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

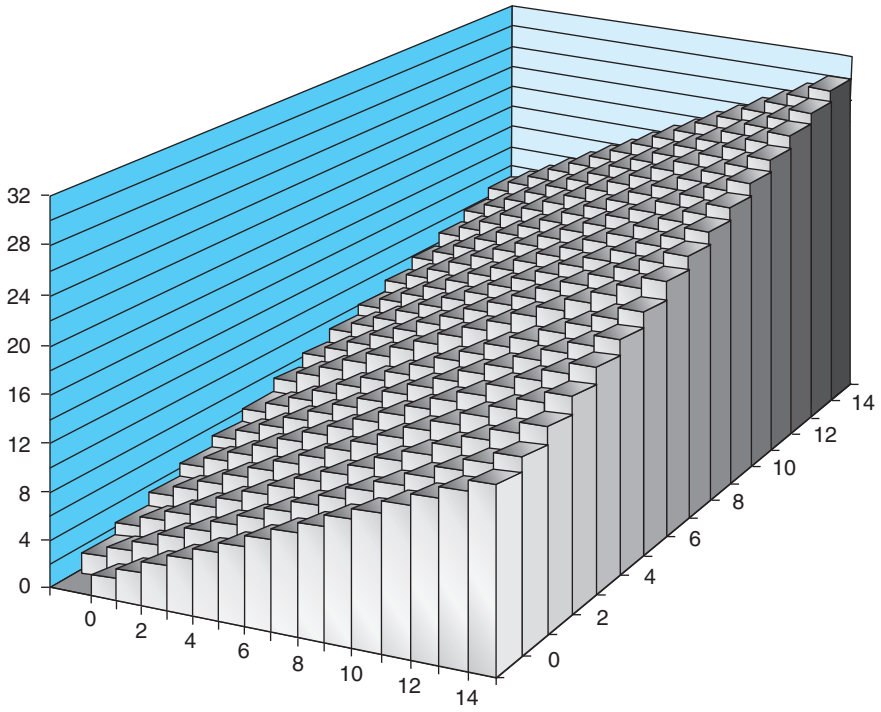

Figure 2.21 **Integer addition.** With a 4-bit word size, the sum could require 5 bits.

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 $+_w^{\mathrm{u}}$ 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 $+^{\mathrm{u}}_{w}$ as follows:

PRINCIPLE: Unsigned addition

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

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

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 $x +^{\mathrm{u}}_{w} y$ 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 $2^w \le x + y < 2^{w+1}$, 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

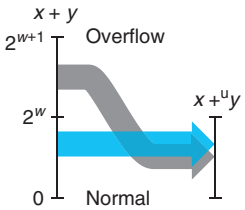

Figure 2.22 Relation between integer addition and unsigned addition. When $x + y$ is greater than $2^w - 1$, the sum overflows.

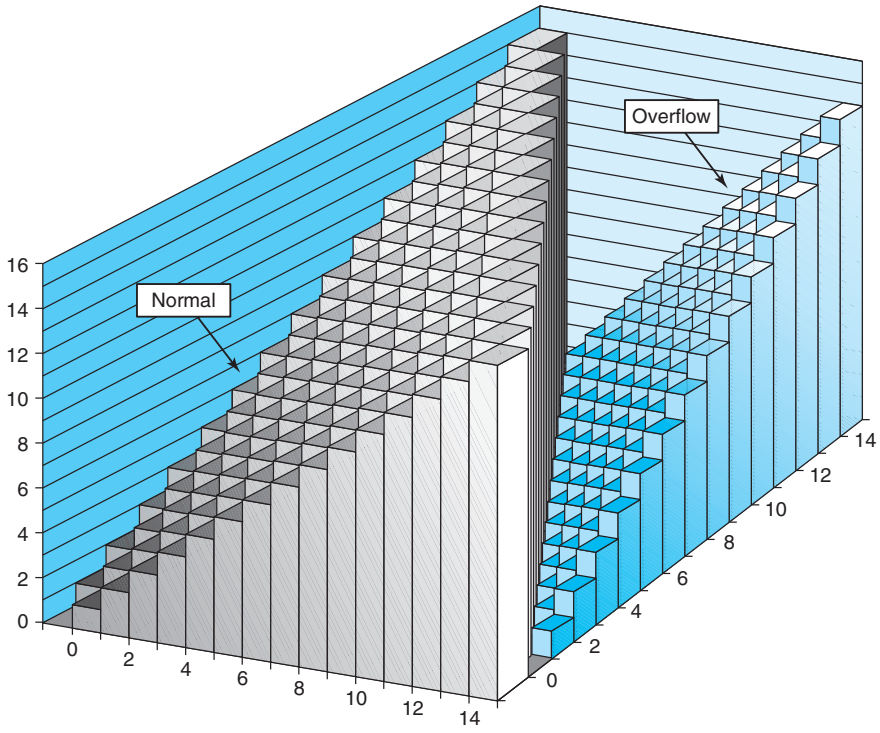

Figure 2.23 Unsigned addition. With a 4-bit word size, addition is performed modulo 16.

