CHAPTER

2

Representing and Manipulating Information

Modern computers store and process information represented as two-valued signals. These lowly binary digits, or *bits*, form the basis of the digital revolution. The familiar decimal, or base-10, representation has been in use for over 1,000 years, having been developed in India, improved by Arab mathematicians in the 12th century, and brought to the West in the 13th century by the Italian mathematician Leonardo Pisano (ca. 1170 to ca. 1250), better known as Fibonacci. Using decimal notation is natural for 10-fingered humans, but binary values work better when building machines that store and process information. Two-valued signals can readily be represented, stored, and transmitted—for example, as the presence or absence of a hole in a punched card, as a high or low voltage on a wire, or as a magnetic domain oriented clockwise or counterclockwise. The electronic circuitry for storing and performing computations on two-valued signals is very simple and reliable, enabling manufacturers to integrate millions, or even billions, of such circuits on a single silicon chip.

In isolation, a single bit is not very useful. When we group bits together and apply some *interpretation* that gives meaning to the different possible bit patterns, however, we can represent the elements of any finite set. For example, using a binary number system, we can use groups of bits to encode nonnegative numbers. By using a standard character code, we can encode the letters and symbols in a document. We cover both of these encodings in this chapter, as well as encodings to represent negative numbers and to approximate real numbers.

We consider the three most important representations of numbers. *Unsigned* encodings are based on traditional binary notation, representing numbers greater than or equal to 0. *Two's-complement* encodings are the most common way to represent *signed* integers, that is, numbers that may be either positive or negative. *Floating-point* encodings are a base-2 version of scientific notation for representing real numbers. Computers implement arithmetic operations, such as addition and multiplication, with these different representations, similar to the corresponding operations on integers and real numbers.

Computer representations use a limited number of bits to encode a number, and hence some operations can *overflow* when the results are too large to be represented. This can lead to some surprising results. For example, on most of today's computers (those using a 32-bit representation for data type `int`), computing the expression

```
200 * 300 * 400 * 500
```

yields −884,901,888. This runs counter to the properties of integer arithmetic—computing the product of a set of positive numbers has yielded a negative result.

On the other hand, integer computer arithmetic satisfies many of the familiar properties of true integer arithmetic. For example, multiplication is associative and commutative, so that computing any of the following C expressions yields −884,901,888:

```
(500  *  400) * (300 * 200)
((500 *  400) * 300) * 200
((200 *  500) * 300) * 400
400   * (200 * (300 * 500))
```

The computer might not generate the expected result, but at least it is consistent!

Floating-point arithmetic has altogether different mathematical properties. The product of a set of positive numbers will always be positive, although overflow will yield the special value $+\infty$. Floating-point arithmetic is not associative due to the finite precision of the representation. For example, the C expression `(3.14+1e20)-1e20` will evaluate to 0.0 on most machines, while `3.14+(1e20-1e20)` will evaluate to 3.14. The different mathematical properties of integer versus floating-point arithmetic stem from the difference in how they handle the finiteness of their representations—integer representations can encode a comparatively small range of values, but do so precisely, while floating-point representations can encode a wide range of values, but only approximately.

By studying the actual number representations, we can understand the ranges of values that can be represented and the properties of the different arithmetic operations. This understanding is critical to writing programs that work correctly over the full range of numeric values and that are portable across different combinations of machine, operating system, and compiler. As we will describe, a number of computer security vulnerabilities have arisen due to some of the subtleties of computer arithmetic. Whereas in an earlier era program bugs would only inconvenience people when they happened to be triggered, there are now legions of hackers who try to exploit any bug they can find to obtain unauthorized access to other people's systems. This puts a higher level of obligation on programmers to understand how their programs work and how they can be made to behave in undesirable ways.

Computers use several different binary representations to encode numeric values. You will need to be familiar with these representations as you progress into machine-level programming in Chapter 3. We describe these encodings in this chapter and show you how to reason about number representations.

We derive several ways to perform arithmetic operations by directly manipulating the bit-level representations of numbers. Understanding these techniques will be important for understanding the machine-level code generated by compilers in their attempt to optimize the performance of arithmetic expression evaluation.

Our treatment of this material is based on a core set of mathematical principles. We start with the basic definitions of the encodings and then derive such properties as the range of representable numbers, their bit-level representations, and the properties of the arithmetic operations. We believe it is important for you to examine the material from this abstract viewpoint, because programmers need to have a clear understanding of how computer arithmetic relates to the more familiar integer and real arithmetic.

The C++ programming language is built upon C, using the exact same numeric representations and operations. Everything said in this chapter about C also holds for C++. The Java language definition, on the other hand, created a new set of standards for numeric representations and operations. Whereas the C standards are designed to allow a wide range of implementations, the Java standard is quite specific on the formats and encodings of data. We highlight the representations and operations supported by Java at several places in the chapter.

Aside How to read this chapter

In this chapter, we examine the fundamental properties of how numbers and other forms of data are represented on a computer and the properties of the operations that computers perform on these data. This requires us to delve into the language of mathematics, writing formulas and equations and showing derivations of important properties.

To help you navigate this exposition, we have structured the presentation to first state a property as a *principle* in mathematical notation. We then illustrate this principle with examples and an informal discussion. We recommend that you go back and forth between the statement of the principle and the examples and discussion until you have a solid intuition for what is being said and what is important about the property. For more complex properties, we also provide a *derivation*, structured much like a mathematical proof. You should try to understand these derivations eventually, but you could skip over them on first reading.

We also encourage you to work on the practice problems as you proceed through the presentation. The practice problems engage you in *active learning*, helping you put thoughts into action. With these as background, you will find it much easier to go back and follow the derivations. Be assured, as well, that the mathematical skills required to understand this material are within reach of someone with a good grasp of high school algebra.

2.1 Information Storage

Rather than accessing individual bits in memory, most computers use blocks of 8 bits, or *bytes*, as the smallest addressable unit of memory. A machine-level program views memory as a very large array of bytes, referred to as *virtual memory*. Every byte of memory is identified by a unique number, known as its *address*, and the set of all possible addresses is known as the *virtual address space*. As indicated by its name, this virtual address space is just a conceptual image presented to the machine-level program. The actual implementation (presented in Chapter 9) uses a combination of dynamic random access memory (DRAM), flash memory, disk storage, special hardware, and operating system software to provide the program with what appears to be a monolithic byte array.

In subsequent chapters, we will cover how the compiler and run-time system partitions this memory space into more manageable units to store the different *program objects*, that is, program data, instructions, and control information. Various mechanisms are used to allocate and manage the storage for different parts of the program. This management is all performed within the virtual address space. For example, the value of a pointer in C—whether it points to an integer, a structure, or some other program object—is the virtual address of the first byte of some block of storage. The C compiler also associates *type* information with each pointer, so that it can generate different machine-level code to access the value stored at the location designated by the pointer depending on the type of that value. Although the C compiler maintains this type information, the actual machine-level program it generates has no information about data types. It simply treats each program object as a block of bytes and the program itself as a sequence of bytes.

Aside The evolution of the C programming language

As was described in an aside on page 4, the C programming language was first developed by Dennis Ritchie of Bell Laboratories for use with the Unix operating system (also developed at Bell Labs). At the time, most system programs, such as operating systems, had to be written largely in assembly code in order to have access to the low-level representations of different data types. For example, it was not feasible to write a memory allocator, such as is provided by the `malloc` library function, in other high-level languages of that era.

The original Bell Labs version of C was documented in the first edition of the book by Brian Kernighan and Dennis Ritchie [60]. Over time, C has evolved through the efforts of several standardization groups. The first major revision of the original Bell Labs C led to the ANSI C standard in 1989, by a group working under the auspices of the American National Standards Institute. ANSI C was a major departure from Bell Labs C, especially in the way functions are declared. ANSI C is described in the second edition of Kernighan and Ritchie's book [61], which is still considered one of the best references on C.

The International Standards Organization took over responsibility for standardizing the C language, adopting a version that was substantially the same as ANSI C in 1990 and hence is referred to as "ISO C90."

This same organization sponsored an updating of the language in 1999, yielding "ISO C99." Among other things, this version introduced some new data types and provided support for text strings requiring characters not found in the English language. A more recent standard was approved in 2011, and hence is named "ISO C11," again adding more data types and features. Most of these recent additions have been *backward compatible*, meaning that programs written according to the earlier standard (at least as far back as ISO C90) will have the same behavior when compiled according to the newer standards.

The GNU Compiler Collection (GCC) can compile programs according to the conventions of several different versions of the C language, based on different command-line options, as shown in Figure 2.1. For example, to compile program `prog.c` according to ISO C11, we could give the command line

```
linux> gcc -std=c11 prog.c
```

The options `-ansi` and `-std=c89` have identical effect—the code is compiled according to the ANSI or ISO C90 standard. (C90 is sometimes referred to as "C89," since its standardization effort began in 1989.) The option `-std=c99` causes the compiler to follow the ISO C99 convention.

