Practice Problem 2.44

Assume we are running code on a 32-bit machine using two's-complement arithmetic for signed values. Right shifts are performed arithmetically for signed values and logically for unsigned values. The variables are declared and initialized as follows:

```
int x = foo();   /* Arbitrary value */
int y = bar();   /* Arbitrary value */

unsigned ux = x;
unsigned uy = y;
```

For each of the following C expressions, either (1) argue that it is true (evaluates to 1) for all values of `x` and `y`, or (2) give values of `x` and `y` for which it is false (evaluates to 0):

A. `(x > 0) || (x-1 < 0)`

B. `(x & 7) != 7 || (x<<29 < 0)`

C. `(x * x) >= 0`

D. `x < 0 || -x <= 0`

E. `x > 0 || -x >= 0`

F. `x+y == uy+ux`

G. `x*~y + uy*ux == -x`

2.4 Floating Point

A floating-point representation encodes rational numbers of the form $V = x \times 2^y$. It is useful for performing computations involving very large numbers ($|V| \gg 0$), numbers very close to 0 ($|V| \ll 1$), and more generally as an approximation to real arithmetic.

Up until the 1980s, every computer manufacturer devised its own conventions for how floating-point numbers were represented and the details of the operations performed on them. In addition, they often did not worry too much about the accuracy of the operations, viewing speed and ease of implementation as being more critical than numerical precision.

All of this changed around 1985 with the advent of IEEE Standard 754, a carefully crafted standard for representing floating-point numbers and the operations performed on them. This effort started in 1976 under Intel's sponsorship with the design of the 8087, a chip that provided floating-point support for the 8086 processor. They hired William Kahan, a professor at the University of California, Berkeley, as a consultant to help design a floating-point standard for its future processors. They allowed Kahan to join forces with a committee generating an industry-wide standard under the auspices of the Institute of Electrical and Electronics Engineers (IEEE). The committee ultimately adopted a standard close to

the one Kahan had devised for Intel. Nowadays, virtually all computers support what has become known as *IEEE floating point*. This has greatly improved the portability of scientific application programs across different machines.

Aside The IEEE

The Institute of Electrical and Electronic Engineers (IEEE—pronounced "Eye-Triple-Eee") is a professional society that encompasses all of electronic and computer technology. It publishes journals, sponsors conferences, and sets up committees to define standards on topics ranging from power transmission to software engineering.

In this section, we will see how numbers are represented in the IEEE floating-point format. We will also explore issues of *rounding*, when a number cannot be represented exactly in the format and hence must be adjusted upward or downward. We will then explore the mathematical properties of addition, multiplication, and relational operators. Many programmers consider floating point to be at best uninteresting and at worst arcane and incomprehensible. We will see that since the IEEE format is based on a small and consistent set of principles, it is really quite elegant and understandable.

2.4.1 Fractional Binary Numbers

A first step in understanding floating-point numbers is to consider binary numbers having fractional values. Let us first examine the more familiar decimal notation. Decimal notation uses a representation of the form $d_m d_{m-1} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots d_{-n}$, where each decimal digit d_i ranges between 0 and 9. This notation represents a value d defined as

$$d = \sum_{i=-n}^{m} 10^i \times d_i$$

The weighting of the digits is defined relative to the decimal point symbol ('.'), meaning that digits to the left are weighted by positive powers of 10, giving integral values, while digits to the right are weighted by negative powers of 10, giving fractional values. For example, 12.34_{10} represents the number $1 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} = 12\frac{34}{100}$.

By analogy, consider a notation of the form $b_m b_{m-1} \cdots b_1 b_0 . b_{-1} b_{-2} \cdots b_{-n-1} b_{-n}$, where each binary digit, or bit, b_i ranges between 0 and 1, as is illustrated in Figure 2.30. This notation represents a number b defined as

$$b = \sum_{i=-n}^{m} 2^i \times b_i \tag{2.19}$$

The symbol '.' now becomes a *binary point*, with bits on the left being weighted by positive powers of 2, and those on the right being weighted by negative powers of 2. For example, 101.11_2 represents the number $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 4 + 0 + 1 + \frac{1}{2} + \frac{1}{4} = 5\frac{3}{4}$.

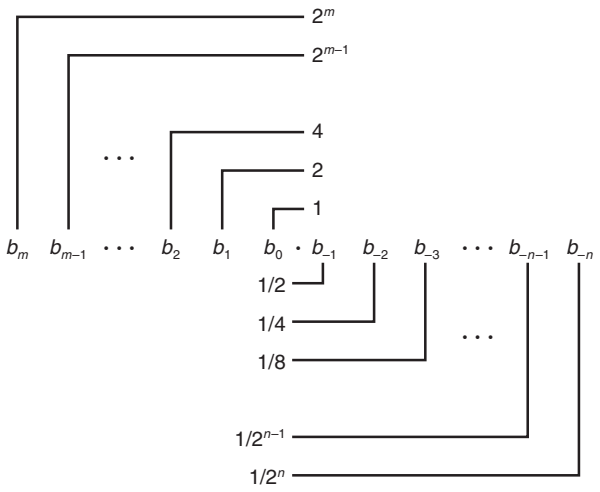


Figure 2.30
Fractional binary representation. Digits to the left of the binary point have weights of the form 2^i, while those to the right have weights of the form $1/2^i$.

One can readily see from Equation 2.19 that shifting the binary point one position to the left has the effect of dividing the number by 2. For example, while 101.11_2 represents the number $5\frac{3}{4}$, 10.111_2 represents the number $2+0+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}=2\frac{7}{8}$. Similarly, shifting the binary point one position to the right has the effect of multiplying the number by 2. For example, 1011.1_2 represents the number $8+0+2+1+\frac{1}{2}=11\frac{1}{2}$.

Note that numbers of the form $0.11\cdots1_2$ represent numbers just below 1. For example, 0.111111_2 represents $\frac{63}{64}$. We will use the shorthand notation $1.0-\epsilon$ to represent such values.

Assuming we consider only finite-length encodings, decimal notation cannot represent numbers such as $\frac{1}{3}$ and $\frac{5}{7}$ exactly. Similarly, fractional binary notation can only represent numbers that can be written $x \times 2^y$. Other values can only be approximated. For example, the number $\frac{1}{5}$ can be represented exactly as the fractional decimal number 0.20. As a fractional binary number, however, we cannot represent it exactly and instead must approximate it with increasing accuracy by lengthening the binary representation:

Representation	Value	Decimal
0.0_2	$\frac{0}{2}$	0.0_{10}
0.01_2	$\frac{1}{4}$	0.25_{10}
0.010_2	$\frac{2}{8}$	0.25_{10}
0.0011_2	$\frac{3}{16}$	0.1875_{10}
0.00110_2	$\frac{6}{32}$	0.1875_{10}
0.001101_2	$\frac{13}{64}$	0.203125_{10}
0.0011010_2	$\frac{26}{128}$	0.203125_{10}
0.00110011_2	$\frac{51}{256}$	0.19921875_{10}

Practice Problem 2.45

Fill in the missing information in the following table:

Fractional value	Binary representation	Decimal representation
$\frac{1}{8}$	0.001	0.125
$\frac{3}{4}$	_____	_____
$\frac{25}{16}$	_____	_____
_____	10.1011	_____
_____	1.001	_____
_____	_____	5.875
_____	_____	3.1875

Practice Problem 2.46

The imprecision of floating-point arithmetic can have disastrous effects. On February 25, 1991, during the first Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks and killed 28 soldiers. The U.S. General Accounting Office (GAO) conducted a detailed analysis of the failure [72] and determined that the underlying cause was an imprecision in a numeric calculation. In this exercise, you will reproduce part of the GAO's analysis.

The Patriot system contains an internal clock, implemented as a counter that is incremented every 0.1 seconds. To determine the time in seconds, the program would multiply the value of this counter by a 24-bit quantity that was a fractional binary approximation to $\frac{1}{10}$. In particular, the binary representation of $\frac{1}{10}$ is the nonterminating sequence $0.000110011[0011]\cdots_2$, where the portion in brackets is repeated indefinitely. The program approximated 0.1, as a value x, by considering just the first 23 bits of the sequence to the right of the binary point: $x = 0.00011001100110011001100$. (See Problem 2.51 for a discussion of how they could have approximated 0.1 more precisely.)

A. What is the binary representation of $0.1 - x$?

B. What is the approximate decimal value of $0.1 - x$?

C. The clock starts at 0 when the system is first powered up and keeps counting up from there. In this case, the system had been running for around 100 hours. What was the difference between the actual time and the time computed by the software?

D. The system predicts where an incoming missile will appear based on its velocity and the time of the last radar detection. Given that a Scud travels at around 2000 meters per second, how far off was its prediction?

Normally, a slight error in the absolute time reported by a clock reading would not affect a tracking computation. Instead, it should depend on the relative time between two successive readings. The problem was that the Patriot software had

been upgraded to use a more accurate function for reading time, but not all of the function calls had been replaced by the new code. As a result, the tracking software used the accurate time for one reading and the inaccurate time for the other [100].

2.4.2 IEEE Floating-Point Representation

Positional notation such as considered in the previous section would not be efficient for representing very large numbers. For example, the representation of 5×2^{100} would consist of the bit pattern 101 followed by 100 zeros. Instead, we would like to represent numbers in a form $x \times 2^y$ by giving the values of x and y.

The IEEE floating-point standard represents a number in a form $V = (-1)^s \times M \times 2^E$:

- The *sign* s determines whether the number is negative ($s = 1$) or positive ($s = 0$), where the interpretation of the sign bit for numeric value 0 is handled as a special case.
- The *significand* M is a fractional binary number that ranges either between 1 and $2 - \epsilon$ or between 0 and $1 - \epsilon$.
- The *exponent* E weights the value by a (possibly negative) power of 2.

The bit representation of a floating-point number is divided into three fields to encode these values:

- The single sign bit `s` directly encodes the sign s.
- The k-bit exponent field `exp` $= e_{k-1} \cdots e_1 e_0$ encodes the exponent E.
- The n-bit fraction field `frac` $= f_{n-1} \cdots f_1 f_0$ encodes the significand M, but the value encoded also depends on whether or not the exponent field equals 0.

Figure 2.31 shows the packing of these three fields into words for the two most common formats. In the single-precision floating-point format (a `float` in C), fields `s`, `exp`, and `frac` are 1, $k = 8$, and $n = 23$ bits each, yielding a 32-bit representation. In the double-precision floating-point format (a `double` in C), fields `s`, `exp`, and `frac` are 1, $k = 11$, and $n = 52$ bits each, yielding a 64-bit representation.

The value encoded by a given bit representation can be divided into three different cases (the latter having two variants), depending on the value of `exp`. These are illustrated in Figure 2.32 for the single-precision format.

Case 1: Normalized Values

This is the most common case. It occurs when the bit pattern of `exp` is neither all zeros (numeric value 0) nor all ones (numeric value 255 for single precision, 2047 for double). In this case, the exponent field is interpreted as representing a signed integer in *biased* form. That is, the exponent value is $E = e - Bias$ where e is the unsigned number having bit representation $e_{k-1} \cdots e_1 e_0$, and $Bias$ is a bias

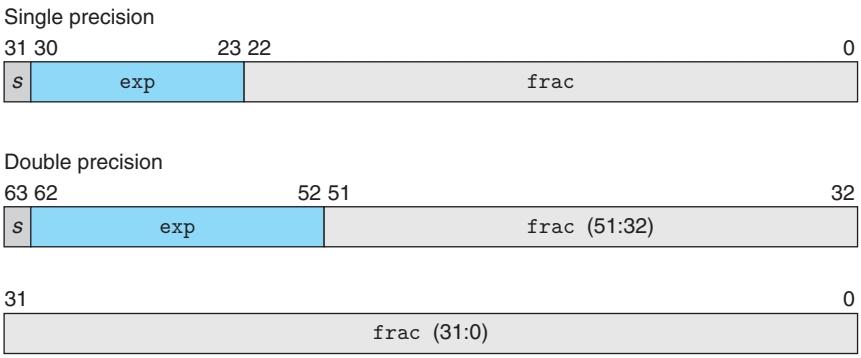


Figure 2.31 **Standard floating-point formats.** Floating-point numbers are represented by three fields. For the two most common formats, these are packed in 32-bit (single precision) or 64-bit (double precision) words.

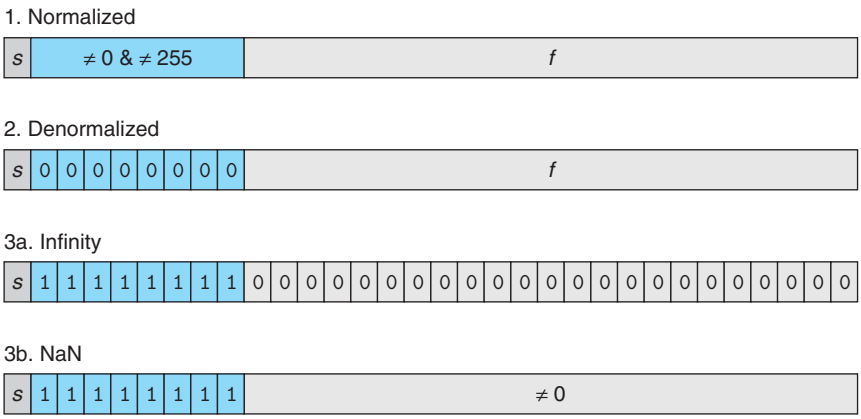


Figure 2.32 **Categories of single-precision, floating-point values.** The value of the exponent determines whether the number is (1) normalized, (2) denormalized, or a (3) special value.

value equal to $2^{k-1} - 1$ (127 for single precision and 1023 for double). This yields exponent ranges from -126 to $+127$ for single precision and -1022 to $+1023$ for double precision.

