Floating Point Numbers

CSE 220: Systems Programming

Ethan Blanton & Carl Alphonce

Department of Computer Science and Engineering

University at Buffalo

Floating Point

Floating point is the counterpoint to integer representation.

It is used to:

- represent rational numbers¶
- approximate real numbers

Binary floating point formats have some surprising properties.

Fixed Point

Floating point has a closely related representation, fixed point.

Fixed point is also used to represent rational and real numbers.

However, it is less flexible than floating point.

We will explore fixed point before floating point.

Planning: Documentation

How do you absorb the documentation?

Read and take notes:

- What is hard?
- What requires more information?
- What can you do right now?

Read:

- The handout
- The README/etc.
- The given code and its comments

Fixed Point

A fixed point number has a fixed number of digits.

A fixed-point number has a maximum magnitude and minimum fractional portion that do not change.

For example, a fixed point number with 3 digits before and after the decimal point might include:

- 003.142
- 099.440
- 107.429

The Binary Point

In binary numbers, we have a binary point.

Just as the decimal point separates 10^0 from 10^{-1}, the binary point separates 2^0 from 2^{-1}.

Do not confuse decimal digit and decimal point!

Likewise, binary digit and binary point.

The Binary Point

Suppose we have a b-bit binary number with bits both before and after the binary point, such that:

- There are w whole-number bits before the binary point
- There are f fractional bits after the binary point
- The largest bit before the point is b_{w-1}
- The smallest bit before the point is b_0
- The largest bit after the point is b_{-1}
- The smallest bit after the point is b_{-f}

$$b_{w-1}, \ldots, b_0, b_{-1}, \ldots, b_{-f}$$

A *w*.*f*-bit Binary Number

The *w* whole-number bits are defined as in integers:

$$b_i, i \geq 0 \doteq b_i \cdot 2^i$$

The *f* fractional-number bits are defined as follows:

$$b_j, j < 0 \doteq b_j \cdot 2^j$$

Thus, its total value is:

$$\sum_{i=0}^{w-1} b_i \cdot 2^i + \sum_{j=-1}^{-f} b_j \cdot 2^j$$

An Example Binary-Point Computation

Consider 11.101b:

$$
\begin{aligned}
11.101\text{b} &= 1 \cdot 2^{1} + 1 \cdot 2^{0} + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \\
&= 2 + 1 + {}^{1}\!/_{2} + 0 + {}^{1}\!/_{8} \\
&= 3\,{}^{5}\!/_{8} \\
&= 3.625
\end{aligned}
$$

What is “Floating Point”?

A floating point number, such as a `float` or `double`, is a number with a variable number of digits before or after the decimal point

(On computers, a variable number of bits before or after the binary point!)

Examples:
3.14159
6.022×10^{23}
6.626×10^{-34}

What is "Floating Point"?

A floating point number, such as a `float` or `double`, is a number with a variable number of digits before or after the decimal point

(On computers, a variable number of bits before or after the binary point!)

Examples:
3.14159
6.022×10^{23}
6.626×10^{-34}

It would take nearly 200 bits to represent all three of these numbers precisely.

What is “Floating Point”?

In order to represent numbers of very small or very large magnitude, floating point allows the point to move.

The number of digits (bits) of precision is fixed.

Some (loose) terms:

- **Significand**: The meaningful digits of a number
- **Exponent**: The “distance” of those digits from zero in powers of the arithmetic base

Floating Point Representation

In base 10, a floating point number is of the form $x \times 10^y$.

If we consider Avogadro's Number (6.022×10^{23}):

- The significand x is 6.022
- The exponent y is 23.

This requires six digits to store, versus 24 digits for 602200000000000000000000.

In base 2, a floating point number is $x \times 2^y$.