As of the writing of this book, when no option is specified, the program will be compiled according to a version of C based on ISO C90, but including some features of C99, some of C11, some of C++, and others specific to GCC. The GNU project is developing a version that combines ISO C11, plus other features, that can be specified with the command-line option `-std=gnu11`. (Currently, this implementation is incomplete.) This will become the default version.

C version	GCC command-line option
GNU 89	*none*, `-std=gnu89`
ANSI, ISO C90	`-ansi`, `-std=c89`
ISO C99	`-std=c99`
ISO C11	`-std=c11`

Figure 2.1 Specifying different versions of C to GCC.

New to C? The role of pointers in C

Pointers are a central feature of C. They provide the mechanism for referencing elements of data structures, including arrays. Just like a variable, a pointer has two aspects: its *value* and its *type*. The value indicates the location of some object, while its type indicates what kind of object (e.g., integer or floating-point number) is stored at that location.

Truly understanding pointers requires examining their representation and implementation at the machine level. This will be a major focus in Chapter 3, culminating in an in-depth presentation in Section 3.10.1.

2.1.1 Hexadecimal Notation

A single byte consists of 8 bits. In binary notation, its value ranges from 00000000_2 to 11111111_2. When viewed as a decimal integer, its value ranges from 0_{10} to 255_{10}. Neither notation is very convenient for describing bit patterns. Binary notation is too verbose, while with decimal notation it is tedious to convert to and from bit patterns. Instead, we write bit patterns as base-16, or *hexadecimal* numbers. Hexadecimal (or simply "hex") uses digits '0' through '9' along with characters 'A' through 'F' to represent 16 possible values. Figure 2.2 shows the decimal and binary values associated with the 16 hexadecimal digits. Written in hexadecimal, the value of a single byte can range from 00_{16} to FF_{16}.

In C, numeric constants starting with `0x` or `0X` are interpreted as being in hexadecimal. The characters 'A' through 'F' may be written in either upper- or lowercase. For example, we could write the number $FA1D37B_{16}$ as `0xFA1D37B`, as `0xfa1d37b`, or even mixing upper- and lowercase (e.g., `0xFa1D37b`). We will use the C notation for representing hexadecimal values in this book.

A common task in working with machine-level programs is to manually convert between decimal, binary, and hexadecimal representations of bit patterns. Converting between binary and hexadecimal is straightforward, since it can be performed one hexadecimal digit at a time. Digits can be converted by referring to a chart such as that shown in Figure 2.2. One simple trick for doing the conversion in your head is to memorize the decimal equivalents of hex digits `A`, `C`, and `F`.

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111
Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Figure 2.2 Hexadecimal notation. Each hex digit encodes one of 16 values.

The hex values B, D, and E can be translated to decimal by computing their values relative to the first three.

For example, suppose you are given the number 0x173A4C. You can convert this to binary format by expanding each hexadecimal digit, as follows:

Hexadecimal	1	7	3	A	4	C
Binary	0001	0111	0011	1010	0100	1100

This gives the binary representation 000101110011101001001100.

Conversely, given a binary number 1111001010110110110011, you convert it to hexadecimal by first splitting it into groups of 4 bits each. Note, however, that if the total number of bits is not a multiple of 4, you should make the *leftmost* group be the one with fewer than 4 bits, effectively padding the number with leading zeros. Then you translate each group of bits into the corresponding hexadecimal digit:

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

Practice Problem 2.1 (solution page 143)

Perform the following number conversions:

A. 0x39A7F8 to binary

B. binary 1100100101111011 to hexadecimal

C. 0xD5E4C to binary

D. binary 1001101110011110110101 to hexadecimal

When a value x is a power of 2, that is, $x = 2^n$ for some nonnegative integer n, we can readily write x in hexadecimal form by remembering that the binary representation of x is simply 1 followed by n zeros. The hexadecimal digit 0 represents 4 binary zeros. So, for n written in the form $i + 4j$, where $0 \leq i \leq 3$, we can write x with a leading hex digit of 1 ($i = 0$), 2 ($i = 1$), 4 ($i = 2$), or 8 ($i = 3$), followed by j hexadecimal 0s. As an example, for $x = 2{,}048 = 2^{11}$, we have $n = 11 = 3 + 4 \cdot 2$, giving hexadecimal representation 0x800.

Practice Problem 2.2 (solution page 143)

Fill in the blank entries in the following table, giving the decimal and hexadecimal representations of different powers of 2:

n	2^n (decimal)	2^n (hexadecimal)
9	512	`0x200`
19	____	____
____	16,384	____
____	____	`0x10000`
17	____	____
____	32	____
____	____	`0x80`

Converting between decimal and hexadecimal representations requires using multiplication or division to handle the general case. To convert a decimal number x to hexadecimal, we can repeatedly divide x by 16, giving a quotient q and a remainder r, such that $x = q \cdot 16 + r$. We then use the hexadecimal digit representing r as the least significant digit and generate the remaining digits by repeating the process on q. As an example, consider the conversion of decimal 314,156:

$$
\begin{aligned}
314{,}156 &= 19{,}634 \cdot 16 + 12 && \text{(C)}\\
19{,}634 &= 1{,}227 \cdot 16 + 2 && \text{(2)}\\
1{,}227 &= 76 \cdot 16 + 11 && \text{(B)}\\
76 &= 4 \cdot 16 + 12 && \text{(C)}\\
4 &= 0 \cdot 16 + 4 && \text{(4)}
\end{aligned}
$$

From this we can read off the hexadecimal representation as `0x4CB2C`.

Conversely, to convert a hexadecimal number to decimal, we can multiply each of the hexadecimal digits by the appropriate power of 16. For example, given the number `0x7AF`, we compute its decimal equivalent as $7 \cdot 16^2 + 10 \cdot 16 + 15 = 7 \cdot 256 + 10 \cdot 16 + 15 = 1{,}792 + 160 + 15 = 1{,}967$.

Practice Problem 2.3 (solution page 144)

A single byte can be represented by 2 hexadecimal digits. Fill in the missing entries in the following table, giving the decimal, binary, and hexadecimal values of different byte patterns:

Decimal	Binary	Hexadecimal
0	0000 0000	`0x00`
167	____	____
62	____	____
188	____	____
____	0011 0111	____
____	1000 1000	____
____	1111 0011	____

Aside Converting between decimal and hexadecimal

For converting larger values between decimal and hexadecimal, it is best to let a computer or calculator do the work. There are numerous tools that can do this. One simple way is to use any of the standard search engines, with queries such as

Convert 0xabcd to decimal

or

123 in hex

Decimal	Binary	Hexadecimal
______	______	0x52
______	______	0xAC
______	______	0xE7

Practice Problem 2.4 (solution page 144)

Without converting the numbers to decimal or binary, try to solve the following arithmetic problems, giving the answers in hexadecimal. *Hint:* Just modify the methods you use for performing decimal addition and subtraction to use base 16.

A. 0x503c + 0x8 = ________

B. 0x503c − 0x40 = ________

C. 0x503c + 64 = ________

D. 0x50ea − 0x503c = ________

2.1.2 Data Sizes

Every computer has a *word size*, indicating the nominal size of pointer data. Since a virtual address is encoded by such a word, the most important system parameter determined by the word size is the maximum size of the virtual address space. That is, for a machine with a w-bit word size, the virtual addresses can range from 0 to $2^w - 1$, giving the program access to at most 2^w bytes.

In recent years, there has been a widespread shift from machines with 32-bit word sizes to those with word sizes of 64 bits. This occurred first for high-end machines designed for large-scale scientific and database applications, followed by desktop and laptop machines, and most recently for the processors found in smartphones. A 32-bit word size limits the virtual address space to 4 gigabytes (written 4 GB), that is, just over 4×10^9 bytes. Scaling up to a 64-bit word size leads to a virtual address space of 16 *exabytes*, or around 1.84×10^{19} bytes.

Most 64-bit machines can also run programs compiled for use on 32-bit machines, a form of backward compatibility. So, for example, when a program `prog.c` is compiled with the directive

```
linux> gcc -m32 prog.c
```

then this program will run correctly on either a 32-bit or a 64-bit machine. On the other hand, a program compiled with the directive

```
linux> gcc -m64 prog.c
```

will only run on a 64-bit machine. We will therefore refer to programs as being either "32-bit programs" or "64-bit programs," since the distinction lies in how a program is compiled, rather than the type of machine on which it runs.

Computers and compilers support multiple data formats using different ways to encode data, such as integers and floating point, as well as different lengths. For example, many machines have instructions for manipulating single bytes, as well as integers represented as 2-, 4-, and 8-byte quantities. They also support floating-point numbers represented as 4- and 8-byte quantities.

