Floating-Point Representation

Hui Chen [a]

[a]CUNY Brooklyn College, Brooklyn, NY, USA

September 10, 2023

Outline

Acknowledgement

The content of most slides come from the authors of the textbook:

Null, Linda, & Lobur, Julia (2018). The essentials of computer organization and architecture (5th ed.). Jones & Bartlett Learning.

Table of Contents

Lesson Objectives

Students are expected to be able to

1. describe the fundamentals of numerical data representation and manipulation in digital computers;
2. convert between various radix systems;
3. convert and perform arithmetic in signed integer representations;
4. explain how errors can occur in computations because of overflow and truncation;
5. *express floating numbers in floating-point representation*;
6. recognize the most popular character codes; and
7. describe the concepts of error detecting and correcting codes.

Table of Contents

Real Number Values

Your math classes introduce the concept of real number

Scientific or business applications deal with real number values.

Signed integer representations are not adequate.

Floating-point representation solves this problem.

Table of Contents

Floating-point Numbers

Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.

For example: $0.5 \times 0.25 = 0.125$

They are often expressed in scientific notation.

For example:

$$0.125 = 1.25 \times 10^{-1}$$

$$5,000,000.0 = 5.0 \times 10^{6}$$

Scientific Notation

Computers use a form of scientific notation for floating-point representation

Numbers written in scientific notation have three components:

$$+1.25 \times 10^{-1}$$

- $+$: sign
- 1.25: mantissa
- 10^{-1}: exponent

Floating-Point Representation

Computer representation of a floating-point number consists of three fixed-size fields:

Sign	Exponent	Significant

which is the standard arrangement of these fields.

Significant vs. mantissa

- "significand" and "mantissa" do not technically mean the same thing despite that many people use these terms interchangeably.
- In this class, "significand" is referred to the fractional part of a floating point number.

Designing Floating-Point Representation

Sign	Exponent	Significant

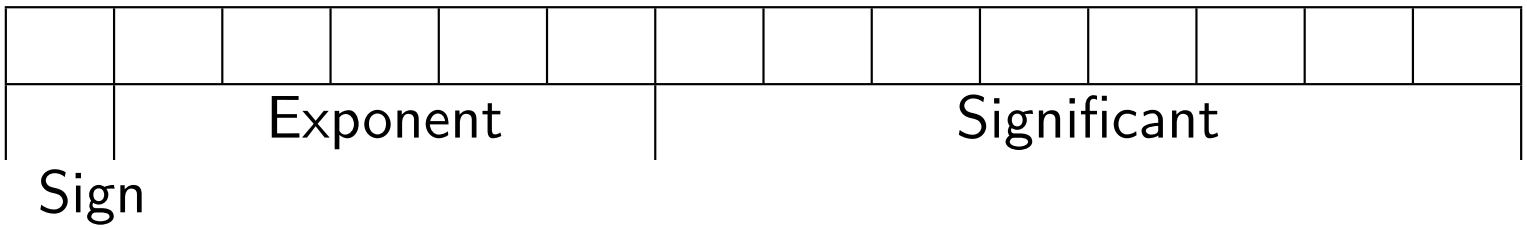
Given N bits to represent a floating point number, we decide

- Sign: 1 bit, indicating the sign of the stored value, typically,
  - 0: positive number
  - 1: negative number
- The size of the exponent field determines the range of values that can be represented.
- The size of the significand determines the precision of the representation.

Example: a 14-bit Simple Model

A hypothetical "Simple Model" to explain the concepts.

- A floating-point number is 14 bits in length.
- The exponent field is 5 bits.
- The significand field is 8 bits.



- The significand is always preceded by *an implied binary point*.
- Thus, the significand always contains a fractional binary value.
- The exponent indicates the power of 2 by which the significand is multiplied.

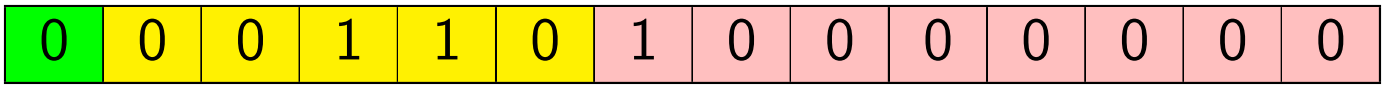
14-bit Simple (but Flawed) Model: Example

Express 32_{10} in the simple 14-bit floating-point model.

