IEEE Floating Point Numbers
Overview

Noah Mendelsohn
Tufts University
Email: noah@cs.tufts.edu
Web: http://www.cs.tufts.edu/~noah
Note: There are excellent detailed descriptions in Bryant and O’Hallaron. These slides are designed to give you an introduction, and some intuition about why IEEE floating point works the way it does.
What’s Wrong with Integers?
Big numbers

Estimated US National Debt as of 7 Oct 2013: $16,753,342,516,305.38*

• What if we had only limited space in which to write?
• Besides, do we really think the 38 cents are meaningful?

We’re more likely to say informally: $16.8 trillion

A scientist would of course write this as: $1.68 \times 10^{13}\

* Source: [http://www.brillig.com/debt_clock/](http://www.brillig.com/debt_clock/)
Big numbers

Estimated US National Debt as of 7 Oct 2013: $16,753,342,516,305.38

* Source: http://www.brillig.com/debt_clock/

• What if we had only limited space in which to write?
• Besides, do we really think the 38 cents are meaningful?

We’re more likely to say informally: $16.8 trillion

A scientist would of course write this as: $1.68 \times 10^{13}$

Key intuition: we can approximate a very large range of numbers using very few digits if we’re willing to lose the less significant digits.
Big numbers

Estimated US National Debt as of 7 Oct 2013: $16,753,342,516,305.38*

Warning: this means we may treat $16,753,342,516,305.38 and $16,753,976,632,595.72 as the same number!

A scientist would of course write this as: $1.68 \times 10^{13}$

* Source: http://www.brillig.com/debt_clock/
Starting with Decimal

IEEE floating point uses *binary* for both fractions and exponents. Students typically have more intuition for decimal, so we’ll introduce the *concepts* with decimal first.
Scientific Notation

Let’s approximate: $16,753,342,516,305.38$ using Scientific Notation

$$1.6753 \times 10^{13}$$
Scientific Notation

Let’s approximate: $16,753,342,516,305.38$ using Scientific Notation

$1.6753 \times 10^{13}$

- Negative numbers: significand is negative
- Very small numbers: exponent is negative

* You may see references to the word “mantissa” – it’s another term for “significand”
Normalization

All of these represent the same number:

\[ .16753 \times 10^{14} \]
\[ .016753 \times 10^{15} \]
\[ 1.6753 \times 10^{13} \]
\[ 16.753 \times 10^{12} \]
\[ 16753. \times 10^9 \]

Whenever possible IEEE floating point numbers are normalized.
Normalization:

Any time we have many equivalent elements and want to compare them

- Equivalence class of objects/values all of which map to same thing in the world of ideas
- Pick a *canonical, normal, representative* element of set based on some criterion
- Whenever you generate or manipulate something, convert it to normal/canonical form
- Comparison is now easy
Floating Point
Computer Data Types
Why floating point types?

- Need a way to work with very wide range of values
- Willing to make same tradeoff as scientific notation
  - We only carry the high order digits – nearby values not distinguished
- What we need
  - Standardized encoding in bits
  - Arithmetic and other operations
- IEEE 754 floating point is a cross-industry standard*
  - 32 and 64 bit encodings: latter carries more digits & wider range of exponents
  - Standardized arithmetic including edge cases – usually in hardware
  - Includes special NaN values for error cases (+/- INFinity) etc.
  - Subnormal numbers implement gradual underflow (advanced topic)
  - Most software systems provide math functions (sqrt, sin, cos, atan, etc., etc.)
- The IA64 machines we use all implement IEEE floating point which are used for the float (32 bit) and double (64 bit) C types
Reminder: how it worked in decimal

**Decimal**

\[
\frac{1}{10} \quad \frac{1}{100} \quad \frac{1}{1000} \quad \frac{1}{10000} \\
\]

\[
.2375 \times 10^1 \\
(\frac{2}{10} + \frac{3}{100} + \frac{7}{1000} + \frac{5}{10000}) \times 10 \\
2 \ 3/8
\]
...but we need to do it in binary!

**Binary**

(1 0/2 0/4 1/8 1/16) * 2

1.0011 x 2^1

2 3/8
Decimal and binary compared

**Decimal**

\[
.2375 \times 10^1 = \frac{1}{10} + \frac{3}{100} + \frac{7}{1000} + \frac{5}{10000} \times 10 = 2 \frac{3}{8}
\]

**Binary**

\[
1.0011 \times 2^1 = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} \times 2 = 2 \frac{3}{8}
\]
IEEE single precision floating point encodes this in 32 bits

$$+1.0011 \times 2^1$$

Huh? That doesn’t look right!
The encoding is a bit tricky

- **Sign**: 1 bit (0 = positive, 1 = negative)
- **Exponent**: 8 bits (true exponent + bias of 127)
- **Significand**: 23 bits (as you would expect but without the leading 1 bit)

Let’s use our example to make this clearer
IEEE floating point encodes this in 32 bits

\[+1.0011 \times 2^1\]

Sign bit: 0 = positive
IEEE floating point encodes this in 32 bits

\[ +1.0011 \times 2^{1} \]

Exponent is 1...
...but we need to add bias

\[ 1 + 127 = 128 \]

Binary: 10000000
IEEE floating point encodes this in 32 bits

Remember, there is always a 1 here for normalized numbers...we don’t waste a bit storing it.
IEEE floating point encodes this in 32 bits

\[ +1.0011 \times 2^1 \]

The fraction is stored "left aligned"
IEEE floating point encodes this in 32 bits

\(+1.0011 \times 2^1\)

Implied “binary point” is left of the fraction

32 bits
A few more details

- **Why all this trickiness?**
  - Saving bits (e.g. not saving the invariant 1)
  - *Integer* compares put positive floating point numbers in sorted order!

- **There’s also a 64 bit format (the C “double” type)**
  - 1 bit sign
  - 11 bit exponent (bias is 1023)
  - 52 bit fraction

- **There are some details we haven’t covered**
  - Exponent of 0 means denormalized number: provides useful values close to zero. There is no implied “1” and the exponent is fixed at -126 for 32 bit and -1022 for 64 bit. These values have less precision.
  - All zeros is zero. If only sign bit on, then negative zero! (yes there is -0!)
  - Exponent of all 1’s denotes special values: +/- Infinity, Exception values, etc. *You should read about this in B&O.*

You may be tested on both 32 & 64 bit formats