The C language supports multiple data formats for both integer and floating-point data. Figure 2.3 shows the number of bytes typically allocated for different C data types. (We discuss the relation between what is guaranteed by the C standard versus what is typical in Section 2.2.) The exact numbers of bytes for some data types depends on how the program is compiled. We show sizes for typical 32-bit and 64-bit programs. Integer data can be either *signed*, able to represent negative, zero, and positive values, or *unsigned*, only allowing nonnegative values. Data type `char` represents a single byte. Although the name `char` derives from the fact that it is used to store a single character in a text string, it can also be used to store integer values. Data types `short`, `int`, and `long` are intended to provide a range of

C declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
`[signed] char`	`unsigned char`	1	1
`short`	`unsigned short`	2	2
`int`	`unsigned`	4	4
`long`	`unsigned long`	4	8
`int32_t`	`uint32_t`	4	4
`int64_t`	`uint64_t`	8	8
`char *`		4	8
`float`		4	4
`double`		8	8

Figure 2.3 Typical sizes (in bytes) of basic C data types. The number of bytes allocated varies with how the program is compiled. This chart shows the values typical of 32-bit and 64-bit programs.

New to C? Declaring pointers

For any data type T, the declaration

```
T *p;
```

indicates that p is a pointer variable, pointing to an object of type T. For example,

```
char *p;
```

is the declaration of a pointer to an object of type char.

sizes. Even when compiled for 64-bit systems, data type int is usually just 4 bytes. Data type long commonly has 4 bytes in 32-bit programs and 8 bytes in 64-bit programs.

To avoid the vagaries of relying on "typical" sizes and different compiler settings, ISO C99 introduced a class of data types where the data sizes are fixed regardless of compiler and machine settings. Among these are data types int32_t and int64_t, having exactly 4 and 8 bytes, respectively. Using fixed-size integer types is the best way for programmers to have close control over data representations.

Most of the data types encode signed values, unless prefixed by the keyword unsigned or using the specific unsigned declaration for fixed-size data types. The exception to this is data type char. Although most compilers and machines treat these as signed data, the C standard does not guarantee this. Instead, as indicated by the square brackets, the programmer should use the declaration signed char to guarantee a 1-byte signed value. In many contexts, however, the program's behavior is insensitive to whether data type char is signed or unsigned.

The C language allows a variety of ways to order the keywords and to include or omit optional keywords. As examples, all of the following declarations have identical meaning:

```
unsigned long
unsigned long int
long unsigned
long unsigned int
```

We will consistently use the forms found in Figure 2.3.

Figure 2.3 also shows that a pointer (e.g., a variable declared as being of type char *) uses the full word size of the program. Most machines also support two different floating-point formats: single precision, declared in C as float, and double precision, declared in C as double. These formats use 4 and 8 bytes, respectively.

Programmers should strive to make their programs portable across different machines and compilers. One aspect of portability is to make the program insensitive to the exact sizes of the different data types. The C standards set lower bounds on the numeric ranges of the different data types, as will be covered later, but there are no upper bounds (except with the fixed-size types). With 32-bit machines and 32-bit programs being the dominant combination from around 1980 until around 2010, many programs have been written assuming the allocations listed for 32-bit programs in Figure 2.3. With the transition to 64-bit machines, many hidden word size dependencies have arisen as bugs in migrating these programs to new machines. For example, many programmers historically assumed that an object declared as type `int` could be used to store a pointer. This works fine for most 32-bit programs, but it leads to problems for 64-bit programs.

2.1.3 Addressing and Byte Ordering

For program objects that span multiple bytes, we must establish two conventions: what the address of the object will be, and how we will order the bytes in memory. In virtually all machines, a multi-byte object is stored as a contiguous sequence of bytes, with the address of the object given by the smallest address of the bytes used. For example, suppose a variable `x` of type `int` has address `0x100`; that is, the value of the address expression `&x` is `0x100`. Then (assuming data type `int` has a 32-bit representation) the 4 bytes of `x` would be stored in memory locations `0x100`, `0x101`, `0x102`, and `0x103`.

For ordering the bytes representing an object, there are two common conventions. Consider a w-bit integer having a bit representation $[x_{w-1}, x_{w-2}, \ldots, x_1, x_0]$, where x_{w-1} is the most significant bit and x_0 is the least. Assuming w is a multiple of 8, these bits can be grouped as bytes, with the most significant byte having bits $[x_{w-1}, x_{w-2}, \ldots, x_{w-8}]$, the least significant byte having bits $[x_7, x_6, \ldots, x_0]$, and the other bytes having bits from the middle. Some machines choose to store the object in memory ordered from least significant byte to most, while other machines store them from most to least. The former convention—where the least significant byte comes first—is referred to as *little endian*. The latter convention—where the most significant byte comes first—is referred to as *big endian*.

Suppose the variable `x` of type `int` and at address `0x100` has a hexadecimal value of `0x01234567`. The ordering of the bytes within the address range `0x100` through `0x103` depends on the type of machine:

Big endian

	0x100	0x101	0x102	0x103	
...	01	23	45	67	...

Little endian

	0x100	0x101	0x102	0x103	
...	67	45	23	01	...

Note that in the word `0x01234567` the high-order byte has hexadecimal value `0x01`, while the low-order byte has value `0x67`.

Most Intel-compatible machines operate exclusively in little-endian mode. On the other hand, most machines from IBM and Oracle (arising from their acquisi-

Aside Origin of "endian"

Here is how Jonathan Swift, writing in 1726, described the history of the controversy between big and little endians:

> . . . Lilliput and Blefuscu . . . have, as I was going to tell you, been engaged in a most obstinate war for six-and-thirty moons past. It began upon the following occasion. It is allowed on all hands, that the primitive way of breaking eggs, before we eat them, was upon the larger end; but his present majesty's grandfather, while he was a boy, going to eat an egg, and breaking it according to the ancient practice, happened to cut one of his fingers. Whereupon the emperor his father published an edict, commanding all his subjects, upon great penalties, to break the smaller end of their eggs. The people so highly resented this law, that our histories tell us, there have been six rebellions raised on that account; wherein one emperor lost his life, and another his crown. These civil commotions were constantly fomented by the monarchs of Blefuscu; and when they were quelled, the exiles always fled for refuge to that empire. It is computed that eleven thousand persons have at several times suffered death, rather than submit to break their eggs at the smaller end. Many hundred large volumes have been published upon this controversy: but the books of the Big-endians have been long forbidden, and the whole party rendered incapable by law of holding employments. (Jonathan Swift. Gulliver's Travels, Benjamin Motte, 1726.)

In his day, Swift was satirizing the continued conflicts between England (Lilliput) and France (Blefuscu). Danny Cohen, an early pioneer in networking protocols, first applied these terms to refer to byte ordering [24], and the terminology has been widely adopted.

tion of Sun Microsystems in 2010) operate in big-endian mode. Note that we said "most." The conventions do not split precisely along corporate boundaries. For example, both IBM and Oracle manufacture machines that use Intel-compatible processors and hence are little endian. Many recent microprocessor chips are *bi-endian*, meaning that they can be configured to operate as either little- or big-endian machines. In practice, however, byte ordering becomes fixed once a particular operating system is chosen. For example, ARM microprocessors, used in many cell phones, have hardware that can operate in either little- or big-endian mode, but the two most common operating systems for these chips—Android (from Google) and IOS (from Apple)—operate only in little-endian mode.

People get surprisingly emotional about which byte ordering is the proper one. In fact, the terms "little endian" and "big endian" come from the book *Gulliver's Travels* by Jonathan Swift, where two warring factions could not agree as to how a soft-boiled egg should be opened—by the little end or by the big. Just like the egg issue, there is no technological reason to choose one byte ordering convention over the other, and hence the arguments degenerate into bickering about sociopolitical issues. As long as one of the conventions is selected and adhered to consistently, the choice is arbitrary.

For most application programmers, the byte orderings used by their machines are totally invisible; programs compiled for either class of machine give identical results. At times, however, byte ordering becomes an issue. The first is when binary data are communicated over a network between different machines. A common problem is for data produced by a little-endian machine to be sent to a big-endian machine, or vice versa, leading to the bytes within the words being in reverse order for the receiving program. To avoid such problems, code written for networking applications must follow established conventions for byte ordering to make sure the sending machine converts its internal representation to the network standard, while the receiving machine converts the network standard to its internal representation. We will see examples of these conversions in Chapter 11.