1. We know that $32 = 2^5$, rewrite it in (binary) scientific notation

$$32_{10} = 1.0_2 \times 2^5 = 0.1_2 \times 2^6$$

2. In a moment, we'll explain why we prefer the second notation (0.1×2^6) versus the first (1.0×2^5). Nevertheless, we have
   - Sign: "+", so 0
   - Exponent: $6_{10} = 110_2$
   - Significant: $1_{10} = 1_2$ with the implied binary point
3. Using this information, we put 110_2 (because $00110_2 = 6_{10}$) in the exponent field and 1 in the significand as shown.



14-bit Simple (but Flawed) Model: Multiple Representations?

Express 32_{10} in the simple 14-bit floating-point model.

We can have equivalent scientific notation, and consequently multiple equivalent floating-point representations.

$$32 = 0.1_2 \times 2^6$$

0	0	0	1	1	0	1	0	0	0	0	0	0	0

$$32 = 0.01_2 \times 2^7$$

0	0	0	1	1	1	0	1	0	0	0	0	0	0

$$32 = 0.001_2 \times 2^8$$

0	0	1	0	0	0	0	0	1	0	0	0	0	0

$$32 = 0.0001_2 \times 2^9$$

0	0	1	0	0	1	0	0	0	1	0	0	0	0

Flawed Model: Multiple Representations Are Bad!

The simple model introduced is good to understand some basics of floating-point representations, but the model is flawed!

- It allows multiple synonymous representations, which
  - waste space, and
  - cause confusion.
- The bigger problem is that the simple model does not allow negative exponents.
  - We have no way to express 0.5 (= 2-1)! (Notice that there is no sign in the exponent field.)

Floating-Point Representation (without the Flaws)

- Normalization: disallows synonymous forms.
- Biased exponent: allow both positive and negative exponents.

Floating-Point Representation: Normalization

Establish a "normalization" rule so that synonymous form are disallowed

- Approach 1 (no implied bits). The first digit of the significand must be 1, with no ones to the left of the radix point.
  - All significands must have the form 0.1xxxxxxxx
  - For example, $4.5 = 100.1_2 \times 2^0 = 1.001 \times 2^2 = 0.1001 \times 2^3$. The last expression is correctly normalized in Approach 1.
- Approach 2 (with implied bits). The significant has an implied 1 to the LEFT of the radix point (IEEE-754).
  - All significands must have the form 1.1xxxxxxxx with the leftmost 1 implied
  - For example, $4.5 = 100.1_2 \times 2^0 = 1.001 \times 2^2$. The significant would include only 001 (with unwritten 1 to the left of the radix point).

Floating-Point Representation: Biased Exponent

The exponent are "unsigned", how do we provide both negative, positive, and 0 exponent?

Floating-Point Representation: Biased Exponent

The exponent are "unsigned", how do we provide both negative, positive, and 0 exponent?

Use excess-M representation

Floating-Point Representation: Biased Exponent

To provide for negative exponents, we will use a biased exponent.

- In our simple model, we have a 5-bit exponent. Since

$$2^{5-1} - 1 = 2^4 - 1 = 15$$

  We adopt the excess-15 representation for the exponent with bias 15.
- which means
  - exponent values less than 15 are negative,
  - exponent values greater than 15 are positive, and
  - the bias exponent value is 0

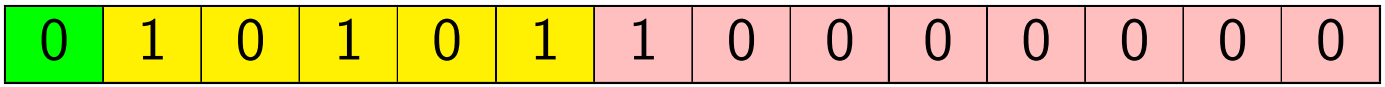
Simple but Improved Model: Example 1

Express 32_{10} in the revised 14-bit floating-point model.

We know that $32_{10} = 1.0_2 \times 2^5 = 0.1_2 \times 2^6$.

1. Sign: 0 for "+"
2. Exponent: To use our excess 15 biased exponent, we add 15 to 6, which yields $21_{10} = 10101_2$.
3. Significant: 0.1_2

So we have:



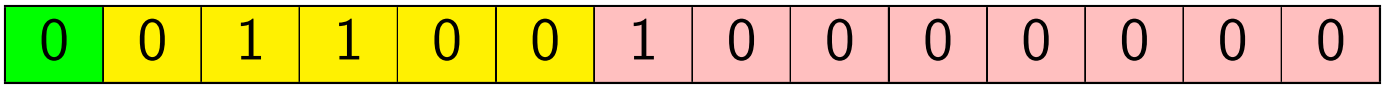
Simple but Improved Model: Example 2

Express 0.0625_{10} in the revised 14-bit floating-point model.

