limits the range of possible values, and the resulting operations can overflow. We have also seen that the two's-complement representation provides a clever way to represent both negative and positive values, while using the same bit-level implementations as are used to perform unsigned arithmetic—operations such as addition, subtraction, multiplication, and even division have either identical or very similar bit-level behaviors, whether the operands are in unsigned or two's-complement form.

We have seen that some of the conventions in the C language can yield some surprising results, and these can be sources of bugs that are hard to recognize or understand. We have especially seen that the `unsigned` data type, while conceptually straightforward, can lead to behaviors that even experienced programmers do not expect. We have also seen that this data type can arise in unexpected ways—for example, when writing integer constants and when invoking library routines.

**Practice Problem 2.44 (solution page 193)**

Assume data type `int` is 32 bits long and uses a two's-complement representation 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 ($|V| \gg 0$),

**Aside** The IEEE

The Institute of Electrical and Electronics Engineers (IEEE—pronounced “eye-triple-ee”) 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. Another example of an IEEE standard is the 802.11 standard for wireless networking.

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.

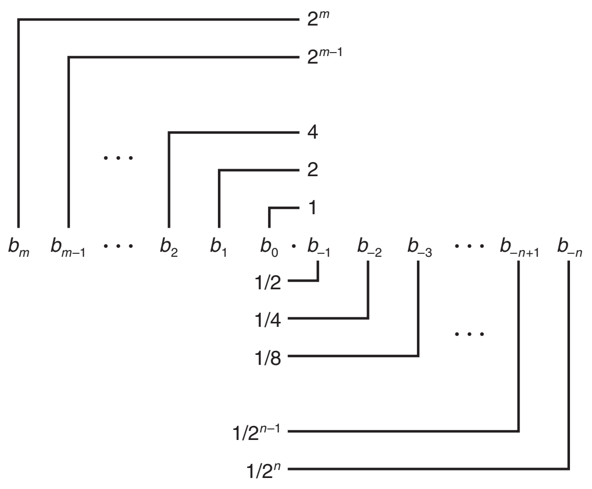
All of this changed around 1985 with the advent of IEEE Standard 754, a carefully crafted standard for representing floating-point numbers and the operations performed on them. This effort started in 1976 under Intel’s sponsorship with the design of the 8087, a chip that provided floating-point support for the 8086 processor. Intel 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.

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

$$d_m \, d_{m-1} \cdots d_1 \, d_0 \, . \, d_{-1} \, d_{-2} \cdots d_{-n}$$



$2^m$
$2^{m-1}$
4
2
1
$b_m$ $b_{m-1}$ · · · $b_2$ $b_1$ $b_0$ . $b_{-1}$ $b_{-2}$ $b_{-3}$ · · · $b_{-n+1}$ $b_{-n}$
1/2
1/4
1/8
$1/2^{n-1}$
$1/2^n$

**Figure 2.31**
**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 $1/2^i$.

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 nonnegative 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 2.31. This notation represents a number $b$ defined as

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

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

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 $0.11 \cdots 1_2$ 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 (solution page 193)

Fill in the missing information in the following table:

| Fractional value | Binary representation | Decimal representation |
|---|---|---|
| $\frac{1}{8}$ | 0.001 | 0.125 |
| $\frac{3}{4}$ | ________ | ________ |
| $\frac{5}{16}$ | ________ | ________ |
| ________ | 10.1011 | ________ |
| ________ | 1.001 | ________ |
| ________ | ________ | 5.875 |
| ________ | ________ | 3.1875 |

## Practice Problem 2.46 (solution page 194)

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 US General

Accounting Office (GAO) conducted a detailed analysis of the failure [76] 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 2,000 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 [103].

### 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 $5 \times 2^{100}$ 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.

