2.1.7 Bit-Level Operations in C

One useful feature of C is that it supports bitwise Boolean operations. In fact, the symbols we have used for the Boolean operations are exactly those used by C: | for OR, & for AND, ~ for NOT, and ^ for EXCLUSIVE-OR. These can be applied to any "integral" data type, including all of those listed in Figure 2.3. Here are some examples of expression evaluation for data type `char`:

C expression	Binary expression	Binary result	Hexadecimal result
`~0x41`	~[0100 0001]	[1011 1110]	`0xBE`
`~0x00`	~[0000 0000]	[1111 1111]	`0xFF`
`0x69 & 0x55`	[0110 1001] & [0101 0101]	[0100 0001]	`0x41`
`0x69 | 0x55`	[0110 1001] | [0101 0101]	[0111 1101]	`0x7D`

As our examples show, the best way to determine the effect of a bit-level expression is to expand the hexadecimal arguments to their binary representations, perform the operations in binary, and then convert back to hexadecimal.

Practice Problem 2.10 (solution page 146)

As an application of the property that $a \hat{} a = 0$ for any bit vector a, consider the following program:

```
1   void inplace_swap(int *x, int *y) {
2       *y = *x ^ *y;  /* Step 1 */
3       *x = *x ^ *y;  /* Step 2 */
4       *y = *x ^ *y;  /* Step 3 */
5   }
```

As the name implies, we claim that the effect of this procedure is to swap the values stored at the locations denoted by pointer variables `x` and `y`. Note that unlike the usual technique for swapping two values, we do not need a third location to temporarily store one value while we are moving the other. There is no performance advantage to this way of swapping; it is merely an intellectual amusement.

Starting with values a and b in the locations pointed to by `x` and `y`, respectively, fill in the table that follows, giving the values stored at the two locations after each step of the procedure. Use the properties of ^ to show that the desired effect is achieved. Recall that every element is its own additive inverse (that is, $a \hat{} a = 0$).

Step	*x	*y
Initially	a	b
Step 1	_____	_____
Step 2	_____	_____
Step 3	_____	_____

Practice Problem 2.11 (solution page 146)

Armed with the function `inplace_swap` from Problem 2.10, you decide to write code that will reverse the elements of an array by swapping elements from opposite ends of the array, working toward the middle.

You arrive at the following function:

```
1  void reverse_array(int a[], int cnt) {
2      int first, last;
3      for (first = 0, last = cnt-1;
4           first <= last;
5           first++,last--)
6           inplace_swap(&a[first], &a[last]);
7  }
```

When you apply your function to an array containing elements 1, 2, 3, and 4, you find the array now has, as expected, elements 4, 3, 2, and 1. When you try it on an array with elements 1, 2, 3, 4, and 5, however, you are surprised to see that the array now has elements 5, 4, 0, 2, and 1. In fact, you discover that the code always works correctly on arrays of even length, but it sets the middle element to 0 whenever the array has odd length.

A. For an array of odd length $\texttt{cnt} = 2k + 1$, what are the values of variables `first` and `last` in the final iteration of function `reverse_array`?

B. Why does this call to function `inplace_swap` set the array element to 0?

C. What simple modification to the code for `reverse_array` would eliminate this problem?

One common use of bit-level operations is to implement *masking* operations, where a mask is a bit pattern that indicates a selected set of bits within a word. As an example, the mask `0xFF` (having ones for the least-significant 8 bits) indicates the low-order byte of a word. The bit-level operation `x & 0xFF` yields a value consisting of the least significant byte of `x`, but with all other bytes set to 0. For example, with `x = 0x89ABCDEF`, the expression would yield `0x000000EF`. The expression `~0` will yield a mask of all ones, regardless of the size of the data representation. The same mask can be written `0xFFFFFFFF` when data type `int` is 32 bits, but it would not be as portable.

Practice Problem 2.12 (solution page 146)

Write C expressions, in terms of variable `x`, for the following values. Your code should work for any word size $w \geq 8$. For reference, we show the result of evaluating the expressions for `x = 0x87654321`, with $w = 32$.

A. The least significant byte of `x`, with all other bits set to 0. [`0x00000021`]

B. All but the least significant byte of `x` complemented, with the least significant byte left unchanged. [`0x789ABC21`]

C. The least significant byte set to all ones, and all other bytes of x left unchanged. [0x876543FF]

Practice Problem 2.13 (solution page 147)

The Digital Equipment VAX computer was a very popular machine from the late 1970s until the late 1980s. Rather than instructions for Boolean operations AND and OR, it had instructions `bis` (bit set) and `bic` (bit clear). Both instructions take a data word `x` and a mask word `m`. They generate a result `z` consisting of the bits of `x` modified according to the bits of `m`. With `bis`, the modification involves setting `z` to 1 at each bit position where `m` is 1. With `bic`, the modification involves setting `z` to 0 at each bit position where `m` is 1.

To see how these operations relate to the C bit-level operations, assume we have functions `bis` and `bic` implementing the bit set and bit clear operations, and that we want to use these to implement functions computing bitwise operations | and ^, without using any other C operations. Fill in the missing code below. *Hint:* Write C expressions for the operations `bis` and `bic`.

```
/* 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 = _____;
    return result;
}

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

2.1.8 Logical Operations in C

C also provides a set of *logical* operators ||, &&, and !, which correspond to the OR, AND, and NOT operations of logic. These can easily be confused with the bit-level operations, but their behavior is quite different. The logical operations treat any nonzero argument as representing TRUE and argument 0 as representing FALSE. They return either 1 or 0, indicating a result of either TRUE or FALSE, respectively. Here are some examples of expression evaluation:

Expression	Result
`!0x41`	`0x00`
`!0x00`	`0x01`
`!!0x41`	`0x01`
`0x69 && 0x55`	`0x01`
`0x69 || 0x55`	`0x01`

Observe that a bitwise operation will have behavior matching that of its logical counterpart only in the special case in which the arguments are restricted to 0 or 1.

A second important distinction between the logical operators '`&&`' and '`||`' versus their bit-level counterparts '`&`' and '`|`' is that the logical operators do not evaluate their second argument if the result of the expression can be determined by evaluating the first argument. Thus, for example, the expression `a && 5/a` will never cause a division by zero, and the expression `p && *p++` will never cause the dereferencing of a null pointer.

Practice Problem 2.14 (solution page 147)

Suppose that `x` and `y` have byte values `0x66` and `0x39`, respectively. Fill in the following table indicating the byte values of the different C expressions:

Expression	Value	Expression	Value
`x & y`	_____	`x && y`	_____
`x | y`	_____	`x || y`	_____
`~x | ~y`	_____	`!x || !y`	_____
`x & !y`	_____	`x && ~y`	_____

Practice Problem 2.15 (solution page 148)

Using only bit-level and logical operations, write a C expression that is equivalent to `x == y`. In other words, it will return 1 when `x` and `y` are equal and 0 otherwise.

2.1.9 Shift Operations in C

C also provides a set of *shift* operations for shifting bit patterns to the left and to the right. For an operand `x` having bit representation $[x_{w-1}, x_{w-2}, \ldots, x_0]$, the C expression `x << k` yields a value with bit representation $[x_{w-k-1}, x_{w-k-2}, \ldots, x_0, 0, \ldots, 0]$. That is, `x` is shifted k bits to the left, dropping off the k most significant bits and filling the right end with k zeros. The shift amount should be a value between 0 and $w - 1$. Shift operations associate from left to right, so `x << j << k` is equivalent to `(x << j) << k`.

There is a corresponding right shift operation, written in C as `x >> k`, but it has a slightly subtle behavior. Generally, machines support two forms of right shift:

Logical. A logical right shift fills the left end with k zeros, giving a result $[0, \ldots, 0, x_{w-1}, x_{w-2}, \ldots x_k]$.

Arithmetic. An arithmetic right shift fills the left end with k repetitions of the most significant bit, giving a result $[x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots x_k]$. This convention might seem peculiar, but as we will see, it is useful for operating on signed integer data.

As examples, the following table shows the effect of applying the different shift operations to two different values of an 8-bit argument x:

Operation	Value 1	Value 2
Argument x	[01100011]	[10010101]
x << 4	[0011*0000*]	[1001*0000*]
x >> 4 (logical)	[*0000*0110]	[*0000*1001]
x >> 4 (arithmetic)	[*0000*0110]	[*1111*1001]

The italicized digits indicate the values that fill the right (left shift) or left (right shift) ends. Observe that all but one entry involves filling with zeros. The exception is the case of shifting [10010101] right arithmetically. Since its most significant bit is 1, this will be used as the fill value.

The C standards do not precisely define which type of right shift should be used with signed numbers—either arithmetic or logical shifts may be used. This unfortunately means that any code assuming one form or the other will potentially encounter portability problems. In practice, however, almost all compiler/machine combinations use arithmetic right shifts for signed data, and many programmers assume this to be the case. For unsigned data, on the other hand, right shifts must be logical.

In contrast to C, Java has a precise definition of how right shifts should be performed. The expression `x >> k` shifts `x` arithmetically by `k` positions, while `x >>> k` shifts it logically.

Practice Problem 2.16 (solution page 148)

Fill in the table below showing the effects of the different shift operations on single-byte quantities. The best way to think about shift operations is to work with binary representations. Convert the initial values to binary, perform the shifts, and then convert back to hexadecimal. Each of the answers should be 8 binary digits or 2 hexadecimal digits.

x		x << 3		Logical x >> 2		Arithmetic x >> 2	
Hex	Binary	Binary	Hex	Binary	Hex	Binary	Hex
0xC3	_____	_____	_____	_____	_____	_____	_____
0x75	_____	_____	_____	_____	_____	_____	_____
0x87	_____	_____	_____	_____	_____	_____	_____
0x66	_____	_____	_____	_____	_____	_____	_____

Aside Shifting by k, for large values of k

For a data type consisting of w bits, what should be the effect of shifting by some value $k \geq w$? For example, what should be the effect of computing the following expressions, assuming data type `int` has $w = 32$:

```
int      lval = 0xFEDCBA98 << 32;
int      aval = 0xFEDCBA98 >> 36;
unsigned uval = 0xFEDCBA98u >> 40;
```

The C standards carefully avoid stating what should be done in such a case. On many machines, the shift instructions consider only the lower $\log_2 w$ bits of the shift amount when shifting a w-bit value, and so the shift amount is computed as $k \bmod w$. For example, with $w = 32$, the above three shifts would be computed as if they were by amounts 0, 4, and 8, respectively, giving results

