.
  - Use delayed branches: Always execute instruction after conditional branch, then jump / don't jump. (This is what MIPS does — meaning that assembler programs we've written don't really represent how things work!)

Slide 19

Pipelining Complications — "Data Hazards"

- Idea is that we need data computed by one instruction before it would normally be available: E.g., two successive R-type instructions, or a load followed by an R-type instruction.
- Several possible solutions:
  - Stall: Just wait until data is available. (Probably not a good solution.)
  - Add hardware for "forwarding": Special hardware to route results to next instruction in addition to regular destination. May or may not be possible.
  - Use delayed loads: Don't allow instruction after "load" to use the result. (This is what original MIPS did.)

Slide 20

Minute Essay

- Many of you have said in essays that homework problems initially seemed daunting but then weren't once you got started on them. I'm curious — does this happen in other CSCI courses too?
- And please reply to my message to CSMajors about scheduling of O/S and Parallel!