Slide 1

Administrivia

- Homework 3 to be on the Web later today. I will send mail. Due a week from Tuesday.
- Next quiz a week from today.
- Appendix B includes description of a “hardware description language” and uses it in examples. Okay to skim/skip.

Slide 2

A Little About Circuit Design — Overview/Recap

- Our goal — to sketch design of implementation(s) of MIPS ISA in terms of simple “logic gates” (things that implement Boolean operators and, or, not).
- Start by looking at “combinational logic blocks” — blocks that map zero or more binary inputs to one or more binary outputs. Examples include single logic gates, “add” circuit from last time, etc. Define a combinational logic block by naming its inputs and outputs and how outputs depend on inputs. Can do this using truth table or Boolean expressions.

  (Other major class of circuits is “sequential logic blocks”, which are similar but have a notion of saved state.)

Slide 3

Two-Level Logic

- Constructing logic blocks that implement arbitrary Boolean algebra expressions could take some thought.
- However, any Boolean-algebra expression can be represented in one of two forms — sum of products or product of sums. (Why? Think about truth-table representation.)

Slide 4

Two-Level Logic Implementations

- So we can define, for any combinational logic block, something that maps n inputs to m outputs by connecting an "array" of AND gates (one for each combination of inputs) to an "array" of OR gates (one for each output). (Example in B.3.)
- Notice that representation in Figure B.3.5 could be changed to represent a different function by changing the positions of the dots — so generic term "programmable logic array" (PLA) makes sense?
- Another standardized way to represent combinational logic block is "ROM" (read-only memory) — for n inputs and m outputs we'd need 2^n entries each consisting of m bits.
- For either of these the process of turning a truth table into implementation can be automated.

Slide 5

"Managing Complexity"

- Worth noting that, as in programming, the discussion will make extensive use of layers of abstraction to build complex things from simple things.
- Just as in programming it's common to define library functions that implement frequently-used operation, we can define some not-so-basic blocks, such as decoders and multiplexors. (See discussion in B.3, especially Figure B.3.2.)

Slide 6

"Don't Care" Inputs/Outputs

- For not-so-small numbers of inputs a full truth table can be big, so it's worthwhile to think about whether there's something simpler that gets the same effect.
- One way to do this — exploit "don't care"s. Input "don't care" arises when both values for an input (in combination with other inputs) give same result. Output "don't care" arises when we aren't interested in output for some combination inputs (maybe it can never occur?). Textbook shows how to use this idea to produce a shorter truth table.
- Exploiting the shorter table, and in general minimizing the complexity of the combinational logic block, can be done manually ("Karnaugh maps") or automatically (various design tools).

Slide 7

Arrays of Logic Elements

- Descriptions so far (except for decoder) have been in terms of single-bit inputs. But often we want to work on word-size collections (e.g., 32 bits of register).
- To do this, we (usually?) can build an "array" of identical logic blocks.
- If inputs/outputs are not in some way connected, can just indicate that input/output values are more than one bit ("bus"). Example — bitwise AND of 32-bit values.
- If inputs/outputs *are* connected, idea still works but picture must indicate connections. Example — addition of 32-bit values using 32 single-bit "adder" blocks, each with three inputs (two operands and carry-in) and two outputs (value and carry-out).

Slide 8

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 9

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 10

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 11

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 12

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).

  Notice also that getting valid output values may take a while for some operations, such as addition — values "flow" through the circuit. Designers of real hardware use clever tricks to speed up addition, such as the one(s) described in B.6. Read if interested!

Slide 13

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 14

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 15

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 16

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).
- "Layers of abstraction" idea mentioned earlier — here's an example.

Slide 17

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 18

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 19

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 20

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. To be continued . . .

Slide 21

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 22

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.