Slide 1

Administrivia

- Grade summaries uploaded to Google Drive in shared-with-you folder(s). I hope this is helpful, and I'm sorry about how long it took!

  Please do double-check that grades are consistent with previously-reported information. I try to avoid mistakes, but.

- Homeworks 5 and 6 posted. Homework 5 is totally optional (extra credit); Homework 6 has two required problems and two optional.

  (Note modified instructions for turning work in — basically same as for Exam 1).

Slide 2

Designing a Processor — Overview, Recap/Review

- Goal of Chapter 4: Sketch design of a hardware implementation of MIPS architecture in terms of some simple building blocks (AND and OR gates, inverters). (Actually only a subset of instructions, but enough to give you the idea?)
- May be useful to keep in mind the goal. 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 3

Circuit Design — Overview

- AND and OR gates implement Boolean-algebra functions of the same names; inverter implements “not”.
- A word about notation: We’ll use the textbook’s notation for Boolean algebra, which alas is (probably?) different from what you used in CSCI 1323.

CSCI 2321	*CSCI 1323*
$a \cdot b$	$a \wedge b$
$a + b$	$a \vee b$
$\overline{a}$	a'

Slide 4

Implementing Logic Gates — Executive-Level Summary

- The ones and zeros of low-level software become two distinct voltages in hardware, and the logic of Boolean algebra is implemented using “switches” (things that connect an input to an output, or not, depending on the state of a control input).
- Currently these switches are (usually?) transistors. A popular technology is CMOS (“Complementary Metal-Oxide Semiconductor”).

Slide 5

CMOS — Executive-Level Summary

- Basic idea is that this technology offers two ways to build "switches", one that "conducts 0s well" and one that "conducts 1s well".
- Can put these together with two sources of constant voltage, one representing 1 and the other 0, to get something that can implement logic gates.
- I wrote up a programmer's take on this, with an example; available under "useful links" on course Web site. (Review?)

Slide 6

Circuit Design, Continued

- Two basic types of blocks:
- "Combinational logic" blocks implement Boolean functions/operations — map input(s) to output(s) without a notion of persistent state. (Think of these as "pure" functions that don't change any variables but can have multiple outputs.)
- "Sequential logic" blocks also implement Boolean functions/operations but include a notion of persistent state. (Think of these as methods in object-oriented programming, which map input(s) to output(s) but also have access to member variables that can be read/written.)

Slide 7

Combinational Logic

- How to specify combinational logic block?
- One way: Truth table with one line for each combination of inputs.
- Another way: Boolean-algebra expression(s) that define output(s) in terms of input(s).
- Example: 1-bit "adder". Inputs are two digits to add and a carry-in, and outputs are sum and carry-out. So we would need a truth table with how many rows? how many columns? (Figure B.5.3.)

  Or we could come up with appropriate Boolean expressions and then turn those into circuits. (Figure B.5.5.)

Slide 8

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 9

Two-Level Logic Implementations (Skim)

- 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 Figure B.3.5.)
- Note 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 10

"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, can define some not-so-basic blocks. Two examples:
- Decoder (Figure B.3.1) maps from n-bit input to 2^n outputs (one for each combination of input bits). (This one we may not use much.)
- Multiplexor (Figure B.3.2) takes n-bit control input and 2^n other inputs and selects one of the inputs based on control input. We'll use this one a lot!

Slide 11

"Don't Care" Inputs/Outputs (Skim)

- 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 of 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 12

Arrays of Logic Elements

- Descriptions so far (except for decoder) have been in terms of single-bit inputs. But often want to work on larger collections (e.g., 32 bits of a register).
- To do this, 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"). Examples: bitwise AND of 32-bit values, Figure B.3.6.
- 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). (Figure shortly.)

Slide 13

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. (Again, we'll look only at a subset.)
- Inputs to operations 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 14

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 15

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 16

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 17

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 overflow detection. 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 a useful subset 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).

  Note 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. Read section B.6 if interested!

Slide 18

Minute Essay

- Questions? For most of you I'm guessing this is mostly new and unfamiliar, but after some exposure I hope it will make sense!