Slide 1

Administrivia

- I *think* I figured out my problems with Zoom sound, so if you want to speak up in class, you can with better prospects that I'll hear you!
- Reminder: Homework 6 due Monday, at 11:59pm. If the lack of office hours this week has been a problem, I could Zoom-meet with you Friday if you're willing. (It's looking like I'm going to need to use Monday's usual office hour for advising as well. Sorry but this happens . . .)
- I'm scheduling a quiz for Monday, over the same material as Homework 6. How to do this . . .

Slide 2

More Administrivia

- I'm thinking do the remaining quizzes (we have three to go!) as follows:

  I'll put the quiz in our shared folder on Google Drive, before class time Monday. You pick a time when you can spend 15 minutes taking the quiz, between then and 11:59pm, take the quiz (usual "open book / notes" rules), and send me your answers as you did for Exam 1. Please don't spend more than 15 minutes on it. I'l plan to end class a bit early Monday so if you want to do the quiz at a time when I can answer any questions, you can do it then.

  Okay? (I'll make that a minute essay question.)
- (If this works I may try the same thing for Exam 2. Scheduling still TBA; I'm going to let it depend in part on experience with at least one quiz.)

Slide 3

Designing a Processor — Overview, Recap/Review

- Goal of Chapter 4: Sketch design of a hardware implementation of a subset of MIPS architecture in terms of simple building blocks (AND and OR gates, inverters).
- Again, big picture is that we need something that can
  - Provide short-term storage of values (registers).
  - Perform arithmetic and logical operations on these values.
  - Provide longer-term storage of values (memory).
  - Transfer data between registers and memory.
  - Repeatedly fetch and execute instructions, allowing for both sequential execution and branching/jumps.

Slide 4

Designing a Processor, Continued

- Key components of the design (Figures 4.1 and 4.2):
  - Something to implement memory.
  - Something to implement instructions: “ALU” (arithmetic/logic unit).
  - Something to implement registers: “register file”.
  - Something to implement fetch/decode/execute cycle: “control logic”.

  The first three together make up the “data path”. Analogy: It’s a puppet, with “control” pulling its strings.

Slide 5

Circuit Design — Recap/Review

- Design (for us anyway) is in terms of AND and OR gates, inverters. Can represent all circuits with explicit diagrams. Note that some blocks (those with no persistent state) can be represented with Boolean expressions and/or truth tables.
- 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 6

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 7

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 8

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 9

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 figures use the “layers of abstraction” idea: E.g., first show details of a “latch”, then show using it as a black box to build something more complex.

Slide 10

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 11

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 12

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 13

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 14

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 . . .

  (Or not! in years past we needed material on finite state machines; now we don’t, so defer for now.)

Slide 15

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 16

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 17

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 18

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 depend on which instruction is being executed?
- Does what I plan for quizzes seem like it can work for you? I want to do this in a way that doesn't depend so much on your having a good environment at a particular time!
- I hear from some students that it can be hard to pay attention to these by-video classes — more distractions, plus one student said it felt more like watching YouTube videos than going to class (and how many play close attention . . .). True for you?

Minute Essay Answer

Slide 19

- 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.)