

## CS61B Lecture #12

### Today:

- Floating point arithmetic (just a taste).

**Readings for Today:** *Programming Into Java*, §5.3, §5.4.

**Readings for next Topic:** *Programming Into Java*, §4.1

Last modified: Fri Sep 28 11:11:08 2001

CS61B: Lecture #12 1

## Floating point

- **Problem:** Need to represent non-integral quantities.
  - Regardless of magnitudes, however, precise representation can require arbitrarily large amounts of space.
  - Unclear what to do about irrational numbers.
  - Must have arithmetic that is fast.
- **Solution:** Don't solve the problem of representing all real numbers.
  - Instead, use fact that most scientific computation only needs results that are "good enough,"
  - That is, that carry at least as many significant digits as the data justify.
  - Settle for a subset of the rationals representable in a fixed number of bits (typical: 32, 64, 80, sometimes 128).
- Many variations on this theme used in the past.
- Majority of machines now support *IEEE binary floating-point arithmetic*,...
- ...which (largely) standardizes on numbers represented and approximations used.

Last modified: Fri Sep 28 11:11:08 2001

CS61B: Lecture #12 2

## Basic Idea: Representation

- Java's two floating-point types each represent just the numbers
$$\pm m \times 2^e$$

$0 \leq m < 2^n$ , for some fixed  $n$  and  $e$  in some fixed range of integers,
- and also  $\pm\infty$  and NaN (Not a Number).
- $n$  is the number of significant bits ( $\approx 3.3 \times$  number of significant base-10 digits).
- For `double`,  $n = 53$  ( $> 15$  significant digits), and  $-1074 \leq e < 972$  (roughly, positive values in the range  $5.0 \times 10^{-324}$  to  $1.8 \times 10^{308}$ ).
- For `float`,  $n = 24$  ( $> 6$  significant digits), and  $-150 \leq e < 105$ , (roughly, positive values in the range  $1.4 \times 10^{-45}$  to  $3.4 \times 10^{38}$ ).
- $\pm\infty$  used for values whose magnitude is larger than this range, including `1.0/0`.
- NaN used for undefined results (e.g.,  $\infty - \infty$ ,  $0/0$ , or any operation involving NaN).
- `-0` compares `==` to `0`, but `1/(-0)` gives  $-\infty$ . Why? Sorry; you'll have to study complex functions.

Last modified: Fri Sep 28 11:11:08 2001

CS61B: Lecture #12 3

## Basic Idea: Arithmetic

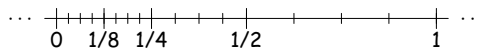
- Basic rule is simple: to compute  $x \oplus y$ , (where  $\oplus$  is  $+$ ,  $-$ ,  $*$ ,  $/$ ) compute the true mathematical result, and then *round* to nearest representable floating-point number.
- Same for literals: is no exact binary equivalent of 0.1, so round to nearest.
- *Unbiased rounding:* Sometimes, result can be exactly halfway between two representable values: choose one with last bit 0.
- (Unfortunately, whole truth is a little more complex: Java implementations allowed to use *more* precision to compute intermediate results.)

Last modified: Fri Sep 28 11:11:08 2001

CS61B: Lecture #12 4

## The Floating Number Line

- Floating-point arithmetic gives same number of significant bits at all values, except near 0.
- Implies that they "stretch out" the further from 0 you go.
- So, if we had only 3 significant bits, and the smallest exponent of 2 were  $2^{-6}$ , we'd see this:



**Note:** corrected from book.

- In particular, we'd get all the 3-bit ( $n$ -bit) integers:
- 
- After which, we start skipping.
  - **Random terminology:** The first  $2^3 - 1$  numbers after 0 in the first diagram ( $2^n - 1$  in general) are called *denormalized*.
  - Unique in having same spacing as the next  $2^3$ . In some other versions of floating point, they are missing.

Last modified: Fri Sep 28 11:11:08 2001

CS61B: Lecture #12 5

## Input of Floating-Point

- To convert from string, `S`, to the floating-point number it denotes, use `Double.parseDouble(S)` or `Float.parseFloat(S)`
- To read, can use `StreamTokenizer` (see project), or (around here) the `UCB I/O` package, which is like `C`'s `scanf`:

```
stdin.scan ("%le"); // Means "read a double"
x = stdin.doubleItem (0);
```

Last modified: Fri Sep 28 11:11:08 2001

CS61B: Lecture #12 6

## Output of Floating Point

- When printing (or converting to `String`, as in `"X=" + X`), Java by default prints "enough" significant digits and decides on whether to use scientific notation based on size.
- You can control this all explicitly:
- In the standard Java library, can use `DecimalFormat` type.
- With the UCB I/O library can use `format` (which imitates the C `printf` functions),
- E.g., If `x` is `3.1415926`, then

```
DecimalFormat form = new DecimalFormat (pattern1);
...
System.out.print (form.format(x));
stdout.format (pattern2).put (x);
```

produces

| Pattern <sub>1</sub> | Pattern <sub>2</sub> | Print  |
|----------------------|----------------------|--------|
| "0.000"              | "%5.3f"              | 3.142  |
| "00.000"             | "%06.3f"             | 03.142 |
| "#0.000"             | "%6.3f"              | 3.142  |
| "#.000"              |                      | 3.142  |
| "0.0E00"             | "%6.1e"              | 3.1E00 |
| ".00E00"             |                      | .31E01 |

Last modified: Fri Sep 28 11:11:08 2001

CS61B: Lecture #12 7

## A Few Points to Remember

- IEEE gives reasonably well-behaved approximations, but they *are* approximations.
- In particular, many decimal fractions not exactly representable in binary (0.1, e.g.)
- Likewise, repeated addition not the same as multiplication.  $N$  adds accumulate  $N$  rounding errors; one multiply accumulates only 1 rounding error. Example:

```
for (double x = 0.0; x < 1.0; x += 0.1) { ... }
```

Loop runs 11 times, not 10.

- When you add large quantities to small, you lose significant digits from the right of the small quantity. Thus, computations like  $X+Y-Z$ , where  $X$  and  $Z$  are nearly equal and much larger than  $Y$  won't be terribly accurate.

Last modified: Fri Sep 28 11:11:08 2001

CS61B: Lecture #12 8