Floating Point Numbers

CS031

September 12, 2011

Motivation

We've seen how unsigned and signed integers are represented by a computer.

We'd like to represent decimal numbers like 3.75_{10} as well.

By learning how these numbers are represented in hardware, we can understand and avoid pitfalls of using them in our code.

Fixed-Point Binary Representation

Fractional numbers are represented in binary much like integers are, but negative exponents and a decimal point are used.

$3.75_{10} = 2+1+0.5+0.25$

$= 2^1+2^0+2^{-1}+2^{-2}$

$= 11.11_2$

Not all numbers have a finite representation:

$0.1_{10} = 0.0625+0.03125+0.0078125+\ldots$

$= 2^{-4}+2^{-5}+2^{-8}+2^{-9}+\ldots$

$= 0.0001100110011\ldots_2$

Computer Representation Goals

Fixed-point representation assumes no space limitations, so it's infeasible here.

Assume we have 32 bits per number.

We want to represent as many decimal numbers as we can, don't want to sacrifice range.

Adding two numbers should be as similar to adding signed integers as possible.

Comparing two numbers should be straightforward and intuitive in this representation.

The Challenges

How many distinct numbers can we represent with 32 bits?

Answer: 2^{32}

We must decide which numbers to represent.

Suppose we want to represent both $3.75_{10} = 11.11_2$ and $7.5_{10} = 111.1_2$. How will the computer distinguish between them in our representation?

Excursus – Scientific Notation

$913.8 = 91.38 \times 10^1 = 9.138 \times 10^2 = 0.9138 \times 10^3$

We call the final 3 representations scientific notation.

It's standard to use the format 9.138×10^2 (exactly one non-zero digit before the decimal point) for a unique representation.

The decimal point is always in the same place in scientific notation, and we can distinguish between 9138 and 913.8.

Floating Point Representation

Floating point is much like scientific notation.

Consider 1.23×10^1

Terminology:

- 1.23 is the mantissa
- 10 is the radix, or base
- 1 is the exponent

We can use a radix of 2 with a binary mantissa for binary notation:

$1.111_2 \times 2^2$ represents $111.1_2 = 7.5_{10}$

Floating Point Normalization

Like in scientific notation, we want exactly one non-zero digit before the decimal point.

We use normalization to ensure that our numbers meet this requirement.

$11011.1_2 \text{ x } 2^{-2}$ normalizes to $1.10111_2 \text{ x } 2^2$.

$0.011_2 \text{ x } 2^3$ normalizes to $1.1_2 \text{ x } 2^1$.

Normalization ensures a unique representation for our numbers.

Floating Point Representation

We can represent many numbers this way:

Range is limited by the exponent

Precision is limited by the mantissa

Addition (in binary):

$1.011 \times 2^{2} + 1.101 \times 2^{-1}$

$= 1011 \times 2^{-1} + 1.101 \times 2^{-1}$ Convert to a common exponent

$= 1100.101 \times 2^{-1}$ Add the mantissas

$= 1.100101 \times 2^{2}$ Normalize the result, if necessary

Floating Point Representation

Comparison:

1) Compare the signs of the mantissa

$$-1.1 \times 2^5 < 1.01 \times 2^2$$

2) If same sign, compare the exponents

$$1.001 \times 2^5 > 1.01 \times 2^4$$

$$-1.11 \times 2^3 > -1 \times 2^4$$

3) If same exponent, compare mantissa

$$1.111 \times 2^5 > 1.11 \times 2^5$$

$$-1.01 \times 2^{-1} > -1.101 \times 2^{-1}$$

$$1.001 \times 2^3 = 1.001 \times 2^3$$

IEEE Representation

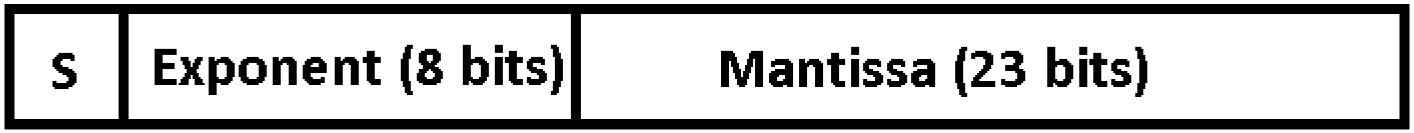
To represent floating point numbers in a computer, distribute the 32 bits between the sign, mantissa, and exponent.

S	**Exponent (8 bits)**	**Mantissa (23 bits, starts with 1)**

For all numbers but 0, the mantissa will start with a 1. Why?

Answer: The mantissa is in binary and must start with a non-zero digit. To gain an extra bit of precision, we won't include that 1 (zero will get a special representation).

IEEE Representation



Sign bit: 0 for positive numbers, 1 for negative numbers

Mantissa: unsigned (like Lecture 1) for ease of adding and comparison

But what about the exponent?

IEEE Representation

The exponent can be negative, so unsigned representation won’t work.

We could use two’s complement.

But which exponent looks bigger:

11111110 or 00000111?

Comparison is not trivial in two’s complement.

Excursus – Excess Notation

Another way to represent signed integers, also called bias notation.

In excess 127 notation, an 8-bit string whose unsigned integer value is M represents M – 127.

Ex: 00000000 represents -127.

11111111 represents 128

What does 01111111 represent?

127 – 127 = 0.

IEEE Representation

Larger numbers look larger in excess 127 notation, so we'll use it for the exponent.

We can now represent numbers.