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 $x +^{\mathrm{u}}_4 y$ 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 $s \doteq x +^{\mathrm{u}}_w y$. 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 $9 +^{\mathrm{u}}_4 12 = 5$. 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 188)

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 188)

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
1			
4			
7			
A			
E			

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 $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$, their sum is in the range $-2^w \leq x + y \leq 2^w - 2$, 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 $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$:

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

This principle is illustrated in Figure 2.24, where the sum $x + y$ is shown on the left, having a value in the range $-2^w \leq x + y \leq 2^w - 2$, 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:

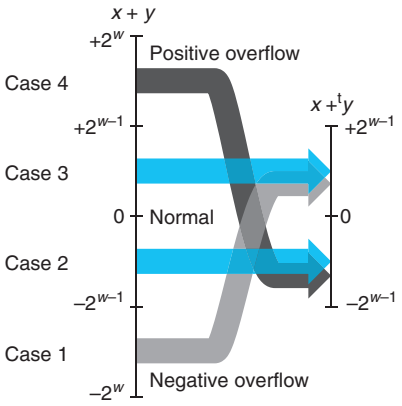

Figure 2.24
Relation between integer and two's-complement addition. When $x + y$ is less than -2^{w-1}, there is a negative overflow. When it is greater than or equal to 2^{w-1}, there is a positive overflow.

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

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 $+_w^{\mathrm{u}}$ is simply addition modulo 2^w, along with the properties of modular addition, we then have

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

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 $x +_w^{\mathrm{t}} y$. 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.

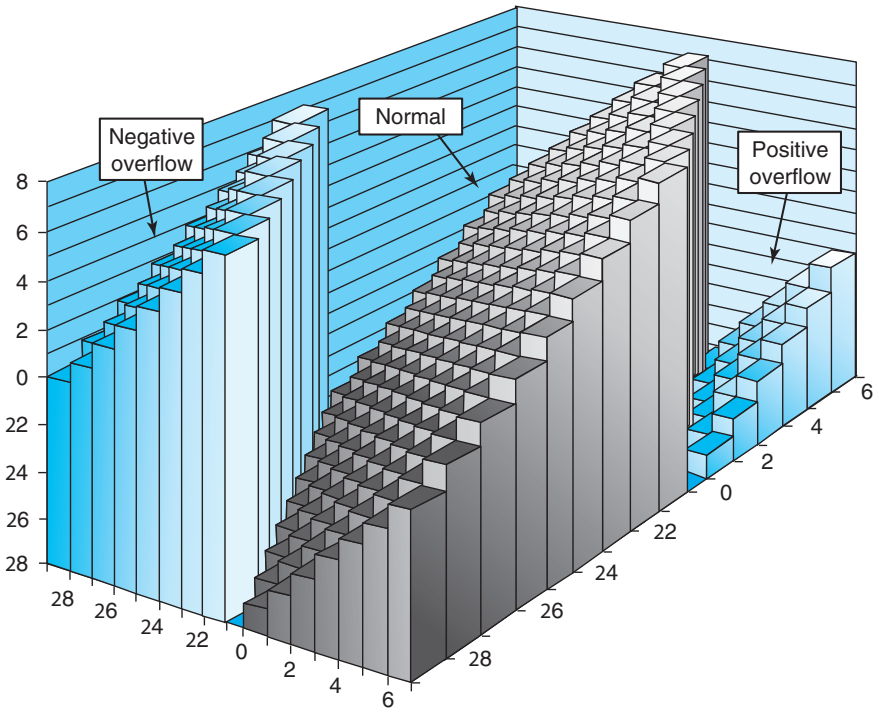

Figure 2.26 Two's-complement addition. With a 4-bit word size, addition can have a negative overflow when $x + y < -8$ and a positive overflow when $x + y \geq 8$.

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 188)

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 189)

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 189)

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 189)

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 189)

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
2			
3			
9			
B			
C			

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:

$\vec{x}$		$\sim\vec{x}$		$incr(\sim\vec{x})$	
[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 $1, 0, \ldots, 0$ 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 $x *^{\mathrm{u}}_w y$.

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 $x *_w^{\mathrm{t}} y = U2T_w((x \cdot y) \bmod 2^w)$. We can apply the operation $T2U_w$ to both sides to get

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

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

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

■

Practice Problem 2.34 (solution page 189)

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 190)

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 190)

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 191)

You are given the task of patching the vulnerability in the XDR code shown in the aside on page 136 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 $x *_w^{\mathrm{u}} 2^k$. ■

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 $x *_w^{\mathrm{t}} 2^k$. ■

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 191)

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<<(`$n-1$`)) + ··· + (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 192)**

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 192)**

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
7	1	1	
30	4	3	
28	2	1	
55	2	2	

Practice Problem 2.41 **(solution page 192)**

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	*0*001100000011010	6,170	6,170.0
4	*0000*001100000011	771	771.25
8	*00000000*00110000	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 $0 \le k < w$. 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 $0 \le r < y$, 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 192)

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 193)

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 193)**

Assume data type `int` is 32 bits long and uses a two's-complement representation for signed values. Right shifts are performed arithmetically for signed values and logically for unsigned values. The variables are declared and initialized as follows:

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

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

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

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

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

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

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

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

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

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

2.4 Floating Point

A floating-point representation encodes rational numbers of the form $V = x \times 2^y$. It is useful for performing computations involving very large numbers ($|V| \gg 0$),