A second case where byte ordering becomes important is when looking at the byte sequences representing integer data. This occurs often when inspecting machine-level programs. As an example, the following line occurs in a file that gives a text representation of the machine-level code for an Intel x86-64 processor:

```
4004d3:  01 05 43 0b 20 00        add    %eax,0x200b43(%rip)
```

This line was generated by a *disassembler*, a tool that determines the instruction sequence represented by an executable program file. We will learn more about disassemblers and how to interpret lines such as this in Chapter 3. For now, we simply note that this line states that the hexadecimal byte sequence `01 05 43 0b 20 00` is the byte-level representation of an instruction that adds a word of data to the value stored at an address computed by adding `0x200b43` to the current value of the *program counter*, the address of the next instruction to be executed. If we take the final 4 bytes of the sequence `43 0b 20 00` and write them in reverse order, we have `00 20 0b 43`. Dropping the leading 0, we have the value `0x200b43`, the numeric value written on the right. Having bytes appear in reverse order is a common occurrence when reading machine-level program representations generated for little-endian machines such as this one. The natural way to write a byte sequence is to have the lowest-numbered byte on the left and the highest on the right, but this is contrary to the normal way of writing numbers with the most significant digit on the left and the least on the right.

A third case where byte ordering becomes visible is when programs are written that circumvent the normal type system. In the C language, this can be done using a *cast* or a *union* to allow an object to be referenced according to a different data type from which it was created. Such coding tricks are strongly discouraged for most application programming, but they can be quite useful and even necessary for system-level programming.

Figure 2.4 shows C code that uses casting to access and print the byte representations of different program objects. We use `typedef` to define data type `byte_pointer` as a pointer to an object of type `unsigned char`. Such a byte pointer references a sequence of bytes where each byte is considered to be a nonnegative integer. The first routine `show_bytes` is given the address of a sequence of bytes, indicated by a byte pointer, and a byte count. The byte count is specified as having data type `size_t`, the preferred data type for expressing the sizes of data structures. It prints the individual bytes in hexadecimal. The C formatting directive `%.2x` indicates that an integer should be printed in hexadecimal with at least 2 digits.

```
#include <stdio.h>

typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, size_t len) {
    int i;
    for (i = 0; i < len; i++)
        printf(" %.2x", start[i]);
    printf("\n");
}

void show_int(int x) {
    show_bytes((byte_pointer) &x, sizeof(int));
}

void show_float(float x) {
    show_bytes((byte_pointer) &x, sizeof(float));
}

void show_pointer(void *x) {
    show_bytes((byte_pointer) &x, sizeof(void *));
}
```

Figure 2.4 Code to print the byte representation of program objects. This code uses casting to circumvent the type system. Similar functions are easily defined for other data types.

Procedures `show_int`, `show_float`, and `show_pointer` demonstrate how to use procedure `show_bytes` to print the byte representations of C program objects of type `int`, `float`, and `void *`, respectively. Observe that they simply pass `show_bytes` a pointer `&x` to their argument `x`, casting the pointer to be of type `unsigned char *`. This cast indicates to the compiler that the program should consider the pointer to be to a sequence of bytes rather than to an object of the original data type. This pointer will then be to the lowest byte address occupied by the object.

These procedures use the C `sizeof` operator to determine the number of bytes used by the object. In general, the expression `sizeof(`T`)` returns the number of bytes required to store an object of type T. Using `sizeof` rather than a fixed value is one step toward writing code that is portable across different machine types.

We ran the code shown in Figure 2.5 on several different machines, giving the results shown in Figure 2.6. The following machines were used:

Linux 32	Intel IA32 processor running Linux.
Windows	Intel IA32 processor running Windows.
Sun	Sun Microsystems SPARC processor running Solaris. (These machines are now produced by Oracle.)
Linux 64	Intel x86-64 processor running Linux.

code/data/show-bytes.c

```
void test_show_bytes(int val) {
    int ival = val;
    float fval = (float) ival;
    int *pval = &ival;
    show_int(ival);
    show_float(fval);
    show_pointer(pval);
}
```

code/data/show-bytes.c

Figure 2.5 Byte representation examples. This code prints the byte representations of sample data objects.

Machine	Value	Type	Bytes (hex)
Linux 32	12,345	`int`	39 30 00 00
Windows	12,345	`int`	39 30 00 00
Sun	12,345	`int`	00 00 30 39
Linux 64	12,345	`int`	39 30 00 00
Linux 32	12,345.0	`float`	00 e4 40 46
Windows	12,345.0	`float`	00 e4 40 46
Sun	12,345.0	`float`	46 40 e4 00
Linux 64	12,345.0	`float`	00 e4 40 46
Linux 32	`&ival`	`int *`	e4 f9 ff bf
Windows	`&ival`	`int *`	b4 cc 22 00
Sun	`&ival`	`int *`	ef ff fa 0c
Linux 64	`&ival`	`int *`	b8 11 e5 ff ff 7f 00 00

Figure 2.6 Byte representations of different data values. Results for `int` and `float` are identical, except for byte ordering. Pointer values are machine dependent.

Our argument 12,345 has hexadecimal representation 0x00003039. For the `int` data, we get identical results for all machines, except for the byte ordering. In particular, we can see that the least significant byte value of 0x39 is printed first for Linux 32, Windows, and Linux 64, indicating little-endian machines, and last for Sun, indicating a big-endian machine. Similarly, the bytes of the `float` data are identical, except for the byte ordering. On the other hand, the pointer values are completely different. The different machine/operating system configurations use different conventions for storage allocation. One feature to note is that the Linux 32, Windows, and Sun machines use 4-byte addresses, while the Linux 64 machine uses 8-byte addresses.

New to C? Naming data types with `typedef`

The `typedef` declaration in C provides a way of giving a name to a data type. This can be a great help in improving code readability, since deeply nested type declarations can be difficult to decipher.

The syntax for `typedef` is exactly like that of declaring a variable, except that it uses a type name rather than a variable name. Thus, the declaration of `byte_pointer` in Figure 2.4 has the same form as the declaration of a variable of type `unsigned char *`.

For example, the declaration

```
typedef int *int_pointer;
int_pointer ip;
```

defines type `int_pointer` to be a pointer to an `int`, and declares a variable `ip` of this type. Alternatively, we could declare this variable directly as

```
int *ip;
```

New to C? Formatted printing with `printf`

The `printf` function (along with its cousins `fprintf` and `sprintf`) provides a way to print information with considerable control over the formatting details. The first argument is a *format string*, while any remaining arguments are values to be printed. Within the format string, each character sequence starting with '%' indicates how to format the next argument. Typical examples include `%d` to print a decimal integer, `%f` to print a floating-point number, and `%c` to print a character having the character code given by the argument.

Specifying the formatting of fixed-size data types, such as `int_32t`, is a bit more involved, as is described in the aside on page 67.

Observe that although the floating-point and the integer data both encode the numeric value 12,345, they have very different byte patterns: 0x00003039 for the integer and 0x4640E400 for floating point. In general, these two formats use different encoding schemes. If we expand these hexadecimal patterns into binary form and shift them appropriately, we find a sequence of 13 matching bits, indicated by a sequence of asterisks, as follows:

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

This is not coincidental. We will return to this example when we study floating-point formats.

New to C? Pointers and arrays

In function `show_bytes` (Figure 2.4), we see the close connection between pointers and arrays, as will be discussed in detail in Section 3.8. We see that this function has an argument `start` of type `byte_pointer` (which has been defined to be a pointer to `unsigned char`), but we see the array reference `start[i]` on line 8. In C, we can dereference a pointer with array notation, and we can reference array elements with pointer notation. In this example, the reference `start[i]` indicates that we want to read the byte that is `i` positions beyond the location pointed to by `start`.

New to C? Pointer creation and dereferencing

In lines 13, 17, and 21 of Figure 2.4 we see uses of two operations that give C (and therefore C++) its distinctive character. The C "address of" operator '&' creates a pointer. On all three lines, the expression `&x` creates a pointer to the location holding the object indicated by variable `x`. The type of this pointer depends on the type of `x`, and hence these three pointers are of type `int *`, `float *`, and `void **`, respectively. (Data type `void *` is a special kind of pointer with no associated type information.)

The cast operator converts from one data type to another. Thus, the cast `(byte_pointer) &x` indicates that whatever type the pointer `&x` had before, the program will now reference a pointer to data of type `unsigned char`. The casts shown here do not change the actual pointer; they simply direct the compiler to refer to the data being pointed to according to the new data type.

Aside Generating an ASCII table

You can display a table showing the ASCII character code by executing the command `man ascii`.

Practice Problem 2.5 (solution page 144)

Consider the following three calls to `show_bytes`:

```
int val = 0x87654321;
byte_pointer valp = (byte_pointer) &val;
show_bytes(valp, 1); /* A. */
show_bytes(valp, 2); /* B. */
show_bytes(valp, 3); /* C. */
```

Indicate the values that will be printed by each call on a little-endian machine and on a big-endian machine:

A. Little endian: ________ Big endian: ________

B. Little endian: ________ Big endian: ________

C. Little endian: ________ Big endian: ________

Practice Problem 2.6 (solution page 145)

Using `show_int` and `show_float`, we determine that the integer 3510593 has hexadecimal representation `0x00359141`, while the floating-point number 3510593.0 has hexadecimal representation `0x4A564504`.