The fraction field `frac` is interpreted as representing the fractional value f, where $0 \leq f < 1$, having binary representation $0.f_{n-1} \cdots f_1 f_0$, that is, with the binary point to the left of the most significant bit. The significand is defined to be $M = 1 + f$. This is sometimes called an *implied leading 1* representation, because we can view M to be the number with binary representation $1.f_{n-1}f_{n-2} \cdots f_0$. This representation is a trick for getting an additional bit of precision for free, since we can always adjust the exponent E so that significand M is in the range $1 \leq M < 2$ (assuming there is no overflow). We therefore do not need to explicitly represent the leading bit, since it always equals 1.

Case 2: Denormalized Values

When the exponent field is all zeros, the represented number is in *denormalized* form. In this case, the exponent value is $E = 1 - Bias$, and the significand value is $M = f$, that is, the value of the fraction field without an implied leading 1.

Aside Why set the bias this way for denormalized values?

Having the exponent value be $1 - Bias$ rather than simply $-Bias$ might seem counterintuitive. We will see shortly that it provides for smooth transition from denormalized to normalized values.

Denormalized numbers serve two purposes. First, they provide a way to represent numeric value 0, since with a normalized number we must always have $M \geq 1$, and hence we cannot represent 0. In fact the floating-point representation of $+0.0$ has a bit pattern of all zeros: the sign bit is 0, the exponent field is all zeros (indicating a denormalized value), and the fraction field is all zeros, giving $M = f = 0$. Curiously, when the sign bit is 1, but the other fields are all zeros, we get the value -0.0. With IEEE floating-point format, the values -0.0 and $+0.0$ are considered different in some ways and the same in others.

A second function of denormalized numbers is to represent numbers that are very close to 0.0. They provide a property known as *gradual underflow* in which possible numeric values are spaced evenly near 0.0.

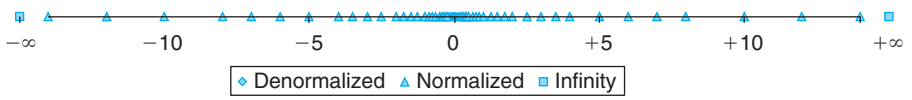
Case 3: Special Values

A final category of values occurs when the exponent field is all ones. When the fraction field is all zeros, the resulting values represent infinity, either $+\infty$ when $s = 0$, or $-\infty$ when $s = 1$. Infinity can represent results that *overflow*, as when we multiply two very large numbers, or when we divide by zero. When the fraction field is nonzero, the resulting value is called a "*NaN*," short for "Not a Number." Such values are returned as the result of an operation where the result cannot be given as a real number or as infinity, as when computing $\sqrt{-1}$ or $\infty - \infty$. They can also be useful in some applications for representing uninitialized data.

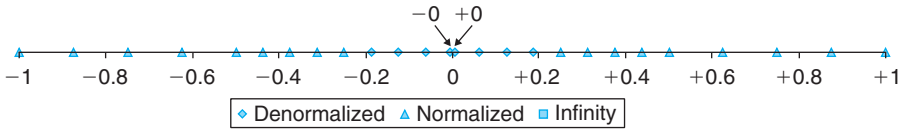
2.4.3 Example Numbers

Figure 2.33 shows the set of values that can be represented in a hypothetical 6-bit format having $k = 3$ exponent bits and $n = 2$ fraction bits. The bias is $2^{3-1} - 1 = 3$. Part A of the figure shows all representable values (other than *NaN*). The two infinities are at the extreme ends. The normalized numbers with maximum magnitude are ± 14. The denormalized numbers are clustered around 0. These can be seen more clearly in part B of the figure, where we show just the numbers between -1.0 and $+1.0$. The two zeros are special cases of denormalized numbers. Observe that the representable numbers are not uniformly distributed—they are denser nearer the origin.

Figure 2.34 shows some examples for a hypothetical 8-bit floating-point format having $k = 4$ exponent bits and $n = 3$ fraction bits. The bias is $2^{4-1} - 1 = 7$. The



(a) Complete range



(b) Values between −1.0 and +1.0

Figure 2.33 **Representable values for 6-bit floating-point format.** There are $k = 3$ exponent bits and $n = 2$ fraction bits. The bias is 3.

		Exponent			Fraction		Value		
Description	Bit representation	e	E	2^E	f	M	$2^E \times M$	V	Decimal
Zero	0 0000 000	0	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{0}{8}$	$\frac{0}{512}$	0	0.0
Smallest pos.	0 0000 001	0	−6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$	$\frac{1}{512}$	0.001953
	0 0000 010	0	−6	$\frac{1}{64}$	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$	$\frac{1}{256}$	0.003906
	0 0000 011	0	−6	$\frac{1}{64}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$	$\frac{3}{512}$	0.005859
	⋮								
Largest denorm.	0 0000 111	0	−6	$\frac{1}{64}$	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$	$\frac{7}{512}$	0.013672
Smallest norm.	0 0001 000	1	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{512}$	$\frac{1}{64}$	0.015625
	0 0001 001	1	−6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$	$\frac{9}{512}$	0.017578
	⋮								
	0 0110 110	6	−1	$\frac{1}{2}$	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$	$\frac{7}{8}$	0.875
	0 0110 111	6	−1	$\frac{1}{2}$	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$	$\frac{15}{16}$	0.9375
One	0 0111 000	7	0	1	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{8}$	1	1.0
	0 0111 001	7	0	1	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	1.125
	0 0111 010	7	0	1	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$	$\frac{5}{4}$	1.25
	⋮								
	0 1110 110	14	7	128	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{1792}{8}$	224	224.0
Largest norm.	0 1110 111	14	7	128	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{1920}{8}$	240	240.0
Infinity	0 1111 000	—	—	—	—	—	—	∞	—

Figure 2.34 **Example nonnegative values for 8-bit floating-point format.** There are $k = 4$ exponent bits and $n = 3$ fraction bits. The bias is 7.

figure is divided into three regions representing the three classes of numbers. The different columns show how the exponent field encodes the exponent E, while the fraction field encodes the significand M, and together they form the represented value $V = 2^E \times M$. Closest to 0 are the denormalized numbers, starting with 0 itself. Denormalized numbers in this format have $E = 1 - 7 = -6$, giving a weight $2^E = \frac{1}{64}$. The fractions f and significands M range over the values 0, $\frac{1}{8}, \ldots, \frac{7}{8}$, giving numbers V in the range 0 to $\frac{1}{64} \times \frac{7}{8} = \frac{7}{512}$.

The smallest normalized numbers in this format also have $E = 1 - 7 = -6$, and the fractions also range over the values 0, $\frac{1}{8}, \ldots \frac{7}{8}$. However, the significands then range from $1 + 0 = 1$ to $1 + \frac{7}{8} = \frac{15}{8}$, giving numbers V in the range $\frac{8}{512} = \frac{1}{64}$ to $\frac{15}{512}$.

Observe the smooth transition between the largest denormalized number $\frac{7}{512}$ and the smallest normalized number $\frac{8}{512}$. This smoothness is due to our definition of E for denormalized values. By making it $1 - Bias$ rather than $-Bias$, we compensate for the fact that the significand of a denormalized number does not have an implied leading 1.

As we increase the exponent, we get successively larger normalized values, passing through 1.0 and then to the largest normalized number. This number has exponent $E = 7$, giving a weight $2^E = 128$. The fraction equals $\frac{7}{8}$, giving a significand $M = \frac{15}{8}$. Thus, the numeric value is $V = 240$. Going beyond this overflows to $+\infty$.

One interesting property of this representation is that if we interpret the bit representations of the values in Figure 2.34 as unsigned integers, they occur in ascending order, as do the values they represent as floating-point numbers. This is no accident—the IEEE format was designed so that floating-point numbers could be sorted using an integer sorting routine. A minor difficulty occurs when dealing with negative numbers, since they have a leading 1, and they occur in descending order, but this can be overcome without requiring floating-point operations to perform comparisons (see Problem 2.83).

Practice Problem 2.47

Consider a 5-bit floating-point representation based on the IEEE floating-point format, with one sign bit, two exponent bits ($k = 2$), and two fraction bits ($n = 2$). The exponent bias is $2^{2-1} - 1 = 1$.

The table that follows enumerates the entire nonnegative range for this 5-bit floating-point representation. Fill in the blank table entries using the following directions:

e: The value represented by considering the exponent field to be an unsigned integer

E: The value of the exponent after biasing

2^E: The numeric weight of the exponent

f: The value of the fraction

M: The value of the significand

$2^E \times M$: The (unreduced) fractional value of the number

V: The reduced fractional value of the number

Decimal: The decimal representation of the number

Express the values of 2^E, f, M, $2^E \times M$, and V either as integers (when possible) or as fractions of the form $\frac{x}{y}$, where y is a power of 2. You need not fill in entries marked "—".

Bits	e	E	2^E	f	M	$2^E \times M$	V	Decimal
0 00 00								
0 00 01								
0 00 10								
0 00 11								
0 01 00								
0 01 01	1	0	1	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	1.25
0 01 10								
0 01 11								
0 10 00								
0 10 01								
0 10 10								
0 10 11								
0 11 00	—	—	—	—	—	—		—
0 11 01	—	—	—	—	—	—		—
0 11 10	—	—	—	—	—	—		—
0 11 11	—	—	—	—	—	—		—

Figure 2.35 shows the representations and numeric values of some important single- and double-precision floating-point numbers. As with the 8-bit format shown in Figure 2.34, we can see some general properties for a floating-point representation with a k-bit exponent and an n-bit fraction:

- The value +0.0 always has a bit representation of all zeros.
- The smallest positive denormalized value has a bit representation consisting of a 1 in the least significant bit position and otherwise all zeros. It has a fraction (and significand) value $M = f = 2^{-n}$ and an exponent value $E = -2^{k-1} + 2$. The numeric value is therefore $V = 2^{-n-2^{k-1}+2}$.
- The largest denormalized value has a bit representation consisting of an exponent field of all zeros and a fraction field of all ones. It has a fraction (and significand) value $M = f = 1 - 2^{-n}$ (which we have written $1 - \epsilon$) and an exponent value $E = -2^{k-1} + 2$. The numeric value is therefore $V = (1 - 2^{-n}) \times 2^{-2^{k-1}+2}$, which is just slightly smaller than the smallest normalized value.

Description	exp	frac	Single precision Value	Single precision Decimal	Double precision Value	Double precision Decimal
Zero	$00\cdots00$	$0\cdots00$	0	0.0	0	0.0
Smallest denorm.	$00\cdots00$	$0\cdots01$	$2^{-23}\times 2^{-126}$	1.4×10^{-45}	$2^{-52}\times 2^{-1022}$	4.9×10^{-324}
Largest denorm.	$00\cdots00$	$1\cdots11$	$(1-\epsilon)\times 2^{-126}$	1.2×10^{-38}	$(1-\epsilon)\times 2^{-1022}$	2.2×10^{-308}
Smallest norm.	$00\cdots01$	$0\cdots00$	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-308}
One	$01\cdots11$	$0\cdots00$	1×2^{0}	1.0	1×2^{0}	1.0
Largest norm.	$11\cdots10$	$1\cdots11$	$(2-\epsilon)\times 2^{127}$	3.4×10^{38}	$(2-\epsilon)\times 2^{1023}$	1.8×10^{308}

Figure 2.35 **Examples of nonnegative floating-point numbers.**

- The smallest positive normalized value has a bit representation with a 1 in the least significant bit of the exponent field and otherwise all zeros. It has a significand value $M = 1$ and an exponent value $E = -2^{k-1} + 2$. The numeric value is therefore $V = 2^{-2^{k-1}+2}$.
- The value 1.0 has a bit representation with all but the most significant bit of the exponent field equal to 1 and all other bits equal to 0. Its significand value is $M = 1$ and its exponent value is $E = 0$.
- The largest normalized value has a bit representation with a sign bit of 0, the least significant bit of the exponent equal to 0, and all other bits equal to 1. It has a fraction value of $f = 1 - 2^{-n}$, giving a significand $M = 2 - 2^{-n}$ (which we have written $2 - \epsilon$). It has an exponent value $E = 2^{k-1} - 1$, giving a numeric value $V = (2 - 2^{-n}) \times 2^{2^{k-1}-1} = (1 - 2^{-n-1}) \times 2^{2^{k-1}}$.

One useful exercise for understanding floating-point representations is to convert sample integer values into floating-point form. For example, we saw in Figure 2.14 that 12,345 has binary representation [11000000111001]. We create a normalized representation of this by shifting 13 positions to the right of a binary point, giving $12345 = 1.1000000111001_2 \times 2^{13}$. To encode this in IEEE single-precision format, we construct the fraction field by dropping the leading 1 and adding 10 zeros to the end, giving binary representation [10000001110010000000000]. To construct the exponent field, we add bias 127 to 13, giving 140, which has binary representation [10001100]. We combine this with a sign bit of 0 to get the floating-point representation in binary of [01000110010000001110010000000000]. Recall from Section 2.1.4 that we observed the following correlation in the bit-level representations of the integer value `12345` (`0x3039`) and the single-precision floating-point value `12345.0` (`0x4640E400`):

```
   0    0    0    0    3    0    3    9
00000000000000000011000000111001
                  *************
             4    6    4    0    E    4    0    0
01000110010000001110010000000000
```

We can now see that the region of correlation corresponds to the low-order bits of the integer, stopping just before the most significant bit equal to 1 (this bit forms the implied leading 1), matching the high-order bits in the fraction part of the floating-point representation.

