Administrivia

- Reminder: Homework 4 due Friday.
- Reminder: Quiz 4 Friday. Topics from what we covered in chapter 3 (data representation, integer arithmetic).

Slide 1

Minute Essay From Last Lecture

- Other operations that can be done on 32-bit values using 32 independent 1-bit logic blocks? Bitwise OR, NOT.
- Addition is, as noted, a little trickier. Multiplication is trickier yet — to use the algorithm described in chapter 3 will require not only logic blocks that can do addition and shifting but also some control logic.

Slide 2

Slide 3

Hardware Description Languages — Executive-Level Summary

- "Hardware description languages" can be used to represent the circuit designs discussed in Appendix C. 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. Simple but illustrative example in Figure 3.4.1. ("Half adder"? means one without a carry-in input.)
- 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. Figures C.6.1 and C.6.2 illustrate the general idea.)

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.

1-Bit ALU

- Figures C.5.1 through C.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 (C.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 5

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 C.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 6

Slide 7

32-Bit ALU from 1-Bit ALUs, Continued

- We said when we first talked about twos' 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 C.5.8! which is Figure C.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 C.5.9).

Slide 8

32-Bit ALU from 1-Bit ALUs, Continued

- Figures C.5.10 and C.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).

Minute Essay

Slide 9

- We just 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?

Minute Essay Answer

Slide 10

- 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.