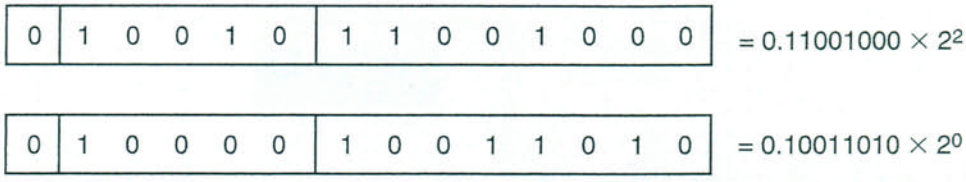
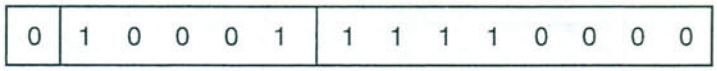
Multiplication and division are carried out using the same rules of exponents applied to decimal arithmetic, such as $2^{-3} \times 2^4 = 2^1$, for example.

EXAMPLE 2.27 Multiply:



Multiplication of 0.11001000 by 0.10011010 yields a product of 0.0111100001010000, and then multiplying by $2^2 \times 2^0 = 2^2$ yields 1.1110000101. Renormalizing and supplying the appropriate exponent, the floating-point product is:



2.5.3 Floating-Point Errors

When we use pencil and paper to solve a trigonometry problem or compute the interest on an investment, we intuitively understand that we are working in the system of real numbers. We know that this system is infinite, because given any pair of real numbers, we can always find another real number that is smaller than one and greater than the other.

Unlike the mathematics in our imaginations, computers are finite systems, with finite storage. When we call upon our computers to carry out floating-point calculations, we are modeling the infinite system of real numbers in a finite system of integers. What we have, in truth, is an *approximation* of the real number system. The more bits we use, the better the approximation. However, there is always some element of error, no matter how many bits we use.

Floating-point errors can be blatant, subtle, or unnoticed. The blatant errors, such as numeric overflow or underflow, are the ones that cause programs to crash. Subtle errors can lead to wildly erroneous results that are often hard to detect before they cause real problems. For example, in our simple model, we can express normalized numbers in the range of $-.11111111_2 \times 2^{15}$ through $+.11111111 \times 2^{15}$. Obviously, we cannot store 2^{-19} or 2^{128}; they simply don't fit. It is not quite so obvious that we cannot accurately store 128.5, which is well within our range. Converting 128.5 to binary, we have 10000000.1, which is 9 bits wide. Our significand can hold only eight. Typically, the low-order bit is dropped or rounded into the next bit. No matter how we handle it, however, we have introduced an error into our system.

We can compute the relative error in our representation by taking the ratio of the absolute value of the error to the true value of the number. Using our example of 128.5, we find:

$$\frac{128.5 - 128}{128.5} = 0.00389105 \approx 0.39\%.$$

Multiplier		Multiplicand	14-Bit Product	Real Product	Error
1000.001 (16.125)	×	0.11101000 = (0.90625)	1110.1001 (14.5625)	14.7784	1.46%
1110.1001 (14.5625)	×	0.11101000 =	1101.0011 (13.1885)	13.4483	1.94%
1101.0011 (13.1885)	×	0.11101000 =	1011.1111 (11.9375)	12.2380	2.46%
1011.1111 (11.9375)	×	0.11101000 =	1010.1101 (10.8125)	11.1366	2.91%
1010.1101 (10.8125)	×	0.11101000 =	1001.1100 (9.75)	10.1343	3.79%
1001.1100 (9.75)	×	0.11101000 =	1000.1101 (8.8125)	8.3922	4.44%

TABLE 2.3 Error Propagation in a 14-Bit Floating-Point Number

If we are not careful, such errors can propagate through a lengthy calculation, causing substantial loss of precision. Table 2.3 illustrates the error propagation as we iteratively multiply 16.24 by 0.91 using our 14-bit model. Upon converting these numbers to 8-bit binary, we see that we have a substantial error from the outset.

As you can see, in six iterations, we have more than tripled the error in the product. Continued iterations will produce an error of 100% because the product eventually goes to zero. Although this 14-bit model is so small that it exaggerates the error, all floating-point systems behave the same way. There is always some degree of error involved when representing real numbers in a finite system, no matter how large we make that system. Even the smallest error can have catastrophic results, particularly when computers are used to control physical events such as in military and medical applications. The challenge to computer scientists is to find efficient algorithms for controlling such errors within the bounds of performance and economics.

