**Slide 1**

## Administrivia

- (Via e-mail.)

**Slide 2**

## Designing a Processor — Overview

- 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 small subset of instructions, but enough to give you the idea?)
- To do this, 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". (Figures B.2.1, B.2.2.)
- 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").
- I wrote up a programmer's take on this, with an example; available under "useful links" on course Web site. Not required reading, but might be interesting?

**Slide 5**

## 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 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).

**Slide 7**

## Combinational Logic, Continued

- Example: Circuit with three inputs ($A$, $B$, $C$), three outputs:
  $D$ true if at least one input true.
  $E$ true f exactly two inputs true.
  $F$ true if all three inputs true.
- Can write truth table or Boolean expression. Textbook presents both, and then a circuit (Figure 3.4).
- This is what I call the "simple dumb way" of producing a circuit — might not produce the smallest circuit, but obviously right.

**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.)
- Example: Circuit with three inputs ($A$, $B$, $C$), three outputs:
  $D$ true if at least one input true.
  $E$ true f exactly two inputs true.
  $F$ true if all three inputs true.
- Can write truth table or Boolean expression. Textbook presents both, and then a circuit (Figure 3.4).
- This is what I call the "simple dumb way" of producing a circuit — might not produce the smallest circuit, but obviously right.

**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: Figure B.3.6 (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). (Figure shortly.)

Slide 13

## Design of an ALU

- One thing we need for a MIPS implementation — something that can do the arithmetic and logic operations in the MIPS instruction set. (Again, look only at a subset.)
- Inputs to operations typically two 32-bit values. Some operations operate on all bits in exactly the same way, independently (e.g., `and`). Others operate on all bits the same way but with dependencies among bits (e.g., `add`). So 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 building up something to performs `and`, `or`, and `add` on 1-bit values (plus carry-in and carry-out values for `add`).
- Result (B.5.6) — 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 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 keep saying that about two's complement notation that it's 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). Clever?

**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 $a - b$. Negative if $a < b$, so use high-order bit of result of $a - b$ to set low-order bit of result.
- Figure B.5.12 shows result plus zero detection.

**Slide 18**

## 32-Bit ALU from 1-Bit ALUs, Continued

- Result is something we can use to do a useful subset of the arithmetic and logic operations of the MIPS ISA.

  Figure B.5.13 shows how "control lines" (Ainvert, Bnegate, Operation) map onto operations of interest.

  Figure B.5.14 shows conventional symbol for whole thing.
- What we can't do with this: 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 19**

## 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.)

  (Briefly review explanation. More on request!)
- 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 20**

## A Very Little Bit About Clocking

- Many (most, currently?) hardware designs are based on 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 21**

## A Very Little Bit About Clocking, Continued

- If input and output of combinational logic are different, all is well. But if they're not (Figure B.7.2)? How can values ever "settle down"?
- So introduce between state element 1 (input) and CL block, some kind of barrier/switch that can either let bits flow or not, and the same thing between CL block and state element 2, with only one of those barriers letting bits flow at a time.

**Slide 22**

## 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 23**

## 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 24**

## 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 25

## 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 26

## 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.
- 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.)

  Details skipped for reasons of time, but read if interested!

**Slide 27**

## 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 need something to control it — which may also involve sequential logic blocks. (In years past this meant another detour, through Appendix B 10 . . . But now we don't, so for reasons of time skip, alas. Interesting stuff!)

**Slide 28**

## 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?
- Have you seen any of this material in another course? (I think ENGR teaches it in at least one course.)

**Slide 29**

## 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.

  (However, as students have sometimes 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.)