Slide 1

Administrivia

- Reminder: If you're going to turn something in for Homework X1, please do so today, or Wednesday at the latest.
- Homework 6 due date moved to next Monday (two requests already for extensions).
- Quiz 5 scheduled for a week from today. More about topics next time.
- (A few comments about Quiz 4.)

Slide 2

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 3

Why Separate Instruction Memory and Data Memory?

- Design shows instruction and data memory separate.
- Why? isn't it all just ones and zeros? Yes, but . . . (Think about it a minute.)

Slide 4

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 5

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.

Slide 6

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, many don't separate instruction memory from data memory).
- Why not? means cycle time is limited by length of longest path through the whole path, while many instructions can be done faster.
- What to do? break up work into multiple pieces . . .

Slide 7

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" because of instruction format(s).
  - Do operation or address calculation.
  - Access data memory.
  - Write register result.
- How does this help? Two possibilities . . .

Slide 8

Simple Multi-Cycle Implementation

- One approach is to stick to the idea of executing one instruction at a time, but break things up so instructions potentially take multiple cycles. (How's *that* going to help? Well . . .)
- Control logic is now going to be 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.)
- However, one potential payoff is skipping unused phases — e.g.., the R-format (arithmetic/logic) instructions don't need to access data memory, and indeed we don't need separate instruction/data memories. A previous edition of the textbook lays out a design for this (review figures briefly).

Slide 9

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, it doesn't make individual instructions faster, but it means you can get more of them done in a given time.
- Like the simple multi-cycle implementation, it means added hardware complexity . . .

Slide 10

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 to somehow partition the datapath so we can have each piece working on a different instruction. But for that to work, we have to add groups of registers between pieces, so we save the results of one step for the next step.
- Ignoring complications ("hazards" — next slides), this gives what's sketched in Figures 4.33 and 4.35.

Slide 11

Pipelining — "Hazards"

- Another potential downside to pipelining (in addition to increased complexity) is that we have to worry about "hazards" — ways in which one instruction might interfere with another.
- Several ways in which things could go wrong . . .

Slide 12

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 go back to separate instruction and data memories.

Slide 13

Pipelining Complications — "Control Hazards"

- Idea is that we need to make a decision but can't yet — e.g., we can't know what instruction should logically follow a conditional branch until we have the branch 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 the assembler programs we've written don't really represent how things work.)

Slide 14

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 a "load" to use the result. (This is what original MIPS did.)

Slide 15

Minute Essay

- One performance advantage of a non-pipelined multi-cycle MIPS implementation is that not all instructions need all phases. Is this true for a pipelined implementation too? (Question based on another "check yourself".)
- Another advantage of a non-pipelined multi-cycle MIPS implementation is that it does not require separate instruction and data memories. Is this true for a pipelined implementation too? (Question based on another "check yourself".)
- Anything noteworthy to report about Homework 5 (the one about circuits and state machines)?

Slide 16

Minute Essay Answer

- It's still true that not all instructions need all phases (e.g., `j` needs only to be fetched and decoded), but this doesn't improve performance because of how pipelining works — it just means that not all steps/phases of the pipeline are in use on every cycle.
- No; since the pipelined implementation has to fetch an instruction on every cycle, it can't also be reading/writing memory unless instruction and data memories are separate.