Practice Problem 2.48

As mentioned in Problem 2.6, the integer 3,510,593 has hexadecimal representation `0x00359141`, while the single-precision, floating-point number 3510593.0 has hexadecimal representation `0x4A564504`. Derive this floating-point representation and explain the correlation between the bits of the integer and floating-point representations.

Practice Problem 2.49

A. For a floating-point format with an n-bit fraction, give a formula for the smallest positive integer that cannot be represented exactly (because it would require an $n+1$-bit fraction to be exact). Assume the exponent field size k is large enough that the range of representable exponents does not provide a limitation for this problem.

B. What is the numeric value of this integer for single-precision format ($n = 23$)?

2.4.4 Rounding

Floating-point arithmetic can only approximate real arithmetic, since the representation has limited range and precision. Thus, for a value x, we generally want a systematic method of finding the "closest" matching value x' that can be represented in the desired floating-point format. This is the task of the *rounding* operation. One key problem is to define the direction to round a value that is halfway between two possibilities. For example, if I have \$1.50 and want to round it to the nearest dollar, should the result be \$1 or \$2? An alternative approach is to maintain a lower and an upper bound on the actual number. For example, we could determine representable values x^- and x^+ such that the value x is guaranteed to lie between them: $x^- \leq x \leq x^+$. The IEEE floating-point format defines four different *rounding modes*. The default method finds a closest match, while the other three can be used for computing upper and lower bounds.

Figure 2.36 illustrates the four rounding modes applied to the problem of rounding a monetary amount to the nearest whole dollar. Round-to-even (also called round-to-nearest) is the default mode. It attempts to find a closest match. Thus, it rounds \$1.40 to \$1 and \$1.60 to \$2, since these are the closest whole dollar values. The only design decision is to determine the effect of rounding values that are halfway between two possible results. Round-to-even mode adopts the

Mode	$1.40	$1.60	$1.50	$2.50	$−1.50
Round-to-even	$1	$2	$2	$2	$−2
Round-toward-zero	$1	$1	$1	$2	$−1
Round-down	$1	$1	$1	$2	$−2
Round-up	$2	$2	$2	$3	$−1

Figure 2.36 **Illustration of rounding modes for dollar rounding.** The first rounds to a nearest value, while the other three bound the result above or below.

convention that it rounds the number either upward or downward such that the least significant digit of the result is even. Thus, it rounds both $1.50 and $2.50 to $2.

The other three modes produce guaranteed bounds on the actual value. These can be useful in some numerical applications. Round-toward-zero mode rounds positive numbers downward and negative numbers upward, giving a value $\hat{x}$ such that $|\hat{x}| \leq |x|$. Round-down mode rounds both positive and negative numbers downward, giving a value x^- such that $x^- \leq x$. Round-up mode rounds both positive and negative numbers upward, giving a value x^+ such that $x \leq x^+$.

Round-to-even at first seems like it has a rather arbitrary goal—why is there any reason to prefer even numbers? Why not consistently round values halfway between two representable values upward? The problem with such a convention is that one can easily imagine scenarios in which rounding a set of data values would then introduce a statistical bias into the computation of an average of the values. The average of a set of numbers that we rounded by this means would be slightly higher than the average of the numbers themselves. Conversely, if we always rounded numbers halfway between downward, the average of a set of rounded numbers would be slightly lower than the average of the numbers themselves. Rounding toward even numbers avoids this statistical bias in most real-life situations. It will round upward about 50% of the time and round downward about 50% of the time.

Round-to-even rounding can be applied even when we are not rounding to a whole number. We simply consider whether the least significant digit is even or odd. For example, suppose we want to round decimal numbers to the nearest hundredth. We would round 1.2349999 to 1.23 and 1.2350001 to 1.24, regardless of rounding mode, since they are not halfway between 1.23 and 1.24. On the other hand, we would round both 1.2350000 and 1.2450000 to 1.24, since 4 is even.

Similarly, round-to-even rounding can be applied to binary fractional numbers. We consider least significant bit value 0 to be even and 1 to be odd. In general, the rounding mode is only significant when we have a bit pattern of the form $XX\cdots X.YY\cdots Y100\cdots$, where X and Y denote arbitrary bit values with the rightmost Y being the position to which we wish to round. Only bit patterns of this form denote values that are halfway between two possible results. As examples, consider the problem of rounding values to the nearest quarter (i.e., 2 bits to the right of the binary point). We would round 10.00011_2 ($2\frac{3}{32}$) down to 10.00_2 (2),

and 10.00110_2 ($2\frac{3}{16}$) up to 10.01_2 ($2\frac{1}{4}$), because these values are not halfway between two possible values. We would round 10.11100_2 ($2\frac{7}{8}$) up to 11.00_2 (3) and 10.10100_2 ($2\frac{5}{8}$) down to 10.10_2 ($2\frac{1}{2}$), since these values are halfway between two possible results, and we prefer to have the least significant bit equal to zero.

Practice Problem 2.50

Show how the following binary fractional values would be rounded to the nearest half (1 bit to the right of the binary point), according to the round-to-even rule. In each case, show the numeric values, both before and after rounding.

A. 10.010_2

B. 10.011_2

C. 10.110_2

D. 11.001_2

Practice Problem 2.51

We saw in Problem 2.46 that the Patriot missile software approximated 0.1 as $x = 0.00011001100110011001100_2$. Suppose instead that they had used IEEE round-to-even mode to determine an approximation x' to 0.1 with 23 bits to the right of the binary point.

A. What is the binary representation of x'?

B. What is the approximate decimal value of $x' - 0.1$?

C. How far off would the computed clock have been after 100 hours of operation?

D. How far off would the program's prediction of the position of the Scud missile have been?

Practice Problem 2.52

Consider the following two 7-bit floating-point representations based on the IEEE floating point format. Neither has a sign bit—they can only represent nonnegative numbers.

1. Format A
   - There are $k = 3$ exponent bits. The exponent bias is 3.
   - There are $n = 4$ fraction bits.
2. Format B
   - There are $k = 4$ exponent bits. The exponent bias is 7.
   - There are $n = 3$ fraction bits.

Below, you are given some bit patterns in Format A, and your task is to convert them to the closest value in Format B. If necessary, you should apply the round-to-even rounding rule. In addition, give the values of numbers given by the Format A

and Format B bit patterns. Give these as whole numbers (e.g., 17) or as fractions (e.g., 17/64).

Format A		Format B	
Bits	Value	Bits	Value
011 0000	1	0111 000	1
101 1110			
010 1001			
110 1111			
000 0001			

2.4.5 Floating-Point Operations

The IEEE standard specifies a simple rule for determining the result of an arithmetic operation such as addition or multiplication. Viewing floating-point values x and y as real numbers, and some operation $\odot$ defined over real numbers, the computation should yield $Round(x \odot y)$, the result of applying rounding to the exact result of the real operation. In practice, there are clever tricks floating-point unit designers use to avoid performing this exact computation, since the computation need only be sufficiently precise to guarantee a correctly rounded result. When one of the arguments is a special value such as -0, ∞, or NaN, the standard specifies conventions that attempt to be reasonable. For example, $1/-0$ is defined to yield $-\infty$, while $1/+0$ is defined to yield $+\infty$.

One strength of the IEEE standard's method of specifying the behavior of floating-point operations is that it is independent of any particular hardware or software realization. Thus, we can examine its abstract mathematical properties without considering how it is actually implemented.

We saw earlier that integer addition, both unsigned and two's complement, forms an abelian group. Addition over real numbers also forms an abelian group, but we must consider what effect rounding has on these properties. Let us define $x +^f y$ to be $Round(x + y)$. This operation is defined for all values of x and y, although it may yield infinity even when both x and y are real numbers due to overflow. The operation is commutative, with $x +^f y = y +^f x$ for all values of x and y. On the other hand, the operation is not associative. For example, with single-precision floating point the expression `(3.14+1e10)-1e10` evaluates to `0.0`—the value 3.14 is lost due to rounding. On the other hand, the expression `3.14+(1e10-1e10)` evaluates to `3.14`. As with an abelian group, most values have inverses under floating-point addition, that is, $x +^f -x = 0$. The exceptions are infinities (since $+\infty - \infty = NaN$), and NaN's, since $NaN +^f x = NaN$ for any x.

The lack of associativity in floating-point addition is the most important group property that is lacking. It has important implications for scientific programmers and compiler writers. For example, suppose a compiler is given the following code fragment:

```
x = a + b + c;
y = b + c + d;
```

The compiler might be tempted to save one floating-point addition by generating the following code:

```
t = b + c;
x = a + t;
y = t + d;
```

However, this computation might yield a different value for `x` than would the original, since it uses a different association of the addition operations. In most applications, the difference would be so small as to be inconsequential. Unfortunately, compilers have no way of knowing what trade-offs the user is willing to make between efficiency and faithfulness to the exact behavior of the original program. As a result, they tend to be very conservative, avoiding any optimizations that could have even the slightest effect on functionality.

On the other hand, floating-point addition satisfies the following monotonicity property: if $a \geq b$ then $x + a \geq x + b$ for any values of a, b, and x other than NaN. This property of real (and integer) addition is not obeyed by unsigned or two's-complement addition.

Floating-point multiplication also obeys many of the properties one normally associates with multiplication. Let us define $x *^{\mathrm{f}} y$ to be $Round(x \times y)$. This operation is closed under multiplication (although possibly yielding infinity or NaN), it is commutative, and it has 1.0 as a multiplicative identity. On the other hand, it is not associative, due to the possibility of overflow or the loss of precision due to rounding. For example, with single-precision floating point, the expression `(1e20*1e20)*1e-20` evaluates to $+\infty$, while `1e20*(1e20*1e-20)` evaluates to `1e20`. In addition, floating-point multiplication does not distribute over addition. For example, with single-precision floating point, the expression `1e20*(1e20-1e20)` evaluates to `0.0`, while `1e20*1e20-1e20*1e20` evaluates to `NaN`.

On the other hand, floating-point multiplication satisfies the following monotonicity properties for any values of a, b, and c other than NaN:

$$a \geq b \text{ and } c \geq 0 \Rightarrow a *^{\mathrm{f}} c \geq b *^{\mathrm{f}} c$$

$$a \geq b \text{ and } c \leq 0 \Rightarrow a *^{\mathrm{f}} c \leq b *^{\mathrm{f}} c$$

In addition, we are also guaranteed that $a *^{\mathrm{f}} a \geq 0$, as long as $a \neq NaN$. As we saw earlier, none of these monotonicity properties hold for unsigned or two's-complement multiplication.

This lack of associativity and distributivity is of serious concern to scientific programmers and to compiler writers. Even such a seemingly simple task as writing code to determine whether two lines intersect in 3-dimensional space can be a major challenge.

2.4.6 Floating Point in C

All versions of C provide two different floating-point data types: `float` and `double`. On machines that support IEEE floating point, these data types correspond to single- and double-precision floating point. In addition, the machines use

the round-to-even rounding mode. Unfortunately, since the C standards do not require the machine to use IEEE floating point, there are no standard methods to change the rounding mode or to get special values such as -0, $+\infty$, $-\infty$, or *NaN*. Most systems provide a combination of include ('`.h`') files and procedure libraries to provide access to these features, but the details vary from one system to another. For example, the GNU compiler GCC defines program constants `INFINITY` (for $+\infty$) and `NAN` (for *NaN*) when the following sequence occurs in the program file:

```
#define _GNU_SOURCE 1
#include <math.h>
```

More recent versions of C, including ISO C99, include a third floating-point data type, `long double`. For many machines and compilers, this data type is equivalent to the `double` data type. For Intel-compatible machines, however, GCC implements this data type using an 80-bit "extended precision" format, providing a much larger range and precision than does the standard 64-bit format. The properties of this format are investigated in Problem 2.85.

Practice Problem 2.53

Fill in the following macro definitions to generate the double-precision values $+\infty$, $-\infty$, and 0:

```
#define POS_INFINITY
#define NEG_INFINITY
#define NEG_ZERO
```

You cannot use any include files (such as `math.h`), but you can make use of the fact that the largest finite number that can be represented with double precision is around 1.8×10^{308}.

When casting values between `int`, `float`, and `double` formats, the program changes the numeric values and the bit representations as follows (assuming a 32-bit `int`):

- From `int` to `float`, the number cannot overflow, but it may be rounded.
- From `int` or `float` to `double`, the exact numeric value can be preserved because `double` has both greater range (i.e., the range of representable values), as well as greater precision (i.e., the number of significant bits).
- From `double` to `float`, the value can overflow to $+\infty$ or $-\infty$, since the range is smaller. Otherwise, it may be rounded, because the precision is smaller.
- From `float` or `double` to `int` the value will be rounded toward zero. For example, 1.999 will be converted to 1, while -1.999 will be converted to -1. Furthermore, the value may overflow. The C standards do not specify a fixed result for this case. Intel-compatible microprocessors designate the

bit pattern $[10\cdots00]$ ($TMin_w$ for word size w) as an *integer indefinite* value. Any conversion from floating point to integer that cannot assign a reasonable integer approximation yields this value. Thus, the expression `(int) +1e10` yields `-21483648`, generating a negative value from a positive one.

Web Aside DATA:IA32-FP Intel IA32 floating-point arithmetic

In the next chapter, we will begin an in-depth study of Intel IA32 processors, the processor found in many of today's personal computers. Here we highlight an idiosyncrasy of these machines that can seriously affect the behavior of programs operating on floating-point numbers when compiled with GCC.

