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	100
$B2U_w$	Function	Binary to unsigned	98
$U2B_w$	Function	Unsigned to binary	100
$U2T_w$	Function	Unsigned to two's complement	107
$T2B_w$	Function	Two's complement to binary	101
$T2U_w$	Function	Two's complement to unsigned	107
$TMin_w$	Constant	Minimum two's-complement value	101
$TMax_w$	Constant	Maximum two's-complement value	101
$UMax_w$	Constant	Maximum unsigned value	99
$+^{\mathrm{t}}_w$	Operation	Two's-complement addition	126
$+^{\mathrm{u}}_w$	Operation	Unsigned addition	121
$*^{\mathrm{t}}_w$	Operation	Two's-complement multiplication	133
$*^{\mathrm{u}}_w$	Operation	Unsigned multiplication	132
$-^{\mathrm{t}}_w$	Operation	Two's-complement negation	131
$-^{\mathrm{u}}_w$	Operation	Unsigned negation	125

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):

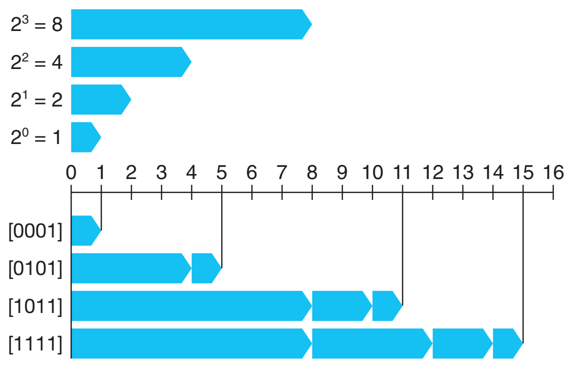

Figure 2.12
Unsigned number examples for $w = 4$. When bit i in the binary representation has value 1, it contributes 2^i to the value.

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:

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

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 $[00 \cdots 0]$ having integer value 0, and the greatest value is given by bit vector $[11 \cdots 1]$ 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:

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

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.

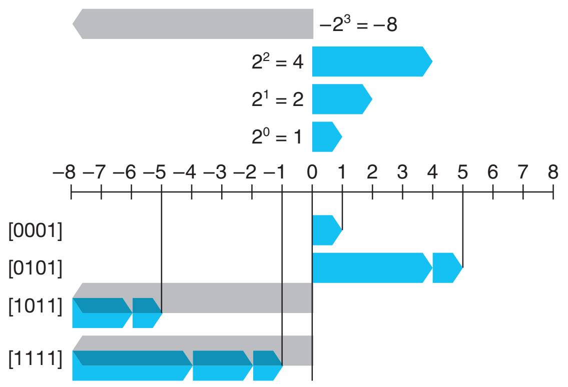

Figure 2.13
Two's-complement number examples for $w = 4$. Bit 3 serves as a sign bit; when set to 1, it contributes $-2^3 = -8$ to the value. This weighting is shown as a leftward-pointing gray bar.

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 $[10 \cdots 0]$ (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 $[01 \cdots 1]$ (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 $B2T_w\colon \{0, 1\}^w \rightarrow \{TMin_w, \ldots, TMax_w\}$. 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 184)

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	$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
`0xA`	[1010]	$2^3 + 2^1 = 10$	$-2^3 + 2^1 = -6$
`0x1`			
`0xB`			
`0x2`			
`0x7`			
`0xC`			

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, $[00 \cdots 0]$ is interpreted as $+0$. The value -0 can be represented in sign-magnitude form as $[10 \cdots 0]$ and in ones' complement as $[11 \cdots 1]$. 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 $[111 \cdots 1] - x$ (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:

```
    short x = 12345;
    short mx = -x;

    show_bytes((byte_pointer) &x, sizeof(short));
    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 185)

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:

```
short    int    v  = -12345;
unsigned short uv = (unsigned short) v;
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:

```
unsigned u = 4294967295u;   /* UMax */
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 185)

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

x	$T2U_4(x)$
-1	
-5	
-6	
-4	
1	
8	

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 $+15$.

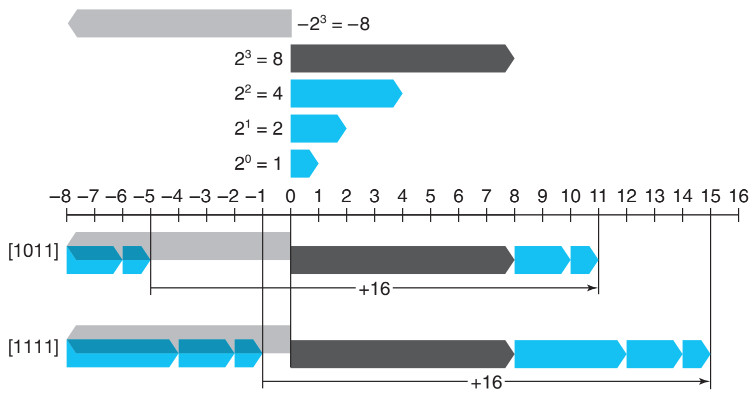

Figure 2.16
Comparing unsigned and two's-complement representations for $w = 4$. The weight of the most significant bit is -8 for two's complement and $+8$ for unsigned, yielding a net difference of 16.

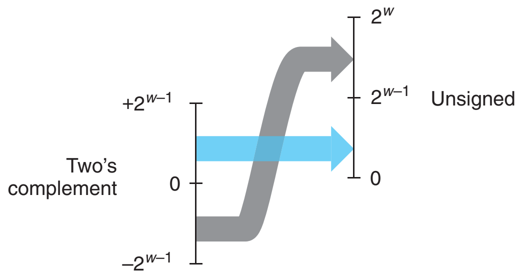

Figure 2.17
Conversion from two's complement to unsigned. Function $T2U$ converts negative numbers to large positive numbers.

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 185)

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 $0 \leq u \leq UMax_w$:

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

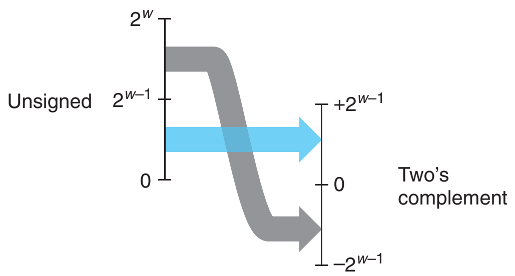

Figure 2.18
Conversion from unsigned to two's complement. Function *U2T* converts numbers greater than $2^{w-1} - 1$ to negative values.

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:

```
    int tx, ty;
    unsigned ux, uy;

    tx = (int) ux;
    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:

```
    int tx, ty;
    unsigned ux, uy;

    tx = ux; /* Cast to signed */
    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:

```
    int x = -1;
    unsigned u = 2147483648; /* 2 to the 31st */

    printf("x = %u = %d\n", x, x);
    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 185)

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 2.21, 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:

```
short sx = -12345;          /* -12345 */
unsigned short usx = sx;    /*  53191 */
int x = sx;                 /* -12345 */
unsigned ux = usx;          /*  53191 */

printf("sx  = %d:\t", sx);
show_bytes((byte_pointer) &sx, sizeof(short));
printf("usx = %u:\t", usx);
show_bytes((byte_pointer) &usx, sizeof(unsigned short));
printf("x   = %d:\t", x);
```

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

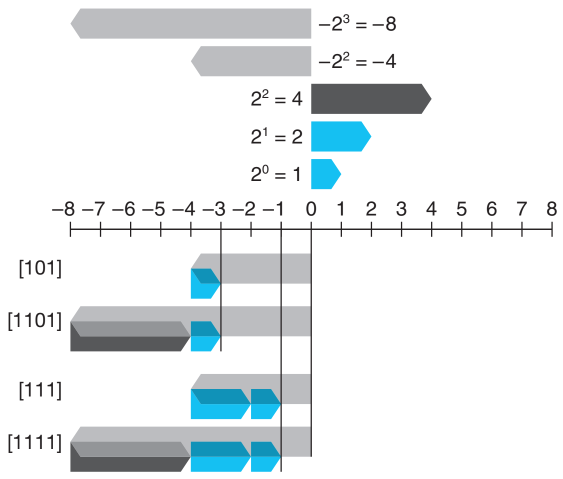

Figure 2.20 **Examples of sign extension from $w = 3$ to $w = 4$.** For $w = 4$, the combined weight of the upper 2 bits is $-8 + 4 = -4$, matching that of the sign bit for $w = 3$.

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 186)

Show that each of the following bit vectors is a two's-complement representation of −4 by applying Equation 2.3:

A. [1100]

B. [11100]

C. [111100]

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:

```
short sx = -12345;        /* -12345   */
unsigned uy = sx;         /* Mystery! */

printf("uy  = %u:\t", uy);
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 186)

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:

```
int x = 53191;
short sx = (short) x;  /* -12345 */
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

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

while the effect for two's-complement numbers is

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

Practice Problem 2.24 (solution page 186)

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
`1`	`1`	1	______	1	______
`3`	`3`	3	______	3	______
`5`	`5`	5	______	5	______
`C`	`4`	12	______	−4	______
`E`	`6`	14	______	−2	______

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 non-intuitive 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 187)

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

```
/* WARNING: This is buggy code */
float sum_elements(float a[], unsigned length) {
    int i;
    float result = 0;

    for (i = 0; i <= length-1; i++)
        result += a[i];
    return result;
}
```

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 187)

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