Slide 1

Administrivia

- Appendix A has many useful details about programming in MIPS assembly language and using SPIM. Time permitting you should at least skim sections sections A.1 through A.6, A.9, the introduction to A.10, and A.11. Notice that A.10 contains reference material on all instructions.
- Notes from a week ago edited to include a little more about floating point.

Slide 2

A Little About Circuit Design

- Goal — sketch design of a (hardware) implementation of MIPS architecture in terms of some simple building blocks (AND and OR gates, inverters).
- Things we'll need:
  - Something to implement instructions: ALU (arithmetic/logic unit).
  - Something to implement registers: register file.
  - Something to implement fetch/decode/execute cycle: control logic.

Slide 3

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. In widely-used "CMOS technology", there are two types of switches, one that's good if the input is "one" and one that's good if the input is "zero". These can be combined to implement logic. Simple example: Inverter. (See link from "useful links" page.)

Slide 4

Circuit Design — Overview Continued

- 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 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 5

Circuit Design — Overview Continued

- "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 output.)
- "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 6

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).
- 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 7

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 8

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 9

"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 10

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 11

Minute Essay

- Bitwise AND is an example of an operation that can be implemented using an array of 32 independent identical logic blocks. What's another operation that can be implemented that way?
- Anything that seems especially puzzling about this material?

Slide 12

Minute Essay Answer

- Any of the other bitwise logical operations (e.g., OR) can be implemented with an array of independent elements. Addition and subtraction, though, require the array elements to be connected.