IA32 processors, like most other processors, have special memory elements called *registers* for holding floating-point values as they are being computed and used. The unusual feature of IA32 is that the floating-point registers use a special 80-bit *extended-precision* format to provide a greater range and precision than the normal 32-bit single-precision and 64-bit double-precision formats used for values held in memory. (See Problem 2.85.) All single- and double-precision numbers are converted to this format as they are loaded from memory into floating-point registers. The arithmetic is always performed in extended precision. Numbers are converted from extended precision to single- or double-precision format as they are stored in memory.

This extension to 80 bits for all register data and then contraction to a smaller format for memory data has some undesirable consequences for programmers. It means that storing a number from a register to memory and then retrieving it back into the register can cause it to change, due to rounding, underflow, or overflow. This storing and retrieving is not always visible to the C programmer, leading to some very peculiar results.

More recent versions of Intel processors, including both IA32 and newer 64-bit machines, provide direct hardware support for single- and double-precision floating-point operations. The peculiarities of the historic IA32 approach will diminish in importance with new hardware and with compilers that generate code based on the newer floating-point instructions.

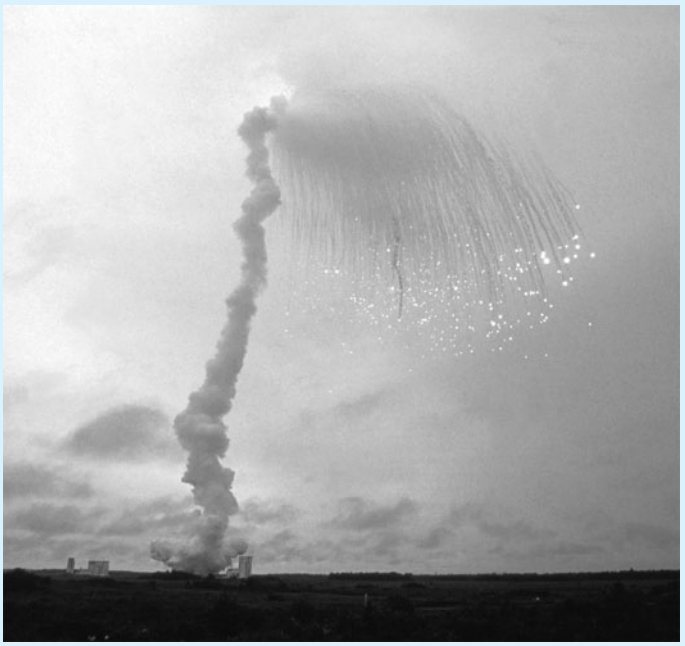
Aside Ariane 5: the high cost of floating-point overflow

Converting large floating-point numbers to integers is a common source of programming errors. Such an error had disastrous consequences for the maiden voyage of the Ariane 5 rocket, on June 4, 1996. Just 37 seconds after liftoff, the rocket veered off its flight path, broke up, and exploded. Communication satellites valued at $500 million were on board the rocket.

A later investigation [69, 39] showed that the computer controlling the inertial navigation system had sent invalid data to the computer controlling the engine nozzles. Instead of sending flight control information, it had sent a diagnostic bit pattern indicating that an overflow had occurred during the conversion of a 64-bit floating-point number to a 16-bit signed integer.

The value that overflowed measured the horizontal velocity of the rocket, which could be more than 5 times higher than that achieved by the earlier Ariane 4 rocket. In the design of the Ariane 4 software, they had carefully analyzed the numeric values and determined that the horizontal velocity

would never overflow a 16-bit number. Unfortunately, they simply reused this part of the software in the Ariane 5 without checking the assumptions on which it had been based.



© Fourmy/REA/SABA/Corbis

Practice Problem 2.54

Assume variables `x`, `f`, and `d` are of type `int`, `float`, and `double`, respectively. Their values are arbitrary, except that neither `f` nor `d` equals $+\infty$, $-\infty$, or *NaN*. For each of the following C expressions, either argue that it will always be true (i.e., evaluate to 1) or give a value for the variables such that it is not true (i.e., evaluates to 0).

A. `x == (int)(double) x`

B. `x == (int)(float) x`

C. `d == (double)(float) d`

D. `f == (float)(double) f`

E. `f == -(-f)`

F. `1.0/2 == 1/2.0`

G. `d*d >= 0.0`

H. `(f+d)-f == d`

2.5 Summary

Computers encode information as bits, generally organized as sequences of bytes. Different encodings are used for representing integers, real numbers, and character strings. Different models of computers use different conventions for encoding numbers and for ordering the bytes within multi-byte data.

The C language is designed to accommodate a wide range of different implementations in terms of word sizes and numeric encodings. Most current machines have 32-bit word sizes, although high-end machines increasingly have 64-bit words. Most machines use two's-complement encoding of integers and IEEE encoding of floating point. Understanding these encodings at the bit level, as well as understanding the mathematical characteristics of the arithmetic operations, is important for writing programs that operate correctly over the full range of numeric values.

When casting between signed and unsigned integers of the same size, most C implementations follow the convention that the underlying bit pattern does not change. On a two's-complement machine, this behavior is characterized by functions $T2U_w$ and $U2T_w$, for a w-bit value. The implicit casting of C gives results that many programmers do not anticipate, often leading to program bugs.

Due to the finite lengths of the encodings, computer arithmetic has properties quite different from conventional integer and real arithmetic. The finite length can cause numbers to overflow, when they exceed the range of the representation. Floating-point values can also underflow, when they are so close to 0.0 that they are changed to zero.

The finite integer arithmetic implemented by C, as well as most other programming languages, has some peculiar properties compared to true integer arithmetic. For example, the expression `x*x` can evaluate to a negative number due to overflow. Nonetheless, both unsigned and two's-complement arithmetic satisfy many of the other properties of integer arithmetic, including associativity, commutativity, and distributivity. This allows compilers to do many optimizations. For example, in replacing the expression `7*x` by `(x<<3)-x`, we make use of the associative, commutative, and distributive properties, along with the relationship between shifting and multiplying by powers of 2.

We have seen several clever ways to exploit combinations of bit-level operations and arithmetic operations. For example, we saw that with two's-complement arithmetic `~x+1` is equivalent to `-x`. As another example, suppose we want a bit pattern of the form $[0, \ldots, 0, 1, \ldots, 1]$, consisting of $w - k$ zeros followed by k ones. Such bit patterns are useful for masking operations. This pattern can be generated by the C expression `(1<<k)-1`, exploiting the property that the desired bit pattern has numeric value $2^k - 1$. For example, the expression `(1<<8)-1` will generate the bit pattern `0xFF`.

Floating-point representations approximate real numbers by encoding numbers of the form $x \times 2^y$. The most common floating-point representation is defined by IEEE Standard 754. It provides for several different precisions, with the most common being single (32 bits) and double (64 bits). IEEE floating point also has representations for special values representing plus and minus infinity, as well as not-a-number.

Floating-point arithmetic must be used very carefully, because it has only limited range and precision, and because it does not obey common mathematical properties such as associativity.

Bibliographic Notes

Reference books on C [48, 58] discuss properties of the different data types and operations. (Of these two, only Steele and Harbison [48] cover the newer features found in ISO C99.) The C standards do not specify details such as precise word sizes or numeric encodings. Such details are intentionally omitted to make it possible to implement C on a wide range of different machines. Several books have been written giving advice to C programmers [59, 70] that warn about problems with overflow, implicit casting to unsigned, and some of the other pitfalls we have covered in this chapter. These books also provide helpful advice on variable naming, coding styles, and code testing. Seacord's book on security issues in C and C++ programs [94], combines information about C programs, how they are compiled and executed, and how vulnerabilities may arise. Books on Java (we recommend the one coauthored by James Gosling, the creator of the language [4]) describe the data formats and arithmetic operations supported by Java.

Most books on logic design [56, 115] have a section on encodings and arithmetic operations. Such books describe different ways of implementing arithmetic circuits. Overton's book on IEEE floating point [78] provides a detailed description of the format as well as the properties from the perspective of a numerical applications programmer.

Homework Problems

2.55 ◆

Compile and run the sample code that uses `show_bytes` (file `show-bytes.c`) on different machines to which you have access. Determine the byte orderings used by these machines.

2.56 ◆

Try running the code for `show_bytes` for different sample values.

2.57 ◆

Write procedures `show_short`, `show_long`, and `show_double` that print the byte representations of C objects of types `short int`, `long int`, and `double`, respectively. Try these out on several machines.

2.58 ◆◆

Write a procedure `is_little_endian` that will return 1 when compiled and run on a little-endian machine, and will return 0 when compiled and run on a big-endian machine. This program should run on any machine, regardless of its word size.

2.59 ◆◆

Write a C expression that will yield a word consisting of the least significant byte of `x`, and the remaining bytes of `y`. For operands `x` = `0x89ABCDEF` and `y` = `0x76543210`, this would give `0x765432EF`.

2.60 ◆◆

Suppose we number the bytes in a w-bit word from 0 (least significant) to $w/8 - 1$ (most significant). Write code for the following C function, which will return an unsigned value in which byte `i` of argument `x` has been replaced by byte `b`:

```
unsigned replace_byte (unsigned x, int i, unsigned char b);
```

Here are some examples showing how the function should work:

```
replace_byte(0x12345678, 2, 0xAB) --> 0x12AB5678
replace_byte(0x12345678, 0, 0xAB) --> 0x123456AB
```

Bit-level integer coding rules

In several of the following problems, we will artificially restrict what programming constructs you can use to help you gain a better understanding of the bit-level, logic, and arithmetic operations of C. In answering these problems, your code must follow these rules:

- Assumptions
  - Integers are represented in two's-complement form.
  - Right shifts of signed data are performed arithmetically.
  - Data type `int` is w bits long. For some of the problems, you will be given a specific value for w, but otherwise your code should work as long as w is a multiple of 8. You can use the expression `sizeof(int)<<3` to compute w.
- Forbidden
  - Conditionals (`if` or `?:`), loops, switch statements, function calls, and macro invocations.
  - Division, modulus, and multiplication.
  - Relative comparison operators (<, >, <=, and >=).
  - Casting, either explicit or implicit.
- Allowed operations
  - All bit-level and logic operations.
  - Left and right shifts, but only with shift amounts between 0 and $w - 1$.
  - Addition and subtraction.
  - Equality (==) and inequality (!=) tests. (Some of the problems do not allow these.)
  - Integer constants `INT_MIN` and `INT_MAX`.

Even with these rules, you should try to make your code readable by choosing descriptive variable names and using comments to describe the logic behind your solutions. As an example, the following code extracts the most significant byte from integer argument `x`:

```
/* Get most significant byte from x */
int get_msb(int x) {
    /* Shift by w-8 */
    int shift_val = (sizeof(int)-1)<<3;
    /* Arithmetic shift */
    int xright = x >> shift_val;
    /* Zero all but LSB */
    return xright & 0xFF;
}
```

2.61 ◆◆

Write C expressions that evaluate to 1 when the following conditions are true, and to 0 when they are false. Assume `x` is of type `int`.

A. Any bit of `x` equals 1.

B. Any bit of `x` equals 0.

C. Any bit in the least significant byte of `x` equals 1.

D. Any bit in the most significant byte of `x` equals 0.

Your code should follow the bit-level integer coding rules (page 120), with the additional restriction that you may not use equality (`==`) or inequality (`!=`) tests.

2.62 ◆◆◆

Write a function `int_shifts_are_arithmetic()` that yields 1 when run on a machine that uses arithmetic right shifts for `int`'s, and 0 otherwise. Your code should work on a machine with any word size. Test your code on several machines.

2.63 ◆◆◆

Fill in code for the following C functions. Function `srl` performs a logical right shift using an arithmetic right shift (given by value `xsra`), followed by other operations not including right shifts or division. Function `sra` performs an arithmetic right shift using a logical right shift (given by value `xsrl`), followed by other operations not including right shifts or division. You may use the computation `8*sizeof(int)` to determine w, the number of bits in data type `int`. The shift amount `k` can range from 0 to $w - 1$.

```
unsigned srl(unsigned x, int k) {
    /* Perform shift arithmetically */
    unsigned xsra = (int) x >> k;
    .
    .
    .

}
```

```
int sra(int x, int k) {
    /* Perform shift logically */
    int xsrl = (unsigned) x >> k;
    .
    .
    .

}
```

2.64 ◆

Write code to implement the following function:

```
/* Return 1 when any odd bit of x equals 1; 0 otherwise.
   Assume w=32. */
int any_odd_one(unsigned x);
```

Your function should follow the bit-level integer coding rules (page 120), except that you may assume that data type `int` has $w = 32$ bits.

2.65 ◆◆◆◆

Write code to implement the following function:

```
/* Return 1 when x contains an odd number of 1s; 0 otherwise.
   Assume w=32. */
int odd_ones(unsigned x);
```

Your function should follow the bit-level integer coding rules (page 120), except that you may assume that data type `int` has $w = 32$ bits.

Your code should contain a total of at most 12 arithmetic, bit-wise, and logical operations.

2.66 ◆◆◆

Write code to implement the following function:

```
/*
 * Generate mask indicating leftmost 1 in x.  Assume w=32.
 * For example 0xFF00 -> 0x8000, and 0x6600 --> 0x4000.
 * If x = 0, then return 0.
 */
int leftmost_one(unsigned x);
```

Your function should follow the bit-level integer coding rules (page 120), except that you may assume that data type `int` has $w = 32$ bits.

Your code should contain a total of at most 15 arithmetic, bit-wise, and logical operations.

Hint: First transform `x` into a bit vector of the form $[0 \cdots 011 \cdots 1]$.