A. Write the binary representations of these two hexadecimal values.

B. Shift these two strings relative to one another to maximize the number of matching bits. How many bits match?

C. What parts of the strings do not match?

2.1.4 Representing Strings

A string in C is encoded by an array of characters terminated by the null (having value 0) character. Each character is represented by some standard encoding, with the most common being the ASCII character code. Thus, if we run our routine `show_bytes` with arguments `"12345"` and 6 (to include the terminating character), we get the result `31 32 33 34 35 00`. Observe that the ASCII code for decimal digit x happens to be `0x3`x, and that the terminating byte has the hex representation `0x00`. This same result would be obtained on any system using ASCII as its character code, independent of the byte ordering and word size conventions. As a consequence, text data are more platform independent than binary data.

Practice Problem 2.7 (solution page 145)

What would be printed as a result of the following call to `show_bytes`?

```
const char *s = "abcdef";
show_bytes((byte_pointer) s, strlen(s));
```

Note that letters 'a' through 'z' have ASCII codes `0x61` through `0x7A`.

2.1.5 Representing Code

Consider the following C function:

```
int sum(int x, int y) {
    return x + y;
}
```

When compiled on our sample machines, we generate machine code having the following byte representations:

Linux 32	`55 89 e5 8b 45 0c 03 45 08 c9 c3`
Windows	`55 89 e5 8b 45 0c 03 45 08 5d c3`
Sun	`81 c3 e0 08 90 02 00 09`
Linux 64	`55 48 89 e5 89 7d fc 89 75 f8 03 45 fc c9 c3`

Aside The Unicode standard for text encoding

The ASCII character set is suitable for encoding English-language documents, but it does not have much in the way of special characters, such as the French 'ç'. It is wholly unsuited for encoding documents in languages such as Greek, Russian, and Chinese. Over the years, a variety of methods have been developed to encode text for different languages. The Unicode Consortium has devised the most comprehensive and widely accepted standard for encoding text. The current Unicode standard (version 7.0) has a repertoire of over 100,000 characters supporting a wide range of languages, including the ancient languages of Egypt and Babylon. To their credit, the Unicode Technical Committee rejected a proposal to include a standard writing for Klingon, a fictional civilization from the television series *Star Trek.*

The base encoding, known as the "Universal Character Set" of Unicode, uses a 32-bit representation of characters. This would seem to require every string of text to consist of 4 bytes per character. However, alternative codings are possible where common characters require just 1 or 2 bytes, while less common ones require more. In particular, the UTF-8 representation encodes each character as a sequence of bytes, such that the standard ASCII characters use the same single-byte encodings as they have in ASCII, implying that all ASCII byte sequences have the same meaning in UTF-8 as they do in ASCII.

The Java programming language uses Unicode in its representations of strings. Program libraries are also available for C to support Unicode.

Here we find that the instruction codings are different. Different machine types use different and incompatible instructions and encodings. Even identical processors running different operating systems have differences in their coding conventions and hence are not binary compatible. Binary code is seldom portable across different combinations of machine and operating system.

A fundamental concept of computer systems is that a program, from the perspective of the machine, is simply a sequence of bytes. The machine has no information about the original source program, except perhaps some auxiliary tables maintained to aid in debugging. We will see this more clearly when we study machine-level programming in Chapter 3.

2.1.6 Introduction to Boolean Algebra

Since binary values are at the core of how computers encode, store, and manipulate information, a rich body of mathematical knowledge has evolved around the study of the values 0 and 1. This started with the work of George Boole (1815–1864) around 1850 and thus is known as *Boolean algebra*. Boole observed that by encoding logic values TRUE and FALSE as binary values 1 and 0, he could formulate an algebra that captures the basic principles of logical reasoning.

The simplest Boolean algebra is defined over the two-element set {0, 1}. Figure 2.7 defines several operations in this algebra. Our symbols for representing these operations are chosen to match those used by the C bit-level operations,

~	
0	1
1	0

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

Figure 2.7 Operations of Boolean algebra. Binary values 1 and 0 encode logic values TRUE and FALSE, while operations ~, &, |, and ^ encode logical operations NOT, AND, OR, and EXCLUSIVE-OR, respectively.

as will be discussed later. The Boolean operation ~ corresponds to the logical operation NOT, denoted by the symbol $\neg$. That is, we say that $\neg P$ is true when P is not true, and vice versa. Correspondingly, ~p equals 1 when p equals 0, and vice versa. Boolean operation & corresponds to the logical operation AND, denoted by the symbol $\wedge$. We say that $P \wedge Q$ holds when both P is true and Q is true. Correspondingly, p & q equals 1 only when $p = 1$ and $q = 1$. Boolean operation | corresponds to the logical operation OR, denoted by the symbol $\vee$. We say that $P \vee Q$ holds when either P is true or Q is true. Correspondingly, p | q equals 1 when either $p = 1$ or $q = 1$. Boolean operation ^ corresponds to the logical operation EXCLUSIVE-OR, denoted by the symbol $\oplus$. We say that $P \oplus Q$ holds when either P is true or Q is true, but not both. Correspondingly, p ^ q equals 1 when either $p = 1$ and $q = 0$, or $p = 0$ and $q = 1$.

Claude Shannon (1916–2001), who later founded the field of information theory, first made the connection between Boolean algebra and digital logic. In his 1937 master's thesis, he showed that Boolean algebra could be applied to the design and analysis of networks of electromechanical relays. Although computer technology has advanced considerably since, Boolean algebra still plays a central role in the design and analysis of digital systems.

We can extend the four Boolean operations to also operate on *bit vectors*, strings of zeros and ones of some fixed length w. We define the operations over bit vectors according to their applications to the matching elements of the arguments. Let a and b denote the bit vectors $[a_{w-1}, a_{w-2}, \ldots, a_0]$ and $[b_{w-1}, b_{w-2}, \ldots, b_0]$, respectively. We define a & b to also be a bit vector of length w, where the ith element equals a_i & b_i, for $0 \leq i < w$. The operations |, ^, and ~ are extended to bit vectors in a similar fashion.

As examples, consider the case where $w = 4$, and with arguments $a = [0110]$ and $b = [1100]$. Then the four operations a & b, a | b, a ^ b, and ~b yield

```
  0110      0110      0110
& 1100    | 1100    ^ 1100    ~ 1100
  ----      ----      ----      ----
  0100      1110      1010      0011
```

Practice Problem 2.8 (solution page 145)

Fill in the following table showing the results of evaluating Boolean operations on bit vectors.

Web Aside DATA:BOOL More on Boolean algebra and Boolean rings

The Boolean operations |, &, and ~ operating on bit vectors of length w form a *Boolean algebra*, for any integer $w > 0$. The simplest is the case where $w = 1$ and there are just two elements, but for the more general case there are 2^w bit vectors of length w. Boolean algebra has many of the same properties as arithmetic over integers. For example, just as multiplication distributes over addition, written $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$, Boolean operation & distributes over |, written a & $(b \mid c) = (a$ & $b) \mid (a$ & $c)$. In addition, however. Boolean operation | distributes over &, and so we can write $a \mid (b$ & $c) = (a \mid b)$ & $(a \mid c)$, whereas we cannot say that $a + (b \cdot c) = (a + b) \cdot (a + c)$ holds for all integers.

When we consider operations ^, &, and ~ operating on bit vectors of length w, we get a different mathematical form, known as a *Boolean ring*. Boolean rings have many properties in common with integer arithmetic. For example, one property of integer arithmetic is that every value x has an *additive inverse* $-x$, such that $x + -x = 0$. A similar property holds for Boolean rings, where ^ is the "addition" operation, but in this case each element is its own additive inverse. That is, a ^ $a = 0$ for any value a, where we use 0 here to represent a bit vector of all zeros. We can see this holds for single bits, since 0 ^ $0 = 1$ ^ $1 = 0$, and it extends to bit vectors as well. This property holds even when we rearrange terms and combine them in a different order, and so $(a$ ^ $b)$ ^ $a = b$. This property leads to some interesting results and clever tricks, as we will explore in Problem 2.10.

