

Administrivia

- Homework 4 on Web. Due next Wednesday. Not accepted past class time Friday (so I can distribute solutions, in preparation for midterm March 8).

Slide 1

Minute Essay From Last Lecture

- Questions:
What instruction would you use to multiply the number in register $\$t0$ by 16, treating it as an unsigned integer. Would you use the same instruction if you wanted to treat it as a signed integer?
What instruction would you use to divide the number in register $\$t0$ by 16, treating it as an unsigned integer. Would you use the same instruction if you wanted to treat it as a signed integer?
- Answers?

Slide 2

Multiplication and Division, Recap

Slide 3

- 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. (See last problem in Homework 4.)
- Required hardware is a 32-bit ALU (which we already talked about how to design), 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.

Representing Real (Non-Integer) Numbers

Slide 4

- 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).

Representing Real Numbers, Continued

Slide 5

- 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, there are 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?)

Representing Real Numbers, Continued

Slide 6

- “IEEE 754 standard” defines formats and operations.
- Picky details:
 - Exponent is actually stored “biased” — actual exponent plus bias (so we only have to store non-negative exponents).
 - Significand doesn’t include leading 1. (Why not?)
 - But then how to represent 0? Agree that exponent of all 0s will represent 0 if significand is 0, else “de-normalized number”.
 - Also, agree that exponent of all 1s will represent +/- “infinity” if significand is 0, else NaN (“not a number” — result of indeterminate or invalid operations such as 0/0).
 - “Single-precision” format has 8 bits for exponent, biased by 127, 23 bits for significand. (Double precision is 8, 1023, 52 respectively.)
- Work through an example . . .

Floating-Point Addition

Slide 7

- How to add two floating-point numbers? Approach is similar to how you'd do this with decimal numbers in scientific notation:
 1. Shift number with smaller exponent to right until exponents match.
 2. Add fractions.
 3. Normalize (get significand back in proper range).
 4. Check for overflow (exponent too big) or underflow (exponent too small).
 5. Round, and renormalize if necessary.
- Work through example.

Minute Essay

Slide 8

- None — quiz.