Floating Point Numbers

Bob Brown
Computer Science Department
Southern Polytechnic State University

Limitations of Binary Integers

The natural arithmetic operand in a computer is the binary integer. However, the range of numbers that can be represented is limited by the computer's word size. We cannot represent very large or very small numbers. For example, in a computer with a 32 bit word, the largest signed number is $2^{31} - 1$. The range is further diminished if some bits of the word are used for fractions. There are techniques for performing integer arithmetic on groups of two or more words, but these are both painful for the programmer and consuming of CPU time.

It is not uncommon for very large and very small numbers to occur in the kinds of problems for which computers are used. These numbers do not lend themselves to representation in integer form, or integer and fraction form. Another approach is needed for problems whose variables are not small integers.

Scientific Notation

Scientists and engineers have developed a compact notation for writing very large or very small numbers. If we wrote it out, the mass of the sun in grams would be a two followed by 33 zeroes. The speed of light in meters per second would be a three followed by eight zeroes. These same numbers, when expressed in scientific notation, are 2×10^{33} and 3×10^{8}. Any number *n* can be expressed as

$$n = f \times 10^{e}$$

where *f* is a fraction and *e* is an exponent. Both *f* and *e* may be negative. If *f* is negative the number *n* is negative. If *e* is negative, the number is less than one.

The essential idea of scientific notation is to separate the significant digits of a number from its magnitude. The number of significant digits is determined by the size of *f* and the range of magnitude is determined by the size of *e*.

We wrote the speed of light as 3×10^{8} meters per second. If that is not precise enough, we can write 2.997×10^{8} to express the same number with four digits of precision.

Floating Point Numbers

Floating-point number systems apply this same idea – separating the significant digits of a number from its magnitude – to representing numbers in computer systems.

Relatively small numbers for the fraction and exponent part provide a way to represent a very wide range with acceptable precision.

In the early days of computers, each manufacturer developed their own floating-point representation. These were incompatible. In some cases, they even produced wrong answers. Floating-point arithmetic has some subtleties which are beyond the scope of this paper.

In 1985, the Institute of Electrical and Electronic Engineers published IEEE Standard 754 for floating-point arithmetic. Virtually all general purpose processors built today have floating-point units which conform to IEEE 754. The examples in this paper describe IEEE 754 floating-point number formats.

Instead of using base ten and powers of ten like scientific notation, IEEE 754 floating-point uses a binary fraction and an exponent that is considered to be a power of two. The format of a single-precision floating-point number is shown in Figure 1. The leftmost bit indicates the sign of the number, with a zero indicating positive and a one indicating negative. The exponent occupies eight bits and is also signed. A negative exponent indicates that the fraction is multiplied by a negative power of two. The exponent is stored as an excess 127 number, which means that the value stored is 127 more than the true value. A stored value of one indicates a true value of -126. A stored value of 254 indicates a true value of +127. Exponents values of zero or 255 (all ones) are used for special purposes described later. The fraction part is a 23-bit binary fraction with the binary point assumed to be to the left of the first bit of the fraction. The approximate range of such a number is $\pm 10^{-38}$ to $\pm 10^{38}$. This is substantially more than we can express using a 32-bit binary integer.

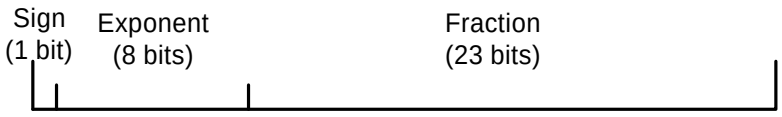

Figure 1. Format of an IEEE 754 single-precision floating-point number.

Normalized Numbers

We represented the speed of light as 2.997×10^8. We could also have written 0.2997×10^9 or 0.02997×10^{10}. We can move the decimal point to the left, adding zeroes as necessary, by increasing the exponent by one for each place the decimal point is moved. Similarly, we can compensate for moving the decimal point to the right by decreasing the exponent. However, if we are dealing with a fixed-size fraction part, as in a computer implementation, leading zeroes in the fraction part cost precision. If we were limited to four digits of fraction, the last example would become 0.0299×10^{10}, a cost of one digit of precision. The same problem can occur in binary fractions.

In order to preserve as many significant digits as possible, floating-point numbers are stored such that the leftmost digit of the fraction part is non-zero. If, after a calcula-

tion, the leftmost digit is not significant (*i.e.* it is zero), the fraction is shifted left and the exponent decreased by one until a significant digit, for binary numbers, a one, is present in the leftmost digit. A floating-point number in that form is called a **normalized** number. There are many possible unnormalized forms for a number, but only one normalized form.

Storing numbers in normalized form provides an opportunity to gain one more significant binary digit in the fraction. If the leftmost digit is known to be one, there is no need to store it; it can be assumed to be present. IEEE 754 takes advantage of this; there is an implied one bit and an implied binary point to the left of the fraction. To emphasize this difference, IEEE 754 refers to the fractional part of a floating-point number as a **significand**.