2.67 ◆◆

You are given the task of writing a procedure `int_size_is_32()` that yields 1 when run on a machine for which an `int` is 32 bits, and yields 0 otherwise. You are not allowed to use the `sizeof` operator. Here is a first attempt:

```
1   /* The following code does not run properly on some machines */
2   int bad_int_size_is_32() {
3       /* Set most significant bit (msb) of 32-bit machine */
4       int set_msb = 1 << 31;
5       /* Shift past msb of 32-bit word */
6       int beyond_msb = 1 << 32;
7
8       /* set_msb is nonzero when word size >= 32
9          beyond_msb is zero when word size <= 32  */
10      return set_msb && !beyond_msb;
11  }
```

When compiled and run on a 32-bit SUN SPARC, however, this procedure returns 0. The following compiler message gives us an indication of the problem:

```
warning: left shift count >= width of type
```

A. In what way does our code fail to comply with the C standard?

B. Modify the code to run properly on any machine for which data type `int` is at least 32 bits.

C. Modify the code to run properly on any machine for which data type `int` is at least 16 bits.

2.68 ◆◆

Write code for a function with the following prototype:

```
/*
 * Mask with least signficant n bits set to 1
 * Examples: n = 6 --> 0x2F, n = 17 --> 0x1FFFF
 * Assume 1 <= n <= w
 */
int lower_one_mask(int n);
```

Your function should follow the bit-level integer coding rules (page 120). Be careful of the case $n = w$.

2.69 ◆◆◆

Write code for a function with the following prototype:

```
/*
 * Do rotating left shift.  Assume 0 <= n < w
 * Examples when x = 0x12345678 and w = 32:
 *    n=4 -> 0x23456781, n=20 -> 0x67812345
 */
unsigned rotate_left(unsigned x, int n);
```

Your function should follow the bit-level integer coding rules (page 120). Be careful of the case $n = 0$.

2.70 ◆◆

Write code for the function with the following prototype:

```
/*
 * Return 1 when x can be represented as an n-bit, 2's complement
 * number; 0 otherwise
 * Assume 1 <= n <= w
 */
int fits_bits(int x, int n);
```

Your function should follow the bit-level integer coding rules (page 120).

2.71 ◆

You just started working for a company that is implementing a set of procedures to operate on a data structure where 4 signed bytes are packed into a 32-bit `unsigned`. Bytes within the word are numbered from 0 (least significant) to 3 (most significant). You have been assigned the task of implementing a function for a machine using two's-complement arithmetic and arithmetic right shifts with the following prototype:

```
/* Declaration of data type where 4 bytes are packed
   into an unsigned */
typedef unsigned packed_t;

/* Extract byte from word.  Return as signed integer */
int xbyte(packed_t word, int bytenum);
```

That is, the function will extract the designated byte and sign extend it to be a 32-bit `int`.

Your predecessor (who was fired for incompetence) wrote the following code:

```
/* Failed attempt at xbyte */
int xbyte(packed_t word, int bytenum)
{
    return (word >> (bytenum << 3)) & 0xFF;
}
```

A. What is wrong with this code?

B. Give a correct implementation of the function that uses only left and right shifts, along with one subtraction.

2.72 ◆◆

You are given the task of writing a function that will copy an integer `val` into a buffer `buf`, but it should do so only if enough space is available in the buffer.

Here is the code you write:

```
/* Copy integer into buffer if space is available */
/* WARNING: The following code is buggy */
```

```
void copy_int(int val, void *buf, int maxbytes) {
    if (maxbytes-sizeof(val) >= 0)
            memcpy(buf, (void *) &val, sizeof(val));
}
```

This code makes use of the library function `memcpy`. Although its use is a bit artificial here, where we simply want to copy an `int`, it illustrates an approach commonly used to copy larger data structures.

You carefully test the code and discover that it *always* copies the value to the buffer, even when `maxbytes` is too small.

A. Explain why the conditional test in the code always succeeds. **Hint:** The `sizeof` operator returns a value of type `size_t`.

B. Show how you can rewrite the conditional test to make it work properly.

2.73 ◆◆

Write code for a function with the following prototype:

```
/* Addition that saturates to TMin or TMax */
int saturating_add(int x, int y);
```

Instead of overflowing the way normal two's-complement addition does, saturating addition returns *TMax* when there would be positive overflow, and *TMin* when there would be negative overflow. Saturating arithmetic is commonly used in programs that perform digital signal processing.

Your function should follow the bit-level integer coding rules (page 120).

2.74 ◆◆

Write a function with the following prototype:

```
/* Determine whether arguments can be subtracted without overflow */
int tsub_ok(int x, int y);
```

This function should return 1 if the computation `x – y` does not overflow.

2.75 ◆◆◆

Suppose we want to compute the complete $2w$-bit representation of $x \cdot y$, where both x and y are unsigned, on a machine for which data type `unsigned` is w bits. The low-order w bits of the product can be computed with the expression `x*y`, so we only require a procedure with prototype

```
unsigned int unsigned_high_prod(unsigned x, unsigned y);
```

that computes the high-order w bits of $x \cdot y$ for unsigned variables.

We have access to a library function with prototype

```
int signed_high_prod(int x, int y);
```

that computes the high-order w bits of $x \cdot y$ for the case where x and y are in two's-complement form. Write code calling this procedure to implement the function for unsigned arguments. Justify the correctness of your solution.

Hint: Look at the relationship between the signed product $x \cdot y$ and the unsigned product $x' \cdot y'$ in the derivation of Equation 2.18.

2.76 ◆◆

Suppose we are given the task of generating code to multiply integer variable `x` by various different constant factors K. To be efficient, we want to use only the operations `+`, `-`, and `<<`. For the following values of K, write C expressions to perform the multiplication using at most three operations per expression.

A. $K = 17$:
B. $K = -7$:
C. $K = 60$:
D. $K = -112$:

2.77 ◆◆

Write code for a function with the following prototype:

```
/* Divide by power of two.  Assume 0 <= k < w-1 */
int divide_power2(int x, int k);
```

The function should compute $x/2^k$ with correct rounding, and it should follow the bit-level integer coding rules (page 120).

2.78 ◆◆

Write code for a function `mul3div4` that, for integer argument `x`, computes `3*x/4`, but following the bit-level integer coding rules (page 120). Your code should replicate the fact that the computation `3*x` can cause overflow.

2.79 ◆◆◆

Write code for a function `threefourths` which, for integer argument `x`, computes the value of $\frac{3}{4}$`x`, rounded toward zero. It should not overflow. Your function should follow the bit-level integer coding rules (page 120).

2.80 ◆◆

Write C expressions to generate the bit patterns that follow, where a^k represents k repetitions of symbol a. Assume a w-bit data type. Your code may contain references to parameters `j` and `k`, representing the values of j and k, but not a parameter representing w.

A. $1^{w-k}0^k$
B. $0^{w-k-j}1^k0^j$

2.81 ◆

We are running programs on a machine where values of type `int` are 32 bits. They are represented in two's complement, and they are right shifted arithmetically. Values of type `unsigned` are also 32 bits.

We generate arbitrary values x and y, and convert them to unsigned values as follows:

```
/* Create some arbitrary values */
int x = random();
int y = random();
/* Convert to unsigned */
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
```

For each of the following C expressions, you are to indicate whether or not the expression *always* yields 1. If it always yields 1, describe the underlying mathematical principles. Otherwise, give an example of arguments that make it yield 0.

A. `(x<y) == (-x>-y)`

B. `((x+y)<<4) + y-x == 17*y+15*x`

C. `~x+~y+1 == ~(x+y)`

D. `(ux-uy) == -(unsigned)(y-x)`

E. `((x >> 2) << 2) <= x`

2.82 ◆◆

Consider numbers having a binary representation consisting of an infinite string of the form $0.y\ y\ y\ y\ y\ y \cdots$, where y is a k-bit sequence. For example, the binary representation of $\frac{1}{3}$ is $0.01010101 \cdots$ $(y = 01)$, while the representation of $\frac{1}{5}$ is $0.001100110011 \cdots$ $(y = 0011)$.

A. Let $Y = B2U_k(y)$, that is, the number having binary representation y. Give a formula in terms of Y and k for the value represented by the infinite string. **Hint:** Consider the effect of shifting the binary point k positions to the right.

B. What is the numeric value of the string for the following values of y?

   (a) 101

   (b) 0110

   (c) 010011

2.83 ◆

Fill in the return value for the following procedure, which tests whether its first argument is less than or equal to its second. Assume the function `f2u` returns an unsigned 32-bit number having the same bit representation as its floating-point argument. You can assume that neither argument is *NaN*. The two flavors of zero, $+0$ and -0, are considered equal.

```
int float_le(float x, float y) {
    unsigned ux = f2u(x);
    unsigned uy = f2u(y);
```

```
    /* Get the sign bits */
    unsigned sx = ux >> 31;
    unsigned sy = uy >> 31;

    /* Give an expression using only ux, uy, sx, and sy */
    return _________ ;
}
```

2.84 ◆

Given a floating-point format with a k-bit exponent and an n-bit fraction, write formulas for the exponent E, significand M, the fraction f, and the value V for the quantities that follow. In addition, describe the bit representation.

A. The number 7.0

B. The largest odd integer that can be represented exactly

C. The reciprocal of the smallest positive normalized value

2.85 ◆

Intel-compatible processors also support an "extended precision" floating-point format with an 80-bit word divided into a sign bit, $k = 15$ exponent bits, a single *integer* bit, and $n = 63$ fraction bits. The integer bit is an explicit copy of the implied bit in the IEEE floating-point representation. That is, it equals 1 for normalized values and 0 for denormalized values. Fill in the following table giving the approximate values of some "interesting" numbers in this format:

	Extended precision	
Description	Value	Decimal
Smallest positive denormalized	_____	_____
Smallest positive normalized	_____	_____
Largest normalized	_____	_____

2.86 ◆

Consider a 16-bit floating-point representation based on the IEEE floating-point format, with one sign bit, seven exponent bits ($k = 7$), and eight fraction bits ($n = 8$). The exponent bias is $2^{7-1} - 1 = 63$.

Fill in the table that follows for each of the numbers given, with the following instructions for each column:

Hex: The four hexadecimal digits describing the encoded form.

M: The value of the significand. This should be a number of the form x or $\frac{x}{y}$, where x is an integer, and y is an integral power of 2. Examples include: 0, $\frac{67}{64}$, and $\frac{1}{256}$.

E: The integer value of the exponent.

V: The numeric value represented. Use the notation x or $x \times 2^z$, where x and z are integers.

As an example, to represent the number $\frac{7}{8}$, we would have $s = 0$, $M = \frac{7}{4}$, and $E = -1$. Our number would therefore have an exponent field of `0x3E` (decimal value $63 - 1 = 62$) and a significand field `0xC0` (binary 11000000_2), giving a hex representation `3EC0`.

You need not fill in entries marked "—".

Description	Hex	M	E	V
-0	_____	_____	_____	—
Smallest value > 2	_____	_____	_____	_____
512	_____	_____	_____	—
Largest denormalized	_____	_____	_____	_____
$-\infty$	_____	—	—	—
Number with hex representation `3BB0`	—	_____	_____	_____

2.87 ◆◆

Consider the following two 9-bit floating-point representations based on the IEEE floating-point format.

1. Format A
   - There is one sign bit.
   - There are $k = 5$ exponent bits. The exponent bias is 15.
   - There are $n = 3$ fraction bits.
2. Format B
   - There is one sign bit.
   - There are $k = 4$ exponent bits. The exponent bias is 7.
   - There are $n = 4$ fraction bits.

Below, you are given some bit patterns in Format A, and your task is to convert them to the closest value in Format B. If rounding is necessary, you should *round toward* $+\infty$. In addition, give the values of numbers given by the Format A and Format B bit patterns. Give these as whole numbers (e.g., 17) or as fractions (e.g., 17/64 or $17/2^6$).

Format A		Format B	
Bits	Value	Bits	Value
`1 01111 001`	$\frac{-9}{8}$	`1 0111 0010`	$\frac{-9}{8}$
`0 10110 011`	_____	_____	_____
`1 00111 010`	_____	_____	_____
`0 00000 111`	_____	_____	_____
`1 11100 000`	_____	_____	_____
`0 10111 100`	_____	_____	_____

2.88 ◆

We are running programs on a machine where values of type `int` have a 32-bit two's-complement representation. Values of type `float` use the 32-bit IEEE format, and values of type `double` use the 64-bit IEEE format.

We generate arbitrary integer values x, y, and z, and convert them to values of type double as follows:

```
/* Create some arbitrary values */
int x = random();
int y = random();
int z = random();
/* Convert to double */
double   dx = (double) x;
double   dy = (double) y;
double   dz = (double) z;
```

For each of the following C expressions, you are to indicate whether or not the expression *always* yields 1. If it always yields 1, describe the underlying mathematical principles. Otherwise, give an example of arguments that make it yield 0. Note that you cannot use an IA32 machine running GCC to test your answers, since it would use the 80-bit extended-precision representation for both float and double.