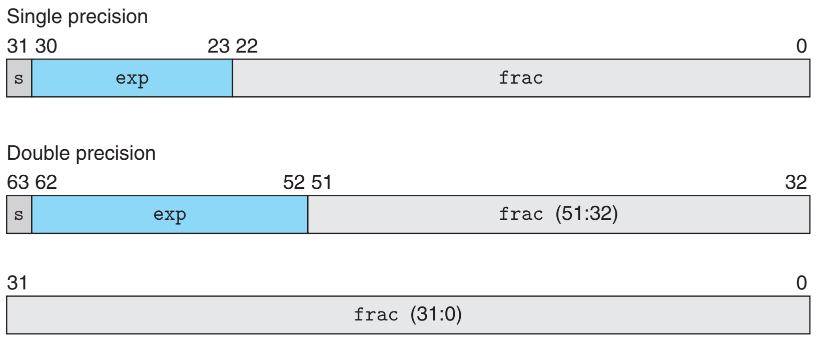


**Figure 2.32 Standard floating-point formats.** Floating-point numbers are represented by three fields. For the two most common formats, these are packed in 32-bit (single-precision) or 64-bit (double-precision) words.

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 2.32 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 2.33 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 $E = e - \mathit{Bias}$, where $e$ is the unsigned number having bit representation $e_{k-1} \cdots e_1 e_0$ and *Bias* is a bias 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 $0 \le f < 1$, having binary representation $0.f_{n-1} \cdots f_1 f_0$, that is, with the

**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.

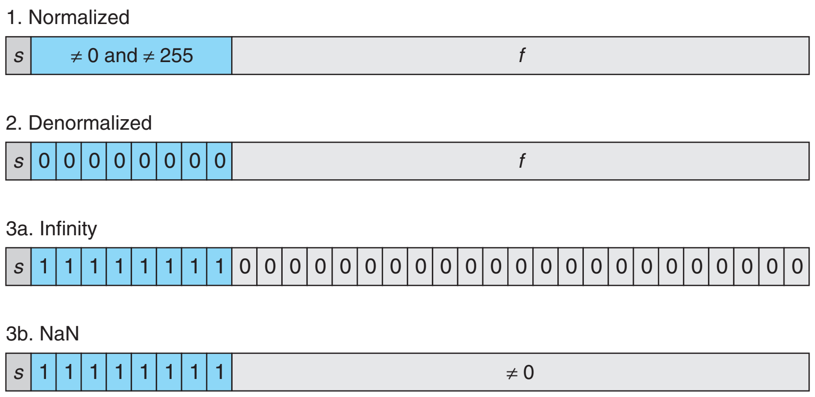


**Figure 2.33 Categories of single-precision floating-point values.** The value of the exponent determines whether the number is (1) normalized, (2) denormalized, or (3) a special value.

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.

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.

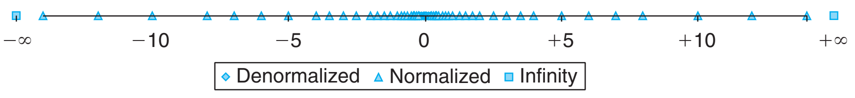
### Case 3: Special Values

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.

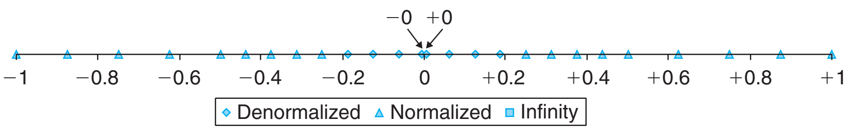
## 2.4.3 Example Numbers

Figure 2.34 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 $\pm 14$. 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 2.35 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 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



(a) Complete range



(b) Values between −1.0 and +1.0

**Figure 2.34 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 positive | 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 denormalized | 0 0000 111 | 0 | −6 | $\frac{1}{64}$ | $\frac{7}{8}$ | $\frac{7}{8}$ | $\frac{7}{512}$ | $\frac{7}{512}$ | 0.013672 |
| Smallest normalized | 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 normalized | 0 1110 111 | 14 | 7 | 128 | $\frac{7}{8}$ | $\frac{15}{8}$ | $\frac{1920}{8}$ | 240 | 240.0 |
| Infinity | 0 1111 000 | — | — | — | — | — | — | $\infty$ | — |

**Figure 2.35 Example nonnegative values for 8-bit floating-point format.** There are $k = 4$ exponent bits and $n = 3$ fraction bits. The bias is 7.