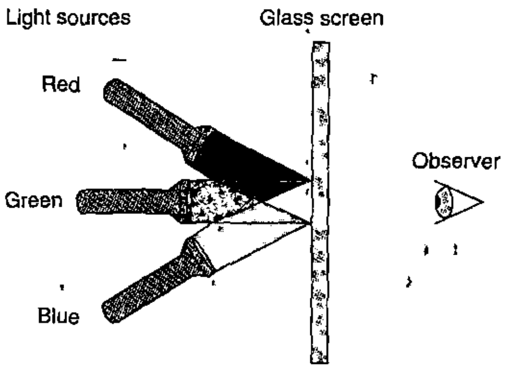
Operation	Result
a	[01101001]
b	[01010101]
~a	________
~b	________
a & b	________
$a \mid b$	________
a ^ b	________

One useful application of bit vectors is to represent finite sets. We can encode any subset $A \subseteq \{0, 1, \ldots, w - 1\}$ with a bit vector $[a_{w-1}, \ldots, a_1, a_0]$, where $a_i = 1$ if and only if $i \in A$. For example, recalling that we write a_{w-1} on the left and a_0 on the right, bit vector $a = [01101001]$ encodes the set $A = \{0, 3, 5, 6\}$, while bit vector $b = [01010101]$ encodes the set $B = \{0, 2, 4, 6\}$. With this way of encoding sets, Boolean operations | and & correspond to set union and intersection, respectively, and ~ corresponds to set complement. Continuing our earlier example, the operation a & b yields bit vector [01000001], while $A \cap B = \{0, 6\}$.

We will see the encoding of sets by bit vectors in a number of practical applications. For example, in Chapter 8, we will see that there are a number of different *signals* that can interrupt the execution of a program. We can selectively enable or disable different signals by specifying a bit-vector mask, where a 1 in bit position i indicates that signal i is enabled and a 0 indicates that it is disabled. Thus, the mask represents the set of enabled signals.

Practice Problem 2.9 (solution page 146)

Computers generate color pictures on a video screen or liquid crystal display by mixing three different colors of light: red, green, and blue. Imagine a simple scheme, with three different lights, each of which can be turned on or off, projecting onto a glass screen:



We can then create eight different colors based on the absence (0) or presence (1) of light sources R, G, and B:

R	G	B	Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

Each of these colors can be represented as a bit vector of length 3, and we can apply Boolean operations to them.

A. The complement of a color is formed by turning off the lights that are on and turning on the lights that are off. What would be the complement of each of the eight colors listed above?

B. Describe the effect of applying Boolean operations on the following colors:

Blue | Green = ______
Yellow & Cyan = ______
Red ^ Magenta = ______

2.1.7 Bit-Level Operations in C

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

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

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

Practice Problem 2.10 (solution page 146)

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

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

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

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

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

Practice Problem 2.11 (solution page 146)

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

You arrive at the following function:

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

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

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

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

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

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

Practice Problem 2.12 (solution page 146)

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

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

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

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

Practice Problem 2.13 (solution page 147)

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

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

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

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

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

2.1.8 Logical Operations in C

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

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

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

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

Practice Problem 2.14 (solution page 147)

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

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

Practice Problem 2.15 (solution page 148)

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

2.1.9 Shift Operations in C

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

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

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

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

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

Operation	Value 1	Value 2
Argument x	[01100011]	[10010101]
x << 4	[0011*0000*]	[0101*0000*]
x >> 4 (logical)	[*0000*0110]	[*0000*1001]
x >> 4 (arithmetic)	[*0000*0110]	[*1111*1001]

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

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

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

Practice Problem 2.16 (solution page 148)

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

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

Aside Shifting by k, for large values of k

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

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

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

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

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

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

Aside Operator precedence issues with shift operations

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

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

2.2 Integer Representations

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

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

Symbol	Type	Meaning	Page
$B2T_w$	Function	Binary to two's complement	64
$B2U_w$	Function	Binary to unsigned	62
$U2B_w$	Function	Unsigned to binary	64
$U2T_w$	Function	Unsigned to two's complement	71
$T2B_w$	Function	Two's complement to binary	65
$T2U_w$	Function	Two's complement to unsigned	71
$TMin_w$	Constant	Minimum two's-complement value	65
$TMax_w$	Constant	Maximum two's-complement value	65
$UMax_w$	Constant	Maximum unsigned value	63
$+^{t}_{w}$	Operation	Two's-complement addition	90
$+^{u}_{w}$	Operation	Unsigned addition	85
$*^{t}_{w}$	Operation	Two's-complement multiplication	97
$*^{u}_{w}$	Operation	Unsigned multiplication	96
$-^{t}_{w}$	Operation	Two's-complement negation	95
$-^{u}_{w}$	Operation	Unsigned negation	89

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

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

2.2.1 Integral Data Types

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

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

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

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

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

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

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

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

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

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

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

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

2.2.2 Unsigned Encodings

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

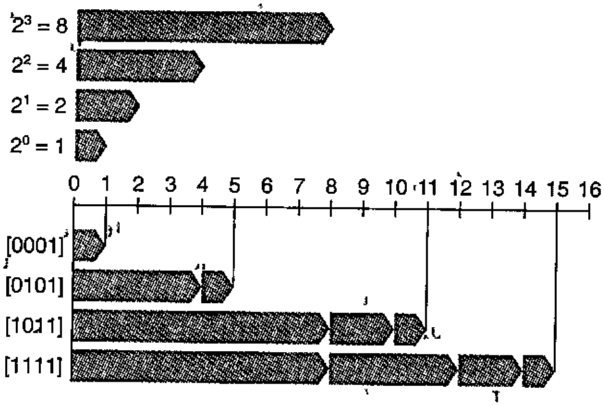


Figure 2.12 Unsigned number examples for $w = 4$. When bit i in the binary representation has value 1, it contributes 2^i to the value.

PRINCIPLE: Definition of unsigned encoding

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

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

■

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

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

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:

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

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

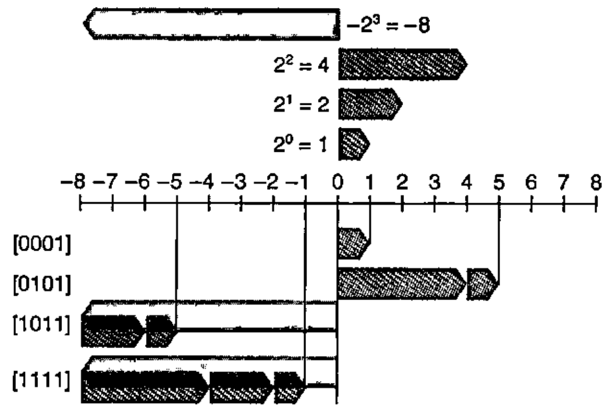


Figure 2.13
Two's-complement number examples for $w = 4$. Bit 3 serves as a sign bit; when set to 1, it contributes $-2^3 = -8$ to the value. This weighting is shown as a leftward-pointing gray bar.

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

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

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

PRINCIPLE: Uniqueness of two's-complement encoding

Function $B2T_w$ is a bijection. ■

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

Practice Problem 2.17 (solution page 148)

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

$\vec{x}$			
Hexadecimal	Binary	$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
0xE	[1110]	$2^3+2^2+2^1=14$	$-2^3+2^2+2^1=-2$
0x0	______	______	______
0x5	______	______	______
0x8	______	______	______
0xD	______	______	______
0xF	______	______	______

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

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

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

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

Aside More on fixed-size integer types

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

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

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

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

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

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

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

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

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

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

Aside Alternative representations of signed numbers

There are two other standard representations for signed numbers:

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

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

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

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

Both of these representations have the curious property that there are two different encodings of the number 0. For both representations, $[00 \cdots 0]$ 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 149)

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

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

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

2.2.4 Conversions between Signed and Unsigned

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

For example, consider the following code:

```
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 149)

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

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

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

PRINCIPLE: Conversion from two's complement to unsigned

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

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

■

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

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

DERIVATION: Conversion from two's complement to unsigned

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

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

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

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

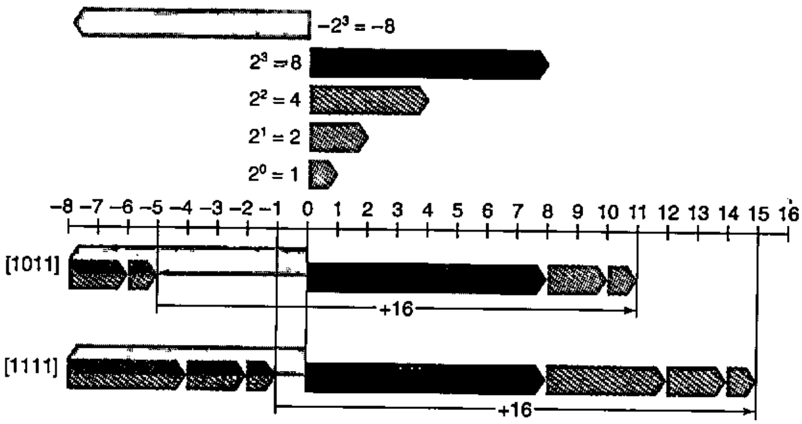


Figure 2.16
Comparing unsigned and two's-complement representations for $w = 4$. The weight of the most significant bit is −8 for two's complement and +8 for unsigned, yielding a net difference of 16.

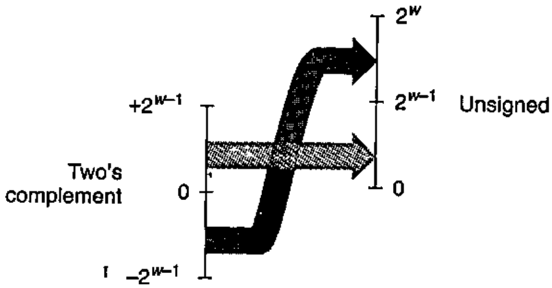


Figure 2.17
Conversion from two's complement to unsigned. Function *T2U* converts negative numbers to large positive numbers.

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

Practice Problem 2.20 (solution page 149)

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

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

PRINCIPLE: Unsigned to two's-complement conversion

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

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

■

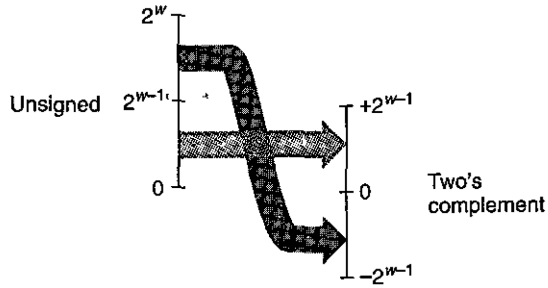


Figure 2.18 Conversion from unsigned to two's complement. Function *U2T* converts numbers greater than $2^{w-1}-1$ to negative values.

This principle can be justified as follows:

DERIVATION: Unsigned to two's-complement conversion

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

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

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

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

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

2.2.5 Signed versus Unsigned in C

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

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

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