```
lval    0xFEDCBA98
aval    0xFFEDCBA9
uval    0x00FEDCBA
```

This behavior is not guaranteed for C programs, however, and so shift amounts should be kept less than the word size.

Java, on the other hand, specifically requires that shift amounts should be computed in the modular fashion we have shown.

Aside Operator precedence issues with shift operations

It might be tempting to write the expression `1<<2 + 3<<4`, intending it to mean `(1<<2) + (3<<4)`. However, in C the former expression is equivalent to `1 << (2+3) << 4`, since addition (and subtraction) have higher precedence than shifts. The left-to-right associativity rule then causes this to be parenthesized as `(1 << (2+3)) << 4`, giving value 512, rather than the intended 52.

Getting the precedence wrong in C expressions is a common source of program errors, and often these are difficult to spot by inspection. When in doubt, put in parentheses!

2.2 Integer Representations

In this section, we describe two different ways bits can be used to encode integers—one that can only represent nonnegative numbers, and one that can represent negative, zero, and positive numbers. We will see later that they are strongly related both in their mathematical properties and their machine-level implementations. We also investigate the effect of expanding or shrinking an encoded integer to fit a representation with a different length.

Figure 2.8 lists the mathematical terminology we introduce to precisely define and characterize how computers encode and operate on integer data. This

Symbol	Type	Meaning	Page
$B2T_w$	Function	Binary to two's complement	64
$B2U_w$	Function	Binary to unsigned	62
$U2B_w$	Function	Unsigned to binary	64
$U2T_w$	Function	Unsigned to two's complement	71
$T2B_w$	Function	Two's complement to binary	65
$T2U_w$	Function	Two's complement to unsigned	71
$TMin_w$	Constant	Minimum two's-complement value	65
$TMax_w$	Constant	Maximum two's-complement value	65
$UMax_w$	Constant	Maximum unsigned value	63
$+^t_w$	Operation	Two's-complement addition	90
$+^u_w$	Operation	Unsigned addition	85
$*^t_w$	Operation	Two's-complement multiplication	97
$*^u_w$	Operation	Unsigned multiplication	96
$-^t_w$	Operation	Two's-complement negation	95
$-^u_w$	Operation	Unsigned negation	89

Figure 2.8 Terminology for integer data and arithmetic operations. The subscript w denotes the number of bits in the data representation. The "Page" column indicates the page on which the term is defined.

terminology will be introduced over the course of the presentation. The figure is included here as a reference.

2.2.1 Integral Data Types

C supports a variety of *integral* data types—ones that represent finite ranges of integers. These are shown in Figures 2.9 and 2.10, along with the ranges of values they can have for "typical" 32- and 64-bit programs. Each type can specify a size with keyword `char`, `short`, `long`, as well as an indication of whether the represented numbers are all nonnegative (declared as `unsigned`), or possibly negative (the default.) As we saw in Figure 2.3, the number of bytes allocated for the different sizes varies according to whether the program is compiled for 32 or 64 bits. Based on the byte allocations, the different sizes allow different ranges of values to be represented. The only machine-dependent range indicated is for size designator `long`. Most 64-bit programs use an 8-byte representation, giving a much wider range of values than the 4-byte representation used with 32-bit programs.

One important feature to note in Figures 2.9 and 2.10 is that the ranges are not symmetric—the range of negative numbers extends one further than the range of positive numbers. We will see why this happens when we consider how negative numbers are represented.

C data type	Minimum	Maximum
[signed] char	−128	127
unsigned char	0	255
short	−32,768	32,767
unsigned short	0	65,535
int	−2,147,483,648	2,147,483,647
unsigned	0	4,294,967,295
long	−2,147,483,648	2,147,483,647
unsigned long	0	4,294,967,295
int32_t	−2,147,483,648	2,147,483,647
uint32_t	0	4,294,967,295
int64_t	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	0	18,446,744,073,709,551,615

Figure 2.9 Typical ranges for C integral data types for 32-bit programs.

C data type	Minimum	Maximum
[signed] char	−128	127
unsigned char	0	255
short	−32,768	32,767
unsigned short	0	65,535
int	−2,147,483,648	2,147,483,647
unsigned	0	4,294,967,295
long	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long	0	18,446,744,073,709,551,615
int32_t	−2,147,483,648	2,147,483,647
uint32_t	0	4,294,967,295
int64_t	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	0	18,446,744,073,709,551,615

Figure 2.10 Typical ranges for C integral data types for 64-bit programs.

The C standards define minimum ranges of values that each data type must be able to represent. As shown in Figure 2.11, their ranges are the same or smaller than the typical implementations shown in Figures 2.9 and 2.10. In particular, with the exception of the fixed-size data types, we see that they require only a

New to C? Signed and unsigned numbers in C, C++, and Java

Both C and C++ support signed (the default) and unsigned numbers. Java supports only signed numbers.

C data type	Minimum	Maximum
`[signed] char`	−127	127
`unsigned char`	0	255
`short`	−32,767	32,767
`unsigned short`	0	65,535
`int`	−32,767	32,767
`unsigned`	0	65,535
`long`	−2,147,483,647	2,147,483,647
`unsigned long`	0	4,294,967,295
`int32_t`	−2,147,483,648	2,147,483,647
`uint32_t`	0	4,294,967,295
`int64_t`	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
`uint64_t`	0	18,446,744,073,709,551,615

Figure 2.11 Guaranteed ranges for C integral data types. The C standards require that the data types have at least these ranges of values.

symmetric range of positive and negative numbers. We also see that data type `int` could be implemented with 2-byte numbers, although this is mostly a throwback to the days of 16-bit machines. We also see that size `long` can be implemented with 4-byte numbers, and it typically is for 32-bit programs. The fixed-size data types guarantee that the ranges of values will be exactly those given by the typical numbers of Figure 2.9, including the asymmetry between negative and positive.

2.2.2 Unsigned Encodings

Let us consider an integer data type of w bits. We write a bit vector as either $\vec{x}$, to denote the entire vector, or as $[x_{w-1}, x_{w-2}, \ldots, x_0]$ to denote the individual bits within the vector. Treating $\vec{x}$ as a number written in binary notation, we obtain the *unsigned* interpretation of $\vec{x}$. In this encoding, each bit x_i has value 0 or 1, with the latter case indicating that value 2^i should be included as part of the numeric value. We can express this interpretation as a function $B2U_w$ (for "binary to unsigned," length w):

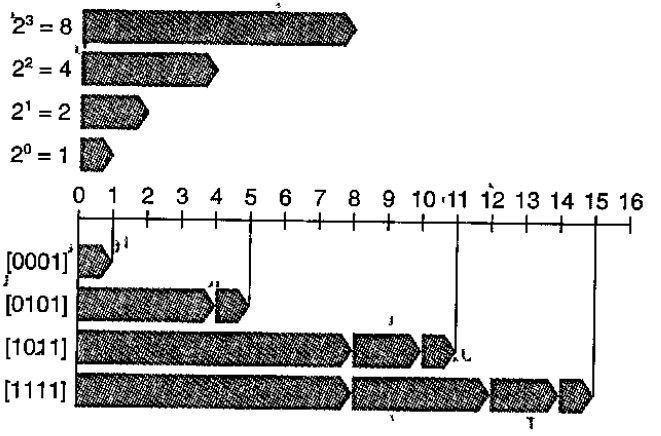


Figure 2.12 Unsigned number examples for $w = 4$. When bit i in the binary representation has value 1, it contributes 2^i to the value.

PRINCIPLE: Definition of unsigned encoding

For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$:

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i \tag{2.1}$$

■

In this equation, the notation $\doteq$ means that the left-hand side is defined to be equal to the right-hand side. The function $B2U_w$ maps strings of zeros and ones of length w to nonnegative integers. As examples, Figure 2.12 shows the mapping, given by $B2U$, from bit vectors to integers for the following cases:

$$\begin{aligned}
B2U_4([0001]) &= 0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 0+0+0+1 = 1\\
B2U_4([0101]) &= 0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 0+4+0+1 = 5\\
B2U_4([1011]) &= 1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 8+0+2+1 = 11\\
B2U_4([1111]) &= 1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 8+4+2+1 = 15
\end{aligned} \tag{2.2}$$

In the figure, we represent each bit position i by a rightward-pointing blue bar of length 2^i. The numeric value associated with a bit vector then equals the sum of the lengths of the bars for which the corresponding bit values are 1.

Let us consider the range of values that can be represented using w bits. The least value is given by bit vector $[00\cdots0]$ having integer value 0, and the greatest value is given by bit vector $[11\cdots1]$ having integer value $UMax_w \doteq \sum_{i=0}^{w-1} 2^i = 2^w - 1$. Using the 4-bit case as an example, we have $UMax_4 = B2U_4([1111]) = 2^4 - 1 = 15$. Thus, the function $B2U_w$ can be defined as a mapping $B2U_w$: $\{0, 1\}^w \rightarrow \{0, \ldots, UMax_w\}$.

The unsigned binary representation has the important property that every number between 0 and $2^w - 1$ has a unique encoding as a w-bit value. For example,

there is only one representation of decimal value 11 as an unsigned 4-bit number—namely, [1011]. We highlight this as a mathematical principle, which we first state and then explain.

PRINCIPLE: Uniqueness of unsigned encoding

Function $B2U_w$ is a bijection. ∎

The mathematical term *bijection* refers to a function f that goes two ways: it maps a value x to a value y where $y = f(x)$, but it can also operate in reverse, since for every y, there is a unique value x such that $f(x) = y$. This is given by the *inverse* function f^{-1}, where, for our example, $x = f^{-1}(y)$. The function $B2U_w$ maps each bit vector of length w to a unique number between 0 and $2^w - 1$, and it has an inverse, which we call $U2B_w$ (for "unsigned to binary"), that maps each number in the range 0 to $2^w - 1$ to a unique pattern of w bits.

2.2.3 Two's-Complement Encodings

For many applications, we wish to represent negative values as well. The most common computer representation of signed numbers is known as *two's-complement* form. This is defined by interpreting the most significant bit of the word to have negative weight. We express this interpretation as a function $B2T_w$ (for "binary to two's complement" length w):

PRINCIPLE: Definition of two's-complement encoding

For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$:

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \tag{2.3}$$

∎

The most significant bit x_{w-1} is also called the *sign bit*. Its "weight" is -2^{w-1}, the negation of its weight in an unsigned representation. When the sign bit is set to 1, the represented value is negative, and when set to 0, the value is nonnegative. As examples, Figure 2.13 shows the mapping, given by $B2T$, from bit vectors to integers for the following cases:

$$\begin{aligned}
B2T_4([0001]) &= -0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 &&= 0+0+0+1 &&= 1\\
B2T_4([0101]) &= -0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 &&= 0+4+0+1 &&= 5\\
B2T_4([1011]) &= -1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 &&= -8+0+2+1 &&= -5\\
B2T_4([1111]) &= -1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 &&= -8+4+2+1 &&= -1
\end{aligned} \tag{2.4}$$

In the figure, we indicate that the sign bit has negative weight by showing it as a leftward-pointing gray bar. The numeric value associated with a bit vector is then given by the combination of the possible leftward-pointing gray bar and the rightward-pointing blue bars.

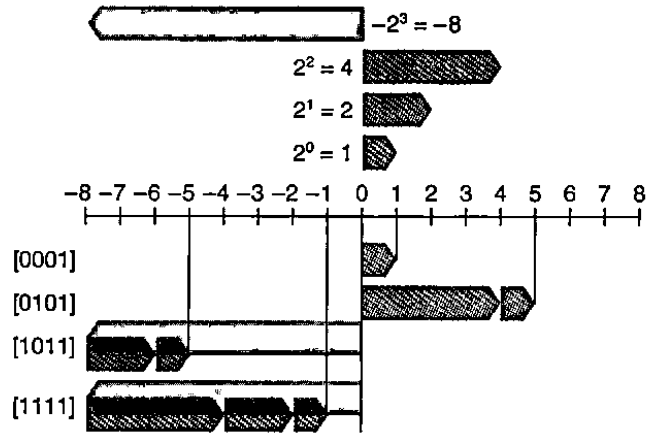


Figure 2.13 Two's-complement number examples for $w = 4$. Bit 3 serves as a sign bit; when set to 1, it contributes $-2^3 = -8$ to the value. This weighting is shown as a leftward-pointing gray bar.

We see that the bit patterns are identical for Figures 2.12 and 2.13 (as well as for Equations 2.2 and 2.4), but the values differ when the most significant bit is 1, since in one case it has weight +8, and in the other case it has weight −8.

Let us consider the range of values that can be represented as a w-bit two's-complement number. The least representable value is given by bit vector $[10\cdots0]$ (set the bit with negative weight but clear all others), having integer value $TMin_w \doteq -2^{w-1}$. The greatest value is given by bit vector $[01\cdots1]$ (clear the bit with negative weight but set all others), having integer value $TMax_w \doteq \sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$. Using the 4-bit case as an example, we have $TMin_4 = B2T_4([1000]) = -2^3 = -8$ and $TMax_4 = B2T_4([0111]) = 2^2 + 2^1 + 2^0 = 4 + 2 + 1 = 7$.

We can see that $B2T_w$ is a mapping of bit patterns of length w to numbers between $TMin_w$ and $TMax_w$, written as $B2T_w$: $\{0, 1\}^w \rightarrow \{TMin_w, \ldots, TMax_w\}$. As we saw with the unsigned representation, every number within the representable range has a unique encoding as a w-bit two's-complement number. This leads to a principle for two's-complement numbers similar to that for unsigned numbers:

PRINCIPLE: Uniqueness of two's-complement encoding

Function $B2T_w$ is a bijection. ■

We define function $T2B_w$ (for "two's complement to binary") to be the inverse of $B2T_w$. That is, for a number x, such that $TMin_w \leq x \leq TMax_w$, $T2B_w(x)$ is the (unique) w-bit pattern that encodes x.

Practice Problem 2.17 (solution page 148)

Assuming $w = 4$, we can assign a numeric value to each possible hexadecimal digit, assuming either an unsigned or a two's-complement interpretation. Fill in the following table according to these interpretations by writing out the nonzero powers of 2 in the summations shown in Equations 2.1 and 2.3:

$\vec{x}$			
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			
0x5			
0x8			
0xD			
0xF			

Figure 2.14 shows the bit patterns and numeric values for several important numbers for different word sizes. The first three give the ranges of representable integers in terms of the values of $UMax_w$, $TMin_w$, and $TMax_w$. We will refer to these three special values often in the ensuing discussion. We will drop the subscript w and refer to the values $UMax$, $TMin$, and $TMax$ when w can be inferred from context or is not central to the discussion.

A few points are worth highlighting about these numbers. First, as observed in Figures 2.9 and 2.10, the two's-complement range is asymmetric: $|TMin| = |TMax| + 1$; that is, there is no positive counterpart to $TMin$. As we shall see, this leads to some peculiar properties of two's-complement arithmetic and can be the source of subtle program bugs. This asymmetry arises because half the bit patterns (those with the sign bit set to 1) represent negative numbers, while half (those with the sign bit set to 0) represent nonnegative numbers. Since 0 is nonnegative, this means that it can represent one less positive number than negative. Second, the maximum unsigned value is just over twice the maximum two's-complement value: $UMax = 2TMax + 1$. All of the bit patterns that denote negative numbers in two's-complement notation become positive values in an unsigned representation.

	Word size w			
Value	8	16	32	64
$UMax_w$	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
	255	65,535	4,294,967,295	18,446,744,073,709,551,615
$TMin_w$	0x80	0x8000	0x80000000	0x8000000000000000
	−128	−32,768	−2,147,483,648	−9,223,372,036,854,775,808
$TMax_w$	0x7F	0x7FFF	0x7FFFFFFF	0x7FFFFFFFFFFFFFFF
	127	32,767	2,147,483,647	9,223,372,036,854,775,807
−1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
0	0x00	0x0000	0x00000000	0x0000000000000000

Figure 2.14 Important numbers. Both numeric values and hexadecimal representations are shown.

Aside More on fixed-size integer types

For some programs, it is essential that data types be encoded using representations with specific sizes. For example, when writing programs to enable a machine to communicate over the Internet according to a standard protocol, it is important to have data types compatible with those specified by the protocol. We have seen that some C data types, especially `long`, have different ranges on different machines, and in fact the C standards only specify the minimum ranges for any data type, not the exact ranges. Although we can choose data types that will be compatible with standard representations on most machines, there is no guarantee of portability.

We have already encountered the 32- and 64-bit versions of fixed-size integer types (Figure 2.3); they are part of a larger class of data types. The ISO C99 standard introduces this class of integer types in the file `stdint.h`. This file defines a set of data types with declarations of the form `int`*N*`_t` and `uint`*N*`_t`, specifying *N*-bit signed and unsigned integers, for different values of *N*. The exact values of *N* are implementation dependent, but most compilers allow values of 8, 16, 32, and 64. Thus, we can unambiguously declare an unsigned 16-bit variable by giving it type `uint16_t`, and a signed variable of 32 bits as `int32_t`.

Along with these data types are a set of macros defining the minimum and maximum values for each value of *N*. These have names of the form `INT`*N*`_MIN`, `INT`*N*`_MAX`, and `UINT`*N*`_MAX`.

Formatted printing with fixed-width types requires use of macros that expand into format strings in a system-dependent manner. So, for example, the values of variables `x` and `y` of type `int32_t` and `uint64_t` can be printed by the following call to `printf`:

```
printf("x = %" PRId32 ", y = %" PRIu64 "\n", x, y);
```

When compiled as a 64-bit program, macro `PRId32` expands to the string `"d"`, while `PRIu64` expands to the pair of strings `"l" "u"`. When the C preprocessor encounters a sequence of string constants separated only by spaces (or other whitespace characters), it concatenates them together. Thus, the above call to `printf` becomes

```
printf("x = %d, y = %lu\n", x, y);
```

Using the macros ensures that a correct format string will be generated regardless of how the code is compiled.

Figure 2.14 also shows the representations of constants -1 and 0. Note that -1 has the same bit representation as *UMax*—a string of all ones. Numeric value 0 is represented as a string of all zeros in both representations.

The C standards do not require signed integers to be represented in two's-complement form, but nearly all machines do so. Programmers who are concerned with maximizing portability across all possible machines should not assume any particular range of representable values, beyond the ranges indicated in Figure 2.11, nor should they assume any particular representation of signed numbers. On the other hand, many programs are written assuming a two's-complement representation of signed numbers, and the "typical" ranges shown in Figures 2.9 and 2.10, and these programs are portable across a broad range of machines and compilers. The file `<limits.h>` in the C library defines a set of constants

Aside Alternative representations of signed numbers

There are two other standard representations for signed numbers:

Ones' complement. This is the same as two's complement, except that the most significant bit has weight $-(2^{w-1} - 1)$ rather than -2^{w-1}.

$$B2O_w(\vec{x}) \doteq -x_{w-1}(2^{w-1} - 1) + \sum_{i=0}^{w-2} x_i 2^i$$

Sign magnitude. The most significant bit is a sign bit that determines whether the remaining bits should be given negative or positive weight:

$$B2S_w(\vec{x}) \doteq (-1)^{x_{w-1}} \cdot \left(\sum_{i=0}^{w-2} x_i 2^i \right)$$

Both of these representations have the curious property that there are two different encodings of the number 0. For both representations, $[00\cdots0]$ is interpreted as $+0$. The value -0 can be represented in sign-magnitude form as $[10\cdots0]$ and in ones' complement as $[11\cdots1]$. Although machines based on ones'-complement representations were built in the past, almost all modern machines use two's complement. We will see that sign-magnitude encoding is used with floating-point numbers.

Note the different position of apostrophes: *two's* complement versus *ones'* complement. The term "two's complement" arises from the fact that for nonnegative x we compute a w-bit representation of $-x$ as $2^w - x$ (a single two). The term "ones' complement" comes from the property that we can compute $-x$ in this notation as $[111\cdots1] - x$ (multiple ones).

delimiting the ranges of the different integer data types for the particular machine on which the compiler is running. For example, it defines constants `INT_MAX`, `INT_MIN`, and `UINT_MAX` describing the ranges of signed and unsigned integers. For a two's-complement machine in which data type `int` has w bits, these constants correspond to the values of $TMax_w$, $TMin_w$, and $UMax_w$.

The Java standard is quite specific about integer data type ranges and representations. It requires a two's-complement representation with the exact ranges shown for the 64-bit case (Figure 2.10). In Java, the single-byte data type is called `byte` instead of `char`. These detailed requirements are intended to enable Java programs to behave identically regardless of the machines or operating systems running them.

To get a better understanding of the two's-complement representation, consider the following code example:

```
1       short x = 12345;
2       short mx = -x;
3
4       show_bytes((byte_pointer) &x, sizeof(short));
5       show_bytes((byte_pointer) &mx, sizeof(short));
```

	12,345		−12,345		53,191	
Weight	Bit	Value	Bit	Value	Bit	Value
1	1	1	1	1	1	1
2	0	0	1	2	1	2
4	0	0	1	4	1	4
8	1	8	0	0	0	0
16	1	16	0	0	0	0
32	1	32	0	0	0	0
64	0	0	1	64	1	64
128	0	0	1	128	1	128
256	0	0	1	256	1	256
512	0	0	1	512	1	512
1,024	0	0	1	1,024	1	1,024
2,048	0	0	1	−2,048	1	2,048
4,096	1	4,096	0	0	0	0
8,192	1	8,192	0	0	0	0
16,384	0	0	1	16,384	1	16,384
±32,768	0	0	1	−32,768	1	32,768
Total		12,345		−12,345		53,191

Figure 2.15 Two's-complement representations of 12,345 and −12,345, and unsigned representation of 53,191. Note that the latter two have identical bit representations.

When run on a big-endian machine, this code prints `30 39` and `cf c7`, indicating that `x` has hexadecimal representation `0x3039`, while `mx` has hexadecimal representation `0xCFC7`. Expanding these into binary, we get bit patterns [0011000000111001] for `x` and [1100111111000111] for `mx`. As Figure 2.15 shows, Equation 2.3 yields values 12,345 and −12,345 for these two bit patterns.

Practice Problem 2.18 (solution page 149)

In Chapter 3, we will look at listings generated by a *disassembler*, a program that converts an executable program file back to a more readable ASCII form. These files contain many hexadecimal numbers, typically representing values in two's-complement form. Being able to recognize these numbers and understand their significance (for example, whether they are negative or positive) is an important skill.

For the lines labeled A–I (on the right) in the following listing, convert the hexadecimal values (in 32-bit two's-complement form) shown to the right of the instruction names (`sub`, `mov`, and `add`) into their decimal equivalents:

```
4004d0:  48 81 ec e0 02 00 00    sub    $0x2e0,%rsp              A.
4004d7:  48 8b 44 24 a8          mov    -0x58(%rsp),%rax         B.
4004dc:  48 03 47 28             add    0x28(%rdi),%rax          C.
4004e0:  48 89 44 24 d0          mov    %rax,-0x30(%rsp)         D.
4004e5:  48 8b 44 24 78          mov    0x78(%rsp),%rax          E.
4004ea:  48 89 87 88 00 00 00    mov    %rax,0x88(%rdi)          F.
4004f1:  48 8b 84 24 f8 01 00    mov    0x1f8(%rsp),%rax         G.
4004f8:  00
4004f9:  48 03 44 24 08          add    0x8(%rsp),%rax
4004fe:  48 89 84 24 c0 00 00    mov    %rax,0xc0(%rsp)          H.
400505:  00
400506:  48 8b 44 d4 b8          mov    -0x48(%rsp,%rdx,8),%rax  I.
```

2.2.4 Conversions between Signed and Unsigned

C allows casting between different numeric data types. For example, suppose variable x is declared as `int` and u as `unsigned`. The expression `(unsigned) x` converts the value of x to an unsigned value, and `(int) u` converts the value of u to a signed integer. What should be the effect of casting signed value to unsigned, or vice versa? From a mathematical perspective, one can imagine several different conventions. Clearly, we want to preserve any value that can be represented in both forms. On the other hand, converting a negative value to unsigned might yield zero. Converting an unsigned value that is too large to be represented in two's-complement form might yield *TMax*. For most implementations of C, however, the answer to this question is based on a bit-level perspective, rather than on a numeric one.

For example, consider the following code:

