Slide 1

Administrivia

- Next homework (short) will be on the Web soon, probably tomorrow. Due next Wednesday. Also notice addition to reading for today.
- About grades, I'm still grading the midterms and homeworks. If I make significant progress today I will e-mail you scores.

Slide 2

Minute Essay From Last Lecture

- Several people mentioned seeing AND/OR gates in CSCI 1323. Maybe with some instructors? Ideas are the same as and/or operators but usage is somewhat different I think.
- Two people mentioned their use in Minecraft(!).

Slide 3

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 4

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 $+/- x.yyy \times 10^z$.
  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 5

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, 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?)

Slide 6

Representing Real Numbers, Continued

- "IEEE 754 standard" defines formats and operations.
- Exponent is actually stored "biased" — actual exponent plus bias (so we only have to store non-negative exponents — simplifies comparisons).
- 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 . . .

Slide 7

Floating-Point Addition

- 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.
- Examples in textbook (worth looking through but not necessary to master details).

Slide 8

Floating-Point Multiplication

- How to multiply two floating-point numbers? Approach is also similar to how you'd do this with decimal numbers in scientific notation:
  1. Add exponents and subtract bias.
  2. Multiply 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.
  6. Set sign bit.

Slide 9

Floating-Point Arithmetic Can Be Strange, Part 1

- Consider the following loop:

```
for (f = 0.0; f != 1.0; f += 0.1)
    printf("f = %g\n", f);
```

What do you think it does?

Why?

Slide 10

Floating-Point Arithmetic Can Be Strange, Part 2

- Consider the following code:

```
double fsmall = 1e-10;
double fbig = 1e10;
double temp1 = fbig;
for (int i = 0; i < 10000; ++i)
    temp1 += fsmall;
double temp2 = 0.0;
for (int i = 0; i < 10000; ++i)
    temp2 += fsmall;
temp2 += fbig;
```

After it runs, is `temp1` equal to `temp2`?

(This has implications for parallel computing, as described in section 3.6.)

Slide 11

Minute Essay

- I mentioned that at least one architecture offered different floating-point formats with the same number of bits (the difference being how many of the bits were used for the different parts of the representation). What advantage(s) would this have? What disadvantage(s)?

Slide 12

Minute Essay Answer

- An advantage is that users would then have a choice between a larger range of possible values and greater precision. A disadvantage is that it's more complicated — both to implement and to use.