```
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 149)

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

Expression	Type	Evaluation
`-2147483647-1 == 2147483648U`	______	______
`-2147483647-1 < 2147483647`	______	______
`-2147483647-1U < 2147483647`	______	______
`-2147483647-1 < -2147483647`	______	______
`-2147483647-1U < -2147483647`	______	______

2.2.6 Expanding the Bit Representation of a Number

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

Web Aside DATA:TMIN Writing *TMin* in C

In Figure 2.19 and in Problem 2.21, 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.

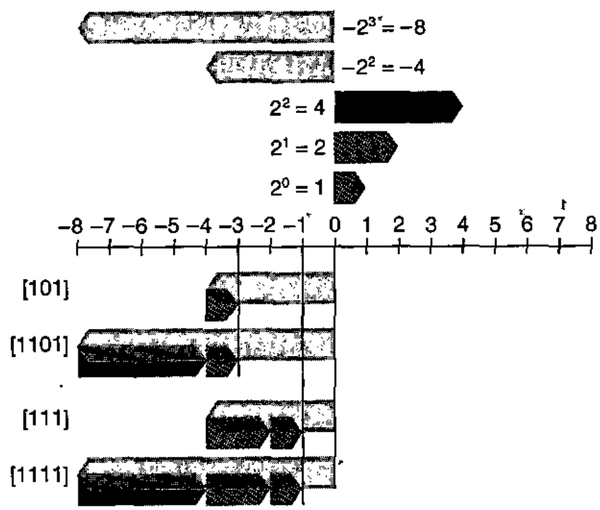


Figure 2.20 Examples of sign extension from $w = 3$ to $w = 4$. For $w = 4$, the combined weight of the upper 2 bits is $-8 + 4 = -4$, matching that of the sign bit for $w = 3$.

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 150)

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

A. [1011]

B. [11011]

C. [111011]

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

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

```
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 150)

Consider the following C functions:

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

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

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

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

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

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

2.2.7 Truncating Numbers

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

```
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

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

while the effect for two's-complement numbers is

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

Practice Problem 2.24 (solution page 150)

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

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

Explain how Equations 2.9 and 2.10 apply to these cases.

2.2.8 Advice on Signed versus Unsigned

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

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

Practice Problem 2.25 (solution page 151)