```
1       short    int   v  = -12345;
2       unsigned short uv = (unsigned short) v;
3       printf("v = %d, uv = %u\n", v, uv);
```

When run on a two's-complement machine, it generates the following output:

```
v = -12345, uv = 53191
```

What we see here is that the effect of casting is to keep the bit values identical but change how these bits are interpreted. We saw in Figure 2.15 that the 16-bit two's-complement representation of −12,345 is identical to the 16-bit unsigned representation of 53,191. Casting from `short` to `unsigned short` changed the numeric value, but not the bit representation.

Similarly, consider the following code:

```
1       unsigned u = 4294967295u;   /* UMax */
2       int      tu = (int) u;
```

```
3       printf("u = %u, tu = %d\n", u, tu);
```

When run on a two's-complement machine, it generates the following output:

```
u = 4294967295, tu = -1
```

We can see from Figure 2.14 that, for a 32-bit word size, the bit patterns representing 4,294,967,295 ($UMax_{32}$) in unsigned form and -1 in two's-complement form are identical. In casting from `unsigned` to `int`, the underlying bit representation stays the same.

This is a general rule for how most C implementations handle conversions between signed and unsigned numbers with the same word size—the numeric values might change, but the bit patterns do not. Let us capture this idea in a more mathematical form. We defined functions $U2B_w$ and $T2B_w$ that map numbers to their bit representations in either unsigned or two's-complement form. That is, given an integer x in the range $0 \leq x < UMax_w$, the function $U2B_w(x)$ gives the unique w-bit unsigned representation of x. Similarly, when x is in the range $TMin_w \leq x \leq TMax_w$, the function $T2B_w(x)$ gives the unique w-bit two's-complement representation of x.

Now define the function $T2U_w$ as $T2U_w(x) \doteq B2U_w(T2B_w(x))$. This function takes a number between $TMin_w$ and $TMax_w$ and yields a number between 0 and $UMax_w$, where the two numbers have identical bit representations, except that the argument has a two's-complement representation while the result is unsigned. Similarly, for x between 0 and $UMax_w$, the function $U2T_w$, defined as $U2T_w(x) \doteq B2T_w(U2B_w(x))$, yields the number having the same two's-complement representation as the unsigned representation of x.

Pursuing our earlier examples, we see from Figure 2.15 that $T2U_{16}(-12{,}345) = 53{,}191$, and that $U2T_{16}(53{,}191) = -12{,}345$. That is, the 16-bit pattern written in hexadecimal as `0xCFC7` is both the two's-complement representation of $-12{,}345$ and the unsigned representation of 53,191. Note also that $12{,}345 + 53{,}191 = 65{,}536 = 2^{16}$. This property generalizes to a relationship between the two numeric values (two's complement and unsigned) represented by a given bit pattern. Similarly, from Figure 2.14, we see that $T2U_{32}(-1) = 4{,}294{,}967{,}295$, and $U2T_{32}(4{,}294{,}967{,}295) = -1$. That is, $UMax$ has the same bit representation in unsigned form as does -1 in two's-complement form. We can also see the relationship between these two numbers: $1 + UMax_w = 2^w$.

We see, then, that function $T2U$ describes the conversion of a two's-complement number to its unsigned counterpart, while $U2T$ converts in the opposite direction. These describe the effect of casting between these data types in most C implementations.

Practice Problem 2.19 (solution page 149)

Using the table you filled in when solving Problem 2.17, fill in the following table describing the function $T2U_4$:

x	$T2U_4(x)$
−8	_____
−3	_____
−2	_____
−1	_____
0	_____
5	_____

The relationship we have seen, via several examples, between the two's-complement and unsigned values for a given bit pattern can be expressed as a property of the function $T2U$:

PRINCIPLE: Conversion from two's complement to unsigned

For x such that $TMin_w \leq x \leq TMax_w$:

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{2.5}$$

■

For example, we saw that $T2U_{16}(-12{,}345) = -12{,}345 + 2^{16} = 53{,}191$, and also that $T2U_w(-1) = -1 + 2^w = UMax_w$.

This property can be derived by comparing Equations 2.1 and 2.3.

DERIVATION: Conversion from two's complement to unsigned

Comparing Equations 2.1 and 2.3, we can see that for bit pattern $\vec{x}$, if we compute the difference $B2U_w(\vec{x}) - B2T_w(\vec{x})$, the weighted sums for bits from 0 to $w-2$ will cancel each other, leaving a value $B2U_w(\vec{x}) - B2T_w(\vec{x}) = x_{w-1}(2^{w-1} - -2^{w-1}) = x_{w-1}2^w$. This gives a relationship $B2U_w(\vec{x}) = B2T_w(\vec{x}) + x_{w-1}2^w$. We therefore have

$$B2U_w(T2B_w(x)) = T2U_w(x) = x + x_{w-1}2^w \tag{2.6}$$

In a two's-complement representation of x, bit x_{w-1} determines whether or not x is negative, giving the two cases of Equation 2.5. ■

As examples, Figure 2.16 compares how functions $B2U$ and $B2T$ assign values to bit patterns for $w = 4$. For the two's-complement case, the most significant bit serves as the sign bit, which we diagram as a leftward-pointing gray bar. For the unsigned case, this bit has positive weight, which we show as a rightward-pointing black bar. In going from two's complement to unsigned, the most significant bit changes its weight from −8 to +8. As a consequence, the values that are negative in a two's-complement representation increase by $2^4 = 16$ with an unsigned representation. Thus, −5 becomes +11, and −1 becomes +15.

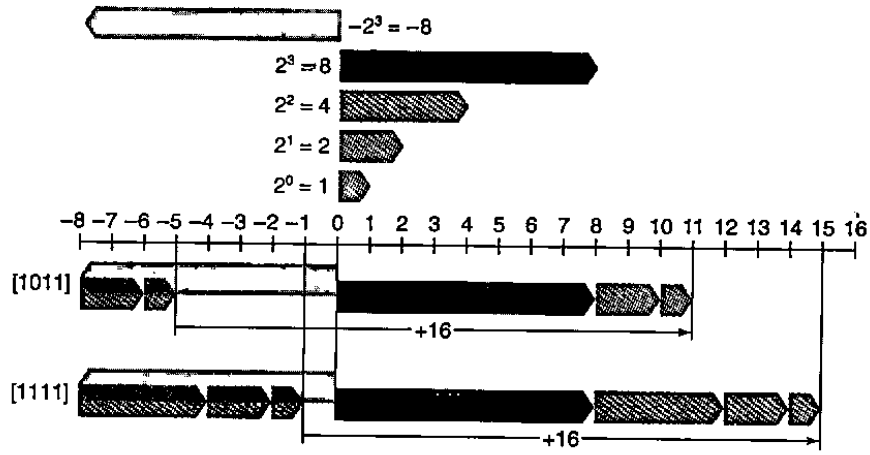


Figure 2.16 Comparing unsigned and two's-complement representations for $w = 4$. The weight of the most significant bit is −8 for two's complement and +8 for unsigned, yielding a net difference of 16.

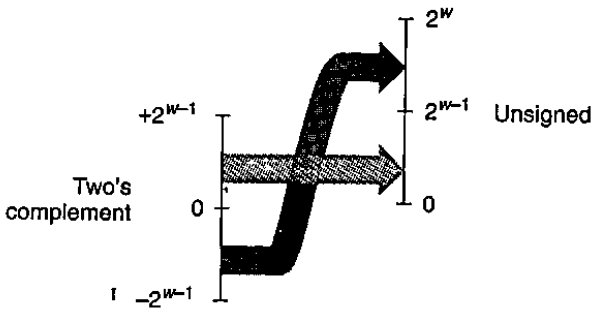


Figure 2.17 Conversion from two's complement to unsigned. Function *T2U* converts negative numbers to large positive numbers.

Figure 2.17 illustrates the general behavior of function *T2U*. As it shows, when mapping a signed number to its unsigned counterpart, negative numbers are converted to large positive numbers, while nonnegative numbers remain unchanged.

Practice Problem 2.20 (solution page 149)

Explain how Equation 2.5 applies to the entries in the table you generated when solving Problem 2.19.

Going in the other direction, we can state the relationship between an unsigned number u and its signed counterpart $U2T_w(u)$:

PRINCIPLE: Unsigned to two's-complement conversion

For u such that $0 \le u \le UMax_w$:

$$U2T_w(u) = \begin{cases} u, & u \le TMax_w \\ u - 2^w, & u > TMax_w \end{cases} \tag{2.7}$$

■

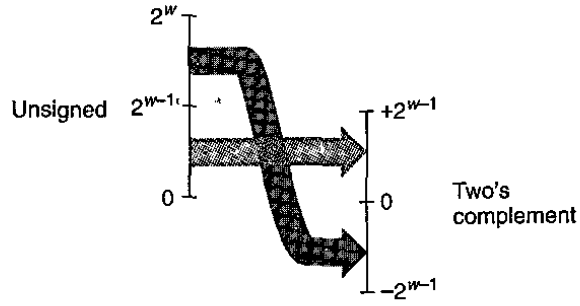


Figure 2.18 Conversion from unsigned to two's complement. Function $U2T$ converts numbers greater than $2^{w-1} - 1$ to negative values.

This principle can be justified as follows:

DERIVATION: Unsigned to two's-complement conversion

Let $\vec{u} = U2B_w(u)$. This bit vector will also be the two's-complement representation of $U2T_w(u)$. Equations 2.1 and 2.3 can be combined to give

$$U2T_w(u) = -u_{w-1}2^w + u \tag{2.8}$$

In the unsigned representation of u, bit u_{w-1} determines whether or not u is greater than $TMax_w = 2^{w-1} - 1$, giving the two cases of Equation 2.7. ∎

The behavior of function $U2T$ is illustrated in Figure 2.18. For small ($\leq TMax_w$) numbers, the conversion from unsigned to signed preserves the numeric value. Large ($> TMax_w$) numbers are converted to negative values.

To summarize, we considered the effects of converting in both directions between unsigned and two's-complement representations. For values x in the range $0 \leq x \leq TMax_w$, we have $T2U_w(x) = x$ and $U2T_w(x) = x$. That is, numbers in this range have identical unsigned and two's-complement representations. For values outside of this range, the conversions either add or subtract 2^w. For example, we have $T2U_w(-1) = -1 + 2^w = UMax_w$—the negative number closest to zero maps to the largest unsigned number. At the other extreme, one can see that $T2U_w(TMin_w) = -2^{w-1} + 2^w = 2^{w-1} = TMax_w + 1$—the most negative number maps to an unsigned number just outside the range of positive two's-complement numbers. Using the example of Figure 2.15, we can see that $T2U_{16}(-12,345) = 65,536 + -12,345 = 53,191$.

2.2.5 Signed versus Unsigned in C

As indicated in Figures 2.9 and 2.10, C supports both signed and unsigned arithmetic for all of its integer data types. Although the C standard does not specify a particular representation of signed numbers, almost all machines use two's complement. Generally, most numbers are signed by default. For example, when declaring a constant such as `12345` or `0x1A2B`, the value is considered signed. Adding character 'U' or 'u' as a suffix creates an unsigned constant; for example, `12345U` or `0x1A2Bu`.

C allows conversion between unsigned and signed. Although the C standard does not specify precisely how this conversion should be made, most systems follow the rule that the underlying bit representation does not change. This rule has the effect of applying the function $U2T_w$ when converting from unsigned to signed, and $T2U_w$ when converting from signed to unsigned, where w is the number of bits for the data type.

Conversions can happen due to explicit casting, such as in the following code:

```
1       int tx, ty;
2       unsigned ux, uy;
3
4       tx = (int) ux;
5       uy = (unsigned) ty;
```

Alternatively, they can happen implicitly when an expression of one type is assigned to a variable of another, as in the following code:

```
1       int tx, ty;
2       unsigned ux, uy;
3
4       tx = ux; /* Cast to signed */
5       uy = ty; /* Cast to unsigned */
```

When printing numeric values with `printf`, the directives %d, %u, and %x are used to print a number as a signed decimal, an unsigned decimal, and in hexadecimal format, respectively. Note that `printf` does not make use of any type information, and so it is possible to print a value of type `int` with directive %u and a value of type `unsigned` with directive %d. For example, consider the following code:

```
1       int x = -1;
2       unsigned u = 2147483648; /* 2 to the 31st */
3
4       printf("x = %u = %d\n", x, x);
5       printf("u = %u = %d\n", u, u);
```

When compiled as a 32-bit program, it prints the following:

```
x = 4294967295 = -1
u = 2147483648 = -2147483648
```

In both cases, `printf` prints the word first as if it represented an unsigned number and second as if it represented a signed number. We can see the conversion routines in action: $T2U_{32}(-1) = UMax_{32} = 2^{32} - 1$ and $U2T_{32}(2^{31}) = 2^{31} - 2^{32} = -2^{31} = TMin_{32}$.

Some possibly nonintuitive behavior arises due to C's handling of expressions containing combinations of signed and unsigned quantities. When an operation is performed where one operand is signed and the other is unsigned, C implicitly casts the signed argument to unsigned and performs the operations

Expression			Type	Evaluation
0	==	0U	Unsigned	1
-1	<	0	Signed	1
-1	<	0U	Unsigned	0 *
2147483647	>	-2147483647-1	Signed	1
2147483647U	>	-2147483647-1	Unsigned	0 *
2147483647	>	(int) 2147483648U	Signed	1 *
-1	>	-2	Signed	1
(unsigned) -1	>	-2	Unsigned	1

Figure 2.19 Effects of C promotion rules. Nonintuitive cases are marked by '*'. When either operand of a comparison is unsigned, the other operand is implicitly cast to unsigned. See Web Aside DATA:TMIN for why we write $TMin_{32}$ as -2,147,483,647-1.

assuming the numbers are nonnegative. As we will see, this convention makes little difference for standard arithmetic operations, but it leads to nonintuitive results for relational operators such as < and >. Figure 2.19 shows some sample relational expressions and their resulting evaluations, when data type `int` has a 32-bit two's-complement representation. Consider the comparison -1 < 0U. Since the second operand is unsigned, the first one is implicitly cast to unsigned, and hence the expression is equivalent to the comparison 4294967295U < 0U (recall that $T2U_w(-1) = UMax_w$), which of course is false. The other cases can be understood by similar analyses.

Practice Problem 2.21 (solution page 149)

Assuming the expressions are evaluated when executing a 32-bit program on a machine that uses two's-complement arithmetic, fill in the following table describing the effect of casting and relational operations, in the style of Figure 2.19:

Expression	Type	Evaluation
-2147483647-1 == 2147483648U	_____	_____
-2147483647-1 < 2147483647	_____	_____
-2147483647-1U < 2147483647	_____	_____
-2147483647-1 < -2147483647	_____	_____
-2147483647-1U < -2147483647	_____	_____

2.2.6 Expanding the Bit Representation of a Number

One common operation is to convert between integers having different word sizes while retaining the same numeric value. Of course, this may not be possible when the destination data type is too small to represent the desired value. Converting from a smaller to a larger data type, however, should always be possible.

Web Aside DATA:TMIN Writing *TMin* in C

In Figure 2.19 and in Problem 2.71, we carefully wrote the value of $TMin_{32}$ as -2,147,483,647-1. Why not simply write it as either -2,147,483,648 or 0x80000000? Looking at the C header file `limits.h`, we see that they use a similar method as we have to write $TMin_{32}$ and $TMax_{32}$:

```
/* Minimum and maximum values a 'signed int' can hold.  */
#define INT_MAX  2147483647
#define INT_MIN  (-INT_MAX - 1)
```

Unfortunately, a curious interaction between the asymmetry of the two's-complement representation and the conversion rules of C forces us to write $TMin_{32}$ in this unusual way. Although understanding this issue requires us to delve into one of the murkier corners of the C language standards, it will help us appreciate some of the subtleties of integer data types and representations.

To convert an unsigned number to a larger data type, we can simply add leading zeros to the representation; this operation is known as *zero extension*, expressed by the following principle:

PRINCIPLE: Expansion of an unsigned number by zero extension

Define bit vectors $\vec{u} = [u_{w-1}, u_{w-2}, \ldots, u_0]$ of width w and $\vec{u}' = [0, \ldots, 0, u_{w-1}, u_{w-2}, \ldots, u_0]$ of width w', where $w' > w$. Then $B2U_w(\vec{u}) = B2U_{w'}(\vec{u}')$. ∎

This principle can be seen to follow directly from the definition of the unsigned encoding, given by Equation 2.1.

For converting a two's-complement number to a larger data type, the rule is to perform a *sign extension*, adding copies of the most significant bit to the representation, expressed by the following principle. We show the sign bit x_{w-1} in blue to highlight its role in sign extension.

PRINCIPLE: Expansion of a two's-complement number by sign extension

Define bit vectors $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$ of width w and $\vec{x}' = [x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]$ of width w', where $w' > w$. Then $B2T_w(\vec{x}) = B2T_{w'}(\vec{x}')$. ∎

As an example, consider the following code:

```
1   short sx = -12345;          /* -12345 */
2   unsigned short usx = sx;    /*  53191 */
3   int x = sx;                 /* -12345 */
4   unsigned ux = usx;          /*  53191 */
5
6   printf("sx  = %d:\t", sx);
7   show_bytes((byte_pointer) &sx, sizeof(short));
8   printf("usx = %u:\t", usx);
9   show_bytes((byte_pointer) &usx, sizeof(unsigned short));
10  printf("x   = %d:\t", x);
```

```
11    show_bytes((byte_pointer) &x, sizeof(int));
12    printf("ux  = %u:\t", ux);
13    show_bytes((byte_pointer) &ux, sizeof(unsigned));
```

When run as a 32-bit program on a big-endian machine that uses a two's-complement representation, this code prints the output

```
sx  = -12345:  cf c7
usx = 53191:   cf c7
x   = -12345:  ff ff cf c7
ux  = 53191:   00 00 cf c7
```

We see that, although the two's-complement representation of −12,345 and the unsigned representation of 53,191 are identical for a 16-bit word size, they differ for a 32-bit word size. In particular, −12,345 has hexadecimal representation `0xFFFFCFC7`, while 53,191 has hexadecimal representation `0x0000CFC7`. The former has been sign extended—16 copies of the most significant bit 1, having hexadecimal representation `0xFFFF`, have been added as leading bits. The latter has been extended with 16 leading zeros, having hexadecimal representation `0x0000`.

As an illustration, Figure 2.20 shows the result of expanding from word size $w = 3$ to $w = 4$ by sign extension. Bit vector [101] represents the value $-4 + 1 = -3$. Applying sign extension gives bit vector [1101] representing the value $-8 + 4 + 1 = -3$. We can see that, for $w = 4$, the combined value of the two most significant bits, $-8 + 4 = -4$, matches the value of the sign bit for $w = 3$. Similarly, bit vectors [111] and [1111] both represent the value −1.

With this as intuition, we can now show that sign extension preserves the value of a two's-complement number.

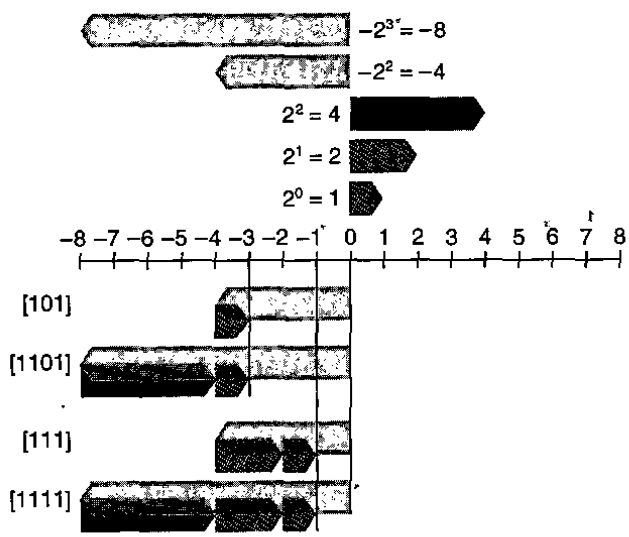


Figure 2.20 Examples of sign extension from $w = 3$ to $w = 4$. For $w = 4$, the combined weight of the upper 2 bits is $-8 + 4 = -4$, matching that of the sign bit for $w = 3$.

DERIVATION: Expansion of a two's-complement number by sign extension

Let $w' = w + k$. What we want to prove is that

$$B2T_{w+k}([\underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ times}}, x_{w-1}, x_{w-2}, \ldots, x_0]) = B2T_w([x_{w-1}, x_{w-2}, \ldots, x_0])$$

The proof follows by induction on k. That is, if we can prove that sign extending by 1 bit preserves the numeric value, then this property will hold when sign extending by an arbitrary number of bits. Thus, the task reduces to proving that

$$B2T_{w+1}([x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]) = B2T_w([x_{w-1}, x_{w-2}, \ldots, x_0])$$

Expanding the left-hand expression with Equation 2.3 gives the following:

$$\begin{aligned} B2T_{w+1}([x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]) &= -x_{w-1}2^w + \sum_{i=0}^{w-1} x_i 2^i \\ &= -x_{w-1}2^w + x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\ &= -x_{w-1}\left(2^w - 2^{w-1}\right) + \sum_{i=0}^{w-2} x_i 2^i \\ &= -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\ &= B2T_w([x_{w-1}, x_{w-2}, \ldots, x_0]) \end{aligned}$$

The key property we exploit is that $2^w - 2^{w-1} = 2^{w-1}$. Thus, the combined effect of adding a bit of weight -2^w and of converting the bit having weight -2^{w-1} to be one with weight 2^{w-1} is to preserve the original numeric value. ∎

Practice Problem 2.22 (solution page 150)

Show that each of the following bit vectors is a two's-complement representation of -5 by applying Equation 2.3:

A. [1011]

B. [11011]

C. [111011]

Observe that the second and third bit vectors can be derived from the first by sign extension.

One point worth making is that the relative order of conversion from one data size to another and between unsigned and signed can affect the behavior of a program. Consider the following code:

```
1    short sx = -12345;        /* -12345   */
2    unsigned uy = sx;         /* Mystery! */
3
4    printf("uy  = %u:\t", uy);
5    show_bytes((byte_pointer) &uy, sizeof(unsigned));
```

When run on a big-endian machine, this code causes the following output to be printed:

```
uy = 4294954951:  ff ff cf c7
```

This shows that, when converting from `short` to `unsigned`, the program first changes the size and then the type. That is, `(unsigned) sx` is equivalent to `(unsigned) (int) sx`, evaluating to 4,294,954,951, not `(unsigned) (unsigned short) sx`, which evaluates to 53,191. Indeed, this convention is required by the C standards.

Practice Problem 2.23 (solution page 150)

Consider the following C functions:

```
int fun1(unsigned word) {
    return (int) ((word << 24) >> 24);
}

int fun2(unsigned word) {
    return ((int) word << 24) >> 24;
}
```

Assume these are executed as a 32-bit program on a machine that uses two's-complement arithmetic. Assume also that right shifts of signed values are performed arithmetically, while right shifts of unsigned values are performed logically.

A. Fill in the following table showing the effect of these functions for several example arguments. You will find it more convenient to work with a hexadecimal representation. Just remember that hex digits 8 through F have their most significant bits equal to 1.

w	fun1(w)	fun2(w)
0x00000076	_____	_____
0x87654321	_____	_____
0x000000C9	_____	_____
0xEDCBA987	_____	_____

B. Describe in words the useful computation each of these functions performs.

2.2.7 Truncating Numbers

Suppose that, rather than extending a value with extra bits, we reduce the number of bits representing a number. This occurs, for example, in the following code:

```
1    int x = 53191;
2    short sx = (short) x;  /* -12345 */
3    int y = sx;            /* -12345 */
```

Casting x to be short will truncate a 32-bit int to a 16-bit short. As we saw before, this 16-bit pattern is the two's-complement representation of −12,345. When casting this back to int, sign extension will set the high-order 16 bits to ones, yielding the 32-bit two's-complement representation of −12,345.

When truncating a w-bit number $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$ to a k-bit number, we drop the high-order $w - k$ bits, giving a bit vector $\vec{x}' = [x_{k-1}, x_{k-2}, \ldots, x_0]$. Truncating a number can alter its value—a form of overflow. For an unsigned number, we can readily characterize the numeric value that will result.

PRINCIPLE: Truncation of an unsigned number

Let $\vec{x}$ be the bit vector $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and let $\vec{x}'$ be the result of truncating it to k bits: $\vec{x}' = [x_{k-1}, x_{k-2}, \ldots, x_0]$. Let $x = B2U_w(\vec{x})$ and $x' = B2U_k(\vec{x}')$. Then $x' = x \bmod 2^k$. ■

The intuition behind this principle is simply that all of the bits that were truncated have weights of the form 2^i, where $i \geq k$, and therefore each of these weights reduces to zero under the modulus operation. This is formalized by the following derivation:

DERIVATION: Truncation of an unsigned number

Applying the modulus operation to Equation 2.1 yields

$$\begin{aligned} B2U_w([x_{w-1}, x_{w-2}, \ldots, x_0]) \bmod 2^k &= \left[\sum_{i=0}^{w-1} x_i 2^i\right] \bmod 2^k \\ &= \left[\sum_{i=0}^{k-1} x_i 2^i\right] \bmod 2^k \\ &= \sum_{i=0}^{k-1} x_i 2^i \\ &= B2U_k([x_{k-1}, x_{k-2}, \ldots, x_0]) \end{aligned}$$

In this derivation, we make use of the property that $2^i \bmod 2^k = 0$ for any $i \geq k$. ■

A similar property holds for truncating a two's-complement number, except that it then converts the most significant bit into a sign bit:

PRINCIPLE: Truncation of a two's-complement number

Let $\vec{x}$ be the bit vector $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and let $\vec{x}$ be the result of truncating it to k bits: $\vec{x}' = [x_{k-1}, x_{k-2}, \ldots, x_0]$. Let $x = B2T_w(\vec{x})$ and $x' = B2T_k(\vec{x}')$. Then $x' = U2T_k(x \bmod 2^k)$. ∎

In this formulation, $x \bmod 2^k$ will be a number between 0 and $2^k - 1$. Applying function $U2T_k$ to it will have the effect of converting the most significant bit x_{k-1} from having weight 2^{k-1} to having weight -2^{k-1}. We can see this with the example of converting value $x = 53{,}191$ from `int` to `short`. Since $2^{16} = 65{,}536 \geq x$, we have $x \bmod 2^{16} = x$. But when we convert this number to a 16-bit two's-complement number, we get $x' = 53{,}191 - 65{,}536 = -12{,}345$.

DERIVATION: Truncation of a two's-complement number

Using a similar argument to the one we used for truncation of an unsigned number shows that

$$B2T_w([x_{w-1}, x_{w-2}, \ldots, x_0]) \bmod 2^k = B2U_k([x_{k-1}, x_{k-2}, \ldots, x_0])$$

That is, $x \bmod 2^k$ can be represented by an unsigned number having bit-level representation $[x_{k-1}, x_{k-2}, \ldots, x_0]$. Converting this to a two's-complement number gives $x' = U2T_k(x \bmod 2^k)$. ∎

Summarizing, the effect of truncation for unsigned numbers is

$$B2U_k([x_{k-1}, x_{k-2}, \ldots, x_0]) = B2U_w([x_{w-1}, x_{w-2}, \ldots, x_0]) \bmod 2^k \quad (2.9)$$

while the effect for two's-complement numbers is

$$B2T_k([x_{k-1}, x_{k-2}, \ldots, x_0]) = U2T_k(B2U_w([x_{w-1}, x_{w-2}, \ldots, x_0]) \bmod 2^k) \quad (2.10)$$

Practice Problem 2.24 (solution page 150)

Suppose we truncate a 4-bit value (represented by hex digits 0 through F) to a 3-bit value (represented as hex digits 0 through 7.) Fill in the table below showing the effect of this truncation for some cases, in terms of the unsigned and two's-complement interpretations of those bit patterns.

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

Explain how Equations 2.9 and 2.10 apply to these cases.

2.2.8 Advice on Signed versus Unsigned

As we have seen, the implicit casting of signed to unsigned leads to some nonintuitive behavior. Nonintuitive features often lead to program bugs, and ones involving the nuances of implicit casting can be especially difficult to see. Since the casting takes place without any clear indication in the code, programmers often overlook its effects.

The following two practice problems illustrate some of the subtle errors that can arise due to implicit casting and the unsigned data type.

Practice Problem 2.25 (solution page 151)

Consider the following code that attempts to sum the elements of an array a, where the number of elements is given by parameter `length`:

```
1   /* WARNING: This is buggy code */
2   float sum_elements(float a[], unsigned length) {
3       int i;
4       float result = 0;
5
6       for (i = 0; i <= length-1; i++)
7           result += a[i];
8       return result;
9   }
```

When run with argument `length` equal to 0, this code should return 0.0. Instead, it encounters a memory error. Explain why this happens. Show how this code can be corrected.

Practice Problem 2.26 (solution page 151)

You are given the assignment of writing a function that determines whether one string is longer than another. You decide to make use of the string library function `strlen` having the following declaration:

```
/* Prototype for library function strlen */
size_t strlen(const char *s);
```

Here is your first attempt at the function:

```
/* Determine whether string s is longer than string t */
/* WARNING: This function is buggy */
int strlonger(char *s, char *t) {
    return strlen(s) - strlen(t) > 0;
}
```

When you test this on some sample data, things do not seem to work quite right. You investigate further and determine that, when compiled as a 32-bit

program, data type `size_t` is defined (via `typedef`) in header file `stdio.h` to be `unsigned`.

A. For what cases will this function produce an incorrect result?

B. Explain how this incorrect result comes about.

C. Show how to fix the code so that it will work reliably.

We have seen multiple ways in which the subtle features of unsigned arithmetic, and especially the implicit conversion of signed to unsigned, can lead to errors or vulnerabilities. One way to avoid such bugs is to never use unsigned numbers. In fact, few languages other than C support unsigned integers. Apparently, these other language designers viewed them as more trouble than they are worth. For example, Java supports only signed integers, and it requires that they be implemented with two's-complement arithmetic. The normal right shift operator >> is guaranteed to perform an arithmetic shift. The special operator >>> is defined to perform a logical right shift.

Unsigned values are very useful when we want to think of words as just collections of bits with no numeric interpretation. This occurs, for example, when packing a word with *flags* describing various Boolean conditions. Addresses are naturally unsigned, so systems programmers find unsigned types to be helpful. Unsigned values are also useful when implementing mathematical packages for modular arithmetic and for multiprecision arithmetic, in which numbers are represented by arrays of words.

2.3 Integer Arithmetic

Many beginning programmers are surprised to find that adding two positive numbers can yield a negative result, and that the comparison $x < y$ can yield a different result than the comparison $x-y < 0$. These properties are artifacts of the finite nature of computer arithmetic. Understanding the nuances of computer arithmetic can help programmers write more reliable code.

2.3.1 Unsigned Addition

Consider two nonnegative integers x and y, such that $0 \leq x, y < 2^w$. Each of these values can be represented by a w-bit unsigned number. If we compute their sum, however, we have a possible range $0 \leq x + y \leq 2^{w+1} - 2$. Representing this sum could require $w + 1$ bits. For example, Figure 2.21 shows a plot of the function $x + y$ when x and y have 4-bit representations. The arguments (shown on the horizontal axes) range from 0 to 15, but the sum ranges from 0 to 30. The shape of the function is a sloping plane (the function is linear in both dimensions). If we were to maintain the sum as a $(w + 1)$-bit number and add it to another value, we may require $w + 2$ bits, and so on. This continued "word size

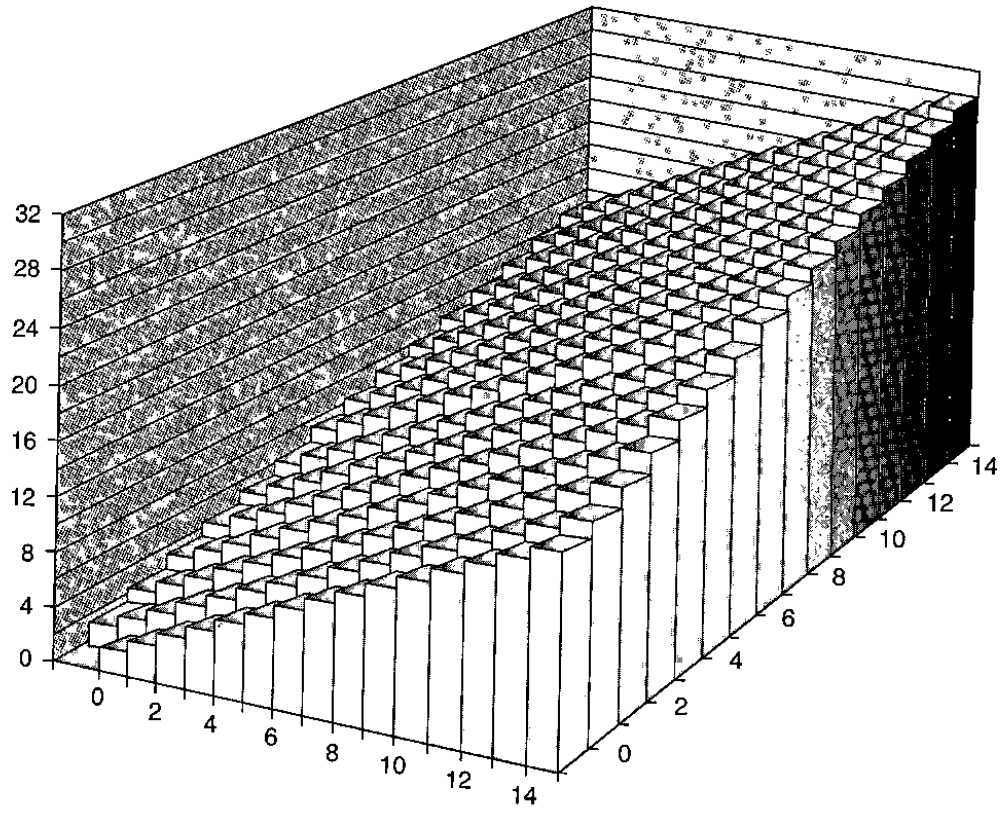


Figure 2.21 Integer addition. With a 4-bit word size, the sum could require 5 bits.

inflation" means we cannot place any bound on the word size required to fully represent the results of arithmetic operations. Some programming languages, such as Lisp, actually support *arbitrary size* arithmetic to allow integers of any size (within the memory limits of the computer, of course.) More commonly, programming languages support fixed-size arithmetic, and hence operations such as "addition" and "multiplication" differ from their counterpart operations over integers.

Let us define the operation $+_w^u$ for arguments x and y, where $0 \leq x, y < 2^w$, as the result of truncating the integer sum $x + y$ to be w bits long and then viewing the result as an unsigned number. This can be characterized as a form of modular arithmetic, computing the sum modulo 2^w by simply discarding any bits with weight greater than 2^{w-1} in the bit-level representation of $x + y$. For example, consider a 4-bit number representation with $x = 9$ and $y = 12$, having bit representations [1001] and [1100], respectively. Their sum is 21, having a 5-bit representation [10101]. But if we discard the high-order bit, we get [0101], that is, decimal value 5. This matches the value 21 mod 16 = 5.

Aside Security vulnerability in getpeername

In 2002, programmers involved in the FreeBSD open-source operating systems project realized that their implementation of the getpeername library function had a security vulnerability. A simplified version of their code went something like this:

```
1  /*
2   * Illustration of code vulnerability similar to that found in
3   * FreeBSD's implementation of getpeername()
4   */
5
6  /* Declaration of library function memcpy */
7  void *memcpy(void *dest, void *src, size_t n);
8
9  /* Kernel memory region holding user-accessible data */
10 #define KSIZE 1024
11 char kbuf[KSIZE];
12
13 /* Copy at most maxlen bytes from kernel region to user buffer */
14 int copy_from_kernel(void *user_dest, int maxlen) {
15     /* Byte count len is minimum of buffer size and maxlen */
16     int len = KSIZE < maxlen ? KSIZE : maxlen;
17     memcpy(user_dest, kbuf, len);
18     return len;
19 }
```

In this code, we show the prototype for library function memcpy on line 7, which is designed to copy a specified number of bytes n from one region of memory to another.

The function copy_from_kernel, starting at line 14, is designed to copy some of the data maintained by the operating system kernel to a designated region of memory accessible to the user. Most of the data structures maintained by the kernel should not be readable by a user, since they may contain sensitive information about other users and about other jobs running on the system, but the region shown as kbuf was intended to be one that the user could read. The parameter maxlen is intended to be the length of the buffer allocated by the user and indicated by argument user_dest. The computation at line 16 then makes sure that no more bytes are copied than are available in either the source or the destination buffer.

Suppose, however, that some malicious programmer writes code that calls copy_from_kernel with a negative value of maxlen. Then the minimum computation on line 16 will compute this value for len, which will then be passed as the parameter n to memcpy. Note, however, that parameter n is declared as having data type size_t. This data type is declared (via typedef) in the library file stdio.h. Typically, it is defined to be unsigned for 32-bit programs and unsigned long for 64-bit programs. Since argument n is unsigned, memcpy will treat it as a very large positive number and attempt to copy that many bytes from the kernel region to the user's buffer. Copying that many bytes (at least 2^{31}) will not actually work, because the program will encounter invalid addresses in the process, but the program could read regions of the kernel memory for which it is not authorized.

Aside Security vulnerability in getpeername *(continued)*

We can see that this problem arises due to the mismatch between data types: in one place the length parameter is signed; in another place it is unsigned. Such mismatches can be a source of bugs and, as this example shows, can even lead to security vulnerabilities. Fortunately, there were no reported cases where a programmer had exploited the vulnerability in FreeBSD. They issued a security advisory "FreeBSD-SA-02:38.signed-error" advising system administrators on how to apply a patch that would remove the vulnerability. The bug can be fixed by declaring parameter `maxlen` to `copy_from_kernel` to be of type `size_t`, to be consistent with parameter `n` of `memcpy`. We should also declare local variable `len` and the return value to be of type `size_t`.

We can characterize operation $+_w^u$ as follows:

PRINCIPLE: Unsigned addition

For x and y such that $0 \leq x, y < 2^w$:

$$x +_w^u y = \begin{cases} x + y, & x + y < 2^w \quad \text{Normal} \\ x + y - 2^w, & 2^w \leq x + y < 2^{w+1} \quad \text{Overflow} \end{cases} \tag{2.11}$$

■

The two cases of Equation 2.11 are illustrated in Figure 2.22, showing the sum $x + y$ on the left mapping to the unsigned w-bit sum $x +_w^u y$ on the right. The normal case preserves the value of $x + y$, while the overflow case has the effect of decrementing this sum by 2^w.

DERIVATION: Unsigned addition

In general, we can see that if $x + y < 2^w$, the leading bit in the $(w + 1)$-bit representation of the sum will equal 0, and hence discarding it will not change the numeric value. On the other hand, if $2^w \leq x + y < 2^{w+1}$, the leading bit in the $(w + 1)$-bit representation of the sum will equal 1, and hence discarding it is equivalent to subtracting 2^w from the sum. ■

An arithmetic operation is said to *overflow* when the full integer result cannot fit within the word size limits of the data type. As Equation 2.11 indicates, overflow

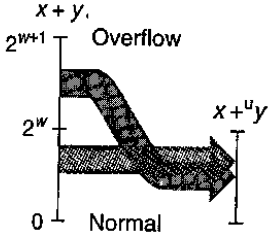


Figure 2.22 Relation between integer addition and unsigned addition. When $x + y$ is greater than $2^w - 1$, the sum overflows.

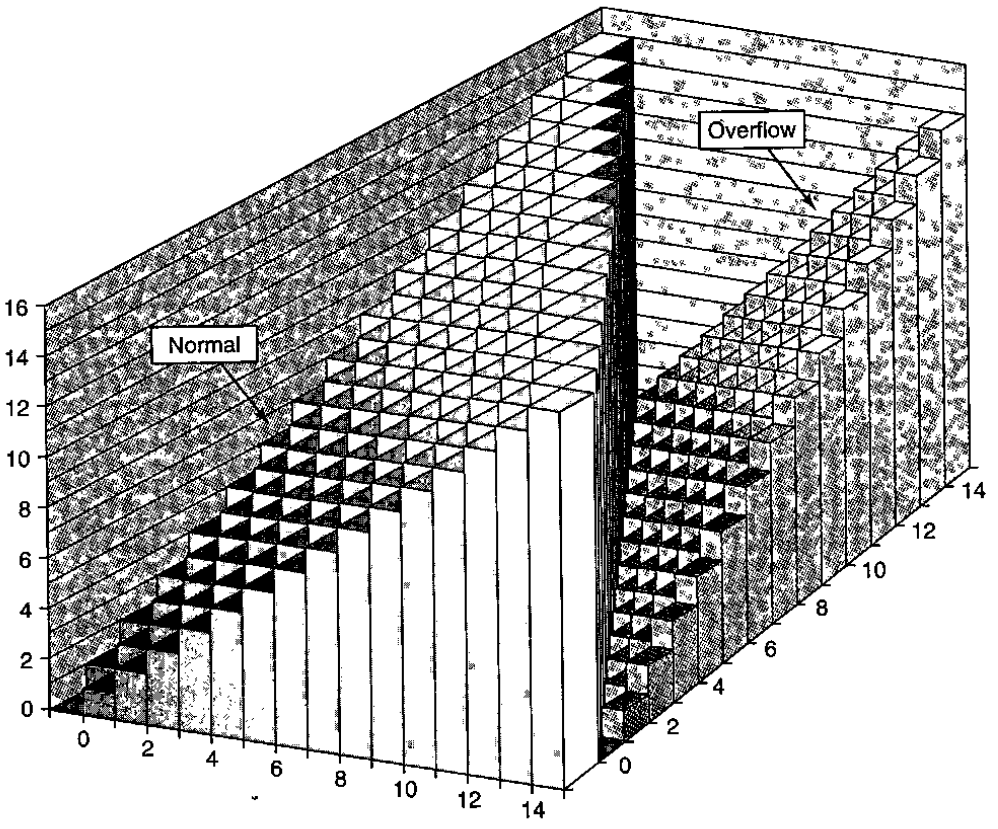


Figure 2.23 Unsigned addition. With a 4-bit word size, addition is performed modulo 16.

occurs when the two operands sum to 2^w or more. Figure 2.23 shows a plot of the unsigned addition function for word size $w = 4$. The sum is computed modulo $2^4 = 16$. When $x + y < 16$, there is no overflow, and $x +_4^u y$ is simply $x + y$. This is shown as the region forming a sloping plane labeled "Normal." When $x + y \geq 16$, the addition overflows, having the effect of decrementing the sum by 16. This is shown as the region forming a sloping plane labeled "Overflow."

When executing C programs, overflows are not signaled as errors. At times, however, we might wish to determine whether or not overflow has occurred.

PRINCIPLE: Detecting overflow of unsigned addition

For x and y in the range $0 \leq x, y \leq UMax_w$, let $s \doteq x +_w^u y$. Then the computation of s overflowed if and only if $s < x$ (or equivalently, $s < y$). ■

As an illustration, in our earlier example, we saw that $9 +_4^u 12 = 5$. We can see that overflow occurred, since $5 < 9$.

DERIVATION: Detecting overflow of unsigned addition

Observe that $x + y \geq x$, and hence if s did not overflow, we will surely have $s \geq x$. On the other hand, if s did overflow, we have $s = x + y - 2^w$. Given that $y < 2^w$, we have $y - 2^w < 0$, and hence $s = x + (y - 2^w) < x$. ■

Practice Problem 2.27 (solution page 152)

Write a function with the following prototype:

```
/* Determine whether arguments can be added without overflow */
int uadd_ok(unsigned x, unsigned y);
```

This function should return 1 if arguments x and y can be added without causing overflow.

Modular addition forms a mathematical structure known as an *abelian group*, named after the Norwegian mathematician Niels Henrik Abel (1802–1829). That is, it is commutative (that's where the "abelian" part comes in) and associative; it has an identity element 0, and every element has an additive inverse. Let us consider the set of w-bit unsigned numbers with addition operation $+_w^u$. For every value x, there must be some value $-_w^u x$ such that $-_w^u x +_w^u x = 0$. This additive inverse operation can be characterized as follows:

PRINCIPLE: Unsigned negation

For any number x such that $0 \leq x < 2^w$, its w-bit unsigned negation $-_w^u x$ is given by the following:

$$-_w^u x = \begin{cases} x, & x = 0 \\ 2^w - x, & x > 0 \end{cases} \tag{2.12}$$

■

This result can readily be derived by case analysis:

DERIVATION: Unsigned negation

When $x = 0$, the additive inverse is clearly 0. For $x > 0$, consider the value $2^w - x$. Observe that this number is in the range $0 < 2^w - x < 2^w$. We can also see that $(x + 2^w - x) \bmod 2^w = 2^w \bmod 2^w = 0$. Hence it is the inverse of x under $+_w^u$. ■

Practice Problem 2.28 (solution page 152)

We can represent a bit pattern of length $w = 4$ with a single hex digit. For an unsigned interpretation of these digits, use Equation 2.12 to fill in the following table giving the values and the bit representations (in hex) of the unsigned additive inverses of the digits shown.

x		$-_4^u x$	
Hex	Decimal	Decimal	Hex
0			
5			
8			
D			
F			

2.3.2 Two's-Complement Addition

With two's-complement addition, we must decide what to do when the result is either too large (positive) or too small (negative) to represent. Given integer values x and y in the range $-2^{w-1} \le x, y \le 2^{w-1} - 1$, their sum is in the range $-2^w \le x + y \le 2^w - 2$, potentially requiring $w + 1$ bits to represent exactly. As before, we avoid ever-expanding data sizes by truncating the representation to w bits. The result is not as familiar mathematically as modular addition, however. Let us define $x +_w^t y$ to be the result of truncating the integer sum $x + y$ to be w bits long and then viewing the result as a two's-complement number.

PRINCIPLE: Two's-complement addition

For integer values x and y in the range $-2^{w-1} \le x, y \le 2^{w-1} - 1$:

$$x +_w^t y = \begin{cases} x + y - 2^w, & 2^{w-1} \le x + y & \text{Positive overflow} \\ x + y, & -2^{w-1} \le x + y < 2^{w-1} & \text{Normal} \\ x + y + 2^w, & x + y < -2^{w-1} & \text{Negative overflow} \end{cases} \quad (2.13)$$

■

This principle is illustrated in Figure 2.24, where the sum $x + y$ is shown on the left, having a value in the range $-2^w \le x + y \le 2^w - 2$, and the result of truncating the sum to a w-bit two's-complement number is shown on the right. (The labels "Case 1" to "Case 4" in this figure are for the case analysis of the formal derivation of the principle.) When the sum $x + y$ exceeds $TMax_w$ (case 4), we say that *positive overflow* has occurred. In this case, the effect of truncation is to subtract 2^w from the sum. When the sum $x + y$ is less than $TMin_w$ (case 1), we say that *negative overflow* has occurred. In this case, the effect of truncation is to add 2^w to the sum.

The w-bit two's-complement sum of two numbers has the exact same bit-level representation as the unsigned sum. In fact, most computers use the same machine instruction to perform either unsigned or signed addition.

DERIVATION: Two's-complement addition

Since two's-complement addition has the exact same bit-level representation as unsigned addition, we can characterize the operation $+_w^t$ as one of converting its arguments to unsigned, performing unsigned addition, and then converting back to two's complement:

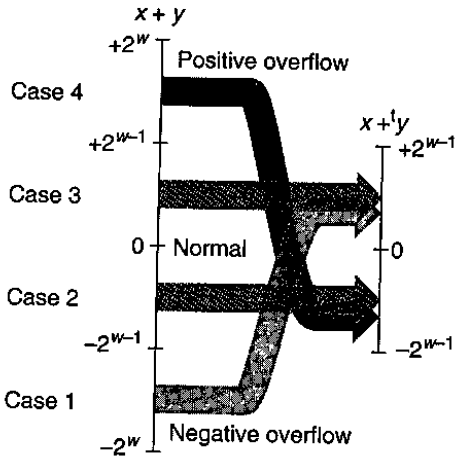


Figure 2.24
Relation between integer and two's-complement addition. When $x + y$ is less than -2^{w-1}, there is a negative overflow. When it is greater than or equal to 2^{w-1}, there is a positive overflow.

$$x +_w^t y = U2T_w(T2U_w(x) +_w^u T2U_w(y)). \tag{2.14}$$

By Equation 2.6, we can write $T2U_w(x)$ as $x_{w-1}2^w + x$ and $T2U_w(y)$ as $y_{w-1}2^w + y$. Using the property that $+_w^u$ is simply addition modulo 2^w, along with the properties of modular addition, we then have

$$\begin{aligned} x +_w^t y &= U2T_w(T2U_w(x) +_w^u T2U_w(y)) \\ &= U2T_w[(x_{w-1}2^w + x + y_{w-1}2^w + y) \bmod 2^w] \\ &= U2T_w[(x + y) \bmod 2^w] \end{aligned}$$

The terms $x_{w-1}2^w$ and $y_{w-1}2^w$ drop out since they equal 0 modulo 2^w.

To better understand this quantity, let us define z as the integer sum $z \doteq x + y$, z' as $z' \doteq z \bmod 2^w$, and z'' as $z'' \doteq U2T_w(z')$. The value z'' is equal to $x +_w^t y$. We can divide the analysis into four cases as illustrated in Figure 2.24:

1. $-2^w \le z < -2^{w-1}$. Then we will have $z' = z + 2^w$. This gives $0 \le z' < -2^{w-1} + 2^w = 2^{w-1}$. Examining Equation 2.7, we see that z' is in the range such that $z'' = z'$. This is the case of negative overflow. We have added two negative numbers x and y (that's the only way we can have $z < -2^{w-1}$) and obtained a nonnegative result $z'' = x + y + 2^w$.
2. $-2^{w-1} \le z < 0$. Then we will again have $z' = z + 2^w$, giving $-2^{w-1} + 2^w = 2^{w-1} \le z' < 2^w$. Examining Equation 2.7, we see that z' is in such a range that $z'' = z' - 2^w$, and therefore $z'' = z' - 2^w = z + 2^w - 2^w = z$. That is, our two's-complement sum z'' equals the integer sum $x + y$.
3. $0 \le z < 2^{w-1}$. Then we will have $z' = z$, giving $0 \le z' < 2^{w-1}$, and hence $z'' = z' = z$. Again, the two's-complement sum z'' equals the integer sum $x + y$.
4. $2^{w-1} \le z < 2^w$. We will again have $z' = z$, giving $2^{w-1} \le z' < 2^w$. But in this range we have $z'' = z' - 2^w$, giving $z'' = x + y - 2^w$. This is the case of positive overflow. We have added two positive numbers x and y (that's the only way we can have $z \ge 2^{w-1}$) and obtained a negative result $z'' = x + y - 2^w$. ∎

x	y	$x + y$	$x +_4^t y$	Case
−8	−5	−13	3	1
[1000]	[1011]	[10011]	[0011]	
−8	−8	−16	0	1
[1000]	[1000]	[10000]	[0000]	
−8	5	−3	−3	2
[1000]	[0101]	[11101]	[1101]	
2	5	7	7	3
[0010]	[0101]	[00111]	[0111]	
5	5	10	−6	4
[0101]	[0101]	[01010]	[1010]	

Figure 2.25 Two's-complement addition examples. The bit-level representation of the 4-bit two's-complement sum can be obtained by performing binary addition of the operands and truncating the result to 4 bits.

As illustrations of two's-complement addition, Figure 2.25 shows some examples when $w = 4$. Each example is labeled by the case to which it corresponds in the derivation of Equation 2.13. Note that $2^4 = 16$, and hence negative overflow yields a result 16 more than the integer sum, and positive overflow yields a result 16 less. We include bit-level representations of the operands and the result. Observe that the result can be obtained by performing binary addition of the operands and truncating the result to 4 bits.

Figure 2.26 illustrates two's-complement addition for word size $w = 4$. The operands range between −8 and 7. When $x + y < -8$, two's-complement addition has a negative overflow, causing the sum to be incremented by 16. When $-8 \leq x + y < 8$, the addition yields $x + y$. When $x + y \geq 8$, the addition has a positive overflow, causing the sum to be decremented by 16. Each of these three ranges forms a sloping plane in the figure.

Equation 2.13 also lets us identify the cases where overflow has occurred:

PRINCIPLE: Detecting overflow in two's-complement addition

For x and y in the range $TMin_w \leq x, y \leq TMax_w$, let $s \doteq x +_w^t y$. Then the computation of s has had positive overflow if and only if $x > 0$ and $y > 0$ but $s \leq 0$. The computation has had negative overflow if and only if $x < 0$ and $y < 0$ but $s \geq 0$. ■

Figure 2.25 shows several illustrations of this principle for $w = 4$. The first entry shows a case of negative overflow, where two negative numbers sum to a positive one. The final entry shows a case of positive overflow, where two positive numbers sum to a negative one.

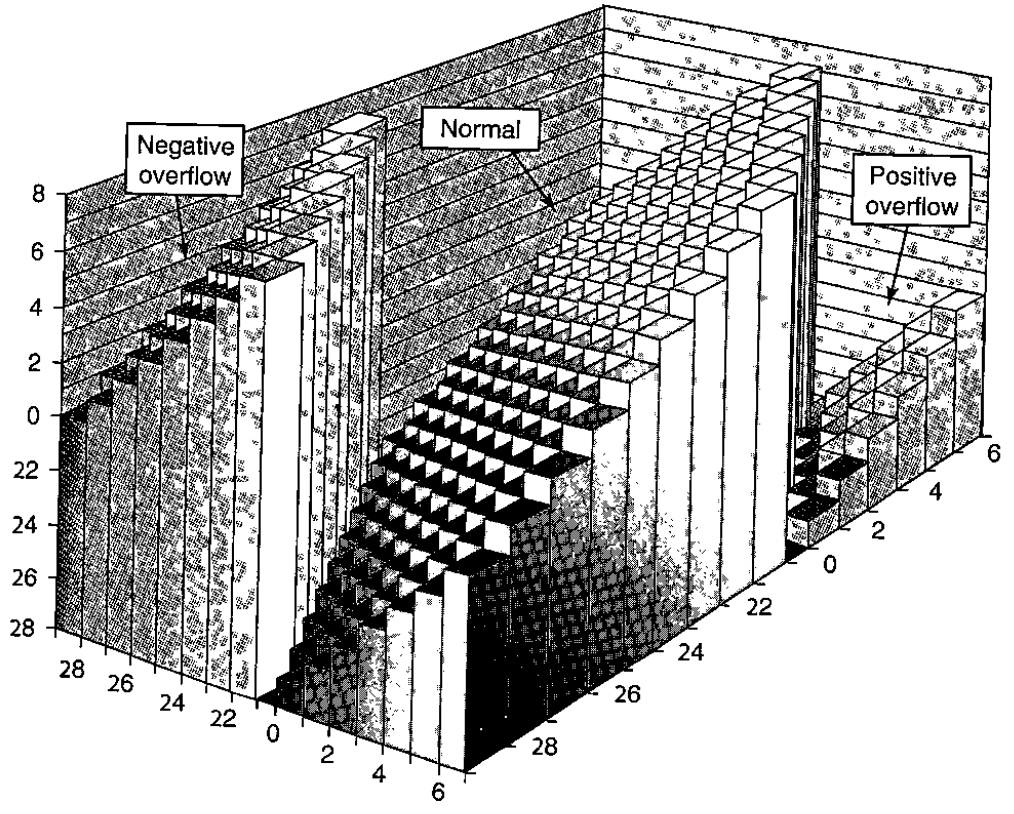


Figure 2.26 Two's-complement addition. With a 4-bit word size, addition can have a negative overflow when $x + y < -8$ and a positive overflow when $x + y \geq 8$.

DERIVATION: Detecting overflow of two's-complement addition

Let us first do the analysis for positive overflow. If both $x > 0$ and $y > 0$ but $s \leq 0$, then clearly positive overflow has occurred. Conversely, positive overflow requires (1) that $x > 0$ and $y > 0$ (otherwise, $x + y < TMax_w$) and (2) that $s \leq 0$ (from Equation 2.13). A similar set of arguments holds for negative overflow. ∎

Practice Problem 2.29 (solution page 152)

Fill in the following table in the style of Figure 2.25. Give the integer values of the 5-bit arguments, the values of both their integer and two's-complement sums, the bit-level representation of the two's-complement sum, and the case from the derivation of Equation 2.13.

x	y	$x + y$	$x +_5^t y$	Case
[10100]	[10001]			

x	y	$x + y$	$x +_5^t y$	Case
[11000]	[11000]			
[10111]	[01000]			
[00010]	[00101]			
[01100]	[00100]			

Practice Problem 2.30 (solution page 153)

Write a function with the following prototype:

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

This function should return 1 if arguments x and y can be added without causing overflow.

Practice Problem 2.31 (solution page 153)

Your coworker gets impatient with your analysis of the overflow conditions for two's-complement addition and presents you with the following implementation of `tadd_ok`:

```
/* Determine whether arguments can be added without overflow */
/* WARNING: This code is buggy. */
int tadd_ok(int x, int y) {
    int sum = x+y;
    return (sum-x == y) && (sum-y == x);
}
```

You look at the code and laugh. Explain why.

Practice Problem 2.32 (solution page 153)

You are assigned the task of writing code for a function `tsub_ok`, with arguments x and y, that will return 1 if computing x-y does not cause overflow. Having just written the code for Problem 2.30, you write the following:

```
/* Determine whether arguments can be subtracted without overflow */
/* WARNING: This code is buggy. */
int tsub_ok(int x, int y) {
```

```
    return tadd_ok(x, -y);
}
```

For what values of x and y will this function give incorrect results? Writing a correct version of this function is left as an exercise (Problem 2.74).

2.3.3 Two's-Complement Negation

We can see that every number x in the range $TMin_w \leq x \leq TMax_w$ has an additive inverse under $+^t_w$, which we denote $-^t_w x$ as follows:

PRINCIPLE: Two's-complement negation

For x in the range $TMin_w \leq x \leq TMax_w$, its two's-complement negation $-^t_w x$ is given by the formula

$$-^t_w x = \begin{cases} TMin_w, & x = TMin_w \\ -x, & x > TMin_w \end{cases} \tag{2.15}$$

■

That is, for w-bit two's-complement addition, $TMin_w$ is its own additive inverse, while any other value x has $-x$ as its additive inverse.

DERIVATION: Two's-complement negation

Observe that $TMin_w + TMin_w = -2^{w-1} + -2^{w-1} = -2^w$. This would cause negative overflow, and hence $TMin_w +^t_w TMin_w = -2^w + 2^w = 0$. For values of x such that $x > TMin_w$, the value $-x$ can also be represented as a w-bit two's-complement number, and their sum will be $-x + x = 0$. ■

Practice Problem 2.33 (solution page 153)

We can represent a bit pattern of length $w = 4$ with a single hex digit. For a two's-complement interpretation of these digits, fill in the following table to determine the additive inverses of the digits shown:

x		$-^t_4 x$	
Hex	Decimal	Decimal	Hex
0			
5			
8			
D			
F			

What do you observe about the bit patterns generated by two's-complement and unsigned (Problem 2.28) negation?

Web Aside DATA:TNEG Bit-level representation of two's-complement negation

There are several clever ways to determine the two's-complement negation of a value represented at the bit level. The following two techniques are both useful, such as when one encounters the value `0xfffffffa` when debugging a program, and they lend insight into the nature of the two's-complement representation.

One technique for performing two's-complement negation at the bit level is to complement the bits and then increment the result. In C, we can state that for any integer value x, computing the expressions `-x` and `~x + 1` will give identical results.

Here are some examples with a 4-bit word size:

$\vec{x}$		$-\vec{x}$		$incr(\sim\vec{x})$	
[0101]	5	[1010]	−6	[1011]	−5
[0111]	7	[1000]	−8	[1001]	−7
[1100]	−4	[0011]	3	[0100]	4
[0000]	0	[1111]	−1	[0000]	0
[1000]	−8	[0111]	7	[1000]	−8

For our earlier example, we know that the complement of `0xf` is `0x0` and the complement of `0xa` is `0x5`, and so `0xfffffffa` is the two's-complement representation of −6.

A second way to perform two's-complement negation of a number x is based on splitting the bit vector into two parts. Let k be the position of the rightmost 1, so the bit-level representation of x has the form $[x_{w-1}, x_{w-2}, \ldots, x_{k+1}, 1, 0, \ldots 0]$. (This is possible as long as $x \neq 0$.) The negation is then written in binary form as $[\sim x_{w-1}, \sim x_{w-2}, \ldots, \sim x_{k+1}, 1, 0, \ldots, 0]$. That is, we complement each bit to the left of bit position k.

We illustrate this idea with some 4-bit numbers, where we highlight the rightmost pattern 1, 0, ..., 0 in italics:

x		$-x$	
[1*100*]	−4	[0*100*]	4
[*1000*]	−8	[*1000*]	−8
[010*1*]	5	[101*1*]	−5
[011*1*]	7	[100*1*]	−7

2.3.4 Unsigned Multiplication

Integers x and y in the range $0 \leq x, y \leq 2^w - 1$ can be represented as w-bit unsigned numbers, but their product $x \cdot y$ can range between 0 and $(2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$. This could require as many as $2w$ bits to represent. Instead, unsigned multiplication in C is defined to yield the w-bit value given by the low-order w bits of the $2w$-bit integer product. Let us denote this value as $x *^u_w y$.

Truncating an unsigned number to w bits is equivalent to computing its value modulo 2^w, giving the following:

PRINCIPLE: Unsigned multiplication

For x and y such that $0 \leq x, y \leq UMax_w$:

$$x *_w^u y = (x \cdot y) \bmod 2^w \tag{2.16}$$

■

2.3.5 Two's-Complement Multiplication

Integers x and y in the range $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$ can be represented as w-bit two's-complement numbers, but their product $x \cdot y$ can range between $-2^{w-1} \cdot (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$ and $-2^{w-1} \cdot -2^{w-1} = 2^{2w-2}$. This could require as many as $2w$ bits to represent in two's-complement form. Instead, signed multiplication in C generally is performed by truncating the $2w$-bit product to w bits. We denote this value as $x *_w^t y$. Truncating a two's-complement number to w bits is equivalent to first computing its value modulo 2^w and then converting from unsigned to two's complement, giving the following:

PRINCIPLE: Two's-complement multiplication

For x and y such that $TMin_w \leq x, y \leq TMax_w$:

$$x *_w^t y = U2T_w((x \cdot y) \bmod 2^w) \tag{2.17}$$

■

We claim that the bit-level representation of the product operation is identical for both unsigned and two's-complement multiplication, as stated by the following principle:

PRINCIPLE: Bit-level equivalence of unsigned and two's-complement multiplication

Let $\vec{x}$ and $\vec{y}$ be bit vectors of length w. Define integers x and y as the values represented by these bits in two's-complement form: $x = B2T_w(\vec{x})$ and $y = B2T_w(\vec{y})$. Define nonnegative integers x' and y' as the values represented by these bits in unsigned form: $x' = B2U_w(\vec{x})$ and $y' = B2U_w(\vec{y})$. Then

$$T2B_w(x *_w^t y) = U2B_w(x' *_w^u y')$$

■

As illustrations, Figure 2.27 shows the results of multiplying different 3-bit numbers. For each pair of bit-level operands, we perform both unsigned and two's-complement multiplication, yielding 6-bit products, and then truncate these to 3 bits. The unsigned truncated product always equals $x \cdot y \bmod 8$. The bit-level representations of both truncated products are identical for both unsigned and two's-complement multiplication, even though the full 6-bit representations differ.

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	5	[101]	3	[011]	15	[001111]	7	[111]
Two's complement	−3	[101]	3	[011]	−9	[110111]	−1	[111]
Unsigned	4	[100]	7	[111]	28	[011100]	4	[100]
Two's complement	−4	[100]	−1	[111]	4	[000100]	−4	[100]
Unsigned	3	[011]	3	[011]	9	[001001]	1	[001]
Two's complement	3	[011]	3	[011]	9	[001001]	1	[001]

Figure 2.27 Three-bit unsigned and two's-complement multiplication examples: Although the bit-level representations of the full products may differ, those of the truncated products are identical.

DERIVATION: Bit-level equivalence of unsigned and two's-complement multiplication

From Equation 2.6, we have $x' = x + x_{w-1}2^w$ and $y' = y + y_{w-1}2^w$. Computing the product of these values modulo 2^w gives the following:

$$\begin{aligned}(x' \cdot y') \bmod 2^w &= [(x + x_{w-1}2^w) \cdot (y + y_{w-1}2^w)] \bmod 2^w \\ &= [x \cdot y + (x_{w-1}y + y_{w-1}x)2^w + x_{w-1}y_{w-1}2^{2w}] \bmod 2^w \\ &= (x \cdot y) \bmod 2^w\end{aligned} \tag{2.18}$$

The terms with weight 2^w and 2^{2w} drop out due to the modulus operator. By Equation 2.17, we have $x *^{\mathrm{t}}_w y = U2T_w((x \cdot y) \bmod 2^w)$. We can apply the operation $T2U_w$ to both sides to get

$$T2U_w(x *^{\mathrm{t}}_w y) = T2U_w(U2T_w((x \cdot y) \bmod 2^w)) = (x \cdot y) \bmod 2^w$$

Combining this result with Equations 2.16 and 2.18 shows that $T2U_w(x *^{\mathrm{t}}_w y) = (x' \cdot y') \bmod 2^w = x' *^{\mathrm{u}}_w y'$. We can then apply $U2B_w$ to both sides to get

$$U2B_w(T2U_w(x *^{\mathrm{t}}_w y)) = T2B_w(x *^{\mathrm{t}}_w y) = U2B_w(x' *^{\mathrm{u}}_w y')$$

■

Practice Problem 2.34 (solution page 153)

Fill in the following table showing the results of multiplying different 3-bit numbers, in the style of Figure 2.27:

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	_____	[100]	_____	[101]	_____	_____	_____	_____
Two's complement	_____	[100]	_____	[101]	_____	_____	_____	_____
Unsigned	_____	[010]	_____	[111]	_____	_____	_____	_____
Two's complement	_____	[010]	_____	[111]	_____	_____	_____	_____

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	_____	[110]	_____	[110]	_____	_____	_____	_____
Two's complement	_____	[110]	_____	[110]	_____	_____	_____	_____

Practice Problem 2.35 (solution page 154)

You are given the assignment to develop code for a function `tmult_ok` that will determine whether two arguments can be multiplied without causing overflow. Here is your solution:

```
/* Determine whether arguments can be multiplied without overflow */
int tmult_ok(int x, int y) {
    int p = x*y;
    /* Either x is zero, or dividing p by x gives y */
    return !x || p/x == y;
}
```

You test this code for a number of values of `x` and `y`, and it seems to work properly. Your coworker challenges you, saying, "If I can't use subtraction to test whether addition has overflowed (see Problem 2.31), then how can you use division to test whether multiplication has overflowed?"

Devise a mathematical justification of your approach, along the following lines. First, argue that the case $x = 0$ is handled correctly. Otherwise, consider w-bit numbers x ($x \neq 0$), y, p, and q, where p is the result of performing two's-complement multiplication on x and y, and q is the result of dividing p by x.

1. Show that $x \cdot y$, the integer product of x and y, can be written in the form $x \cdot y = p + t2^w$, where $t \neq 0$ if and only if the computation of p overflows.
2. Show that p can be written in the form $p = x \cdot q + r$, where $|r| < |x|$.
3. Show that $q = y$ if and only if $r = t = 0$.

Practice Problem 2.36 (solution page 154)

For the case where data type `int` has 32 bits, devise a version of `tmult_ok` (Problem 2.35) that uses the 64-bit precision of data type `int64_t`, without using division.

Practice Problem 2.37 (solution page 155)

You are given the task of patching the vulnerability in the XDR code shown in the aside on page 100 for the case where both data types `int` and `size_t` are 32 bits. You decide to eliminate the possibility of the multiplication overflowing by computing the number of bytes to allocate using data type `uint64_t`. You replace

Aside Security vulnerability in the XDR library

In 2002, it was discovered that code supplied by Sun Microsystems to implement the XDR library, a widely used facility for sharing data structures between programs, had a security vulnerability arising from the fact that multiplication can overflow without any notice being given to the program.

Code similar to that containing the vulnerability is shown below:

```
1  /* Illustration of code vulnerability similar to that found in
2   * Sun's XDR library.
3   */
4  void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
5      /*
6       * Allocate buffer for ele_cnt objects, each of ele_size bytes
7       * and copy from locations designated by ele_src
8       */
9      void *result = malloc(ele_cnt * ele_size);
10     if (result == NULL)
11         /* malloc failed */
12         return NULL;
13     void *next = result;
14     int i;
15     for (i = 0; i < ele_cnt; i++) {
16         /* Copy object i to destination */
17         memcpy(next, ele_src[i], ele_size);
18         /* Move pointer to next memory region */
19         next += ele_size;
20     }
21     return result;
22 }
```

The function `copy_elements` is designed to copy `ele_cnt` data structures, each consisting of `ele_size` bytes into a buffer allocated by the function on line 9. The number of bytes required is computed as `ele_cnt * ele_size`.

Imagine, however, that a malicious programmer calls this function with `ele_cnt` being 1,048,577 ($2^{20} + 1$) and `ele_size` being 4,096 (2^{12}) with the program compiled for 32 bits. Then the multiplication on line 9 will overflow, causing only 4,096 bytes to be allocated, rather than the 4,294,971,392 bytes required to hold that much data. The loop starting at line 15 will attempt to copy all of those bytes, overrunning the end of the allocated buffer, and therefore corrupting other data structures. This could cause the program to crash or otherwise misbehave.

The Sun code was used by almost every operating system and in such widely used programs as Internet Explorer and the Kerberos authentication system. The Computer Emergency Response Team (CERT), an organization run by the Carnegie Mellon Software Engineering Institute to track security vulnerabilities and breaches, issued advisory "CA-2002-25," and many companies rushed to patch their code. Fortunately, there were no reported security breaches caused by this vulnerability.

A similar vulnerability existed in many implementations of the library function `calloc`. These have since been patched. Unfortunately, many programmers call allocation functions, such as `malloc`, using arithmetic expressions as arguments, without checking these expressions for overflow. Writing a reliable version of `calloc` is left as an exercise (Problem 2.76).

the original call to `malloc` (line 9) as follows:

```
uint64_t asize =
    ele_cnt * (uint64_t) ele_size;
void *result = malloc(asize);
```

Recall that the argument to `malloc` has type `size_t`.

A. Does your code provide any improvement over the original?

B. How would you change the code to eliminate the vulnerability?

2.3.6 Multiplying by Constants

Historically, the integer multiply instruction on many machines was fairly slow, requiring 10 or more clock cycles, whereas other integer operations—such as addition, subtraction, bit-level operations, and shifting—required only 1 clock cycle. Even on the Intel Core i7 Haswell we use as our reference machine, integer multiply requires 3 clock cycles. As a consequence, one important optimization used by compilers is to attempt to replace multiplications by constant factors with combinations of shift and addition operations. We will first consider the case of multiplying by a power of 2, and then we will generalize this to arbitrary constants.

PRINCIPLE: Multiplication by a power of 2

Let x be the unsigned integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$. Then for any $k \geq 0$, the $w+k$-bit unsigned representation of $x2^k$ is given by $[x_{w-1}, x_{w-2}, \ldots, x_0, 0, \ldots, 0]$, where k zeros have been added to the right. ■

So, for example, 11 can be represented for $w = 4$ as [1011]. Shifting this left by $k = 2$ yields the 6-bit vector [101100], which encodes the unsigned number $11 \cdot 4 = 44$.

DERIVATION: Multiplication by a power of 2

This property can be derived using Equation 2.1:

$$\begin{aligned} B2U_{w+k}([x_{w-1}, x_{w-2}, \ldots, x_0, 0, \ldots, 0]) &= \sum_{i=0}^{w-1} x_i 2^{i+k} \\ &= \left[\sum_{i=0}^{w-1} x_i 2^i\right] \cdot 2^k \\ &= x2^k \end{aligned}$$

■

When shifting left by k for a fixed word size, the high-order k bits are discarded, yielding

$$[x_{w-k-1}, x_{w-k-2}, \ldots, x_0, 0, \ldots, 0]$$

but this is also the case when performing multiplication on fixed-size words. We can therefore see that shifting a value left is equivalent to performing unsigned multiplication by a power of 2:

PRINCIPLE: Unsigned multiplication by a power of 2

For C variables x and k with unsigned values x and k, such that $0 \leq k < w$, the C expression x << k yields the value $x *^{u}_{w} 2^{k}$. ■

Since the bit-level operation of fixed-size two's-complement arithmetic is equivalent to that for unsigned arithmetic, we can make a similar statement about the relationship between left shifts and multiplication by a power of 2 for two's-complement arithmetic:

PRINCIPLE: Two's-complement multiplication by a power of 2

For C variables x and k with two's-complement value x and unsigned value k, such that $0 \leq k < w$, the C expression x << k yields the value $x *^{t}_{w} 2^{k}$. ■

Note that multiplying by a power of 2 can cause overflow with either unsigned or two's-complement arithmetic. Our result shows that even then we will get the same effect by shifting. Returning to our earlier example, we shifted the 4-bit pattern [1011] (numeric value 11) left by two positions to get [101100] (numeric value 44). Truncating this to 4 bits gives [1100] (numeric value $12 = 44 \bmod 16$).

Given that integer multiplication is more costly than shifting and adding, many C compilers try to remove many cases where an integer is being multiplied by a constant with combinations of shifting, adding, and subtracting. For example, suppose a program contains the expression x*14. Recognizing that $14 = 2^3 + 2^2 + 2^1$, the compiler can rewrite the multiplication as (x<<3) + (x<<2) + (x<<1), replacing one multiplication with three shifts and two additions. The two computations will yield the same result, regardless of whether x is unsigned or two's complement, and even if the multiplication would cause an overflow. Even better, the compiler can also use the property $14 = 2^4 - 2^1$ to rewrite the multiplication as (x<<4) - (x<<1), requiring only two shifts and a subtraction.

Practice Problem 2.38 (solution page 155)

As we will see in Chapter 3, the LEA instruction can perform computations of the form (a<<k) + b, where k is either 0, 1, 2, or 3, and b is either 0 or some program value. The compiler often uses this instruction to perform multiplications by constant factors. For example, we can compute 3*a as (a<<1) + a.

Considering cases where b is either 0 or equal to a, and all possible values of k, what multiples of a can be computed with a single LEA instruction?

Generalizing from our example, consider the task of generating code for the expression x * K, for some constant K. The compiler can express the binary representation of K as an alternating sequence of zeros and ones:

$$[(0\ldots0)\ (1\ldots1)\ (0\ldots0)\cdots(1\ldots1)]$$

For example, 14 can be written as $[(0\ldots0)(111)(0)]$. Consider a run of ones from bit position n down to bit position m ($n \geq m$). (For the case of 14, we have $n = 3$ and $m = 1$.) We can compute the effect of these bits on the product using either of two different forms:

Form A: (x<<n) + (x<<($n-1$)) + $\cdots$ + (x<<m)

Form B: (x<<($n+1$)) - (x<<m)

By adding together the results for each run, we are able to compute x * K without any multiplications. Of course, the trade-off between using combinations of shifting, adding, and subtracting versus a single multiplication instruction depends on the relative speeds of these instructions, and these can be highly machine dependent. Most compilers only perform this optimization when a small number of shifts, adds, and subtractions suffice.

Practice Problem 2.39 (solution page 156)

How could we modify the expression for form B for the case where bit position n is the most significant bit?

Practice Problem 2.40 (solution page 156)

For each of the following values of K, find ways to express x * K using only the specified number of operations, where we consider both additions and subtractions to have comparable cost. You may need to use some tricks beyond the simple form A and B rules we have considered so far.

K	Shifts	Add/Subs	Expression
6	2	1	_____
31	1	1	_____
−6	2	1	_____
55	2	2	_____

Practice Problem 2.41 (solution page 156)

For a run of ones starting at bit position n down to bit position m ($n \geq m$), we saw that we can generate two forms of code, A and B. How should the compiler decide which form to use?

2.3.7 Dividing by Powers of 2

Integer division on most machines is even slower than integer multiplication—requiring 30 or more clock cycles. Dividing by a power of 2 can also be performed

k	>> k (binary)	Decimal	$12{,}340/2^k$
0	0011000000110100	12,340	12,340.0
1	0001100000011010	6,170	6,170.0
4	0000001100000011	771	771.25
8	0000000000110000	48	48.203125

Figure 2.28 Dividing unsigned numbers by powers of 2. The examples illustrate how performing a logical right shift by k has the same effect as dividing by 2^k and then rounding toward zero.

using shift operations, but we use a right shift rather than a left shift. The two different right shifts—logical and arithmetic—serve this purpose for unsigned and two's-complement numbers, respectively.

Integer division always rounds toward zero. To define this precisely, let us introduce some notation. For any real number a, define $\lfloor a \rfloor$ to be the unique integer a' such that $a' \leq a < a' + 1$. As examples, $\lfloor 3.14 \rfloor = 3$, $\lfloor -3.14 \rfloor = -4$, and $\lfloor 3 \rfloor = 3$. Similarly, define $\lceil a \rceil$ to be the unique integer a' such that $a' - 1 < a \leq a'$. As examples, $\lceil 3.14 \rceil = 4$, $\lceil -3.14 \rceil = -3$, and $\lceil 3 \rceil = 3$. For $x \geq 0$ and $y > 0$, integer division should yield $\lfloor x/y \rfloor$, while for $x < 0$ and $y > 0$, it should yield $\lceil x/y \rceil$. That is, it should round down a positive result but round up a negative one.

The case for using shifts with unsigned arithmetic is straightforward, in part because right shifting is guaranteed to be performed logically for unsigned values.

PRINCIPLE: Unsigned division by a power of 2

For C variables x and k with unsigned values x and k, such that $0 \leq k < w$, the C expression x >> k yields the value $\lfloor x/2^k \rfloor$. ■

As examples, Figure 2.28 shows the effects of performing logical right shifts on a 16-bit representation of 12,340 to perform division by 1, 2, 16, and 256. The zeros shifted in from the left are shown in italics. We also show the result we would obtain if we did these divisions with real arithmetic. These examples show that the result of shifting consistently rounds toward zero, as is the convention for integer division.

DERIVATION: Unsigned division by a power of 2

Let x be the unsigned integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and let k be in the range $0 \leq k < w$. Let x' be the unsigned number with $w - k$ bit representation $[x_{w-1}, x_{w-2}, \ldots, x_k]$, and let x'' be the unsigned number with k-bit representation $[x_{k-1}, \ldots, x_0]$. We can therefore see that $x = 2^k x' + x''$, and that $0 \leq x'' < 2^k$. It therefore follows that $\lfloor x/2^k \rfloor = x'$.

Performing a logical right shift of bit vector $[x_{w-1}, x_{w-2}, \ldots, x_0]$ by k yields the bit vector

$$[0, \ldots, 0, x_{w-1}, x_{w-2}, \ldots, x_k]$$

k	>> k (binary)	Decimal	$-12{,}340/2^k$
0	1100111111001100	−12,340	−12,340.0
1	1110011111100110	−6,170	−6,170.0
4	1111110011111100	−772	−771.25
8	1111111111001111	−49	−48.203125

Figure 2.29 Applying arithmetic right shift. The examples illustrate that arithmetic right shift is similar to division by a power of 2, except that it rounds down rather than toward zero.

This bit vector has numeric value x', which we have seen is the value that would result by computing the expression x >> k. ■

The case for dividing by a power of 2 with two's-complement arithmetic is slightly more complex. First, the shifting should be performed using an *arithmetic* right shift, to ensure that negative values remain negative. Let us investigate what value such a right shift would produce.

PRINCIPLE: Two's-complement division by a power of 2, rounding down

Let C variables x and k have two's-complement value x and unsigned value k, respectively, such that $0 \leq k < w$. The C expression x >> k, when the shift is performed arithmetically, yields the value $\lfloor x/2^k \rfloor$. ■

For $x \geq 0$, variable x has 0 as the most significant bit, and so the effect of an arithmetic shift is the same as for a logical right shift. Thus, an arithmetic right shift by k is the same as division by 2^k for a nonnegative number. As an example of a negative number, Figure 2.29 shows the effect of applying arithmetic right shift to a 16-bit representation of −12,340 for different shift amounts. For the case when no rounding is required ($k = 1$), the result will be $x/2^k$. When rounding is required, shifting causes the result to be rounded downward. For example, the shifting right by four has the effect of rounding −771.25 down to −772. We will need to adjust our strategy to handle division for negative values of x.

DERIVATION: Two's-complement division by a power of 2, rounding down

Let x be the two's-complement integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and let k be in the range $0 \leq k < w$. Let x' be the two's-complement number represented by the $w - k$ bits $[x_{w-1}, x_{w-2}, \ldots, x_k]$, and let x'' be the *unsigned* number represented by the low-order k bits $[x_{k-1}, \ldots, x_0]$. By a similar analysis as the unsigned case, we have $x = 2^k x' + x''$ and $0 \leq x'' < 2^k$, giving $x' = \lfloor x/2^k \rfloor$. Furthermore, observe that shifting bit vector $[x_{w-1}, x_{w-2}, \ldots, x_0]$ right *arithmetically* by k yields the bit vector

$$[x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_k]$$

which is the sign extension from $w - k$ bits to w bits of $[x_{w-1}, x_{w-2}, \ldots, x_k]$. Thus, this shifted bit vector is the two's-complement representation of $\lfloor x/2^k \rfloor$. ■

k	Bias	−12,340 + bias (binary)	>> k (binary)	Decimal	$-12{,}340/2^k$
0	0	1100111111001100	1100111111001100	−12,340	−12,340.0
1	1	1100111111001101	1110011111100110	−6,170	−6,170.0
4	15	1100111111011011	1111110011111101	−771	−771.25
8	255	1101000011001011	1111111111010000	−48	−48.203125

Figure 2.30 Dividing two's-complement numbers by powers of 2. By adding a bias before the right shift, the result is rounded toward zero.

We can correct for the improper rounding that occurs when a negative number is shifted right by "biasing" the value before shifting.

PRINCIPLE: Two's-complement division by a power of 2, rounding up

Let C variables x and k have two's-complement value x and unsigned value k, respectively, such that $0 \leq k < w$. The C expression `(x + (1 << k) - 1) >> k`, when the shift is performed arithmetically, yields the value $\lceil x/2^k \rceil$. ■

Figure 2.30 demonstrates how adding the appropriate bias before performing the arithmetic right shift causes the result to be correctly rounded. In the third column, we show the result of adding the bias value to −12,340, with the lower k bits (those that will be shifted off to the right) shown in italics. We can see that the bits to the left of these may or may not be incremented. For the case where no rounding is required ($k = 1$), adding the bias only affects bits that are shifted off. For the cases where rounding is required, adding the bias causes the upper bits to be incremented, so that the result will be rounded toward zero.

The biasing technique exploits the property that $\lceil x/y \rceil = \lfloor (x + y - 1)/y \rfloor$ for integers x and y such that $y > 0$. As examples, when $x = -30$ and $y = 4$, we have $x + y - 1 = -27$ and $\lceil -30/4 \rceil = -7 = \lfloor -27/4 \rfloor$. When $x = -32$ and $y = 4$, we have $x + y - 1 = -29$ and $\lceil -32/4 \rceil = -8 = \lfloor -29/4 \rfloor$.

DERIVATION: Two's-complement division by a power of 2, rounding up

To see that $\lceil x/y \rceil = \lfloor (x + y - 1)/y \rfloor$, suppose that $x = qy + r$, where $0 \leq r < y$, giving $(x + y - 1)/y = q + (r + y - 1)/y$, and so $\lfloor (x + y - 1)/y \rfloor = q + \lfloor (r + y - 1)/y \rfloor$. The latter term will equal 0 when $r = 0$ and 1 when $r > 0$. That is, by adding a bias of $y - 1$ to x and then rounding the division downward, we will get q when y divides x and $q + 1$ otherwise.

Returning to the case where $y = 2^k$, the C expression `x + (1 << k) - 1` yields the value $x + 2^k - 1$. Shifting this right arithmetically by k therefore yields $\lceil x/2^k \rceil$. ■

These analyses show that for a two's-complement machine using arithmetic right shifts, the C expression

