Slide 1

Administrivia

- Quiz 3 Tuesday. More questions from chapter 2, possibly including material about binary representation of instructions. (Why another quiz so soon? Might help you prepare for midterm.)
- Review sheet for midterm on the Web soon.
- Reminder: Homework 2 officially due today; accepted Tuesday without penalty.

Slide 2

Minute Essay From Last Lecture

- The review(?) of arithmetic in/and different bases was review for most people. If not for you, and you're still confused after a try at reading up, ask.

Slide 3

Implementing Arithmetic — Preview

- In the next chapter we start talking about hardware design (though still at a somewhat abstract level).
- For now it may be useful to know that the low-level building blocks are entities that can evaluate Boolean expressions — very simple ones at the lowest level, and slightly more complex ones one level up.
- So for example we can implement addition by first making a "one-bit adder" that maps three inputs (two operands and carry-in) to two outputs (result and carry-out), and then chaining together 32 of them.
- (Multiplication and division may need to be more complex, involving multiple steps and control-flow logic.)

Slide 4

More Arithmetic — Multiplication

- As with addition, first think through how we do this "by hand" in base 10. (Review terminology: In $a \times b$, call a the "multiplicand" and b the "multiplier".) Example?
- We can do the same thing in base 2, but it's simpler, no? computing the partial results is easier. This gives the textbook's first algorithm, figure 3.5. (Work through example if time permits.)

  Notice also that overflow could be a lot worse here — so normally we'll compute a result twice as big as the inputs.

  (We can do better — later.)
- What about signs? Algorithm works, if we extend the sign bit when we shift right.

Slide 5

Multiplication, Continued

- In MIPS architecture, 64-bit product / work area is kept two special-purpose registers (`lo` and `hi`). Two instructions needed to do a multiplication and get the result:

  ```
  mult rs1, rs2
  mflo rdest
  ```

  Assembler provides a "pseudoinstruction":

  ```
  mul rdest, rs1, rs2
  ```

- Notice, however, that a "smart" compiler might turn some multiplications into shifts. (Which ones?)

Slide 6

Division

- As with other arithmetic, first think through how we do this "by hand" in base 10. (Review terminology: We divide "dividend" a by "divisor" b to produce quotient q and remainder r, where $a = bq + r$ and $0 \leq |r| < b$.) Example?

  We can do the same thing in base 2; this gives the algorithm in figure 3.10. (Work through example if time permits.)

  (Here too we can do better — later).

- What about signs? Simplest solution is (they say!) to perform division on non-negative numbers and then fix up signs of the result if need be.

Slide 7

Division, Continued

- In MIPS architecture, 64-bit work area for quotient and remainder is kept in same two special-purpose registers used for multiplication (`lo` and `hi`). After division, quotient is in `lo` and remainder is in `hi`. Two (or more) instructions needed to do a division and get the result:

```
div rs1, rs2
mflo rq
mfhi rr
```

  Assembler provides a "pseudoinstruction":

```
div rdest, rs1, rs2
```

- Notice, however, that a "smart" compiler might turn some divisions into shifts. (Which ones?)

Slide 8

Integer Multiplication and Division, Recap

- Algorithms for both operations are based on how you do things "by hand", with some modifications to permit simpler hardware. It's not critical to understand the details, but probably useful to work through an example to believe that it works.
- Required hardware is something that can add two 32-bit numbers, a 64-bit "work area", something to do right and left shifts of the 64-bit area, and some control logic.
- MIPS architecture uses "special registers" `lo` and `hi` for the 64-bit work area. This is where the results end up. There are instructions to multiply, to divide, and to move from the special registers. ("Move from" explains the names of the instructions.)

Slide 9

Representing Real (Non-Integer) Numbers

- Approach is based on a binary version of "scientific notation":

  In base 10, we can write numbers in the form $+/- x.yyyy \times 10^z$.

  E.g., $428 = 4.28 \times 10^2$, or $-.0012 = -1.2 \times 10^{-3}$.

- We can do the same thing in base 2. Examples:

  $32 = 1.0_2 \times 2^5$

  $-3 = -1.1_2 \times 2^1$

  $1/2 = 1.0_2 \times 2^{-1}$

  $3/8 = 1.1_2 \times 2^{-2}$

- This is "floating point" (as opposed to "fixed point", which would allow for non-integers but wouldn't allow as much flexibility — wide range, all with reasonable precision).

Slide 10

Representing Real Numbers, Continued

- In base 10, we can completely specify a number by giving its sign, a number in the range $0 \leq x < 10$ (the "significand" or "mantissa"), and the exponent for 10. Same idea applies in base 2.
- So, most/all "floating-point formats" have a bit for the sign, some bits for the significand, and some bits for the exponent. Different choices are possible, even with the same total number of bits; (at least) one architecture (VAX) even supported more than one format with the same number of bits(!).
- With integers, number of bits limits the range of numbers that can be represented. With "floating-point" numbers, two limiting factors — number of bits for the significand (which limits what?), and number of bits for the exponent (which limits what?).

  (Does this suggest why the VAX designers offered two formats?)

  (To be continued . . .)

Slide 11

Minute Essay

- None — quiz.