Slide 1

Administrivia

- Reminder: Homework 5 due today.
- Remaining quizzes scheduled. First of them next Monday. More about topics next time.
- Homework 6 posted. Due in a week.
- Remember that you do get attendance points for sending me responses to the video lecture minute essays. As of my writing this, of the 42 students enrolled, 21 responses for 3/22, 17 for 3/25.

Slide 2

Designing a Processor — Review

- "Big picture" of Figure 4.1 consists of two kinds of components:
- Combinational logic blocks (ALU and adders): Map inputs to outputs with no notion of persistent state. We looked at the textbook's design of a simplified ALU. But where do those inputs come from, and what happens to the outputs? Well . . .
- Sequential logic blocks: Include a notion of persistent state, and can be built around "memory elements". Look at those next . . .

Slide 3

Memory Elements

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

  First, a little about clocking . . .

Slide 4

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 clock cycle and written out at a different point in the cycle — "synchronous" circuit. (So does that mean "asynchronous" circuits are also possible? Yes, though well outside the scope of this course. Research area.)

Slide 5

A Very Little Bit About Clocking, Continued

- Overall scheme as in Figure B.7.2. (Could be clearer.) Idea is that we want, between state element 1 (input) and the CL block some kind of barrier/switch that can either let bits flow or not, and the same thing between the CL block and state element 2, with only one of those barriers letting bits flow at a time.
- 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 6

Memory Elements, Continued

- Figure B.8.2 shows such a barrier ("latch") — circuit that stores one bit and only samples data input when clock input is 1. Details interesting but not really crucial for this course!
- Notice how this 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 7

Register Files

- (Note 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", logic block that holds many values and allows us to read and write them. Figures B.8.7 and following give more details (next slides), and this should look like something that would be useful in implementing MIPS instructions with register operands, no?

Slide 8

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 9

Register Files, Continued

- Figure B.8.7 shows "big picture".
- Figures B.8.8 and B.8.9 show some of details. Note that looks sort of like top-down design as used in the world of programming: Start at fairly high level of abstraction and then fill in details.

Slide 10

SRAM and DRAM

- What about RAM (Random Access Memory)? in some ways 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 transistors 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 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 13

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.

Slide 14

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 15

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 16

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 17

Minute Essay

- We sketched a somewhat-simple design for a 32-bit ALU. We could make a 64-bit ALU in much the same way. Comparing the two in terms of how long it would take to do each of the discussed operations, which would you guess to be faster (if either)?
- Does the answer to the previous question depend on which instruction is being executed?

Slide 18

Minute Essay Answer

- The 64-bit ALU will be slower for some operations (such as `add`), since "values" have "flow" through 64 1-bit ALUs rather than 32.

  (However, as one student pointed out, if the ALU is doing all the operations anyway even though only one is being used, in some sense they do all take the same amount of time.)