```
(x<0 ? x+(1<<k)-1 : x) >> k
```

will compute the value $x/2^k$.

Practice Problem 2.42 (solution page 156)

Write a function `div16` that returns the value `x/16` for integer argument `x`. Your function should not use division, modulus, multiplication, any conditionals (`if` or `?:`), any comparison operators (e.g., `<`, `>`, or `==`), or any loops. You may assume that data type `int` is 32 bits long and uses a two's-complement representation, and that right shifts are performed arithmetically.

We now see that division by a power of 2 can be implemented using logical or arithmetic right shifts. This is precisely the reason the two types of right shifts are available on most machines. Unfortunately, this approach does not generalize to division by arbitrary constants. Unlike multiplication, we cannot express division by arbitrary constants K in terms of division by powers of 2.

Practice Problem 2.43 (solution page 157)

In the following code, we have omitted the definitions of constants `M` and `N`:

```
#define M       /* Mystery number 1 */
#define N       /* Mystery number 2 */
int arith(int x, int y) {
    int result = 0;
    result = x*M + y/N; /* M and N are mystery numbers. */
    return result;
}
```

We compiled this code for particular values of `M` and `N`. The compiler optimized the multiplication and division using the methods we have discussed. The following is a translation of the generated machine code back into C:

```
/* Translation of assembly code for arith */
int optarith(int x, int y) {
    int t = x;
    x <<= 5;
    x -= t;
    if (y < 0) y += 7;
    y >>= 3;  /* Arithmetic shift */
    return x+y;
}
```

What are the values of `M` and `N`?

2.3.8 Final Thoughts on Integer Arithmetic

As we have seen, the "integer" arithmetic performed by computers is really a form of modular arithmetic. The finite word size used to represent numbers

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 157)

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

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

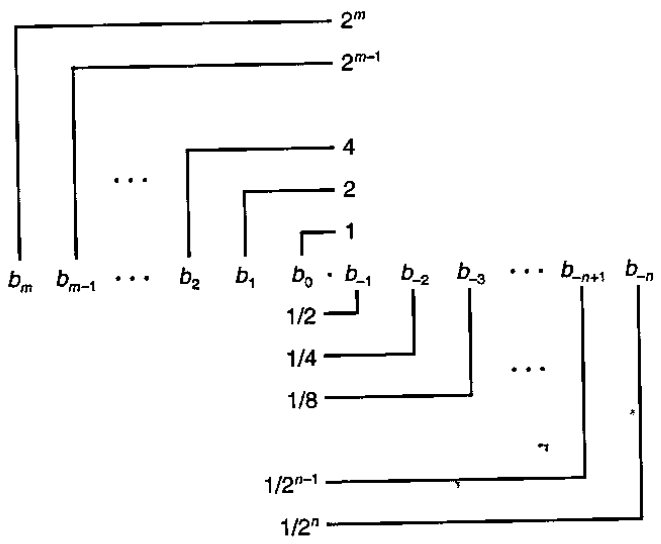


Figure 2.31 Fractional binary representation. Digits to the left of the binary point have weights of the form 2^i, while those to the right have weights of the form $1/2^i$.

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

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

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 $0.11\cdots1_2$ 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 (solution page 157)

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 158)

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: $x = 0.00011001100110011001100_2$. (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 5×2^{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.

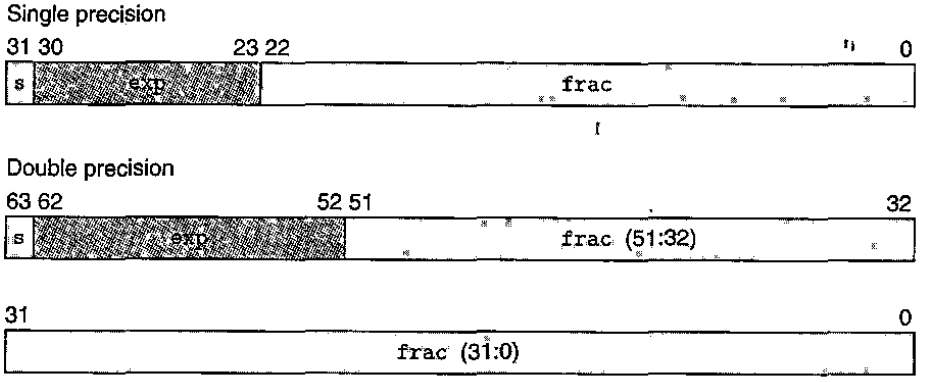


Figure 2.32 Standard floating-point formats. Floating-point numbers are represented by three fields. For the two most common formats, these are packed in 32-bit (single-precision) or 64-bit (double-precision) words.

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 $E = e - Bias$, 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 $0 \leq f < 1$, 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.

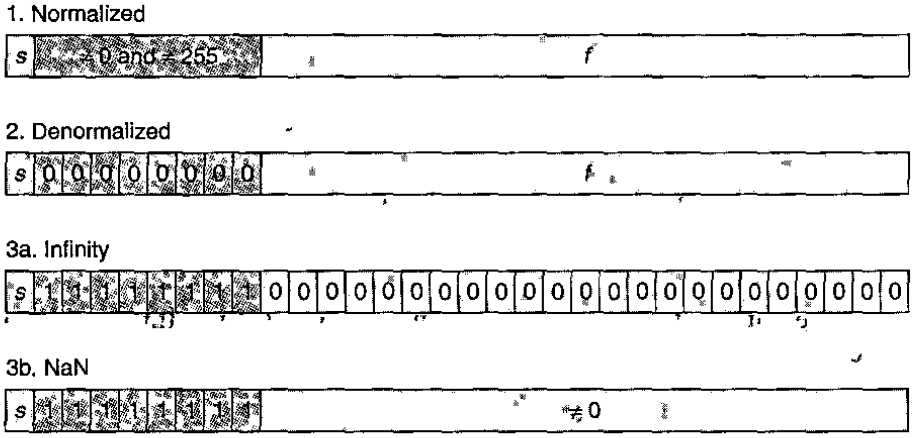


Figure 2.33 Categories of single-precision floating-point values. The value of the exponent determines whether the number is (1) normalized, (2) denormalized, or (3) a special value.

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.

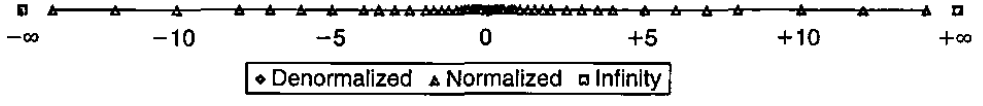
Case 3: Special Values

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.

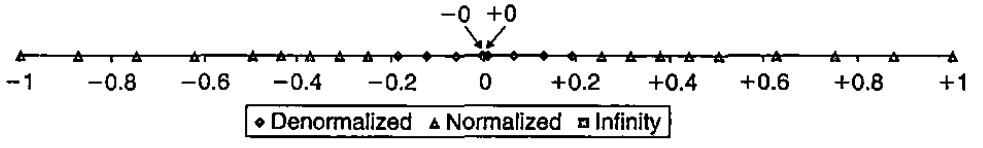
2.4.3 Example Numbers

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 ± 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	—	—	—	—	—	—	∞	—

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 158)

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	———	———	———	———	———	———	———	———

Bits	e	E	2^E	f	M	$2^E \times M$	V	Decimal
0 01 01			1	0	1	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{5}{4}$
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.

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

Figure 2.36 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 2.15 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 159)

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 159)

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 159)

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 (solution page 159)

We saw in Problem 2.46 that the Patriot missile software approximated 0.1 as $x = 0.0001100110011001100110011001100_2$. 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 160)

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 NaNs, 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 $x +^f a \geq x +^f b$ 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 $x *^f y$ 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:

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

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

In addition, we are also guaranteed that $a *^f a \geq 0$, 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 160)

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 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 $[10\cdots00]$ ($TMin_w$ 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.

Practice Problem 2.54 (solution page 160)

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

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 [73, 33] 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 five 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.

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 $x \times 2^y$. IEEE Standard 754 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 [45, 61] discuss properties of the different data types and operations. Of these two, only Steele and Harbison [45] cover the newer features found in ISO C99. There do not yet seem to be any books that cover the features found in ISO C11. 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, 74] 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 [97] 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 [5]) describe the data formats and arithmetic operations supported by Java.

Most books on logic design [58, 116] have a section on encodings and arithmetic operations. Such books describe different ways of implementing arithmetic circuits. Overton's book on IEEE floating point [82] 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`, `long`, 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 >=).
- 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`.
  - Casting between data types `int` and `unsigned`, either explicitly or implicitly.

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 128), 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 data type `int` and yields 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 128), 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 128), except that you may assume that data type `int` has $w = 32$ bits.

Your code should contain a total of at most 12 arithmetic, bitwise, 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 128), except that you may assume that data type `int` has $w = 32$ bits.

Your code should contain a total of at most 15 arithmetic, bitwise, 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 --> 0x3F, n = 17 --> 0x1FFFF
 * Assume 1 <= n <= w
 */
int lower_one_mask(int n);
```

Your function should follow the bit-level integer coding rules (page 128). 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 128). 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 128).

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 128).

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 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 ◆

The library function `calloc` has the following declaration:

```
void *calloc(size_t nmemb, size_t size);
```

According to the library documentation, "The `calloc` function allocates memory for an array of `nmemb` elements of `size` bytes each. The memory is set to zero. If `nmemb` or `size` is zero, then `calloc` returns `NULL`."

Write an implementation of `calloc` that performs the allocation by a call to `malloc` and sets the memory to zero via `memset`. Your code should not have any vulnerabilities due to arithmetic overflow, and it should work correctly regardless of the number of bits used to represent data of type `size_t`.

As a reference, functions `malloc` and `memset` have the following declarations:

```
void *malloc(size_t size);
void *memset(void *s, int c, size_t n);
```

2.77 ◆◆
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. $K = -112$

2.78 ◆◆
Write code for a function with the following prototype:

```
/* Divide by power of 2. 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 128).

2.79 ◆◆
Write code for a function `mul3div4` that, for integer argument x, computes 3 * x/4 but follows the bit-level integer coding rules (page 128). Your code should replicate the fact that the computation 3*x can cause overflow.

2.80 ◆◆◆
Write code for a function `threefourths` that, 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 128).

2.81 ◆◆
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. $0^{w-k-j}1^k0^j$

2.82 ◆
We are running programs 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`

2.83 ◆◆

Consider numbers having a binary representation consisting of an infinite string of the form $0.yyyyyy\cdots$, 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

2.84 ◆

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           ;
}
```

2.85 ◆

Given a floating-point format with a k-bit exponent and an n-bit fraction, write formulas for the exponent E, the 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

2.86 ◆

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	_____	_____

This format can be used in C programs compiled for Intel-compatible machines by declaring the data to be of type `long double`. However, it forces the compiler to generate code based on the legacy 8087 floating-point instructions. The resulting program will most likely run much slower than would be the case for data type `float` or `double`.

2.87 ◆

The 2008 version of the IEEE floating-point standard, named IEEE 754-2008, includes a 16-bit "half-precision" floating-point format. It was originally devised by computer graphics companies for storing data in which a higher dynamic range is required than can be achieved with 16-bit integers. This format has 1 sign bit, 5 exponent bits ($k = 5$), and 10 fraction bits ($n = 10$). The exponent bias is $2^{5-1} - 1 = 15$.

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.

D: The (possibly approximate) numerical value, as is printed using the `%f` formatting specification of `printf`.

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 01110_2 (decimal value $15 - 1 = 14$) and a significand field of 1100000000_2, giving a hex representation 3B00. The numerical value is 0.875.

You need not fill in entries marked —.

Description	Hex	M	E	V	D
-0	_____	_____	_____	-0	-0.0
Smallest value > 2	_____	_____	_____	_____	_____
512	_____	_____	_____	512	512.0
Largest denormalized	_____	_____	_____	_____	_____
$-\infty$	_____	—	—	$-\infty$	$-\infty$
Number with hex representation 3BB0	3BB0	_____	_____	_____	_____

2.88 ◆◆

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

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

In the following table, 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	_____	_____	_____

2.89 ◆

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`

2.90 ◆

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);
}
```

2.91 ◆

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 2.83.

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.

To 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;
}
```

2.92 ◆◆

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_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.

2.93 ◆◆

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.

2.94 ◆◆◆

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.

2.95 ◆◆◆

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.

2.96 ◆◆◆◆

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.

2.97 ◆◆◆◆

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 37)

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 100110111001110110101 to hexadecimal:

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

Solution to Problem 2.2 (page 37)

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 38)
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 39)
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 48)
This problem tests your understanding of the byte representation of data and the two different byte orderings.

A.	Little endian: 21	Big endian: 87
B.	Little endian: 21 43	Big endian: 87 65
C.	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 49)
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 49)
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 51)
This problem is a drill to help you become more familiar with Boolean operations.

Operation	Result
a	[01101001]
b	[01010101]
~*a*	[10010110]
~*b*	[10101010]
a & *b*	[01000001]
a | *b*	[01111101]
a ^ *b*	[00111100]

Solution to Problem 2.9 (page 53)
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 54)
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 55)
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 55)
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 when data type `int` is 32 bits.

Solution to Problem 2.13 (page 56)

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 57)

This problem highlights the relation between bit-level Boolean operations and logical operations in C. A common programming error is to use a bit-level operation when a logical 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 57)

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 58)

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 65)

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.

Hexadecimal	Binary		
$\vec{x}$		$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
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 69)

For a 32-bit word, any value consisting of 8 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 7 digits. Filling this out with a leading zero gives 0x08048337, a positive number.

```
4004d0:  48 81 ec e0 02 00 00    sub   $0x2e0,%rsp              A. 736
4004d7:  48 8b 44 24 a8          mov   -0x58(%rsp),%rax         B. -88
4004dc:  48 03 47 28             add   0x28(%rdi),%rax          C.  40
4004e0:  48 89 44 24 d0          mov   %rax,-0x30(%rsp)         D. -48
4004e5:  48 8b 44 24 78          mov   0x78(%rsp),%rax          E. 120
4004ea:  48 89 87 88 00 00 00    mov   %rax,0x88(%rdi)          F. 136
4004f1:  48 8b 84 24 f8 01 00    mov   0x1f8(%rsp),%rax         G. 504
4004f8:  00
4004f9:  48 03 44 24 08          add   0x8(%rsp),%rax
4004fe:  48 89 84 24 c0 00 00    mov   %rax,0xc0(%rsp)          H. 192
400505:  00
400506:  48 8b 44 d4 b8          mov   -0x48(%rsp,%rdx,8),%rax  I. -72
```

Solution to Problem 2.19 (page 71)

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 73)

This exercise tests your understanding of Equation 2.5.

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 76)

This problem reinforces your understanding of the relation between two's-complement and unsigned representations, as well as 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 79)

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 80)

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 82)

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 83)

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 83)

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 89)

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 89)

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 $(-_4^u x) + x = 16$. 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 93)

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 94)

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 94)

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 94)

This function will give correct values, except when y is *TMin*. In this case, we will have -y also equal to *TMin*, and so the call to function `tadd_ok` will indicate overflow when x is negative and no overflow when x is nonnegative. In fact, the opposite is true: `tsub_ok(x, TMin)` should yield 0 when x is negative and 1 when it is nonnegative.

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 95)

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		$-_4 x$	
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 98)

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 complement	−4	[100]	−3	[101]	12	[001100]	−4	[100]
Unsigned	2	[010]	7	[111]	14	[001110]	6	[110]
Two's complement	2	[010]	−1	[111]	−2	[111110]	−2	[110]
Unsigned	6	[110]	6	[110]	36	[100100]	4	[100]
Two's complement	−2	[110]	−2	[110]	4	[000100]	−4	[100]

Solution to Problem 2.35 (page 99)

It is 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 2.6, 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 99)

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 the arguments can be multiplied
2     without overflow */
3  int tmult_ok(int x, int y) {
4      /* Compute product without overflow */
5      int64_t pll = (int64_t) x*y;
6      /* See if casting to int preserves value */
7      return pll == (int) pll;
8  }
```

Note that the casting on the right-hand side of line 5 is critical. If we instead wrote the line as

```
int64_t 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 99)

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 9):

```
uint64_t required_size = ele_cnt * (uint64_t) 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 102)

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 103)
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 103)
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 103)
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 $m > 0$. 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 $m = 0$, 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 107)
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 107)

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 108)

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 2,147,483,647 ($TMax_{32}$).

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 111)

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

$\frac{1}{8}$	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 111)

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 the 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. $0.343 \times 2{,}000 \approx 687$ meters.

Solution to Problem 2.47 (page 117)

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 119)

Hexadecimal 0x359141 is equivalent to binary [11010110010001010000001]. Shifting this right 21 places gives $1.10101100100010100000001_2 \times 2^{21}$. We form the fraction field by dropping the leading 1 and adding two zeros, 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 120)

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 zeros, 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 121)

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 122)

A. Looking at the nonterminating sequence for $\frac{1}{10}$, we see that the 2 bits to the right of the rounding position are 1, so a better approximation to $\frac{1}{10}$ 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. $0.086 \times 2{,}000 \approx 171$ meters.

Solution to Problem 2.52 (page 122)

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 125)

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 125)

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.