Slide 1

Administrivia

- Grade summaries mailed. Overall averages were low, but not unusual for this course. Scores on midterm particularly disappointing, at least for some.

  "Will there be a curve?" not as such but I usually end up drawing boundaries between letter grades lower than the strict 90/80/etc. scheme would dictate.

  "Can I (or how much can I) improve my grade?" Yes; still plenty of points in play (three quizzes, another 100-point exam, more homework). Also usually I offer an opportunity to get extra credit at the end of the semester, plus possibly extra-point questions on individual assignments.
- Homework 5 on the Web. Due next Wednesday.
- Quiz 4 next Wednesday.

Slide 2

Minute Essay From Last Lecture

- Many people got the answer I had in mind — the "ripple effect" means that some operations take longer on 64 bits than on 32 — but some thought a 64-bit ALU would be faster. Something I'm not thinking of??
- One student pointed out that if the ALU always does all the operations but only selects one for output, doesn't that mean all operations take the same amount of time? in some sense yes!

Slide 3

Circuit Design — Review

- Idea here is to design circuits that can implement various things, using AND gates, OR gates, and inverters.
- "Combinational logic" blocks implement Boolean functions/operations — map input(s) to output(s) without a notion of persistent state. We looked at some examples last time.
- "Sequential logic" blocks also implement Boolean functions/operations but do include a notion of persistent state — such as what's needed for registers.

Slide 4

32-Bit ALU — Review

- Last time we looked at the textbook's design of a simplified 32-BIT ALU that can do most of the arithmetic and logic operations of the MIPS ISA. (Look more carefully at `slt`?)
- Can think of how this works as follows: Values (voltages?) at inputs flow through the circuits to produce outputs. Changes in input producce changes in output, after some nonzero delay that depends at least in part on how many gates there are between input and output (hence the "ripple effect".
- Where do inputs come from? something that can store values.

Slide 5

Memory Elements

- Next step is to come up with a logic block that can hold a value:
  - Inputs are old value, "set" (to 1), "reset" (to 0).
  - Outputs are value, negation of value.
- An unclocked logic block that can do this — Figure B.8.1. ("Unclocked"? more about clocking next.)
- But in a typical design, you want to use these both as inputs and outputs to combination-logic blocks (think for example about how a MIPS `add` on registers should work). How is this possible? how could values ever "settle down"?

  First, a little about clocking . . .

Slide 6

A Very Little Bit About Clocking

- Many (most, currently?) hardware designs are based on the idea of a "clock" — something that generates regular signal changes and can be used to control when updates to state elements happen.
- As sketched in section B.7 — inputs/outputs to combinational logic block are connected to state elements. Input values are "sampled" at one point in the clock cycle and written out at a different point in the cycle — "synchronous" circuit. (So does that mean "asynchronous" circuits are also possible? yes, but well beyond the scope of this course.)
- Why do this? as a way to avoid race conditions.
- One implication, though, is that the clock cycle has to be long enough for the slowest combinational logic block!

Slide 7

Memory Elements, Continued

- Can then extend the circuit of Figure B.8.1 to something that only samples (data) input when clock input is 1 (“D latch”, Figure B.8.2) and further to something whose output only changes when clock input is 0 (“D flip-flop”, Figure B.8.4).
- Notice how these figures use the “layers of abstraction” idea — first show details of a “latch”, then show using it as a black box to build something more complex.

Slide 8

Register Files

- (Notice here that “file” here has essentially nothing in common with what we usually mean by “file” in CS.)
- So now we have something that can read/write/save one bit, and we know (in principle) how to control when its value is read and written. But what we want is a bunch of “registers” that can each read/write/save 32 bits.
- Usual approach — “register file”, a logic block that holds a bunch of values and allows us to read and write them. Figures in section B.9 give more details (next slide) — and this should look like something that would be useful in implementing MIPS instructions with register operands, no?

Slide 9

Register Files, Continued

- Inputs:
  - Two (multi-bit) register numbers saying which registers we want to "read" (use as input to some operation).
  - One (multi-bit) register number saying which register we (might) want to "write" (change the value of).
  - One (32-bit) value to (maybe) save in a register.
  - A "yes do a write" bit.
- Outputs:
  - Two (32-bit) values representing the contents of the two registers selected by the "read register" numbers used as input.

Slide 10

SRAM and DRAM

- What about RAM (Random Access Memory)? in some ways this is much like a register file, but with a single address rather than three register numbers, as shown in Figure B.9.1.
- Internal details . . . Two options (at least):
  - Static RAM ("SRAM"), which maintains state as long as there's power and is pretty similar to the implementation of a register file.
  - Dynamic RAM ("DRAM"), which makes use of capacitors as well as transisters and has to be refreshed periodically.

  (Guess which one "costs" more.)

Slide 11

The Big Picture, Revisited

- We've sketched what we need for the "datapath" part of a MIPS processor — combinational logic blocks to perform arithmetic/logic operations (ALU), sequential logic blocks to store information (register file, RAM).
- Now we need something to control it — which may also involve sequential logic blocks. So another detour through Appendix B . . .

Slide 12

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 the theory course, 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).
- Appendix B example — controlling a traffic light. (Figures B.10.1 through B.10.3 and surrounding text.)

Slide 13

Implementing the MIPS Architecture

- Goal of Chapter 4 is to show how we could use the low-level building blocks described in Appendix B to implement a proof-of-concept subset of the architecture (instructions, registers, etc.) we've defined.
- "Proof of concept"? yes, the subset we'll implement may not be enough to do anything useful or interesting, but it should be enough to illustrate how we could implement the rest of the architecture.

Slide 14

Subset to Implement

- Representative memory-access instructions (`lw`, `sw`).
- Representative arithmetic/logical instructions (`add`, `sub`, `and`, `or`, `slt`).
- Representative control-flow instructions (`beq`, `j`).

Slide 15

Overview

- Very simplified view of what a processor does: Fetch next instruction. Figure out what it is and execute it. Lather, rinse, repeat.

  Implicit in this description is a notion of "next instruction", which normally moves through the stored program in sequence but not always (e.g., for control-flow instructions).

- What we have to work with: Two kinds of "logic blocks" described in Appendix B. (To be continued . . .)

Slide 16

Minute Essay

- How did the exam compare to your expectations, with regard to length, difficulty, topics . . .
- If you didn't do well, why do you think you didn't? was it not understanding the material, not reviewing everything that was asked about, not being able to work quickly enough, something else?