

Computer arithmetic

Computers store and process numbers in binary (base 2) form consisting of two digits 0 and 1. Each binary digit (zero or one) is called a *bit* and a collection of 8 bits is called a *byte*. The left hand most bit in the representation of a number is called the *most significant bit* (MSB) while the right hand most bit is called the *least significant bit* (LSB).

Conversions between binary and decimal systems

We will first see how to convert a real number in binary form into its decimal equivalent. The key idea is that the binary digits to the left of the binary decimal point increase in factors of 2 for each place moved over. Similarly, the digits to the right of the binary decimal point decrease in factors of 2 for each place moved over. Consider the real number $(101.1)_2$ in binary form. We have

$$(101.1)_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} = (5.5)_{10}.$$

Now consider the reverse operation of converting a real decimal number into its binary equivalent. We will illustrate the technique on the number $(20.5625)_{10}$. Consider the integer portion 20 of 20.5625.

$$\frac{20}{2} = 10 \text{ (remainder 0)} \rightarrow 0$$

$$\frac{10}{2} = 5 \text{ (remainder 0)} \rightarrow 0$$

$$\frac{5}{2} = 2 \text{ (remainder 1)} \rightarrow 1$$

$$\frac{2}{2} = 1 \text{ (remainder 0)} \rightarrow 0$$

$$\frac{1}{2} = 0 \text{ (remainder 1)} \rightarrow 1$$

This gives $(20)_{10} = (10100)_2$ (read results in reverse order). Now consider the fractional portion 0.5625 of 20.5625.

$$\begin{array}{rcl}
 0.5625 < 1 & \rightarrow & 0. \\
 2 \times 0.5625 = 1.1250 > 1 & \rightarrow & 1 \text{ (remove 1 from 1.1250)} \\
 2 \times 0.125 = 0.25 < 1 & \rightarrow & 0 \\
 2 \times 0.25 = 0.5 < 1 & \rightarrow & 0 \\
 2 \times 0.5 = 1.0 = 1 & \rightarrow & 1 \text{ (remove 1 from 1 which gives 0 - finished)}
 \end{array}$$

Therefore, $(0.5625)_{10} = (0.1001)_2$. Combining our two results, we have $(20.5625)_{10} = (10100.1001)_2$.

Exercise: Represent $(0.1)_{10}$ in binary form.

Floating point representation

In *floating point notation*, a real number x is expressed in decimal as

$$x = \pm S \times 10^E, \text{ where } 1 \leq S < 10.$$

and E is an integer. The numbers S and E are called the *significand* and the *exponent*, respectively. For instance, $6519.23 = 6.51923 \times 10^3$ and $0.00000391 = 3.91 \times 10^{-6}$. The fractional portion of the significand is called the *mantissa*.

Binary numbers can similarly be represented in floating point notation as

$$x = \pm S \times 2^E, \text{ where } S = 1.$$

and E is a binary integer. We will illustrate how to represent the decimal number 50 as a binary floating point number

$$\begin{aligned}
 50 &= \frac{50}{2^5} \times 2^5 \text{ (50 lies between } 2^5 = 32 \text{ and } 2^6 = 64) \\
 &= 1.5625 \times 2^5 \\
 &= (1.1001)_2 \times 2^{(101)_2}.
 \end{aligned}$$

The process used to obtain the floating point representation of a binary number is also called *normalization* and the new number is said to be in *normalized* form.

IEEE 754 standards for representing floating point numbers

Floating point numbers are stored in *single precision* (short) or *double precision* (long) formats. In single precision format, a number is stored in a string of 32 bits (4 bytes), and in double precision in a string of 64 bits (8 bytes). In both cases, the MSB (first bit to the left) stores the sign (0 for positive and 1 for negative) of the number. The next 8 bits in single precision (11 bits in double precision) are used for storing the exponent (actually the biased exponent as we discuss later!). Finally, the last 23 bits in single precision (52 bits in double precision) are used for storing the mantissa. The value of the mantissa is entered in binary form. The value of the exponent is entered with a bias. The bias is introduced in order to avoid using one of the bits for the sign of the exponent (which can be positive and negative too!)

In the single precision format, 127 is added to the exponent of the normalized floating point number that is being represented. Here is the reason why: The

range of the exponent field bit strings for normalized single precision numbers varies between 00000001 to 11111110 (the string 00000000 is used to represent 0 and the string 11111111 is used to represent $\pm\infty$). These strings represent the decimal numbers $1, \dots, 254$, and so we choose a bias of 127 to represent exponents from $E_{\min} = -126 = 1 - 127$ to $E_{\max} = 127 = (254 - 127)$. Similarly in double precision, 1023 is added to the exponent of the normalized floating point number that is being represented.

For example, the number 22.5 is represented in single precision as

$$\begin{aligned} 22.5 &= \frac{22.5}{2^4} \times 2^4 \\ &= 1.40625 \times 2^4 \\ &= (1.01101)_2 \times 2^{(100)_2} \\ &= 0 \text{ (sign bit) } 10000011 \text{ (8 bit biased exponent 131) } 0110100000000000000000 \text{ (23 bit mantissa)}. \end{aligned}$$

Exercise: Write 22.5 in double precision.

