

CSE 220: Systems Programming

Floating Point Numbers

Ethan Blanton

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.

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}

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

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:

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

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 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 $x \times 2^y$.

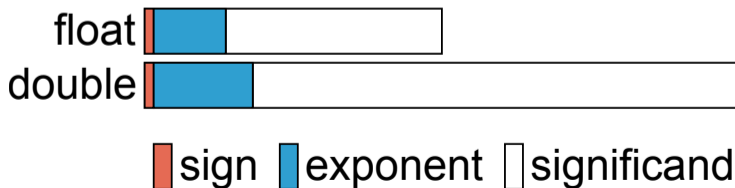
IEEE 754 Floating Point

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!

x (the significand) is stored:

- 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 0 is $x = 1.0$

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

This means that an exponent of 0 is stored as 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 00000000 00000000 00000000`

Remembering our significand and exponent storage rules, this means:

$x = 1.0$ (x is stored as significant digits **after the point**: $1 + 0$)

$y = 1$ (y is stored plus 127: $128 - 127$)

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

(We didn't use 1.0 because it's kind of a special case.)

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 be clean 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

¹See “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.

License

Copyright 2018, 2019, 2020 Ethan Blanton, All Rights Reserved.
Copyright 2019 Karthik Dantu, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see <https://www.cse.buffalo.edu/~eblanton/>.