Slide 1

Administrivia

- (By e-mail.)

Slide 2

Multi-Cycle Implementations — Recap/Review

- We've sketched out a simple implementation of a subset of the MIPS instruction set, in which all instructions are completed in a single cycle.
- But the single-cycle restriction has some annoying consequences, among them that the single cycle has to be long enough for bits to flow through the longest path through the whole circuit, even if for some instructions some parts aren't used.
- We could do better by relaxing that restriction . . .

Slide 3

Multi-Cycle Implementations — Recap/Review, Continued

- First step is to break instruction execution into "phases":
  - Fetch instruction.
  - Read register operands and "decode" instruction (generate control signals).
  - Do operation or address calculation.
  - Access data memory.
  - Write register result.
- Can then use these phases as basis for a simple multi-cycle implementation (one instruction at a time, but skip unneeded phases) or a pipelined implementation.

Slide 4

Pipelined Implementation

- Idea here is 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 5

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(!). (See Figure 4.33.)
- 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 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).

Slide 6

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 7

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 8

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 9

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 10

Pipelined Implementation — Some Details

- Figures 4.36 through 4.40 show some details of how this implementation works for different groups of instructions. Textbook's notation is that state elements whose right side is highlighted (blue) are being read, and those whose left side is highlighted are being written.
- Note that we now spot a flaw in the design: At the point where we need "which register to write to?", it's no longer correct. Figure 4.41 shows how to correct.

Slide 11

Pipelined Implementation — What's Left

- Need to be explicit about exactly what's needed for those "registers" between stages, but should sort of be common sense(?).
- Need to generate control signals, as in single-cycle implementation — and here, need to also add (some of) them to those interstage registers. Figure 4.51 shows result.
- Need to deal with data and control hazards. (Structural hazards don't exist for MIPS ISA — well, assuming we have separate instruction/data memories, as in the single-cycle implementation.)

  Textbook shows many details, interesting but a bit much for this course. But good to get key ideas . . .

Slide 12

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 13

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.