Suppose we are given the 32 bit string SCF , where S , C , and F represent the sign bit, exponent bit string, and mantissa bit string, respectively. This bit string represents the floating point number $(-1)^S(1.F)_2 \times 2^{(C-01111111)_2}$ in single precision (note that $(127)_{10} = (01111111)_2$). Similarly, the 64 bit string SCF represents the floating point number $(-1)^S(1.F)_2 \times 2^{(C-011111111111)_2}$.

Exercise: What floating point number does the 64 bit string

01000000001110111001000100

represent?

The smallest positive normalized single format number is

$$\begin{aligned} N_{\min} &= (1.000\dots 0)_2 \times 2^{-126} \\ &\approx 1.2 \times 10^{-38}. \end{aligned}$$

The largest positive normalized double format number is

$$\begin{aligned} N_{\max} &= (1.111\dots 1)_2 \times 2^{127} \\ &\approx 3.4 \times 10^{38}. \end{aligned}$$

Numbers smaller than N_{\min} result in *underflow* while numbers larger than N_{\max} result in *overflow* on the computer.

Exercise: Show that the smallest and largest positive normalized double format numbers are $N_{\min} \approx 2.2 \times 10^{-308}$ and $N_{\max} \approx 1.8 \times 10^{308}$, respectively.

Precision and Machine Epsilon of IEEE single and double formats

The first single format number larger than 1 is $1 + 2^{-23}$ (Why?). Similarly, the first double format number larger than 1 is $1 + 2^{-52}$. Thus, single precision corresponds to $2^{-23} \approx 10^{-7}$, i.e., about 7 digits of precision. Similarly, double precision corresponds to $2^{-52} \approx 10^{-16}$, i.e., about 16 digits of precision. Because of its greater precision, the double format is preferred for most applications in scientific computing, although the single format provides an efficient way to store more quantities of data. We will refer to $\epsilon = 2^{-52} = 2.22 \times 10^{-16}$ as the *machine epsilon* in this course. In general, if p is the precision (23 for single format and 52 for double format) of the computer arithmetic, then $\epsilon = 2^{-p}$.

Rounding

Suppose a real number x cannot be expressed exactly as a single or a double format number (not a floating point number), then we need to approximate x by something else on the computer. Let p (23 for single format and 52 for double format) denote the precision. Suppose, we have a number x that is not an floating point number, i.e.,

$$x = (1.b_1b_2, \dots, b_p b_{p+1} \dots)_2 \times 2^E.$$

The closest floating point number less than or equal to x is

$$x_- = (1.b_1b_2, \dots, b_p)_2 \times 2^E$$

that is obtained by discarding the bits b_{p+1} , etc in the binary representation of x . Similarly,

$$x_+ = ((1.b_1b_2, \dots, b_p)_2 + (0.00, \dots, 01)_2) \times 2^E$$

is the next floating point number that is bigger than x (Why?). The two commonly used methods for rounding a non-floating point number x into its floating point approximation $\text{fl}(x)$ are:

1. **Round towards zero:** We have $\text{fl}(x) = x_-$ if $x \geq 0$ and $\text{fl}(x) = x_+$ if $x < 0$.
2. **Round to nearest:** In this case, $\text{fl}(x)$ is the closest floating point number x_- or x_+ that is nearest to x .

This is similar to the rounding of decimal numbers: For instance, suppose we have to round $x = 6.5926$ to 3 digits of precision (3 digits after the decimal point), then we have $\text{fl}(x) = 6.592$ if round towards zero is employed and $\text{fl}(x) = 6.593$ if round to nearest is employed.

Exercise: What are the IEEE single format binary representations for $(0.1)_{10}$ with either of the two rounding modes?

We can show the following:

Theorem 1 *The relative rounding error associated with rounding is given by*

$$\frac{|\text{fl}(x) - x|}{|x|} \leq 2^{-p} = \epsilon \text{ (round towards zero)}$$
$$\frac{|\text{fl}(x) - x|}{|x|} \leq 2^{-p-1} = \frac{\epsilon}{2} \text{ (round to nearest)}$$

where ϵ is the machine epsilon.

Arithmetic operations on the computer

Suppose, we want to add two numbers x and y on the computer. The IEEE standard stipulates that this operation is performed on the computer as follows:

$$x \oplus y = \text{fl}(\text{fl}(x) + \text{fl}(y)).$$

This should be interpreted as follows: If x and y are not floating point numbers, then we first find their floating point equivalents $\text{fl}(x)$ and $\text{fl}(y)$ (these depend on the rounding mode used!). We then add $\text{fl}(x)$ and $\text{fl}(y)$ and if the result $\text{fl}(x) + \text{fl}(y)$ is not a floating point number, then we round it too. Subtraction, multiplication, and division are performed similarly.

Exercise: Look at Example 3 on pages 21-22 of Burden and Faires [1].

References

- [1] R.L. BURDEN AND J. DOUGLAS FAIRES, *Numerical Analysis*, 8th edition, Thomson Brooks/Cole, 2005. (See Section 1.2)
- [2] D. GOLDBERG, *What every computer scientist should know about floating point arithmetic*, ACM Computing Surveys, 23(1991), pp. 5-48. Available at <http://cch.loria.fr/documentation/IEEE754/ACM/goldberg.pdf>. (See suggested readings on course webpage).
- [3] M.L. OVERTON, *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM, Philadelphia, 2001. (See Chapters 1-5)