2.5.4 The IEEE-754 Floating-Point Standard

The floating-point model that we have been using in this section is designed for simplicity and conceptual understanding. We could extend this model to include whatever number of bits we wanted. Until the 1980s, these kinds of decisions were purely arbitrary, resulting in numerous incompatible representations across various manufacturers' systems. In 1985, the IEEE published a floating-point standard for both single- and double-precision floating-point numbers. This standard is officially known as IEEE-754 (1985) and includes two formats: **single precision** and **double precision**.

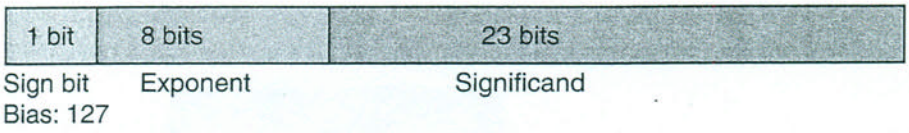


FIGURE 2.2 IEEE-754 Single Precision Floating-Point Representation

The IEEE-754 single-precision standard uses an excess 127 bias over an 8-bit exponent. The significand assumes an implied 1 *to the left* of the radix point and is 23 bits. This implied 1 is referred to as the **hidden bit** or **hidden 1**, and allows an actual significand of 23 + 1 = 24 bits. With the sign bit included, the total word size is 32 bits, as shown in Figure 2.2.

We mentioned earlier that IEEE-754 makes an exception to the rule of normalization. Because this standard assumes an implied 1 to the left of the radix point, the leading bit in the significand can indeed be zero. For example, the number $5.5 = 101.1 = .1011 \times 2^3$. IEEE-754 assumes an implied 1 to the left of the radix point and thus represents 5.5 as 1.011×2^2. Because the 1 is implied, the significand is 011 and does not begin with a 1.

Table 2.4 shows the single-precision representation of several floating-point numbers, including some special ones. One should note that zero is not directly representable in the given format, due to a required hidden bit in the significand. Therefore, zero is a special value denoted using an exponent of all zeros and a significand of all zeros. IEEE-754 does allow for both −0 and +0, although they are equal values. For this reason, programmers should use caution when comparing a floating-point value to zero.

When the exponent is 255, the quantity represented is ± infinity (which has a zero significand) or "not a number" (which has a non-zero significand). "Not a number," or NaN, is used to represent a value that is not a real number (such as

Floating Point Number	Single Precision Representation
1.0	0 01111111 00000000000000000000000
0.5	0 10000000 00000000000000000000000
19.5	0 10000011 00111000000000000000000
-3.75	1 10000000 11100000000000000000000
Zero	0 00000000 00000000000000000000000
± Infinity	0/1 11111111 00000000000000000000000
NaN	0/1 11111111 any non-zero significand
Denormalized Number	0/1 00000000 any non-zero significand

TABLE 2.4 Some Example IEEE-754 Single-Precision Floating-Point Numbers

the square root of a negative number) or as an error indicator (such as in a division by zero error).

Under the IEEE-754 standard, most numeric values are normalized and have an implicit leading 1 in their significands (that is assumed to be to the left of the radix point). Another important convention is when the exponent is all zeros but the significand is non-zero. This represents a **denormalized** number in which there is no hidden bit assumed.

The largest magnitude value we can represent (forget the sign for the time being) with the single precision floating point format is $2^{127} \times 1.11111111111111111111111_2$ (let's call this MAX). We can't use an exponent of all ones because that is reserved for NaN. The smallest magnitude number we can represent is $2^{-127} \times .00000000000000000000001_2$ (let's call this MIN). We can use an exponent of all zeros (which means the number is denormalized) since the significand is non-zero (and represents 2^{-23}). Due to the preceding special values and the limited number of bits, there are four numerical ranges that single-precision floating-point numbers cannot represent: negative numbers less than −MAX (negative overflow); negative numbers greater than −MIN (negative underflow); positive numbers less than +MIN (positive underflow); and positive numbers greater than +MAX (positive overflow).

Double-precision numbers use a signed 64-bit word consisting of an 11-bit exponent and a 52-bit significand. The bias is 1023. The range of numbers that can be represented in the IEEE double-precision model is shown in Figure 2.3. NaN is indicated when the exponent is 2047. Representations for zero and infinity correspond to the single-precision model.