A. `(float) x == (float) dx`

B. `dx - dy == (double) (x-y)`

C. `(dx + dy) + dz == dx + (dy + dz)`

D. `(dx * dy) * dz == dx * (dy * dz)`

E. `dx / dx == dz / dz`

2.89 ◆

You have been assigned the task of writing a C function to compute a floating-point representation of 2^x. You decide that the best way to do this is to directly construct the IEEE single-precision representation of the result. When x is too small, your routine will return 0.0. When x is too large, it will return $+\infty$. Fill in the blank portions of the code that follows to compute the correct result. Assume the function u2f returns a floating-point value having an identical bit representation as its unsigned argument.

```
float fpwr2(int x)
{

    /* Result exponent and fraction */
    unsigned exp, frac;
    unsigned u;

    if (x < _________) {
        /* Too small.  Return 0.0 */
        exp =  _________;
        frac = _________;
    } else if (x < _________) {
```

```
        /* Denormalized result */
        exp = ________;
        frac = ________;
    } else if (x < ________) {
        /* Normalized result. */
        exp = ________;
        frac = ________;
    } else {
        /* Too big.  Return +oo */
        exp = ________;
        frac = ________;
    }

    /* Pack exp and frac into 32 bits */
    u = exp << 23 | frac;
    /* Return as float */
    return u2f(u);
}
```

2.90 ◆

Around 250 B.C., the Greek mathematician Archimedes proved that $\frac{223}{71} < \pi < \frac{22}{7}$. Had he had access to a computer and the standard library `<math.h>`, he would have been able to determine that the single-precision floating-point approximation of π has the hexadecimal representation `0x40490FDB`. Of course, all of these are just approximations, since π is not rational.

A. What is the fractional binary number denoted by this floating-point value?

B. What is the fractional binary representation of $\frac{22}{7}$? **Hint:** See Problem 2.82.

C. At what bit position (relative to the binary point) do these two approximations to π diverge?

Bit-level floating-point coding rules

In the following problems, you will write code to implement floating-point functions, operating directly on bit-level representations of floating-point numbers. Your code should exactly replicate the conventions for IEEE floating-point operations, including using round-to-even mode when rounding is required.

Toward this end, we define data type `float_bits` to be equivalent to `unsigned`:

```
/* Access bit-level representation floating-point number */
typedef unsigned float_bits;
```

Rather than using data type `float` in your code, you will use `float_bits`. You may use both `int` and `unsigned` data types, including unsigned and integer constants and operations. You may not use any unions, structs, or arrays. Most

significantly, you may not use any floating-point data types, operations, or constants. Instead, your code should perform the bit manipulations that implement the specified floating-point operations.

The following function illustrates the use of these coding rules. For argument f, it returns ± 0 if f is denormalized (preserving the sign of f) and returns f otherwise.

```
/* If f is denorm, return 0.  Otherwise, return f */
float_bits float_denorm_zero(float_bits f) {
    /* Decompose bit representation into parts */
    unsigned sign = f>>31;
    unsigned exp =  f>>23 & 0xFF;
    unsigned frac = f     & 0x7FFFFF;
    if (exp == 0) {
        /* Denormalized.  Set fraction to 0 */
        frac = 0;
    }
    /* Reassemble bits */
    return (sign << 31) | (exp << 23) | frac;
}
```

2.91 ◆◆

Following the bit-level floating-point coding rules, implement the function with the following prototype:

```
/* Compute -f.  If f is NaN, then return f. */
float_bits float_negate(float_bits f);
```

For floating-point number f, this function computes $-f$. If f is NaN, your function should simply return f.

Test your function by evaluating it for all 2^{32} values of argument `f` and comparing the result to what would be obtained using your machine's floating-point operations.

2.92 ◆◆

Following the bit-level floating-point coding rules, implement the function with the following prototype:

```
/* Compute |f|.  If f is NaN, then return f. */
float_bits float_absval(float_bits f);
```

For floating-point number f, this function computes $|f|$. If f is NaN, your function should simply return f.

Test your function by evaluating it for all 2^{32} values of argument `f` and comparing the result to what would be obtained using your machine's floating-point operations.

2.93 ◆◆◆

Following the bit-level floating-point coding rules, implement the function with the following prototype:

```
/* Compute 2*f.  If f is NaN, then return f. */
float_bits float_twice(float_bits f);
```

For floating-point number f, this function computes $2.0 \cdot f$. If f is *NaN*, your function should simply return f.

Test your function by evaluating it for all 2^{32} values of argument `f` and comparing the result to what would be obtained using your machine's floating-point operations.

2.94 ◆◆◆

Following the bit-level floating-point coding rules, implement the function with the following prototype:

```
/* Compute 0.5*f.  If f is NaN, then return f. */
float_bits float_half(float_bits f);
```

For floating-point number f, this function computes $0.5 \cdot f$. If f is *NaN*, your function should simply return f.

Test your function by evaluating it for all 2^{32} values of argument `f` and comparing the result to what would be obtained using your machine's floating-point operations.

2.95 ◆◆◆◆

Following the bit-level floating-point coding rules, implement the function with the following prototype:

```
/*
 * Compute (int) f.
 * If conversion causes overflow or f is NaN, return 0x80000000
 */
int float_f2i(float_bits f);
```

For floating-point number f, this function computes `(int)` f. Your function should round toward zero. If f cannot be represented as an integer (e.g., it is out of range, or it is *NaN*), then the function should return `0x80000000`.

Test your function by evaluating it for all 2^{32} values of argument `f` and comparing the result to what would be obtained using your machine's floating-point operations.

2.96 ◆◆◆◆

Following the bit-level floating-point coding rules, implement the function with the following prototype:

```
/* Compute (float) i */
float_bits float_i2f(int i);
```

For argument `i`, this function computes the bit-level representation of `(float) i`.

Test your function by evaluating it for all 2^{32} values of argument `f` and comparing the result to what would be obtained using your machine's floating-point operations.

Solutions to Practice Problems

Solution to Problem 2.1 (page 35)

Understanding the relation between hexadecimal and binary formats will be important once we start looking at machine-level programs. The method for doing these conversions is in the text, but it takes a little practice to become familiar.

A. `0x39A7F8` to binary:

Hexadecimal	3	9	A	7	F	8
Binary	0011	1001	1010	0111	1111	1000

B. Binary 1100100101111011 to hexadecimal:

Binary	1100	1001	0111	1011
Hexadecimal	C	9	7	B

C. `0xD5E4C` to binary:

Hexadecimal	D	5	E	4	C
Binary	1101	0101	1110	0100	1100

D. Binary 100110111001111010101 to hexadecimal:

Binary	10	0110	1110	0111	1011	0101
Hexadecimal	2	6	E	7	B	5

Solution to Problem 2.2 (page 35)

This problem gives you a chance to think about powers of 2 and their hexadecimal representations.

n	2^n (Decimal)	2^n (Hexadecimal)
9	512	0x200
19	524,288	0x80000
14	16,384	0x4000
16	65,536	0x10000
17	131,072	0x20000
5	32	0x20
7	128	0x80

Solution to Problem 2.3 (page 36)

This problem gives you a chance to try out conversions between hexadecimal and decimal representations for some smaller numbers. For larger ones, it becomes much more convenient and reliable to use a calculator or conversion program.

Decimal	Binary	Hexadecimal
0	0000 0000	`0x00`
$167 = 10 \cdot 16 + 7$	1010 0111	`0xA7`
$62 = 3 \cdot 16 + 14$	0011 1110	`0x3E`
$188 = 11 \cdot 16 + 12$	1011 1100	`0xBC`
$3 \cdot 16 + 7 = 55$	0011 0111	`0x37`
$8 \cdot 16 + 8 = 136$	1000 1000	`0x88`
$15 \cdot 16 + 3 = 243$	1111 0011	`0xF3`
$5 \cdot 16 + 2 = 82$	0101 0010	`0x52`
$10 \cdot 16 + 12 = 172$	1010 1100	`0xAC`
$14 \cdot 16 + 7 = 231$	1110 0111	`0xE7`

Solution to Problem 2.4 (page 37)

When you begin debugging machine-level programs, you will find many cases where some simple hexadecimal arithmetic would be useful. You can always convert numbers to decimal, perform the arithmetic, and convert them back, but being able to work directly in hexadecimal is more efficient and informative.

A. `0x503c + 0x8 = 0x5044`. Adding `8` to hex `c` gives `4` with a carry of `1`.

B. `0x503c − 0x40 = 0x4ffc`. Subtracting `4` from `3` in the second digit position requires a borrow from the third. Since this digit is `0`, we must also borrow from the fourth position.

C. `0x503c + 64 = 0x507c`. Decimal 64 (2^6) equals hexadecimal `0x40`.

D. `0x50ea − 0x503c = 0xae`. To subtract hex `c` (decimal 12) from hex `a` (decimal 10), we borrow 16 from the second digit, giving hex `e` (decimal 14). In the second digit, we now subtract `3` from hex `d` (decimal 13), giving hex `a` (decimal 10).

Solution to Problem 2.5 (page 45)

This problem tests your understanding of the byte representation of data and the two different byte orderings.

Little endian: `21`	Big endian: `87`
Little endian: `21 43`	Big endian: `87 65`
Little endian: `21 43 65`	Big endian: `87 65 43`

Recall that `show_bytes` enumerates a series of bytes starting from the one with lowest address and working toward the one with highest address. On a little-endian machine, it will list the bytes from least significant to most. On a big-endian machine, it will list bytes from the most significant byte to the least.

Solution to Problem 2.6 (page 46)

This problem is another chance to practice hexadecimal to binary conversion. It also gets you thinking about integer and floating-point representations. We will explore these representations in more detail later in this chapter.

A. Using the notation of the example in the text, we write the two strings as follows:

```
   0   0   3   5   9   1   4   1
00000000001101011001000101000001
          ********************
    4   A   5   6   4   5   0   4
01001010010101100100010100000100
```

B. With the second word shifted two positions to the right relative to the first, we find a sequence with 21 matching bits.

C. We find all bits of the integer embedded in the floating-point number, except for the most significant bit having value 1. Such is the case for the example in the text as well. In addition, the floating-point number has some nonzero high-order bits that do not match those of the integer.

Solution to Problem 2.7 (page 46)

It prints `61 62 63 64 65 66`. Recall also that the library routine `strlen` does not count the terminating null character, and so `show_bytes` printed only through the character '`f`'.

Solution to Problem 2.8 (page 49)

This problem is a drill to help you become more familiar with Boolean operations.

Operation	Result
a	[01101001]
b	[01010101]
$\sim a$	[10010110]
$\sim b$	[10101010]
a & b	[01000001]
a | b	[01111101]
a ^ b	[00111100]

Solution to Problem 2.9 (page 50)

This problem illustrates how Boolean algebra can be used to describe and reason about real-world systems. We can see that this color algebra is identical to the Boolean algebra over bit vectors of length 3.

A. Colors are complemented by complementing the values of R, G, and B. From this, we can see that White is the complement of Black, Yellow is the complement of Blue, Magenta is the complement of Green, and Cyan is the complement of Red.

B. We perform Boolean operations based on a bit-vector representation of the colors. From this we get the following:

Blue (001)	`|`	Green (010)	=	Cyan (011)
Yellow (110)	`&`	Cyan (011)	=	Green (010)
Red (100)	`^`	Magenta (101)	=	Blue (001)

Solution to Problem 2.10 (page 51)

This procedure relies on the fact that Exclusive-Or is commutative and associative, and that $a \hat{} a = 0$ for any a.

Step	`*x`	`*y`
Initially	a	b
Step 1	a	$a \hat{} b$
Step 2	$a \hat{} (a \hat{} b) = (a \hat{} a) \hat{} b = b$	$a \hat{} b$
Step 3	b	$b \hat{} (a \hat{} b) = (b \hat{} b) \hat{} a = a$

See Problem 2.11 for a case where this function will fail.

Solution to Problem 2.11 (page 52)

This problem illustrates a subtle and interesting feature of our inplace swap routine.

A. Both `first` and `last` have value k, so we are attempting to swap the middle element with itself.

B. In this case, arguments `x` and `y` to `inplace_swap` both point to the same location. When we compute `*x ^ *y`, we get 0. We then store 0 as the middle element of the array, and the subsequent steps keep setting this element to 0. We can see that our reasoning in Problem 2.10 implicitly assumed that `x` and `y` denote different locations.

C. Simply replace the test in line 4 of `reverse_array` to be `first < last`, since there is no need to swap the middle element with itself.

Solution to Problem 2.12 (page 53)

Here are the expressions:

A. `x & 0xFF`

B. `x ^ ~0xFF`

C. `x | 0xFF`

These expressions are typical of the kind commonly found in performing low-level bit operations. The expression `~0xFF` creates a mask where the 8 least-significant bits equal 0 and the rest equal 1. Observe that such a mask will be generated regardless of the word size. By contrast, the expression `0xFFFFFF00` would only work on a 32-bit machine.

Solution to Problem 2.13 (page 53)

These problems help you think about the relation between Boolean operations and typical ways that programmers apply masking operations. Here is the code:

```
/* Declarations of functions implementing operations bis and bic */
int bis(int x, int m);
int bic(int x, int m);

/* Compute x|y using only calls to functions bis and bic */
int bool_or(int x, int y) {
  int result = bis(x,y);
  return result;
}

/* Compute x^y using only calls to functions bis and bic */
int bool_xor(int x, int y) {
  int result = bis(bic(x,y), bic(y,x));
  return result;
}
```

The `bis` operation is equivalent to Boolean OR—a bit is set in `z` if either this bit is set in `x` or it is set in `m`. On the other hand, `bic(x, m)` is equivalent to `x&~m`; we want the result to equal 1 only when the corresponding bit of `x` is 1 and of `m` is 0.

Given that, we can implement `|` with a single call to `bis`. To implement `^`, we take advantage of the property

$$x \hat{} y = (x \,\&\, \sim y) \mid (\sim x \,\&\, y).$$

Solution to Problem 2.14 (page 54)

This problem highlights the relation between bit-level Boolean operations and logic operations in C. A common programming error is to use a bit-level operation when a logic one is intended, or vice versa.