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 2.35 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 occur in descending order, but this can be overcome without requiring floating-point operations to perform comparisons (see Problem 2.84).

### Practice Problem 2.47 (solution page 194)

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 |

| Bits | $e$ | $E$ | $2^E$ | $f$ | $M$ | $2^E \times M$ | $V$ | Decimal |
|---|---|---|---|---|---|---|---|---|
| 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 2.36 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 2.35, 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.
- 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

| | | | Single precision | | Double precision | |
|---|---|---|---|---|---|---|
| Description | `exp` | `frac` | Value | Decimal | Value | Decimal |
| Zero | $00 \cdots 00$ | $0 \cdots 00$ | $0$ | $0.0$ | $0$ | $0.0$ |
| Smallest denormalized | $00 \cdots 00$ | $0 \cdots 01$ | $2^{-23} \times 2^{-126}$ | $1.4 \times 10^{-45}$ | $2^{-52} \times 2^{-1022}$ | $4.9 \times 10^{-324}$ |
| Largest denormalized | $00 \cdots 00$ | $1 \cdots 11$ | $(1-\epsilon) \times 2^{-126}$ | $1.2 \times 10^{-38}$ | $(1-\epsilon) \times 2^{-1022}$ | $2.2 \times 10^{-308}$ |
| Smallest normalized | $00 \cdots 01$ | $0 \cdots 00$ | $1 \times 2^{-126}$ | $1.2 \times 10^{-38}$ | $1 \times 2^{-1022}$ | $2.2 \times 10^{-308}$ |
| One | $01 \cdots 11$ | $0 \cdots 00$ | $1 \times 2^{0}$ | $1.0$ | $1 \times 2^{0}$ | $1.0$ |
| Largest normalized | $11 \cdots 10$ | $1 \cdots 11$ | $(2-\epsilon) \times 2^{127}$ | $3.4 \times 10^{38}$ | $(2-\epsilon) \times 2^{1023}$ | $1.8 \times 10^{308}$ |

**Figure 2.36 Examples of nonnegative floating-point numbers.**

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 2.15 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 $12{,}345 = 1.1000000111001_2 \times 2^{13}$. 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 2.1.3 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 (solution page 195)

As mentioned in Problem 2.6, the integer 3,510,593 has hexadecimal representation `0x00359141`, while the single-precision floating-point number 3,510,593.0 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** **(solution page 195)**

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 2.37 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 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

| 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 2.37 **Illustration of rounding modes for dollar rounding.** The first rounds to a nearest value, while the other three bound the result above or below.

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** **(solution page 195)**

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. $10.111_2$

B. $11.010_2$

C. $11.000_2$

D. $10.110_2$

### Practice Problem 2.51 (solution page 195)

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 (solution page 196)

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 $\mathit{Round}(x \odot y)$, 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 $-0$, $\infty$, or $\mathit{NaN}$, 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 $x +^{\mathrm{f}} y$ to be $\mathit{Round}(x + y)$. 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 $x +^{\mathrm{f}} y = y +^{\mathrm{f}} x$ 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, $x +^{\mathrm{f}} -x = 0$. The exceptions are infinities (since $+\infty - \infty = \mathit{NaN}$), and $\mathit{NaN}$s, since $\mathit{NaN} +^{\mathrm{f}} x = \mathit{NaN}$ 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 $x +^{\mathrm{f}} a \geq x +^{\mathrm{f}} b$ 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*:

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

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 three-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>
```

## Practice Problem 2.53 (solution page 196)

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 $1.8 \times 10^{308}$.

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

- 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 $[10 \cdots 00]$ ($TMin_w$ 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.

## Practice Problem 2.54 (solution page 196)

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. Machines with 64-bit word sizes have become increasingly common, replacing the 32-bit machines that dominated the market for around 30 years. Because 64-bit machines can also run programs compiled for 32-bit machines, we have focused on the distinction between 32- and 64-bit programs, rather than machines. The advantage of 64-bit programs is that they can go beyond the 4 GB address limitation of 32-bit programs.

Most machines encode signed numbers using a two's-complement representation and encode floating-point numbers using IEEE Standard 754. 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