At a slight cost in performance, most FPUs use only the 64-bit model so that only one set of specialized circuits needs to be designed and implemented.

Virtually every recently designed computer system has adopted the IEEE-754 floating-point model. Unfortunately, by the time this standard came along, many mainframe computer systems had established their own floating-point systems. Changing to the newer system has taken decades for well-established architectures such as IBM mainframes, which now support both their traditional floating-point system and IEEE-754. Before 1998, however, IBM systems had been using the same architecture for floating-point arithmetic that the original System/360 used in 1964. One would expect that both systems will continue to

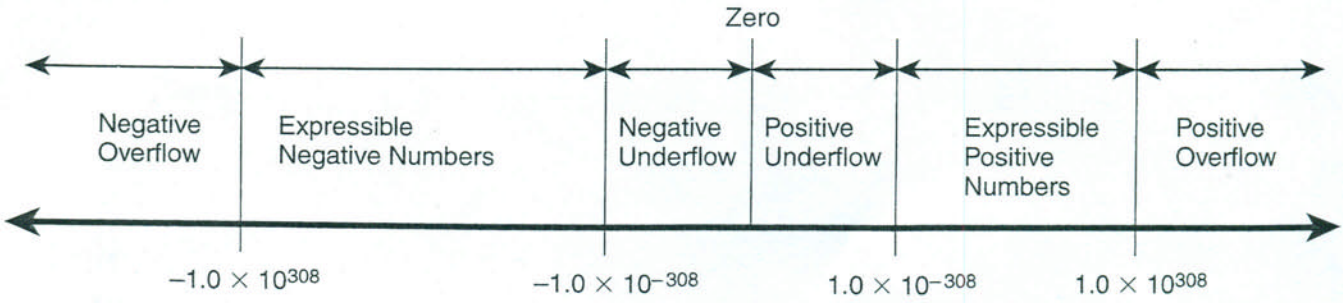


FIGURE 2.3 Range of IEEE-754 Double-Precision Numbers

be supported, owing to the substantial amount of older software that is running on these systems.

2.5.5 Range, Precision, and Accuracy

When discussing floating-point numbers it is important to understand the terms *range, precision,* and *accuracy.* Range is very straightforward, because it represents the interval from the smallest value in a given format to the largest value in that same format. For example, the range of 16-bit two's complement integers is −32768 to +32767. The range of IEEE-754 double-precision floating-point numbers is given in Figure 2.3. Even with this large range, we know there are infinitely many numbers that do not exist within the range specified by IEEE-754. The reason floating point numbers work at all is that there will always be a number in this range that is *close to* the number you want.

People have no problem understanding range, but accuracy and precision are often confused with each other. Accuracy refers to how close a number is to its true value; for example, we can't represent 0.1 in floating point, but we can find a number in the range that is relatively close, or reasonably accurate, to 0.1. Precision, on the other hand, deals with how much information we have about a value and the amount of information used to represent the value. 1.666 is a number with four decimal digits of precision; 1.6660 is the same exact number with five decimal digits of precision. The second number is not more accurate than the first.

Accuracy must be put into context—to know how accurate a value is, one must know how close it is to its intended target or "true value." We can't look at two numbers and immediately declare the first is more accurate than the second simply because the first has more digits of precision.

Although they are separate, accuracy and precision are related. Higher precision often allows a value to be more accurate, but that is not always the case. For example, we can represent the value 1 as an integer, a single-precision floating point, or a double-precision floating point, but each is equally (exactly) accurate. As another example, consider 3.13333 as an estimate for pi. It has 6 digits of precision, yet is accurate to only two digits. Adding more precision will do nothing to increase the accuracy.

On the other hand, when multiplying 0.4×0.3, our accuracy depends on our precision. If we allow only one decimal place for precision, our result is 0.1 (which is close to, but not exactly, the product). If we allow two decimal places of precision, we get 0.12, which accurately reflects the answer.

2.5.6 Additional Problems with Floating-Point Numbers

We have seen that floating-point numbers can overflow and underflow. In addition, we know that a floating-point number may not exactly represent the value we wish, as is the case with the rounding error that occurs with the binary floating-point representation for the decimal number 0.1. As we have seen, these rounding errors can propagate, resulting in substantial problems.