Expression	Value	Expression	Value
`x & y`	`0x20`	`x && y`	`0x01`
`x | y`	`0x7F`	`x || y`	`0x01`
`~x | ~y`	`0xDF`	`!x || !y`	`0x00`
`x & !y`	`0x00`	`x && ~y`	`0x01`

Solution to Problem 2.15 (page 54)

The expression is `!(x ^ y)`.

That is, `x^y` will be zero if and only if every bit of `x` matches the corresponding bit of `y`. We then exploit the ability of `!` to determine whether a word contains any nonzero bit.

There is no real reason to use this expression rather than simply writing `x == y`, but it demonstrates some of the nuances of bit-level and logical operations.

Solution to Problem 2.16 (page 56)

This problem is a drill to help you understand the different shift operations.

x		x << 3		(Logical) x >> 2		(Arithmetic) x >> 2	
Hex	Binary	Binary	Hex	Binary	Hex	Binary	Hex
0xC3	[11000011]	[00011000]	0x18	[00110000]	0x30	[11110000]	0xF0
0x75	[01110101]	[10101000]	0xA8	[00011101]	0x1D	[00011101]	0x1D
0x87	[10000111]	[00111000]	0x38	[00100001]	0x21	[11100001]	0xE1
0x66	[01100110]	[00110000]	0x30	[00011001]	0x19	[00011001]	0x19

Solution to Problem 2.17 (page 61)

In general, working through examples for very small word sizes is a very good way to understand computer arithmetic.

The unsigned values correspond to those in Figure 2.2. For the two's-complement values, hex digits 0 through 7 have a most significant bit of 0, yielding nonnegative values, while hex digits 8 through F have a most significant bit of 1, yielding a negative value.

$\vec{x}$			
Hexadecimal	Binary	$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
0xE	[1110]	$2^3 + 2^2 + 2^1 = 14$	$-2^3 + 2^2 + 2^1 = -2$
0x0	[0000]	0	0
0x5	[0101]	$2^2 + 2^0 = 5$	$2^2 + 2^0 = 5$
0x8	[1000]	$2^3 = 8$	$-2^3 = -8$
0xD	[1101]	$2^3 + 2^2 + 2^0 = 13$	$-2^3 + 2^2 + 2^0 = -3$
0xF	[1111]	$2^3 + 2^2 + 2^1 + 2^0 = 15$	$-2^3 + 2^2 + 2^1 + 2^0 = -1$

Solution to Problem 2.18 (page 64)

For a 32-bit machine, any value consisting of eight hexadecimal digits beginning with one of the digits 8 through f represents a negative number. It is quite common to see numbers beginning with a string of f's, since the leading bits of a negative number are all ones. You must look carefully, though. For example, the number 0x8048337 has only seven digits. Filling this out with a leading zero gives 0x08048337, a positive number.

```
8048337:  81 ec b8 01 00 00     sub    $0x1b8,%esp              A.   440
804833d:  8b 55 08              mov    0x8(%ebp),%edx
8048340:  83 c2 14              add    $0x14,%edx               B.    20
8048343:  8b 85 58 fe ff ff     mov    0xfffffe58(%ebp),%eax    C.  -424
8048349:  03 02                 add    (%edx),%eax
804834b:  89 85 74 fe ff ff     mov    %eax,0xfffffe74(%ebp)    D.  -396
8048351:  8b 55 08              mov    0x8(%ebp),%edx
8048354:  83 c2 44              add    $0x44,%edx               E.    68
8048357:  8b 85 c8 fe ff ff     mov    0xfffffec8(%ebp),%eax    F.  -312
```

```
804835d:  89 02                 mov    %eax,(%edx)
804835f:  8b 45 10              mov    0x10(%ebp),%eax          G.   16
8048362:  03 45 0c              add    0xc(%ebp),%eax           H.   12
8048365:  89 85 ec fe ff ff     mov    %eax,0xfffffeec(%ebp)    I.   -276
804836b:  8b 45 08              mov    0x8(%ebp),%eax
804836e:  83 c0 20              add    $0x20,%eax               J.   32
8048371:  8b 00                 mov    (%eax),%eax
```

Solution to Problem 2.19 (page 67)

The functions *T2U* and *U2T* are very peculiar from a mathematical perspective. It is important to understand how they behave.

We solve this problem by reordering the rows in the solution of Problem 2.17 according to the two's-complement value and then listing the unsigned value as the result of the function application. We show the hexadecimal values to make this process more concrete.

$\vec{x}$ (hex)	x	$T2U_4(x)$
0x8	−8	8
0xD	−3	13
0xE	−2	14
0xF	−1	15
0x0	0	0
0x5	5	5

Solution to Problem 2.20 (page 68)

This exercise tests your understanding of Equation 2.6.

For the first four entries, the values of x are negative and $T2U_4(x) = x + 2^4$. For the remaining two entries, the values of x are nonnegative and $T2U_4(x) = x$.

Solution to Problem 2.21 (page 70)

This problem reinforces your understanding of the relation between two's-complement and unsigned representations, and the effects of the C promotion rules. Recall that $TMin_{32}$ is −2,147,483,648, and that when cast to unsigned it becomes 2,147,483,648. In addition, if either operand is unsigned, then the other operand will be cast to unsigned before comparing.

Expression	Type	Evaluation
`-2147483647-1 == 2147483648U`	unsigned	1
`-2147483647-1 < 2147483647`	signed	1
`-2147483647-1U < 2147483647`	unsigned	0
`-2147483647-1 < -2147483647`	signed	1
`-2147483647-1U < -2147483647`	unsigned	1

Solution to Problem 2.22 (page 74)

This exercise provides a concrete demonstration of how sign extension preserves the numeric value of a two's-complement representation.

A.	[1011]:	$-2^3 + 2^1 + 2^0$	=	$-8 + 2 + 1$	=	-5
B.	[11011]:	$-2^4 + 2^3 + 2^1 + 2^0$	=	$-16 + 8 + 2 + 1$	=	-5
C.	[111011]:	$-2^5 + 2^4 + 2^3 + 2^1 + 2^0$	=	$-32 + 16 + 8 + 2 + 1$	=	-5

Solution to Problem 2.23 (page 74)

The expressions in these functions are common program "idioms" for extracting values from a word in which multiple bit fields have been packed. They exploit the zero-filling and sign-extending properties of the different shift operations. Note carefully the ordering of the cast and shift operations. In `fun1`, the shifts are performed on unsigned variable `word`, and hence are logical. In `fun2`, shifts are performed after casting `word` to `int`, and hence are arithmetic.

A.

`w`	`fun1(w)`	`fun2(w)`
`0x00000076`	`0x00000076`	`0x00000076`
`0x87654321`	`0x00000021`	`0x00000021`
`0x000000C9`	`0x000000C9`	`0xFFFFFFC9`
`0xEDCBA987`	`0x00000087`	`0xFFFFFF87`

B. Function `fun1` extracts a value from the low-order 8 bits of the argument, giving an integer ranging between 0 and 255. Function `fun2` extracts a value from the low-order 8 bits of the argument, but it also performs sign extension. The result will be a number between -128 and 127.

Solution to Problem 2.24 (page 76)

The effect of truncation is fairly intuitive for unsigned numbers, but not for two's-complement numbers. This exercise lets you explore its properties using very small word sizes.

Hex		Unsigned		Two's complement	
Original	Truncated	Original	Truncated	Original	Truncated
0	0	0	0	0	0
2	2	2	2	2	2
9	1	9	1	-7	1
B	3	11	3	-5	3
F	7	15	7	-1	-1

As Equation 2.9 states, the effect of this truncation on unsigned values is to simply find their residue, modulo 8. The effect of the truncation on signed values is a bit more complex. According to Equation 2.10, we first compute the modulo 8 residue of the argument. This will give values 0 through 7 for arguments 0 through 7, and also for arguments -8 through -1. Then we apply function $U2T_3$ to these residues, giving two repetitions of the sequences 0 through 3 and -4 through -1.

Solution to Problem 2.25 (page 77)

This problem is designed to demonstrate how easily bugs can arise due to the implicit casting from signed to unsigned. It seems quite natural to pass parameter `length` as an unsigned, since one would never want to use a negative length. The stopping criterion `i <= length-1` also seems quite natural. But combining these two yields an unexpected outcome!

Since parameter `length` is unsigned, the computation $0 - 1$ is performed using unsigned arithmetic, which is equivalent to modular addition. The result is then $UMax$. The $\leq$ comparison is also performed using an unsigned comparison, and since any number is less than or equal to $UMax$, the comparison always holds! Thus, the code attempts to access invalid elements of array `a`.

The code can be fixed either by declaring `length` to be an `int`, or by changing the test of the `for` loop to be `i < length`.

Solution to Problem 2.26 (page 77)

This example demonstrates a subtle feature of unsigned arithmetic, and also the property that we sometimes perform unsigned arithmetic without realizing it. This can lead to very tricky bugs.

A. *For what cases will this function produce an incorrect result?* The function will incorrectly return 1 when `s` is shorter than `t`.

B. *Explain how this incorrect result comes about.* Since `strlen` is defined to yield an unsigned result, the difference and the comparison are both computed using unsigned arithmetic. When `s` is shorter than `t`, the difference `strlen(s) - strlen(t)` should be negative, but instead becomes a large, unsigned number, which is greater than 0.

C. *Show how to fix the code so that it will work reliably.* Replace the test with the following:

```
return strlen(s) > strlen(t);
```

Solution to Problem 2.27 (page 81)

This function is a direct implementation of the rules given to determine whether or not an unsigned addition overflows.

```
/* Determine whether arguments can be added without overflow */
int uadd_ok(unsigned x, unsigned y) {
    unsigned sum = x+y;
    return sum >= x;
}
```

Solution to Problem 2.28 (page 82)

This problem is a simple demonstration of arithmetic modulo 16. The easiest way to solve it is to convert the hex pattern into its unsigned decimal value. For nonzero values of x, we must have $(-_4^{\mathrm{u}} x) + x = 16$. Then we convert the complemented value back to hex.

x		$-_4^u x$	
Hex	Decimal	Decimal	Hex
0	0	0	0
5	5	11	B
8	8	8	8
D	13	3	3
F	15	1	1

Solution to Problem 2.29 (page 86)

This problem is an exercise to make sure you understand two's-complement addition.

x	y	$x + y$	$x +_5^t y$	Case
−12	−15	−27	5	1
[10100]	[10001]	[100101]	[00101]	
−8	−8	−16	−16	2
[11000]	[11000]	[110000]	[10000]	
−9	8	−1	−1	2
[10111]	[01000]	[111111]	[11111]	
2	5	7	7	3
[00010]	[00101]	[000111]	[00111]	
12	4	16	−16	4
[01100]	[00100]	[010000]	[10000]	

Solution to Problem 2.30 (page 86)

This function is a direct implementation of the rules given to determine whether or not a two's-complement addition overflows.

```
/* Determine whether arguments can be added without overflow */
int tadd_ok(int x, int y) {
    int sum = x+y;
    int neg_over = x <  0 && y <  0 && sum >= 0;
    int pos_over = x >= 0 && y >= 0 && sum <  0;
    return !neg_over && !pos_over;
}
```

Solution to Problem 2.31 (page 86)

Your coworker could have learned, by studying Section 2.3.2, that two's-complement addition forms an abelian group, and so the expression `(x+y)-x` will evaluate to `y` regardless of whether or not the addition overflows, and that `(x+y)-y` will always evaluate to `x`.

Solution to Problem 2.32 (page 87)

This function will give correct values, except when `y` is *TMin*. In this case, we will have `-y` also equal to *TMin*, and so function `tadd_ok` will consider there to be

negative overflow any time `x` is negative. In fact, `x-y` does not overflow for these cases.

One lesson to be learned from this exercise is that *TMin* should be included as one of the cases in any test procedure for a function.

Solution to Problem 2.33 (page 87)

This problem helps you understand two's-complement negation using a very small word size.

For $w = 4$, we have $TMin_4 = -8$. So -8 is its own additive inverse, while other values are negated by integer negation.

x		$-_4^t x$	
Hex	Decimal	Decimal	Hex
0	0	0	0
5	5	−5	B
8	−8	−8	8
D	−3	3	3
F	−1	1	1

The bit patterns are the same as for unsigned negation.

Solution to Problem 2.34 (page 90)

This problem is an exercise to make sure you understand two's-complement multiplication.

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	4	[100]	5	[101]	20	[010100]	4	[100]
Two's comp.	−4	[100]	−3	[101]	12	[001100]	−4	[100]
Unsigned	2	[010]	7	[111]	14	[001110]	6	[110]
Two's comp.	2	[010]	−1	[111]	−2	[111110]	−2	[110]
Unsigned	6	[110]	6	[110]	36	[100100]	4	[100]
Two's comp.	−2	[110]	−2	[110]	4	[000100]	−4	[100]

Solution to Problem 2.35 (page 90)

It's not realistic to test this function for all possible values of `x` and `y`. Even if you could run 10 billion tests per second, it would require over 58 years to test all combinations when data type `int` is 32 bits. On the other hand, it is feasible to test your code by writing the function with data type `short` or `char` and then testing it exhaustively.

Here's a more principled approach, following the proposed set of arguments:

1. We know that $x \cdot y$ can be written as a $2w$-bit two's-complement number. Let u denote the unsigned number represented by the lower w bits, and v denote the two's-complement number represented by the upper w bits. Then, based on Equation 2.3, we can see that $x \cdot y = v2^w + u$.

We also know that $u = T2U_w(p)$, since they are unsigned and two's-complement numbers arising from the same bit pattern, and so by Equation 2.5, we can write $u = p + p_{w-1}2^w$, where p_{w-1} is the most significant bit of p. Letting $t = v + p_{w-1}$, we have $x \cdot y = p + t2^w$.