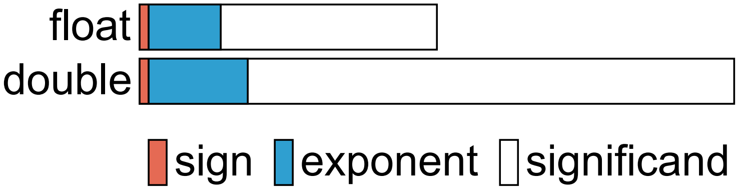
IEEE 754 Floating Point

IEEE Standard 754 defines a particular floating point format.

If a floating point number is $x \times 2^y$, in IEEE 754:

- A single precision number (`float`) has a 23-bit x and 8-bit y
- A double precision number (`double`) is 52-bit x and 11-bit y

Each has a one-bit sign.



Storing IEEE 754 Components

However, x and y are not stored directly!

Instead, we store x' and y', where:

x (the significand) is stored as x':

- Normalized to a value right of the binary point
- With an assumed leading 1 preceding the binary point

This means that a stored significand of $x' = 0$ is $x = 1.0$

y (the exponent) is stored as $y' = y + 127$.

This means that an exponent of $y = 0$ is stored as $y' = 127$.

Using dump_mem()

We have previously used dump_mem() to analyze integers.

We will now use it to look at floating point.

Dumping a float looks like this:

```
float f = 1.0;
dump_mem(&f, sizeof(float));
```

Note that &f is of type float *, but can be passed to void *.

Examining Floats

```
float f1 = 2.0f;
float f2 = 0.2f;

dump_mem(&f1, sizeof(f1));
dump_mem(&f2, sizeof(f2));
```

Examining Floats

```
float f1 = 2.0f;
float f2 = 0.2f;

dump_mem(&f1, sizeof(f1));
dump_mem(&f2, sizeof(f2));
```

Output:

```
00 00 00 40
cd cc 4c 3e
```

Deconstructing 2.0

Why is `2.0f` `0x40000000`?

0 10000000 0000000 00000000 00000000

Remembering our significand and exponent storage rules:

$x' = 0$ so $x = 1.0$ (only zero bits after the point)
$y' = 128$ so $y = 1$ (that is, $y = y' - 127$)

Thus: $1.0 \times 2^1 = 2.0$

(We didn't use 1.0 because its representation is more surprising.)

Deconstructing 0.2

This became 0x3e4ccccd, or:
0 01111100 1001100 11001100 11001101

Is this surprising?

Deconstructing 0.2

This became 0x3e4ccccd, or:
0 01111100 1001100 11001100 11001101

Is this surprising?

What just happened?

Deconstructing 0.2

This became 0x3e4ccccd, or:
0 01111100 1001100 11001100 11001101

Is this surprising?

What just happened?

The significand isn’t decimal!

It’s after the binary point.

Fractions cleanly represented in decimal, like $^1/_5$, may not terminate in binary — sort of like $^1/_3$ in decimal.

More Floating Point

IEEE 754 is more complicated than we covered here.
(You’ll read more about it in the text.)

We have covered the big ideas, however.

Some important implications to consider:

- Very large (either positive or negative) floating point numbers become imprecise because of that $\times 2^y$ factor.
- Very small (close to zero) floating point numbers become imprecise for the same reason.
- Double precision numbers can still be quite large and precise!
- The possible floating point values are unevenly spaced.[1]

[1] See “Denormalized Values” in your text for a caveat.

Summary

- Numbers can have fractional portions
- Both fixed and floating point representations can be calculated in both binary and decimal
- IEEE 754 standardizes a floating point representation
- Floating point numbers have fixed precision, but variable magnitude

References I

Required Readings

[1] Randal E. Bryant and David R. O’Hallaron. *Computer Science: A Programmer’s Perspective*. Third Edition. Chapter 2: 2.4 Intro, 2.4.1-2.4.3, 2.4.6, 2.5. Pearson, 2016.

Optional Readings

[2] Fabien Sanglard. *Game Engine Black Book: Wolfenstein 3D*. Chapter 2, Section 2.1.3; `https://fabiensanglard.net/gebb/index.html` (free pay-what-you-want PDF). Fabien Sanglard, 2020.

License