Although rounding is undesirable, it is understandable. In addition to this rounding problem, however, floating-point arithmetic differs from real number arithmetic in two relatively disturbing, and not necessarily intuitive, ways. First, floating-point arithmetic is not always associative. This means for three floating-point numbers a, b, and c, that

$$(a + b) + c \neq a + (b + c)$$

The same holds true for associativity under multiplication. Although in many cases the left hand side will equal the right hand side, there is no guarantee. Floating-point arithmetic is also not distributive:

$$a \times (b + c) \neq ab + ac$$

Although results can vary depending on compiler (we used Gnu C), declaring the doubles $a = 0.1$, $b = 0.2$, and $c = 0.3$ illustrates the above inequalities nicely. We encourage you to find three additional floating-point numbers to illustrate that floating-point arithmetic is neither associative nor distributive.

What does this all mean to you as a programmer? Programmers should use extra care when using the equality operator on floating point numbers. This implies that they should be avoided in controlling looping structures such as `do...while` and `for` loops. It is good practice to declare a "nearness to x" epsilon (e.g., `epsilon` = 1.0×10^{-20}) and then test an absolute value as in

```
if(abs(x) < epsilon) then...\\ It's close enough if we've defined
                            \\ epsilon correctly!
```

Floating-Point Ops or Oops?

In this chapter we have introduced floating-point numbers and the means by which computers represent them. We have touched upon floating-point rounding errors (studies in numerical analysis will provide further depth on this topic) and the fact that floating-point numbers don't obey the standard associative and distributive laws. But just how serious are these issues? To answer this question, we introduce three major floating-point blunders.

In 1994, when Intel introduced the Pentium microprocessor, number crunchers around the world noticed something weird was happening. Calculations involving double-precision divisions and certain bit patterns were producing incorrect results. Although the flawed chip was slightly inaccurate for some pairs of numbers, other instances were more extreme. For example, if $x = 4{,}195{,}835$ and $y = 3{,}145{,}727$, finding $z = x - (x/y) \times y$ *should* produce a z of 0. The Intel 286, 386, and 486 chips gave exactly that result. Even taking into account the possibility of floating-point round-off error, the value of z should have been about 9.3×10^{-10}. But on the new Pentium, z was equal to 256!

Once Intel was informed of the problem, research and testing revealed the flaw to be an omission in the chip's design. The Pentium was using the radix-4

SRT algorithm for speedy division, which necessitated a 1066-element table. Once implemented in silicon, 5 of those table entries were 0 and should have been +2.

Although the Pentium bug was a public relations debacle for Intel, it was not a catastrophe for those using the chip. In fact, it was a minor thing compared to the programming mistakes with floating-point numbers that have resulted in disasters in areas from off-shore oil drilling, to stock markets, to missile defense. The list of actual disasters that resulted from floating-point errors is very long. The following two instances are among the worst of them.

During the Persian Gulf War of 1991, the United States relied on Patriot missiles to track and intercept cruise missiles and Scud missiles. One of these missiles failed to track an incoming Scud missile, allowing the Scud to hit an American army barracks, killing 28 people and injuring many more. After an investigation, it was determined the failure of the Patriot missile was due to using too little precision to allow the missile to accurately determine the incoming Scud velocity.

The Patriot missile uses radar to determine the location of an object. If the internal weapons control computer identifies the object as something that should be intercepted, calculations are performed to predict the air space in which the object should be located at a specific time. This prediction is based on the object's known velocity and time of last detection.

The problem was in the clock, which measured time in tenths of seconds. But, the time since boot was stored as an integer number of seconds (determined by multiplying the elapsed time by 1/10). For predicting where an object would be at a specific time, the time and velocity needed to be real numbers. It was no problem to convert the integer to a real number; however, using 24-bit registers for its calculations, the Patriot was limited in the precision of this operation. The potential problem is easily seen when one realizes 1/10 in binary is:

```
0.0001100110011001100110011001100...
```

When the elapsed time was small, this "chopping error" was insignificant and caused no problems. The Patriot was designed to be on for only a few minutes at a time, so this limit of 24-bit precision would be of no consequence. The problem was that during the Gulf War, the missiles were on for days. The longer a missile was on, the larger the error became, and the more probable that the inaccuracy of the prediction calculation would cause an unsuccessful interception. And this is precisely what happened on February 25, 1991 when a failed interception resulted in 28 people dead—a failed interception due to loss of precision (required for accuracy) in floating-point numbers. It is estimated that the Patriot missile had been operational about 100 hours, introducing a rounding error in the time conversion of about 0.34 seconds, which translates to approximately half a kilometer of travel for a Scud missile.