

# Floating-point representation

A measurement can have a fractional value, such as 13.25 mm. However, computer memory contains only bits: zeros and ones. How can these bits represent the fractional part of a number?

We will start with binary fractions, introduce the basic idea of floating-point representation, and then examine the IEEE 754 binary32 format.

## 1. How do we write fractions in binary?

In decimal notation, the positions after the decimal point have weights of one tenth, one hundredth, one thousandth, and so on.

Binary notation follows the same principle, but each successive position has half the weight of the previous one:

| Position      | Immediately before the point | First fractional digit | Second fractional digit | Third fractional digit | Fourth fractional digit |
|---------------|------------------------------|------------------------|-------------------------|------------------------|-------------------------|
| Weight        | 1                            | 1/2                    | 1/4                     | 1/8                    | 1/16                    |
| Decimal value | 1                            | 0.5                    | 0.25                    | 0.125                  | 0.0625                  |

For example, the binary fraction **0.101** has the value:

$$1 \times 1/2 + 0 \times 1/4 + 1 \times 1/8 = 0.625.$$

To distinguish between number systems, we will use a subscript: 2 means binary, and 10 means decimal.

Therefore:

$$0.101_2 = 0.625_{10}.$$

This is not the same as the decimal number 0.101. In both cases, the point separates the integer and fractional parts, but the positional weights are different.

In binary notation, this separator is called the **binary point**.

### Example: writing 13.25 in binary

Consider the integer and fractional parts separately:

- $13 = 8 + 4 + 1$ , so the integer part is 1101.
- $0.25 = 1/4$ , so the fractional part is 01.

Therefore:

$$13.25_{10} = 1101.01_2.$$

Check:  $8 + 4 + 1 + 1/4 = 13.25$ .

## A general method: multiply the fractional part by two

We can find the binary digits of a fraction by repeatedly multiplying by two.

At each step:

1. Multiply the current fractional part by 2.
2. Record the integer part of the result: it will be 0 or 1.
3. Continue with only the remaining fractional part.

For example, let us convert 0.625:

| Step | Multiplication          | Next binary digit | Remaining fractional part |
|------|-------------------------|-------------------|---------------------------|
| 1    | $0.625 \times 2 = 1.25$ | 1                 | 0.25                      |
| 2    | $0.25 \times 2 = 0.5$   | 0                 | 0.5                       |
| 3    | $0.5 \times 2 = 1.0$    | 1                 | 0                         |

Read the digits **in the order in which they were produced**:

$$0.625_{10} = 0.101_2.$$

The remaining fraction has become zero, so the conversion is complete.

Unlike the repeated division method for integers, we do not reverse the order of these digits.

## 2. Where do we store the binary point?

There is no actual point character between the bits in memory. Its position must be defined by an interpretation rule.

One option is to decide in advance how many bits represent the fractional part. This is called **fixed-point representation**.

For example, suppose we use eight unsigned bits and agree that the last four bits represent the fractional part.

The stored pattern 00110100 is then interpreted as 0011.0100:

$$2 + 1 + 1/4 = 3.25.$$

Another way to describe the same rule is to read the stored bits as the unsigned integer 52 and divide it by 16:

$$52 / 16 = 3.25.$$

In this simple format:

- The distance between adjacent representable values is always  $1/16 = 0.0625$ .
- The largest representable value is  $255/16 = 15.9375$ .

Using more fractional bits would give a finer resolution. However, with the same total number of bits, fewer bits would remain for the integer part.

**Fixed-point representation uses a constant step size.** It can be useful when the measurement range and the required resolution are known in advance.

But what if we need to work with both very small and very large numbers?

### 3. The basic idea of floating-point representation

In engineering, we often write numbers using powers of ten:

- $13\,250 = 1.325 \times 10^4$ .
- $0.001325 = 1.325 \times 10^{-3}$ .

In both examples, 1.325 contains the significant digits. The power of ten determines the scale of the number.

The **exponent** is the number written above and to the right of the base: 4 and  $-3$  in these examples.

A negative exponent means division:

- $10^{-3} = 1/1000$ .
- $2^{-3} = 1/8$ .

**Floating-point representation** uses this idea. It stores the sign, the significant digits, and the exponent separately.

The position of the point depends on the exponent, which is why we say that the point “floats”.

For binary floating-point numbers, the base is **2**, rather than 10.

#### Putting the number into a standard form

We have already found the binary representation of 13.25:

**$1101.01_2$** .

Moving the binary point three positions to the left gives  $1.10101$ . This makes the number eight times smaller, so we must multiply it by  $2^3$ , or 8, to preserve its value:

**$1101.01_2 = 1.10101_2 \times 2^3$** .

For a nonzero number, we rearrange its magnitude so that exactly one digit, a 1, appears before the binary point.

