Slide 1

Administrivia

- Homework 3 coming soon, tomorrow I hope. I will send mail.
- Textbook includes description of a “hardware description language” and uses it in examples. Okay to skim/skip.

Slide 2

Minute Essay From Last Lecture

- (See notes from last time.)
- One person commented that the diagrams are intimidating at first glance but make sense on more-careful inspection. Agreed!

Slide 3

Hardware Description Languages — Executive-Level Summary

- "Hardware description languages" can be used to represent the circuit designs discussed in Appendix B. Useful as description/specification and also as input to tools that can generate logic blocks.
- Two commonly-used ones are Verilog and VHDL; textbook uses Verilog. Discussion and examples in section B.4.
- Syntactically, Verilog looks more or less like C, but there's (at least) one significant difference: It needs to represent not only sequences of assignments (where each one completes before the next one starts) but also blocks of assignments that execute in parallel. (Think in terms of values flowing through the pictures we've been drawing — fast but not infinitely so, so where possible we want to do things simultaneously rather than in sequence.)

Slide 4

Design of an ALU

- One of the things we need for a MIPS implementation is something that can do the arithmetic and logic operations in the MIPS instruction set.
- Inputs to operations are typically two 32-bit values. Some operations can be done by operating on all bits in exactly the same way and independently (e.g., `and`). Others can be done by operating on all bits in the same way but with dependencies among bits (e.g., `add`). So we will design a "1-bit ALU" and then figure out how to connect 32 of them to make the full 32-bit logic block.

Slide 5

1-Bit ALU

- Figures B.5.1 through B.5.6 show how we can build up something that performs `and`, `or`, and `add` on 1-bit values (plus carry-in and carry-out values for `add`).
- Result (B.5.6) is a logic block with inputs
  - two 1-bit operands
  - 2-bit "which operation?"
  - 1-bit carry-in

  and outputs
  - 1-bit result
  - 1-bit carry-out

Slide 6

32-Bit ALU from 1-Bit ALUs

- Now we want to connect 32 of these 1-bit ALUs to make a 32-bit ALU.
- Figure B.5.7 shows how:
  - Connect operand inputs of each 1-bit ALU to individual bits of 32-bit operand, and similarly for 32-bit result.
  - Connect "which operation?" input (common to all) to "which operation?" input of each 1-bit ALU.

Slide 7

32-Bit ALU from 1-Bit ALUs, Continued

- We said when we first talked about two's complement notation that it was attractive because once you build something that can add, you can easily extend it to something that can subtract, right?
- Conceptually, we can compute $a - b$ by adding a to $-b$, and we can compute $-b$ by reversing all the bits of b and adding one — which is just what's shown in Figure B.5.8! which is Figure B.5.7 plus one more input, which:
  - if 0, makes the initial carry-in 0 and uses b as is.
  - if 1, makes the initial carry-in 1 and flips bits of b.
- We can apply a similar idea (adding an input that lets us use a as is or "flipped") to implement `nor` (Figure B.5.9).

Slide 8

32-Bit ALU from 1-Bit ALUs, Continued

- Figures B.5.10 and B.5.11 and accompanying text show how to extend the design to implement `slt` and also an overflow detector. Executive-level summary: Calculate $a - b$ and use high-order bit of result of that operation to set low-order bit of result.
- Result is something we can use to do pretty much all of the arithmetic and logic operations of the MIPS ISA. Exceptions are shifts (but those don't seem like they'd be too hard) and multiplication/division (which do, so skip for now).

Slide 9

Memory Elements

- So now we (sort of) know how to design logic blocks that use switches/gates to compute output bits from input bits.
- But where do those input bits come from, and where do the output bits go? "state elements" — things that can save values.
- (Keep in mind that the goal here is to get a sense of how you can build something that stores a value out of gates/switches. Details are (I think!) very interesting but can to some extent be skimmed.)

Slide 10

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 11

Memory Elements, Continued

- Idea here 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.

Slide 12

Memory Elements, Continued

- Can then extend this 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 we're starting with simple things and using them to construct more complicated things — much as you do in writing software. "Hm!" ?

Slide 13

Register Files

- So now we have something that can read/write/save one bit. But what we want is a bunch of "registers" that can each read/write/save 32 bits. What to do?
- 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 three register operands, no?

Slide 14

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 15

SRAM and DRAM

- What about RAM (Random Access Memory)? in some ways, extension of register-file idea — Figure B.9.1.
- Internal details are different, though, and there are two options:
  - Static RAM (“SRAM”), which maintains state as long as there’s power.
  - Dynamic RAM (“DRAM”), which has to be refreshed periodically.

  (Guess which one “costs” more.)

Slide 16

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) and store information (register file).
- Now we need something to control it — a sequential logic block.

Slide 17

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).
- 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 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 to the previous question depend on which instruction is being executed?

Slide 19

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.