We know that $0.0625_{10} = 1.0_2 \times 2^{-4} = 0.1_2 \times 2^{-3}$.

1. Sign: 0 for "+"
2. Exponent: To use our excess 15 biased exponent, we add 15 to -3, which yields $12_{10} = 01100_2$.
3. Significant: 0.1_2

So we have:



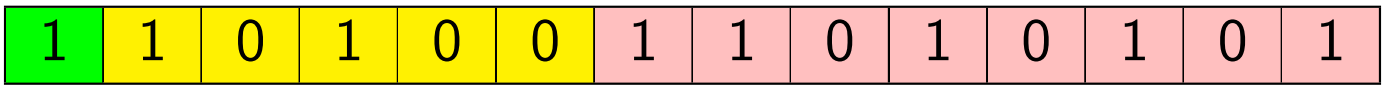
Simple but Improved Model: Example 3

Express -26.625_{10} in the revised 14-bit floating-point model.

We know that
$26.625_{10} = 11010.101_2 = 11010.101_2 \times 2^0 = 0.11010101_2 \times 2^5$.

1. Sign: 1 for "-"
2. Exponent: To use our excess 15 biased exponent, we add 15 to 5, which yields $20_{10} = 10100_2$.
3. Significant: 0.11010101_2

So we have:



IEEE Standard

The IEEE has established a standard called IEEE-754 for floating-point numbers, which are followed by most programming languages.

Common Name	Name	Width	Significand Bits	Exponent Bits	Java
Half Precision	Binary16	16	10	5	
Single Precision	Binary32	32	23	8	`float`
Double Precision	Binary64	64	52	11	`double`
Quadruple Precision	Binary128	128	112	15	
Octuple Precision	Binary256	256	236	19	

IEEE Standard: Significant With Implied Bits

In IEEE Standard, the significant has an implied 1 to the LEFT of the radix point (IEEE-754).

- All significands must have the form 1.1xxxxxxxx with the leftmost 1 implied
- For example, $4.5 = 100.1_2 \times 2^0 = 1.001 \times 2^2$. The significant would include only 001 (with unwritten 1 to the left of the radix point).

IEEE Standard: Example

Express -3.75 as a floating-point number using IEEE single precision.

First, let's normalize the number according to IEEE rules:

$$-3.75_{10} = -11.11_2 = -1.111 \times 2^1$$

So,

- Sign: 1 for "-"
- Exponent: The bias is $01111111_2 = 2^7 - 1 = 127_{10}$. The exponent in excess-127 is $(127 + 1)_{10} = 128_{10} = 10000000_2$
- Significand: $.111$ with implied 1 to the left of the radix point

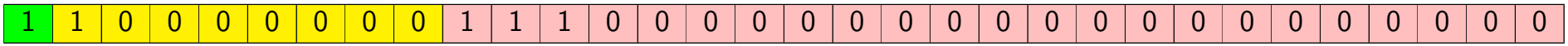


Table of Contents

IEEE Standard: Special Values

Using the IEEE-754 single precision floating point standard:

- An exponent of $255_10 = 11111111$ indicates a special value.
  - If the significand is zero, the value is $-\infty$ (- (infinity)) or $+\infty$ (+ (infinity)) according to the sign bit.
  - If the significand is nonzero, the value is `NaN`, "not a number," often used to flag an error condition.

Using the double precision standard:

- An exponent of $2047_10 = 11111111111$ indicates a special value. The rest are the same.

Floating Point Representation: +0.0 and -0.0

Both the 14-bit model that we have presented and the IEEE-754 floating point standard allow two representations for zero.

- Zero is indicated by all zeros in the exponent and the significand, but the sign bit can be either 0 or 1.

Although negative zero and positive zero do not have equal bit values, programming languages often evaluate `+0.0 == -0.0` to be true.

An interesting and informing discussion is in Java API documentation: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Double.html#equivalenceRelation

Table of Contents

Floating Point Representation: Arithmetic: Addition

Floating-point addition and subtraction are done using methods analogous to how we perform calculations using pencil and paper.

1. Express both operands in the same exponential power,
2. then add/subtract the numbers, preserving the exponent in the sum.
3. If the exponent requires adjustment, we do so at the end of the calculation.

Floating-Point Representation: Addition: Example 1

Find the sum of 12_{10} and 1.25_{10} using the 14-bit "simple but improved" floating-point model.