This is called **normalized form**, and the rearrangement is called **normalization**.

The  $1.10101$  part is called the **significand**. Some teaching materials call it the **mantissa**.

The significand contains the significant binary digits. In normalized binary form, its value is at least 1 but less than 2.

For a number smaller than one, normalization gives a negative exponent:

$$0.00101_2 = 1.01_2 \times 2^{-3} = 0.15625_{10}.$$

Here, we moved the point three positions to the right. To preserve the value, we divide by eight.

#### 4. IEEE 754: what does each bit store?

The basic idea does not yet specify how many bits should be used for the sign, exponent, and significand.

To interpret data consistently, computers need a shared set of rules.

**IEEE 754** is a standard for floating-point representation and arithmetic. It defines storage formats, rounding rules, and special values, among other things.

We will examine its 32-bit binary format, called **binary32**. This is also known as **single precision**.

The 32 bits are divided into three consecutive groups. A group of bits with a specific purpose is called a **field**.

| Field           | Size    | Purpose                                                              |
|-----------------|---------|----------------------------------------------------------------------|
| Sign            | 1 bit   | 0: positive sign; 1: negative sign                                   |
| Biased exponent | 8 bits  | Stores the exponent with an offset, explained below                  |
| Fraction        | 23 bits | Stores the bits after the binary point in the normalized significand |

The total is  $1 + 8 + 23 = 32$  bits, or four bytes.

The following explanations apply to **normal, finite, nonzero numbers**. Zero and other special cases will be discussed separately.

##### Why store only the bits after the point?

A normalized binary significand always begins with  $1.$

We therefore do not need to store this leading 1 explicitly. When reading the number, we automatically put it back before the stored fractional bits.

For example, from the significand  $1.10101$ , only  $10101$  is placed in the fraction field. The remaining positions are filled with zeros on the right.

The omitted 1 is called the **implicit leading bit**, or **hidden bit**.

As a result, the 23 stored fraction bits provide **24 significant binary digits** for normal numbers.

The significand and the stored fraction field are therefore not exactly the same thing.

### How can we store a negative exponent?

The exponent can be positive or negative. However, the binary32 exponent field is **not interpreted using two's complement**.

Instead, we add **127** to the actual exponent and store the resulting non-negative integer.

This offset is called the **bias**.

- **When storing:** stored exponent = actual exponent + 127.
- **When reading:** actual exponent = stored exponent – 127.

| Actual exponent | Stored value | Stored value in eight bits |
|-----------------|--------------|----------------------------|
| –3              | 124          | 01111100                   |
| 0               | 127          | 01111111                   |
| +3              | 130          | 10000010                   |

For normal numbers, the stored exponent can range from 1 to 254. This corresponds to actual exponents from –126 to +127.

Stored exponent values 0 and 255 are reserved for special cases.

## 5. Complete example: storing –13.25 in binary32

### Step 1: determine the sign

The number is negative, so the **sign bit is 1**.

We now work with its magnitude, 13.25.

### Step 2: convert to binary and normalize

From our earlier calculation:

$$13.25_{10} = 1101.01_2 = 1.10101_2 \times 2^3.$$

The actual exponent is **3**, and the significand is **1.10101**.

### Step 3: construct the exponent field

Add the bias:

$$3 + 127 = 130.$$

Since  $130 = 128 + 2$ , the eight exponent bits are:

**10000010.**

#### Step 4: construct the fraction field

Omit the implicit leading 1 from 1.10101.

The remaining 10101 contains five bits. Add 18 zeros on the right to obtain a 23-bit fraction field:

**10101000000000000000000.**

No rounding is needed because all significant digits fit.

#### Step 5: combine the fields

| Sign: 1 bit | Exponent: 8 bits | Fraction: 23 bits       |
|-------------|------------------|-------------------------|
| 1           | 10000010         | 10101000000000000000000 |

The complete 32-bit pattern is:

1 10000010 10101000000000000000000

The spaces show the field boundaries; they are not stored characters.

We have written the pattern from the sign bit down to the lowest fraction bit. The order of bytes in memory is a separate issue.

**Important:** the negative sign is stored in a separate bit. To represent  $-13.25$ , we do not invert the complete pattern for  $+13.25$  and add 1. Two's complement is the different representation introduced for signed integers.

#### Checking the result: what do these 32 bits mean?

1. The sign bit is 1, so the result is negative.
2. The exponent field is 10000010, or 130. The actual exponent is  $130 - 127 = 3$ .
3. Restore the implicit leading 1 before the fraction bits: 1.10101.
4. The significand is  $1 + 1/2 + 1/8 + 1/32 = 1.65625$ .
5. Multiply by  $2^3$ :  $1.65625 \times 8 = 13.25$ .
6. Apply the negative sign:  **$-13.25$** .