Range of Floating Point Numbers

Although the range of a single-precision floating-point number is $\pm 10^{-38}$ to $\pm 10^{38}$, it is important to remember that there are still only 2^{32} distinct values. The floating-point system can not represent every possible real number. Instead, it approximates the real numbers by a series of points. If the result of a calculation is not one of the numbers that can be represented exactly, what is stored is the nearest number that *can* be represented. This process is called **rounding**, and it introduces error in floating-point calculations. Since rounding down is as likely as rounding up, the cumulative effect of rounding error is generally negligible.

The spacing between floating-point numbers is not constant. Clearly, the difference between 0.10×2^1 and 0.11×2^1 is far less than the difference between 0.10×2^{127} and 0.11×2^{127}. If the difference between numbers is expressed as a percentage of the number, the distances are similar throughout the range, and the relative error due to rounding is about the same for small numbers as for large.

Not only cannot all real numbers be expressed exactly, there are whole ranges of numbers that cannot be represented. Consider the real number line as shown in Figure 2. The number zero can be represented exactly because it is defined by the standard. The positive numbers that can be represented fall approximately in the range 2^{-126} to 2^{+127}.

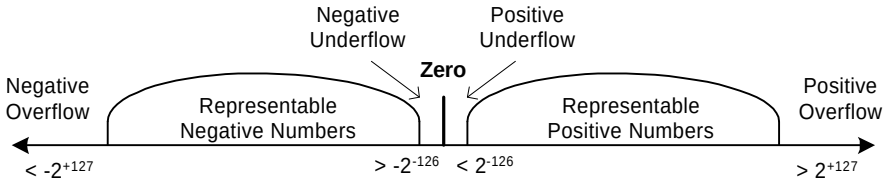

Figure 2. Zones of floating-point numbers along the real number line.

Numbers greater than 2^{+127} cannot be represented; this is called **positive overflow**. A similar range of negative numbers can be represented. Numbers to the left of that range cannot be represented; this is **negative overflow**.

There is also a range of numbers near zero that cannot be represented. The smallest positive number that can be represented in normalized form is 1.0×2^{-126}. The condition of trying to represent smaller numbers is called **positive underflow**. The same condition on the negative side of zero is called **negative underflow**. Prior to IEEE 754, manufacturers just set such results to zero, or signaled an error exception. The IEEE standard provides a more graceful way of handling such conditions: the requirement that numbers be normalized is relaxed near zero. The exponent is allowed to become zero, representing 2^{-127}, the implicit one at the left of the binary point becomes a zero, and the fraction part is allowed to have leading zeroes. Such a number approaches zero with increasing loss of significant digits.

More on IEEE 754

In addition to the single-precision floating-point numbers we have discussed, IEEE 754 specifies a double-precision number of 64 bits. The double-precision format has one bit of sign, eleven bits of exponent, and 52 bits of fraction. This gives it a range of approximately $\pm 10^{-308}$ to $\pm 10^{308}$. There is also an 80-bit extended-precision representation used mainly internally to floating-point processors.

Each format provides a way to represent special numbers in addition to the regular normalized floating-point format. An exponent of zero with a non-zero fraction is the **denormalized** form discussed above to handle positive and negative overflow. A number with both exponent and fraction of zero represents the number zero. Both positive and negative representations of zero are possible.

Positive and negative overflow are handled by providing a representation for infinity. This is a positive or negative sign bit, an exponent of all ones, and a fraction of zero. This representation is also used for the result of division by zero. Arithmetic operations on infinity behave in a predictable way.

Finally, all ones in the exponent and a non-zero fraction represents **Not a Number**, also called **NaN**.

Bibliography

Stallings, William, *Computer Organization and Architecture, Fourth Edition,* Prentice-Hall, 1996.

Tanenbaum, Andrew S., *Structured Computer Organization, Third Edition,* Prentice-Hall, 1990.

Exercises

FP-1. Rewrite the following numbers in normalized form and express the answer using decimal numbers in scientific notation. State the rule for normalizing a floating-point number.

a) 6.02257×10^{23} b) 0.0005×10^{6} c) 427×10^{0} d) 3.14159

FP-2. Add the following pairs of numbers; express the sums in normalized form using scientific notation. State the rule for addition of floating point numbers.

a) $0.137 \times 10^{2} + 0.420 \times 10^{2}$ b) $0.288 \times 10^{3} + 0.650 \times 10^{4}$

FP-3. Multiply the following pairs of numbers; express the products in normalized form. State the rule for multiplication of floating point numbers.

a) $(0.137 \times 10^{2}) \times (0.420 \times 10^{2)}$ b) $(0.288 \times 10^{3}) \times (0.650 \times 10^{4})$

FP-4. Add this pair of numbers. Normalize the result and write the fraction as a finite-precision number with four decimal digits of precision, *i.e.* four digits to the right of the decimal point. Explain why you got the result you did. What observation can you make about addition and subtraction of floating-point numbers?

$0.5000 \times 10^{8} + 0.4321 \times 10^{2}$

FP-5. Convert 0.640×10^{2} to an IEEE 754 single-precision floating point number; separate the various parts of the number with vertical lines and label them as shown in Figure 1. Show your work.

FP-6. Convert the number ¾ to an IEEE 754 single-precision floating point number; separate the various parts of the number with vertical lines and label them as shown in Figure 1. Show your work.

FP-7. Explain the concept of *underflow*.

FP-8. What is the essential idea behind scientific notation and floating-point numbers?