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

```
/* 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 151)

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

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

Here is your first attempt at the function:

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

When you test this on some sample data, things do not seem to work quite right. You investigate further and determine that, when compiled as a 32-bit program, data type `size_t` is defined (via `typedef`) in header file `stdio.h` to be `unsigned`.

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

B. Explain how this incorrect result comes about.

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

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

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

2.3 Integer Arithmetic

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

2.3.1 Unsigned Addition

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

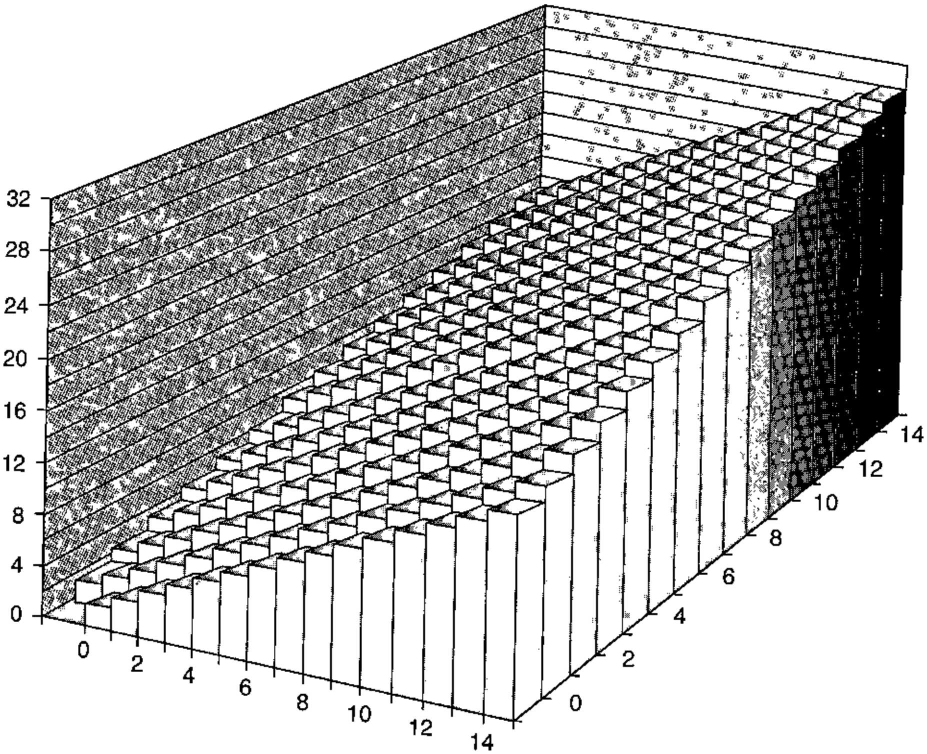


Figure 2.21 Integer addition. With a 4-bit word size, the sum could require 5 bits.

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

Let us define the operation $+^{\mathrm{u}}_{w}$ 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 $0 \leq x, y < 2^w$:

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

■

The two cases of Equation 2.11 are illustrated in Figure 2.22, showing the sum $x + y$ on the left mapping to the unsigned w-bit sum $x +^{\mathrm{u}}_{w} y$ 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 $2^w \leq x + y < 2^{w+1}$, the leading bit in the $(w + 1)$-bit representation of the sum will equal 1, and hence discarding it is equivalent to subtracting 2^w from the sum. ■

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

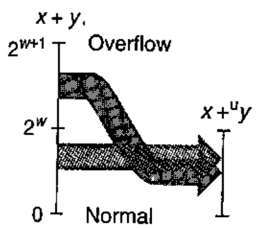


Figure 2.22 Relation between integer addition and unsigned addition. When $x + y$ is greater than $2^w - 1$, the sum overflows.

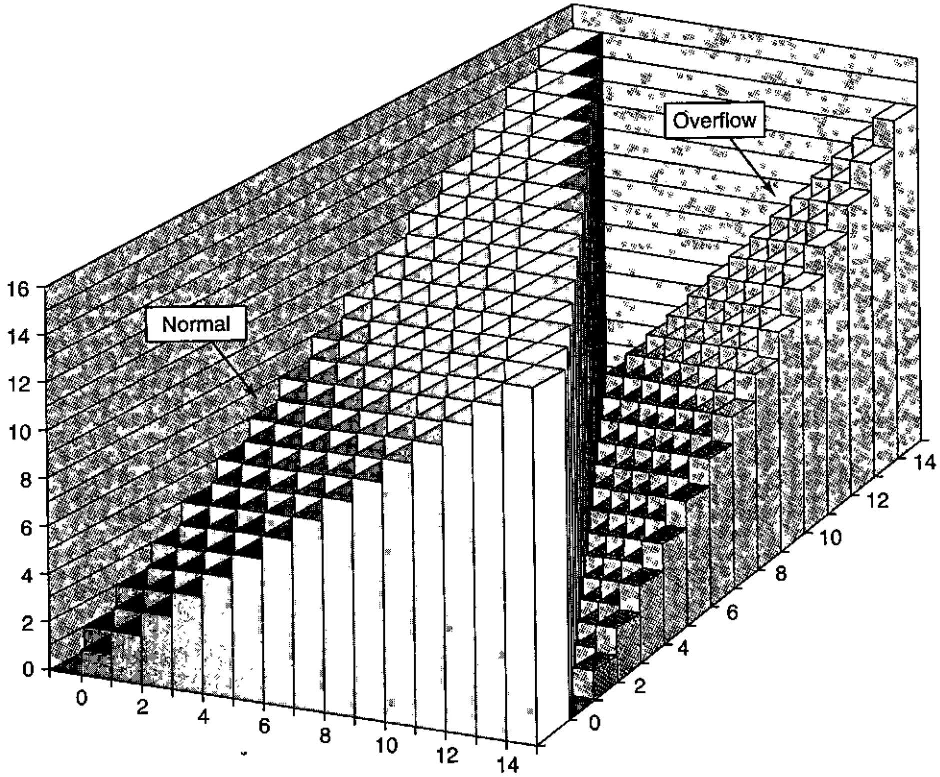


Figure 2.23 Unsigned addition. With a 4-bit word size, addition is performed modulo 16.

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

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

PRINCIPLE: Detecting overflow of unsigned addition

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

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

DERIVATION: Detecting overflow of unsigned addition

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

Practice Problem 2.27 (solution page 152)

Write a function with the following prototype:

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

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

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

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

■

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 $+^{\mathrm{u}}_{w}$. ■

Practice Problem 2.28 (solution page 152)

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

x		$-^{u}_{4}\, x$	
Hex	Decimal	Decimal	Hex
0	____	____	____
5	____	____	____
8	____	____	____
D	____	____	____
F	____	____	____

2.3.2 Two's-Complement Addition

With two's-complement addition, we must decide what to do when the result is either too large (positive) or too small (negative) to represent. Given integer values x and y in the range $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$, 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 $x +^{t}_{w} y$ 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 $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$:

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

■

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 $+^{t}_{w}$ as one of converting its arguments to unsigned, performing unsigned addition, and then converting back to two's complement:

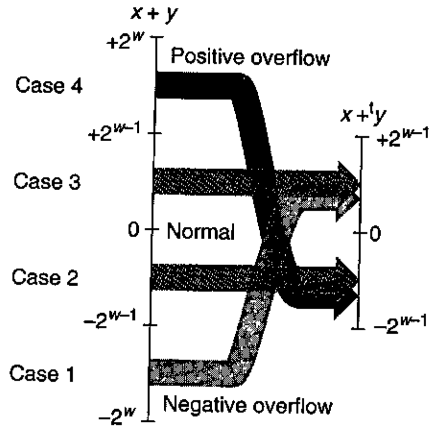


Figure 2.24
Relation between integer and two's-complement addition. When $x + y$ is less than -2^{w-1}, there is a negative overflow. When it is greater than or equal to 2^{w-1}, there is a positive overflow.

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

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

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

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

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 $s \doteq x +^{t}_{w} y$. Then the computation of s has had positive overflow if and only if $x > 0$ and $y > 0$ but $s \leq 0$. The computation has had negative overflow if and only if $x < 0$ and $y < 0$ but $s \geq 0$. ■

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

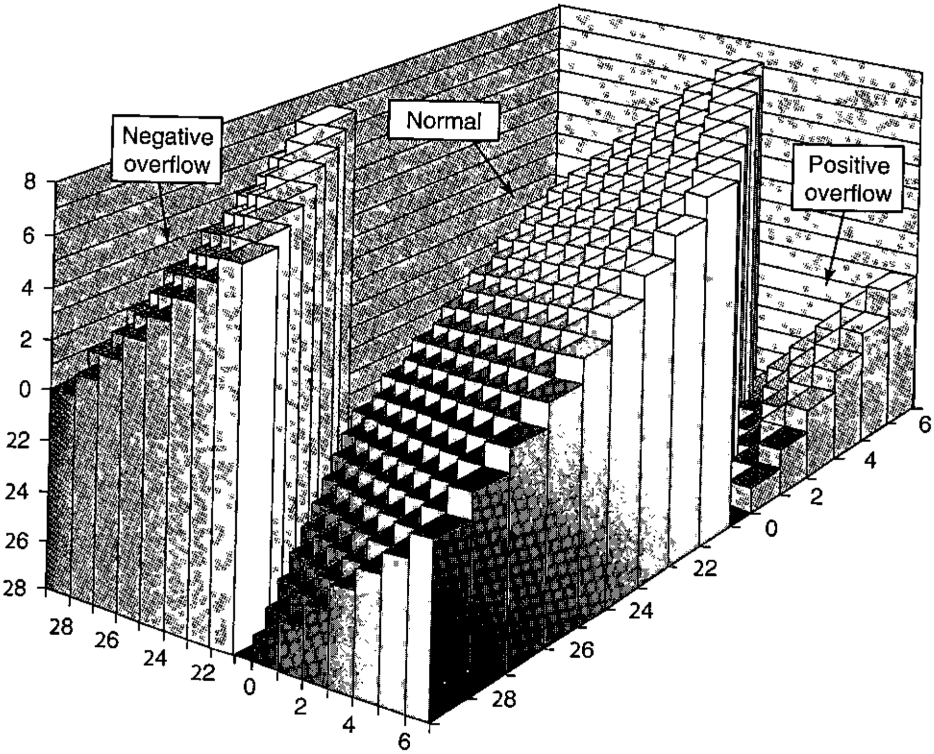


Figure 2.26 Two's-complement addition. With a 4-bit word size, addition can have a negative overflow when $x + y < -8$ and a positive overflow when $x + y \geq 8$.

DERIVATION: Detecting overflow of two's-complement addition

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

Practice Problem 2.29 (solution page 152)

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

x	y	$x + y$	$x +^t_5 y$	Case
______	______	______	______	______
[10100]	[10001]	______	______	______

x	y	$x + y$	$x +^t_5 y$	Case
______	______	______	______	______
[11000]	[11000]	______	______	______
______	______	______	______	______
[10111]	[01000]	______	______	______
______	______	______	______	______
[00010]	[00101]	______	______	______
______	______	______	______	______
[01100]	[00100]	______	______	______

Practice Problem 2.30 (solution page 153)

Write a function with the following prototype:

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

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

Practice Problem 2.31 (solution page 153)

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

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

You look at the code and laugh. Explain why.

Practice Problem 2.32 (solution page 153)

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

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

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

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

2.3.3 Two's-Complement Negation

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

PRINCIPLE: Two's-complement negation

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

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

■

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

DERIVATION: Two's-complement negation

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

Practice Problem 2.33 (solution page 153)

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

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

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

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

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

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

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

$\vec{x}$		$\sim\vec{x}$		$incr(\sim\vec{x})$	
[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 $x *^{u}_{w} y$.

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 $0 \leq x, y \leq UMax_w$:

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

■

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 $x *^{\mathrm{t}}_{w} y$. 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 $TMin_w \leq x, y \leq TMax_w$:

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

■

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

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

■

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

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

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

$$U2B_w(T2U_w(x *^{t}_{w} y)) = T2B_w(x *^{t}_{w} y) = U2B_w(x' *^{u}_{w} y')$$

■

Practice Problem 2.34 (solution page 153)

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

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

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

Practice Problem 2.35 (solution page 154)

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

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

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

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

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

Practice Problem 2.36 (solution page 154)

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

Practice Problem 2.37 (solution page 155)

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

Aside Security vulnerability in the XDR library

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

Code similar to that containing the vulnerability is shown below:

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

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

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

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

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

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

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

Recall that the argument to malloc has type size_t.

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

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

2.3.6 Multiplying by Constants

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

PRINCIPLE: Multiplication by a power of 2

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

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

DERIVATION: Multiplication by a power of 2

This property can be derived using Equation 2.1:

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

■

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

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

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

PRINCIPLE: Unsigned multiplication by a power of 2

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

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

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

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

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

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

Practice Problem 2.38 (solution page 155)

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

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

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

$$[(0 \ldots 0)\,(1 \ldots 1)\,(0 \ldots 0) \cdots (1 \ldots 1)]$$

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

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

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

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

Practice Problem 2.39 (solution page 156)

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

Practice Problem 2.40 (solution page 156)

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

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

Practice Problem 2.41 (solution page 156)

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

2.3.7 Dividing by Powers of 2

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

k	>> k (binary)	Decimal	$12,340/2^k$
0	0011000000110100	12,340	12,340.0
1	*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	110011111100110*1*	*1*110011111100110	−6,170	−6,170.0
4	15	110011111101*1011*	*1111*110011111101	−771	−771.25
8	255	11010000*11001011*	*11111111*11010000	−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 156)

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

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

Practice Problem 2.43 (solution page 157)

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

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

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

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

What are the values of M and N?

2.3.8 Final Thoughts on Integer Arithmetic

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

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

Practice Problem 2.44 (solution page 157)

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

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

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

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

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

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

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

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

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

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

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

2.4 Floating Point

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