#### Expressing the rule as a formula

Let:

- **s** be the sign bit.

- **E** be the unsigned integer value of the exponent field.
- **F** be the unsigned integer value of the 23-bit fraction field.

For a normal binary32 number:

$$\text{value} = (-1)^s \times (1 + F / 2^{23}) \times 2^{E-127}.$$

This formula combines the steps we have just performed separately.

It applies in this form only when  $1 \leq E \leq 254$ .

## 6. Another example: a small positive number

Let us represent **0.15625**.

Since  $0.15625 = 1/8 + 1/32$ , its binary form is  $0.00101$ .

Normalized:

$$1.01_2 \times 2^{-3}.$$

- The sign bit is **0**, because the number is positive.
- The actual exponent is **-3**. The stored exponent is  $-3 + 127 = \mathbf{124}$ , or  $01111100$ .
- The fraction bits after the implicit leading 1 are  $01$ . Pad them with zeros on the right to make 23 bits.

| Sign | Exponent | Fraction                |
|------|----------|-------------------------|
| 0    | 01111100 | 01000000000000000000000 |

To check the result, the significand is 1.25 and the power of two is 1/8:

$$1.25 / 8 = 0.15625.$$

## 7. Why can some numbers not be stored exactly?

In decimal notation,  $1/3$  has an infinite expansion:

**0.3333...**

Binary notation also has fractions with infinite expansions, but they are not exactly the same fractions as in decimal notation.

For example:

$$0.1_{10} = 0.0001100110011..._2.$$

After the first fractional digit, the group  $0011$  repeats.

We can see the repetition using multiplication by two:

| Step | Multiplication       | Output bit | Remaining fraction |
|------|----------------------|------------|--------------------|
| 1    | $0.1 \times 2 = 0.2$ | 0          | 0.2                |
| 2    | $0.2 \times 2 = 0.4$ | 0          | 0.4                |
| 3    | $0.4 \times 2 = 0.8$ | 0          | 0.8                |
| 4    | $0.8 \times 2 = 1.6$ | 1          | 0.6                |
| 5    | $0.6 \times 2 = 1.2$ | 1          | 0.2                |

After step 5, the remaining fraction is 0.2 again. The same sequence will therefore repeat indefinitely.

A finite number of bits can store only a finite number of digits, so we must choose an **approximation**.

In binary32, the representable value closest to 0.1 is approximately:

**0.10000000149.**

This is not a computer malfunction. It is similar to writing 0.333 instead of the exact value  $1/3$ .

### The basic rounding rule

The default IEEE 754 rounding rule for binary arithmetic selects the **nearest representable value**.

If the exact value is halfway between two representable values, it selects the one whose last retained binary digit is 0.

This is called **round to nearest, ties to even**.

Consequently, storing a number is not always a matter of simply discarding the remaining digits. Rounding can change the digits that are retained.

### Precision does not mean a fixed number of decimal places

Normal binary32 numbers have 24 significant binary digits.

As the exponent increases, those digits represent larger positional weights. Therefore, the spacing between representable numbers generally increases with their magnitude.

For example, in binary32:

- From 1 up to 2, the spacing between consecutive representable values is  $2^{-23}$ , approximately 0.0000001192.
- From  $2^{24} = 16\,777\,216$  up to  $2^{25}$ , the spacing is **2**.

The integer **16 777 217** therefore cannot be stored exactly in binary32, even though it is nowhere near the largest value the format can represent.

Its representable neighbours are 16 777 216 and 16 777 218. The default rounding rule selects 16 777 216.

**Range and precision are different properties.** A value can be within the representable range without being exactly representable.

## 8. Zero, very small numbers, and special values

A normal significand always has an implicit leading 1. This rule cannot represent zero.

The standard therefore assigns different interpretations to certain exponent patterns.

In the following table, E is the unsigned value of the exponent field, and F is the unsigned value of the fraction field.

| Stored exponent, E | Fraction, F | Meaning                                                                     |
|--------------------|-------------|-----------------------------------------------------------------------------|
| 1–254              | Any         | A normal, finite, nonzero number with an implicit leading 1                 |
| 0                  | 0           | Zero: +0 or –0, depending on the sign bit                                   |
| 0                  | Nonzero     | A subnormal number: a nonzero value with a magnitude below the normal range |
| 255                | 0           | Positive or negative infinity, depending on the sign bit                    |
| 255                | Nonzero     | NaN: a special “Not a Number” value                                         |

### Subnormal numbers

For **subnormal numbers**, the leading bit is 0 rather than 1, and the exponent of two is fixed at –126.