Ex: -3.75_{10} $= -1 * (2^1+2^0+2^{-1}+2^{-2})$

$= -11.11_2 = -1.111_2 \times 2^1$

Sign bit: 1 (negative)

Exponent: 10000000 (128-127=1)

Mantissa: 11100000000000000000000

Answer: 1100 0000 0111 0000 0000 0000 0000 0000

IEEE Representation

What about converting to decimal?

0011 1110 1010 1000 0000 0000 0000 0000

Sign bit:	0 (positive)
Exponent:	01111101 (125-127=-2)
Mantissa:	01010000000000000000000

Answer: $1.0101_2 \times 2^{-2}$

$= 0.010101_2$

$= 0.25+0.0625+0.015625$

$= 0.328125$

Properties of IEEE

Infinitely many reals, finitely many IEEE floating point numbers.

Cannot represent some numbers, like 1/3 or 1/10.

More numbers are closer to zero than to 2^{127}. (Why?)

Special representation needed for zero.

Properties of IEEE

Representation of zero:

0000 0000 0000 0000 0000 0000 0000 0000

Positive/Negative Infinity:

?111 1111 1000 0000 0000 0000 0000 0000

NaN: (with non-zero mantissa)

0111 1111 1??? ???? ???? ???? ???? ????

Denormalized number: a number of the form

?000 0000 0??? ???? ???? ???? ???? ????

with a non-zero mantissa does not assume a hidden 1.

Properties of IEEE

Biggest positive number (roughly 2^{128}):

0111 1111 0111 1111 1111 1111 1111 1111

Smallest positive number (2^{-126}):

0000 0000 1000 0000 0000 0000 0000 0000

Negative number furthest from zero (roughly -2^{128}):

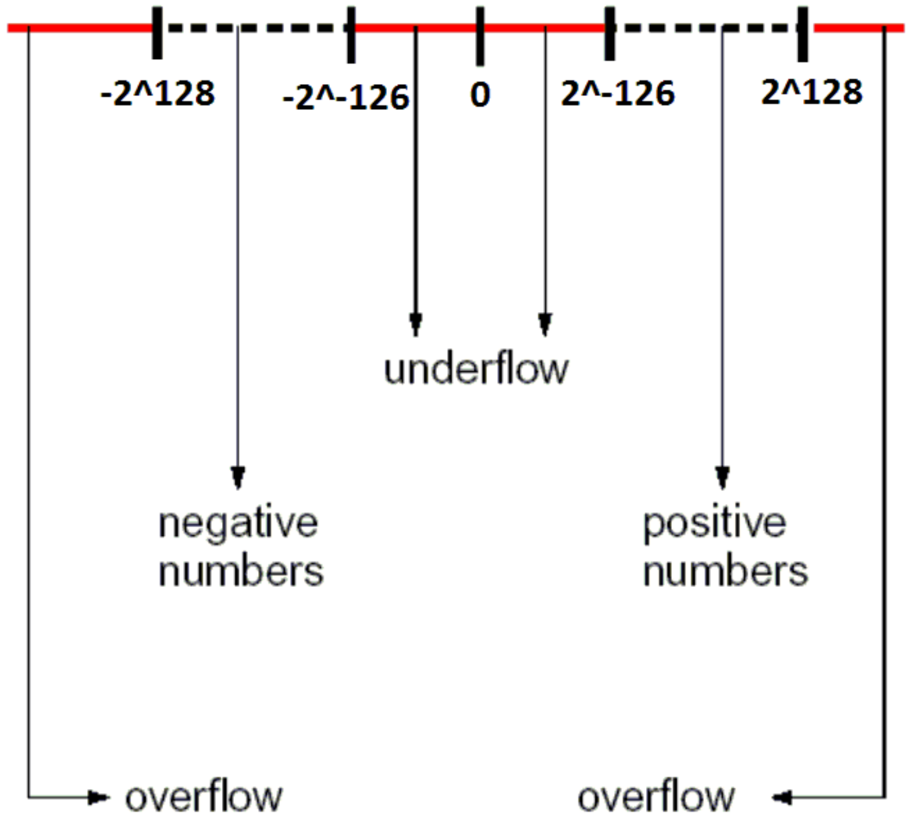
1111 1111 0111 1111 1111 1111 1111 1111

Negative number closest to zero (-2^{-126}):

1000 0000 0100 0000 0000 0000 0000 0000

Problems with IEEE

Not all ranges can be represented.



(Figure not to scale)

Problems with IEEE

Addition is not necessarily associative:

Let $x = -1.1 \times 2^{120}$, $y = 1.1 \times 2^{120}$, $z = 1.0$

$$
\begin{aligned}
(x+y)+z &= (-1.1 \times 2^{120} + 1.1 \times 2^{120}) + 1.0 \\
&= 0 + 1.0 = 1.0 \\
x+(y+z) &= -1.1 \times 2^{120} + (1.1 \times 2^{120} + 1.0) \\
&= -1.1 \times 2^{120} + 1.1 \times 2^{120} = 0.0
\end{aligned}
$$

Rounding errors:

$0.1+0.1+0.1 \mathrel{!=} 0.3$

$2^{100} + 2^{-100} = 2^{100}$

Where Do We Go Next?

- We have covered IEEE-754 Single Precision Floating Point representation, which is what C++ uses for the float type and close to what Java uses. Other IEEE standards exist: Java’s double type uses 64-bit, excess 1023, 11 bits for exponent.
- Careful analysis can identify places where errors may occur.
- We can avoid writing code that could result in floating point errors.