When $t = 0$, we have $x \cdot y = p$; the multiplication does not overflow. When $t \neq 0$, we have $x \cdot y \neq p$; the multiplication does overflow.

2. By definition of integer division, dividing p by nonzero x gives a quotient q and a remainder r such that $p = x \cdot q + r$, and $|r| < |x|$. (We use absolute values here, because the signs of x and r may differ. For example, dividing -7 by 2 gives quotient -3 and remainder -1.)

3. Suppose $q = y$. Then we have $x \cdot y = x \cdot y + r + t2^w$. From this, we can see that $r + t2^w = 0$. But $|r| < |x| \leq 2^w$, and so this identity can hold only if $t = 0$, in which case $r = 0$.

Suppose $r = t = 0$. Then we will have $x \cdot y = x \cdot q$, implying that $y = q$.

When x equals 0, multiplication does not overflow, and so we see that our code provides a reliable way to test whether or not two's-complement multiplication causes overflow.

Solution to Problem 2.36 (page 91)

With 64 bits, we can perform the multiplication without overflowing. We then test whether casting the product to 32 bits changes the value:

```
1   /* Determine whether arguments can be multiplied without overflow */
2   int tmult_ok(int x, int y) {
3       /* Compute product without overflow */
4       long long pll = (long long) x*y;
5       /* See if casting to int preserves value */
6       return pll == (int) pll;
7   }
```

Note that the casting on the right-hand side of line 4 is critical. If we instead wrote the line as

```
long long pll = x*y;
```

the product would be computed as a 32-bit value (possibly overflowing) and then sign extended to 64 bits.

Solution to Problem 2.37 (page 92)

A. This change does not help at all. Even though the computation of `asize` will be accurate, the call to `malloc` will cause this value to be converted to a 32-bit unsigned number, and so the same overflow conditions will occur.

B. With `malloc` having a 32-bit unsigned number as its argument, it cannot possibly allocate a block of more than 2^{32} bytes, and so there is no point attempting to allocate or copy this much memory. Instead, the function

should abort and return NULL, as illustrated by the following replacement to the original call to malloc (line 10):

```
long long unsigned required_size =
    ele_cnt * (long long unsigned) ele_size;
size_t request_size = (size_t) required_size;
if (required_size != request_size)
    /* Overflow must have occurred.  Abort operation */
    return NULL;
void *result = malloc(request_size);
if (result == NULL)
    /* malloc failed */
    return NULL;
```

Solution to Problem 2.38 (page 93)

In Chapter 3, we will see many examples of the LEA instruction in action. The instruction is provided to support pointer arithmetic, but the C compiler often uses it as a way to perform multiplication by small constants.

For each value of k, we can compute two multiples: 2^k (when b is 0) and $2^k + 1$ (when b is a). Thus, we can compute multiples 1, 2, 3, 4, 5, 8, and 9.

Solution to Problem 2.39 (page 94)

The expression simply becomes `-(x<<m)`. To see this, let the word size be w so that $n = w-1$. Form B states that we should compute `(x<<w) - (x<<m)`, but shifting x to the left by w will yield the value 0.

Solution to Problem 2.40 (page 94)

This problem requires you to try out the optimizations already described and also to supply a bit of your own ingenuity.

K	Shifts	Add/Subs	Expression
6	2	1	`(x<<2) + (x<<1)`
31	1	1	`(x<<5) - x`
–6	2	1	`(x<<1) - (x<<3)`
55	2	2	`(x<<6) - (x<<3) - x`

Observe that the fourth case uses a modified version of form B. We can view the bit pattern [110111] as having a run of 6 ones with a zero in the middle, and so we apply the rule for form B, but then we subtract the term corresponding to the middle zero bit.

Solution to Problem 2.41 (page 94)

Assuming that addition and subtraction have the same performance, the rule is to choose form A when $n = m$, either form when $n = m + 1$, and form B when $n > m + 1$.

The justification for this rule is as follows. Assume first that $m > 1$. When $n = m$, form A requires only a single shift, while form B requires two shifts and a subtraction. When $n = m + 1$, both forms require two shifts and either an addition or a subtraction. When $n > m + 1$, form B requires only two shifts and one subtraction, while form A requires $n - m + 1 > 2$ shifts and $n - m > 1$ additions. For the case of $m = 1$, we get one fewer shift for both forms A and B, and so the same rules apply for choosing between the two.

Solution to Problem 2.42 (page 97)

The only challenge here is to compute the bias without any testing or conditional operations. We use the trick that the expression `x >> 31` generates a word with all ones if `x` is negative, and all zeros otherwise. By masking off the appropriate bits, we get the desired bias value.

```
int div16(int x) {
    /* Compute bias to be either 0 (x >= 0) or 15 (x < 0) */
    int bias = (x >> 31) & 0xF;
    return (x + bias) >> 4;
}
```

Solution to Problem 2.43 (page 98)

We have found that people have difficulty with this exercise when working directly with assembly code. It becomes more clear when put in the form shown in `optarith`.

We can see that `M` is 31; `x*M` is computed as `(x<<5)-x`.

We can see that `N` is 8; a bias value of 7 is added when `y` is negative, and the right shift is by 3.

Solution to Problem 2.44 (page 99)

These "C puzzle" problems provide a clear demonstration that programmers must understand the properties of computer arithmetic:

A. `(x > 0) || (x-1 < 0)`
False. Let `x` be −2,147,483,648 ($TMin_{32}$). We will then have `x-1` equal to 2147483647 ($TMax_{32}$).

B. `(x & 7) != 7 || (x<<29 < 0)`
True. If `(x & 7) != 7` evaluates to 0, then we must have bit x_2 equal to 1. When shifted left by 29, this will become the sign bit.

C. `(x * x) >= 0`
False. When `x` is 65,535 (`0xFFFF`), `x*x` is −131,071 (`0xFFFE0001`).

D. `x < 0 || -x <= 0`
True. If `x` is nonnegative, then `-x` is nonpositive.

E. `x > 0 || -x >= 0`
False. Let `x` be −2,147,483,648 ($TMin_{32}$). Then both `x` and `-x` are negative.

F. `x+y == uy+ux`
True. Two's-complement and unsigned addition have the same bit-level behavior, and they are commutative.

G. `x*~y + uy*ux == -x`
True. `~y` equals `-y-1`. `uy*ux` equals `x*y`. Thus, the left hand side is equivalent to `x*-y-x+x*y`.

Solution to Problem 2.45 (page 102)

Understanding fractional binary representations is an important step to understanding floating-point encodings. This exercise lets you try out some simple examples.

Fractional value	Binary representation	Decimal representation
$\frac{1}{8}$	0.001	0.125
$\frac{3}{4}$	0.11	0.75
$\frac{25}{16}$	1.1001	1.5625
$\frac{43}{16}$	10.1011	2.6875
$\frac{9}{8}$	1.001	1.125
$\frac{47}{8}$	101.111	5.875
$\frac{51}{16}$	11.0011	3.1875

One simple way to think about fractional binary representations is to represent a number as a fraction of the form $\frac{x}{2^k}$. We can write this in binary using the binary representation of x, with the binary point inserted k positions from the right. As an example, for $\frac{25}{16}$, we have $25_{10} = 11001_2$. We then put the binary point four positions from the right to get 1.1001_2.

Solution to Problem 2.46 (page 102)

In most cases, the limited precision of floating-point numbers is not a major problem, because the *relative* error of the computation is still fairly low. In this example, however, the system was sensitive to the *absolute* error.

A. We can see that $0.1 - x$ has binary representation

$$0.000000000000000000000001100[1100]\cdots_2$$

B. Comparing this to the binary representation of $\frac{1}{10}$, we can see that it is simply $2^{-20} \times \frac{1}{10}$, which is around 9.54×10^{-8}.

C. $9.54 \times 10^{-8} \times 100 \times 60 \times 60 \times 10 \approx 0.343$ seconds.

D. $0.343 \times 2000 \approx 687$ meters.

Solution to Problem 2.47 (page 107)

Working through floating-point representations for very small word sizes helps clarify how IEEE floating point works. Note especially the transition between denormalized and normalized values.

Bits	e	E	2^E	f	M	$2^E \times M$	V	Decimal
0 00 00	0	0	1	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{0}{4}$	0	0.0
0 00 01	0	0	1	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	0.25
0 00 10	0	0	1	$\frac{2}{4}$	$\frac{2}{4}$	$\frac{2}{4}$	$\frac{1}{2}$	0.5
0 00 11	0	0	1	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	0.75
0 01 00	1	0	1	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{4}{4}$	1	1.0
0 01 01	1	0	1	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	$\frac{5}{4}$	1.25
0 01 10	1	0	1	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{6}{4}$	$\frac{3}{2}$	1.5
0 01 11	1	0	1	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{7}{4}$	$\frac{7}{4}$	1.75
0 10 00	2	1	2	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{8}{4}$	2	2.0
0 10 01	2	1	2	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{10}{4}$	$\frac{5}{2}$	2.5
0 10 10	2	1	2	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{12}{4}$	3	3.0
0 10 11	2	1	2	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{14}{4}$	$\frac{7}{2}$	3.5
0 11 00	—	—	—	—	—	—	∞	—
0 11 01	—	—	—	—	—	—	*NaN*	—
0 11 10	—	—	—	—	—	—	*NaN*	—
0 11 11	—	—	—	—	—	—	*NaN*	—

Solution to Problem 2.48 (page 110)

Hexadecimal value `0x359141` is equivalent to binary [11010110010000101000001]. Shifting this right 21 places gives $1.10101100100001010000001_2 \times 2^{21}$. We form the fraction field by dropping the leading 1 and adding two 0s, giving [10101100100010100000100]. The exponent is formed by adding bias 127 to 21, giving 148 (binary [10010100]). We combine this with a sign field of 0 to give a binary representation

[01001010010101100100010100000100].

We see that the matching bits in the two representations correspond to the low-order bits of the integer, up to the most significant bit equal to 1 matching the high-order 21 bits of the fraction:

```
   0   0   3   5   9   1   4   1
00000000001101011001000101000001
          **********************
    4   A   5   6   4   5   0   4
01001010010101100100010100000100
```

Solution to Problem 2.49 (page 110)

This exercise helps you think about what numbers cannot be represented exactly in floating point.

A. The number has binary representation 1, followed by n 0s, followed by 1, giving value $2^{n+1} + 1$.

B. When $n = 23$, the value is $2^{24} + 1 = 16{,}777{,}217$.

Solution to Problem 2.50 (page 112)

Performing rounding by hand helps reinforce the idea of round-to-even with binary numbers.

Original		Rounded	
10.010_2	$2\frac{1}{4}$	10.0	2
10.011_2	$2\frac{3}{8}$	10.1	$2\frac{1}{2}$
10.110_2	$2\frac{3}{4}$	11.0	3
11.001_2	$3\frac{1}{8}$	11.0	3

Solution to Problem 2.51 (page 112)

A. Looking at the nonterminating sequence for 1/10, we can see that the 2 bits to the right of the rounding position are 1, and so a better approximation to 1/10 would be obtained by incrementing x to get $x' = 0.0001100110011001100110_2$, which is larger than 0.1.

B. We can see that $x' - 0.1$ has binary representation:

$$0.000000000000000000000000[1100].$$

Comparing this to the binary representation of $\frac{1}{10}$, we can see that it is $2^{-22} \times \frac{1}{10}$, which is around 2.38×10^{-8}.

C. $2.38 \times 10^{-8} \times 100 \times 60 \times 60 \times 10 \approx 0.086$ seconds, a factor of 4 less than the error in the Patriot system.

D. $0.343 \times 2000 \approx 171$ meters.

Solution to Problem 2.52 (page 112)

This problem tests a lot of concepts about floating-point representations, including the encoding of normalized and denormalized values, as well as rounding.

Format A		Format B		
Bits	Value	Bits	Value	Comments
011 0000	1	0111 000	1	
101 1110	$\frac{15}{2}$	1001 111	$\frac{15}{2}$	
010 1001	$\frac{25}{32}$	0110 100	$\frac{3}{4}$	Round down
110 1111	$\frac{31}{2}$	1011 000	16	Round up
000 0001	$\frac{1}{64}$	0001 000	$\frac{1}{64}$	Denorm → norm

Solution to Problem 2.53 (page 115)

In general, it is better to use a library macro rather than inventing your own code. This code seems to work on a variety of machines, however.

We assume that the value `1e400` overflows to infinity.

```
#define POS_INFINITY 1e400
#define NEG_INFINITY (-POS_INFINITY)
#define NEG_ZERO (-1.0/POS_INFINITY)
```

Solution to Problem 2.54 (page 117)

Exercises such as this one help you develop your ability to reason about floating-point operations from a programmer's perspective. Make sure you understand each of the answers.

A. `x == (int)(double) x`
Yes, since `double` has greater precision and range than `int`.

B. `x == (int)(float) x`
No. For example, when `x` is *TMax*.

C. `d == (double)(float) d`
No. For example, when `d` is `1e40`, we will get $+\infty$ on the right.

D. `f == (float)(double) f`
Yes, since `double` has greater precision and range than `float`.

E. `f == -(-f)`
Yes, since a floating-point number is negated by simply inverting its sign bit.

F. `1.0/2 == 1/2.0`
Yes, the numerators and denominators will both be converted to floating-point representations before the division is performed.

G. `d*d >= 0.0`
Yes, although it may overflow to $+\infty$.

H. `(f+d)-f == d`
No, for example when `f` is `1.0e20` and `d` is 1.0, the expression `f+d` will be rounded to `1.0e20`, and so the expression on the left-hand side will evaluate to 0.0, while the right-hand side will be 1.0.