Overton, Numerical Computing with IEEE Floating Point Arithmetic

Chapter 3

Computer Representation of Numbers

What is the best way to represent numbers on the computer? Let us start by considering integers. Typically, integers are stored using a 32-bit word, so we confine our attention to this case. If we were concerned only with nonnegative integers, the representation would be easy: a bitstring specifying the binary representation of the integer. For example, the integer 71 (see (2.1)) would be stored as

$$\boxed{00000000000000000000000001000111}.$$

The nonnegative integers that we can represent in this way range from 0 (a bitstring of 32 zeros) to $2^{32}-1$ (a bitstring of 32 ones). The number 2^{32} is too big, since its binary representation consists of a one followed by 32 zeros.

Signed Integers via 2's Complement

In fact, we need to be able to represent negative integers in addition to positive integers and 0. The most obvious idea is *sign-and-modulus*: use one of the 32 bits to represent the sign, and use the remaining 31 bits to store the magnitude of the integer, which may then range from 0 to $2^{31}-1$. However, nearly all machines use a more clever representation called 2's *complement*.[5] A nonnegative integer x, where $0 \le x \le 2^{31}-1$, is stored as the binary representation of x, but a negative integer $-y$, where $1 \le y \le 2^{31}$, is stored as the binary representation of the positive integer

$$2^{32} - y. \tag{3.1}$$

For example, the integer -71 is stored as

$$\boxed{11111111111111111111111110111001}.$$

In order to see that this is correct, let us add the 2's complement representations for 71 and -71 together:

$$\begin{array}{rl} & (\ 00000000000000000000000001000111\ )_2 \\ + & (\ 11111111111111111111111110111001\ )_2 \\ = & (\ 100000000000000000000000000000000\ )_2. \end{array}$$

[5] There is a third system called 1's complement, where a negative integer $-y$ is stored as the binary representation of $2^{32}-y-1$. This system was used by some supercomputers in the 1960s and 1970s but is now obsolete.

Adding in binary by hand is like adding in decimal. Proceed bitwise right to left; when 1 and 1 are added together, the result is 10 (base 2), so the resulting bit is set to 0 and the 1 is carried over to the next bit to the left. The sum of the representations for 71 and for -71 is thus the bitstring for 2^{32}, as required by the definition (3.1). The bit in the leftmost position of the sum cannot be stored in the 32-bit word and is called an *overflow bit*. If it is discarded, the result is 0—exactly what we want for the result of $71 + (-71)$. This is the motivation for the 2's complement representation.

Exercise 3.1 *Using a 32-bit word, how many different integers can be represented by* (a) *sign and modulus;* (b) *2's complement? Express the answer using powers of 2. For which of these two systems is the representation for zero unique?*

Exercise 3.2 *Suppose we wish to store integers using only a 16-bit half-word (2 bytes). This is called a* short integer *format. What is the range of integers that can be stored using 2's complement? Express the answer using powers of 2 and also translate the numbers into decimal notation.*

Exercise 3.3 *Using an 8-bit format for simplicity, give the 2's complement representation for the following integers:* 1, 10, 100, -1, -10, *and* -100. *Verify that addition of a negative number to its positive counterpart yields zero, as required, when the overflow bit is discarded.*

Exercise 3.4 *Show that if an integer x between -2^{31} and $2^{31}-1$ is represented using 2's complement in a 32-bit word, the leftmost bit is 1 if x is negative and 0 if x is positive or 0.*

Exercise 3.5 *An easy way to convert the representation of a nonnegative integer x to the 2's complement representation for $-x$ begins by changing all 0 bits to 1s and all 1 bits to 0s. One more step is necessary to complete the process; what is it, and why?*

All computers provide hardware instructions for adding integers. If two positive integers are added together, the result may give an integer greater than or equal to 2^{31}. In this case, we say that *integer overflow* occurs. One would hope that this leads to an informative error message for the user, but whether or not this happens depends on the programming language and compiler being used. In some cases, the overflow bits may be discarded and the programmer must be alert to prevent this from happening.[6] The same problem may occur if two negative integers are added together, giving a negative integer with magnitude greater than 2^{31}.

On the other hand, if two integers with opposite sign are added together, integer overflow cannot occur, although an overflow bit may arise when the 2's complement bitstrings are added together. Consider the operation

$$x + (-y),$$

where

$$0 \le x \le 2^{31} - 1 \quad \text{and} \quad 1 \le y \le 2^{31}.$$

Clearly, it is possible to store the desired result $x - y$ without integer overflow. The result may be positive, negative, or zero, depending on whether $x > y$, $x = y$, or $x < y$. Now let us see what happens if we add the 2's complement representations for

[6]The IEEE floating point standard, to be introduced in the next chapter, says nothing about requirements for integer arithmetic.

x and $-y$, i.e., the bitstrings for the nonnegative numbers x and $2^{32} - y$. We obtain the bitstring for

$$2^{32} + x - y = 2^{32} - (y - x).$$

If $x \geq y$, the leftmost bit of the result is an overflow bit, corresponding to the power 2^{32}, but this bit can be discarded, giving the correct result $x - y$. If $x < y$, the result fits in 32 bits with no overflow bit, and we have the desired result, since it represents the negative value $-(y - x)$ in 2's complement.

This demonstrates an important property of 2's complement representation: no special hardware is needed for integer subtraction. The addition hardware can be used once the negative number $-y$ has been represented using 2's complement.

Exercise 3.6 *Show the details for the integer sums* $50 + (-100)$, $100 + (-50)$, *and* $50 + 50$, *using an 8-bit format.*

Besides addition, the other standard hardware operations on integer operands are multiplication and division, where by the latter, we mean integer division, yielding an integer quotient. Multiplication may give integer overflow. Integer division by zero normally leads to program termination and an error message for the user.

Exercise 3.7 *(D. Goldberg) Besides division by zero, is there any other division operation that could result in integer overflow?*

Fixed Point

Now let us turn to the representation of nonintegral real numbers. Rational numbers could be represented by pairs of integers, the numerator and denominator. This has the advantage of accuracy but the disadvantage of being very inconvenient for arithmetic. Systems that represent rational numbers in this way are said to be *symbolic* rather than numeric. However, for most numerical computing purposes, real numbers, whether rational or irrational, are approximately stored using the binary representation of the number. There are two possible methods, called fixed point and floating point.

In *fixed point* representation, the computer word may be viewed as divided into three fields: one 1-bit field for the sign of the number, one field of bits for the binary representation of the number before the binary point, and one field of bits for the binary representation after the binary point. For example, in a 32-bit word with field widths of 15 and 16, respectively, the number 11/2 (see (2.2)) would be stored as

0	000000000000101	1000000000000000

while the number 1/10 would be approximately stored as

0	000000000000000	0001100110011001

The fixed point system is severely limited by the size of the numbers it can store. In the example just given, only numbers ranging in size from (exactly) 2^{-16} to (slightly less than) 2^{15} could be stored. This is not adequate for many applications. Therefore, fixed point representation is rarely used for numerical computing.

Floating Point

Floating point representation is based on *exponential* (or *scientific*) notation. In exponential notation, a nonzero real number x is expressed in decimal as

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