Slide 1

Administrivia

- (None.)

Slide 2

Multi-Cycle Implementations

- So, we have a sketch for an implementation that executes one instruction per cycle. But the textbook says it's not used (though might be for small/simple processors, and useful pedagogically).
- Why not? means cycle time is limited by length of longest path through the whole circuit, while many instructions can be done faster. (Plus it's spectacularly bad for some.)
- What to do? break up work into multiple pieces . . .

Slide 3

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 4

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 5

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, (So different instructions might take different numbers of cycles — as discussed way back in Chapter 1.)
- Also, we don't need separate instruction/data memories. (And we can possibly also eliminate some duplicate elements that are there only because of what we'll call shortly "structural hazards".)
- However, control logic becomes more complex: Must do everything we were doing before, plus keep track of which phase we're in. We can do that with a finite state machine, discussed in Appendix B, which we didn't make time for — but then, if we're skipping details of multi-cycle (as the textbook does — currently), it's not as critical.

  But a few words . . .

Slide 6

Finite State Machines

- Typically represent sequential logic blocks as "finite state machines", consisting of
  - Input(s).
  - Output(s).
  - Current state (one of a set of possible states).

  (For those of you who've taken Theory: These are the finite automata probably covered there.)
- Define FSM by Boolean expressions that map
  - Current state and input(s) to next state.
  - Current state and (optionally) input(s) to output(s).

Slide 7

Finite State Machines

- Appendix B example: Controlling a traffic light. (Figures B.10.1 through B.10.3 and surrounding text.)
- In general, idea is to:
  - Assign numbers to states, and figure out how many bits are needed to represent this (only one for example, more if more than two states).
  - Write down Boolean expressions for bits of next state (one for each bit) based on bits of current state and inputs.
  - Write down Boolean expressions for output bits based on bits of current state and inputs.
- (And once we have truth tables, we can get circuits!)

Slide 8

FSMs and Multi-Cycle

- Idea is to define FSM with one state for each stage of the design.
- Inputs are outputs of previous step; outputs include control signals.
- (Some previous editions of the textbook laid out a fairly detailed design for this. It was interesting if also something of a pain!)
- Why not used as much any more? Almost surely because the only performance gain is here is that some instructions take less time, while these days is on better performance through more parallelism. (Hardware designers haven't figured out to make single circuits go faster, but they can put more on a chip, and how else to use that?)

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 . . .

Slide 10

Pipelined Implementation, Continued

- It 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 11

Pipelining — Implementation Overview

- First might note again 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), 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!"?
- To be continued . . .

Slide 12

Minute Essay

- Questions?