This allows nonzero values smaller in magnitude than the smallest normal number, but with fewer significant digits.

The smallest positive binary32 value is:

$2^{-149}$ , approximately  $1.4 \times 10^{-45}$ .

The formula for a subnormal value is:

$$\text{value} = (-1)^s \times (F / 2^{23}) \times 2^{-126}.$$

There is no implicit leading 1 in this formula.

### Infinity and NaN

**Infinity** can appear, for example, when a result is too large to represent as a finite number. It is a special value, not an extremely large finite number.

**NaN** stands for **Not a Number**. It can indicate an operation that has no ordinary numerical result, such as 0/0.

Depending on the programming environment, such an operation may produce a special value or an error.

## Positive and negative zero

+0 and −0 compare as equal in ordinary numerical equality comparisons. However, their bit patterns differ, and some operations distinguish between them.

This differs from two's complement integer representation, which has only one representation of zero.

## 9. What changes with 64-bit floating-point representation?

Another common IEEE 754 format is **binary64**, also called **double precision**.

It uses the same basic idea, but allocates more bits to both the exponent and the fraction.

| Property                                     | binary32                   | binary64                    |
|----------------------------------------------|----------------------------|-----------------------------|
| Total size                                   | 32 bits, 4 bytes           | 64 bits, 8 bytes            |
| Sign                                         | 1 bit                      | 1 bit                       |
| Exponent field                               | 8 bits                     | 11 bits                     |
| Stored fraction bits                         | 23 bits                    | 52 bits                     |
| Significant binary digits for normal numbers | 24                         | 53                          |
| Exponent bias                                | 127                        | 1023                        |
| Approximate decimal precision                | About 7 significant digits | About 16 significant digits |
| Largest positive finite value, approximately | $3.4 \times 10^{38}$       | $1.8 \times 10^{308}$       |

A **significant digit** is not the same as a digit after the decimal point.

For example, 1.234567 and 1 234 567 both contain seven significant decimal digits.

The decimal precision figures in the table are approximate descriptions, not guarantees that every decimal number with that many digits can be stored exactly.

Binary64 provides much greater precision, but **it still cannot store 0.1 exactly**, because the binary expansion of 0.1 is infinite.

## 10. What should we consider in engineering calculations?

- **Floating-point results may contain rounding error.** This is different from the measurement error of an instrument, although both may matter in a calculation.
- **A small change can disappear when added to a large value.** If the exact result rounds back to the original representable value, the stored value does not change.
- **The order of calculations can matter.** Rounding of intermediate results means that regrouping additions can change the final result.
- **Calculated or measured values are often compared using a tolerance.** The tolerance should reflect the units, the magnitude of the values, and the accuracy required by the task.

For example, suppose we consider two length results sufficiently close if they differ by no more than 0.001 mm.

We then check whether the absolute value of their difference is at most 0.001 mm.

For values on a different scale, we may also need a relative tolerance: a tolerance expressed in relation to the magnitude of the values.

Displaying more decimal places does not improve the accuracy of the original measurement or of an already completed calculation.

## Review questions and exercises

1. Write 0.375 as a binary fraction.
2. Write 6.5 in normalized binary form.
3. Give the three fields of +6.5 in IEEE 754 binary32 format.
4. Which bits change if we store  $-6.5$  instead of  $+6.5$ ?
5. Why can neither binary32 nor binary64 store 0.1 exactly?
6. How can an integer be within the range of a floating-point format but still not be exactly representable?

## Answers

1.  $0.375 = 1/4 + 1/8$ , so its binary representation is  **$0.011_2$** .
2.  $6.5 = 110.1_2 = \mathbf{1.101_2 \times 2^2}$ .
3. Sign: 0; exponent:  $2 + 127 = 129$ , or 10000001; fraction: 1010000000000000000000.
4. Only the sign bit changes, from 0 to 1.
5. The binary expansion of 0.1 repeats indefinitely, but the number of available storage bits is finite.
6. Range limits the magnitude of values, while the length of the significand limits precision. For large values, the spacing between consecutive representable numbers can be greater than one.

## Sources and further reading

- [Oracle Numerical Computation Guide: IEEE formats and special values](#)
- [David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic](#)

This section introduces the basic principles of binary floating-point representation. It does not cover all IEEE 754 formats or the full set of arithmetic rules.

From:  
<https://edu.iit.uni-miskolc.hu/> - Institute of Information Science - University of Miskolc

Permanent link:  
[https://edu.iit.uni-miskolc.hu/tanszek:oktatas:techcomm:floating-point\\_representation?rev=1791830311](https://edu.iit.uni-miskolc.hu/tanszek:oktatas:techcomm:floating-point_representation?rev=1791830311)

Last update: 2026/10/12 18:38