1. We find $12_{10} = 0.1100_2 \times 2^4$, and $1.25_{10} = 0.101_2 \times 2^1 = 0.000101_2 \times 2^4$. Express them in our model.
2. Add the numbers

	0	1	0	0	1	1	1	1	0	0	0	0	0	0
+	0	1	0	0	1	1	0	0	0	1	0	1	0	0
=	0	1	0	0	1	1	1	1	0	1	0	1	0	0

Thus, the result is $0.110101_2 \times 2^4$.

Floating-Point Representation: Subtraction: Example 2

Find the sum of 12_{10} and -1.25_{10} using the 14-bit "simple but improved" floating-point model.

Floating Point Representation: Arithmetic: Multiplication

Floating-point multiplication is also carried out in a manner akin to how we perform multiplication using pencil and paper.

- We multiply the two operands and add their exponents.
- If the exponent requires adjustment, we do so at the end of the calculation.

Floating Point Representation: Arithmetic: Multiplication: Example 3

Find the product of 12_{10} and 1.25_{10} using the 14-bit floating-point model.

1. We find $12_10 = 0.1100_2 \times 2^4$. And $1.25_{10} = 0.101_2 \times 2^1$.
2. Thus, our product is $0.0111100_2 \times 2^5 = 0.1111 \times 2^4$.
3. The normalized product requires an exponent of $(15 + 4)_{10} = 19_{10} = 10011_2$

Table of Contents

Floating-Point Errors

No matter how many bits we use in a floating-point representation, our model must be finite.

But the real number system is, of course, infinite, so our models can give nothing more than an approximation of a real value.

At some point, every model breaks down, introducing errors into our calculations.

By using a greater number of bits in our model, we can reduce these errors, but we can never totally eliminate them.

Be Aware of Floating-Point Errors

Our job becomes one of reducing error, or at least being aware of the possible magnitude of error in our calculations.

We must also be aware that errors can compound through repetitive arithmetic operations.

Floating-Point Errors Can be Amplified!

For example, our 14-bit model cannot exactly represent the decimal value 128.5. In binary, it is 9 bits wide: $10000000.1_2 = 128.5_{10}$

When we try to express 128.5_{10} in our 14-bit model, we lose the low-order bit, giving a relative error of:

$$\frac{128.5 - 128}{128.5} \approx 0.39\%$$

If we had a procedure that repetitively added 0.5 to 128.5, we would have an error of nearly 2% after only four iterations.

Tips to Reduce Floating Point Errors

Floating-point errors can be reduced when we use operands that are similar in magnitude.

If we were repetitively adding 0.5 to 128.5, it would have been better to iteratively add 0.5 to itself and then add 128.5 to this sum.

In this example, the error was caused by loss of the low-order bit.

Loss of the high-order bit is more problematic.

Be Aware of Floating-Point Overflow/Underflow

Floating-point overflow and underflow can cause programs to crash.

- Overflow occurs when there is no room to store the high-order bits resulting from a calculation.
- Underflow occurs when a value is too small to store, possibly resulting in division by zero.

Experienced programmers know that it's better for a program to crash than to have it produce incorrect, but plausible, results.

Range, Precision, and Accuracy

When discussing floating-point numbers, it is important to understand the terms range, precision, and accuracy.

- The range of a numeric integer format is the difference between the largest and smallest values that can be expressed.
- Accuracy refers to how closely a numeric representation approximates a true value.
- The precision of a number indicates how much information we have about a value.

Most of the time, greater precision leads to better accuracy, but this is not always true.

- For example: 3.1333 is a value of pi that is accurate to two digits, but has 5 digits of precision.

Commutative or Distributive?

Because of truncated bits, you cannot always assume that a particular floating point operation is commutative or distributive. This means that we cannot assume:

- $(a + b) + c = a + (b + c)$ or
- $a(b + c) = ab + ac$

Testing Equality?

To test a floating point value for equality to some other number, it is best to declare a "nearness to x" epsilon value.

- For example, instead of checking to see if floating point x is equal to 2 as follows:

```
if (x == 2.) {
    //...
}
```

it is better to use:

```
if (abs(x - 2) < epsilon) {
    // ...
}
```

(assuming we have epsilon defined correctly!)

Table of Contents

Summary and Q&A

You are expected to be able to

1. *express floating numbers in floating-point representation*;

Any questions on:

- Floating-Point Representation
- Example of a Simple (but Flawed) Model
- Simple but Improved Model
- IEEE Floating Point Standard
- Special Values
- Floating Point Arithmetic
- Floating Point Errors