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 1000 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, better known as Fibonacci. Using decimal notation is natural for ten-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 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 encodings 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-two 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, 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 $+\infty$. On the other hand, 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`.

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.

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 give you some practice reasoning 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 when compiling arithmetic expressions.

Our treatment of this material is very mathematical. 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 this material from such an abstract viewpoint, because programmers need to have a solid understanding of how computer arithmetic relates to the more familiar integer and real arithmetic. Although it may appear intimidating, the mathematical treatment requires just an understanding of basic algebra. We recommend you work the practice problems as a way to solidify the connection between the formal treatment and some real-life examples.

Aside: How to read this chapter.

If you find equations and formulas daunting, do not let that stop you from getting the most out of this chapter! We provide full derivations of mathematical ideas for completeness, but the best way to read this material is often to skip over the derivation on your initial reading. Instead, try working out a few simple examples (for example, the practice problems) to build your intuition, and then see how the mathematical derivation reinforces your intuition.

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 standard is 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.

2.1 Information Storage

Rather than accessing individual bits in a memory, most computers use blocks of eight 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 10) uses a combination of random-access memory (RAM), disk storage, special hardware, and operating system software to provide the program with what appears to be a monolithic byte array.

One task of a compiler and the run-time system is to subdivide 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 unit—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.

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.

2.1.1 Hexadecimal Notation

A single byte consists of eight 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 2.1 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 lower case. For example, we could write the number $FA1D37B_{16}$ as `0xFA1D37B`,

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 2.1 Hexadecimal notation. Each Hex digit encodes one of 16 values.

as `0xfa1d37b`, or even mixing upper and lower case, 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 2.1. One simple trick for doing the conversion in your head is to memorize the decimal equivalents of hex digits `A`, `C`, and `F`. 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 four bits each. Note, however, that if the total number of bits is not a multiple of four, you should make the *leftmost* group be the one with fewer than four bits, effectively padding the number with leading 0s. Then you translate each group of four bits into the corresponding hexadecimal digit:

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

Practice Problem 2.1

Perform the following number conversions:

A. `0x8F7A93` to binary.

B. Binary 1011011110011100 to hexadecimal.

C. `0xC4E5D` to binary.

D. Binary 1101011011011111100110 to hexadecimal.

When a value x is a power of two, that is, $x = 2^n$ for some n, we can readily write x in hexadecimal form by remembering that the binary representation of x is simply 1 followed by n 0s. The hexadecimal digit `0` represents four binary 0s. 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 `0`s. As an example, for $x = 2048 = 2^{11}$, we have $n = 11 = 3 + 4 \cdot 2$, giving hexadecimal representation `0x800`.

Practice Problem 2.2

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)
11	2048	0x800
7		
	8192	
		0x20000
16		
	256	
		0x20

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 314156:

$$\begin{array}{rcll}
314156 & = & 19634 \cdot 16 + 12 & (\texttt{C}) \\
19634 & = & 1227 \cdot 16 + 2 & (\texttt{2}) \\
1227 & = & 76 \cdot 16 + 11 & (\texttt{B}) \\
76 & = & 4 \cdot 16 + 12 & (\texttt{C}) \\
4 & = & 0 \cdot 16 + 4 & (\texttt{4})
\end{array}$$

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 $7 \cdot 16^2 + 10 \cdot 16 + 15 = 7 \cdot 256 + 10 \cdot 16 + 15 = 1792 + 160 + 15 = 1967$.

Practice Problem 2.3

A single byte can be represented by two 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	00000000	00
55		
136		
243		
	01010010	
	10101100	
	11100111	
		A7
		3E
		BC

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. For example, the following script in the Perl language converts a list of numbers from decimal to hexadecimal:

code/data/d2h

```
#!/usr/local/bin/perl
# Convert list of decimal numbers into hex

for ($i = 0; $i < @ARGV; $i++) {
    printf("%d\t= 0x%x\n", $ARGV[$i], $ARGV[$i]);
}
```

code/data/d2h

Once this file has been set to be executable, the command:

```
unix> ./d2h 100 500 751
```

yields output:

```
100 = 0x64
500 = 0x1f4
751 = 0x2ef
```

Similarly, the following script converts from hexadecimal to decimal:

code/data/h2d

```
#!/usr/local/bin/perl
# Convert list of hex numbers into decimal

for ($i = 0; $i < @ARGV; $i++) {
  $val = hex($ARGV[$i]);
  printf("0x%x = %d\n", $val, $val);
}
```

code/data/h2d

Practice Problem 2.4

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. `0x502c` + `0x8` =

B. `0x502c` − `0x30` =

C. `0x502c` + 64 =

D. `0x50da` − `0x502c` =

2.1.2 Words

Every computer has a *word size*, indicating the nominal size of integer and 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 an n-bit word size, the virtual addresses can range from 0 to $2^n - 1$, giving the program access to at most 2^n bytes.

Most computers today have a 32-bit word size. This limits the virtual address space to 4 gigabytes (written 4 GB), that is, just over 4×10^9 bytes. Although this is ample space for most applications, we have reached the point where many large-scale scientific and database applications require larger amounts of storage. Consequently, high-end machines with 64-bit word sizes are becoming increasingly commonplace as storage costs decrease.

2.1.3 Data Sizes

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 two-, four-, and eight-byte quantities. They also support floating-point numbers represented as four and eight-byte quantities.

C declaration	Typical 32-bit	Compaq Alpha
`char`	1	1
`short int`	2	2
`int`	4	4
`long int`	4	8
`char *`	4	8
`float`	4	4
`double`	8	8

Figure 2.2 Sizes (in bytes) of C numeric data types. The number of bytes allocated varies with machine and compiler.

The C language supports multiple data formats for both integer and floating-point data. The C 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. The C data type `int` can also be prefixed by the qualifiers `long` and `short`, providing integer representations of various sizes. Figure 2.2 shows the number of bytes allocated for various C data types. The exact number depends on both the machine and the compiler. We show two representative cases: a typical 32-bit machine, and the Compaq Alpha architecture, a 64-bit machine targeting high end applications. Most 32-bit machines use the allocations indicated as "typical." Observe that "short" integers have two-byte allocations, while an unqualified `int` is 4 bytes. A "long" integer uses the full word size of the machine.

Figure 2.2 also shows that a pointer (e.g., a variable declared as being of type "`char *`") uses the full word size of the machine. 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 four and eight bytes, respectively.

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`.

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 standard sets lower bounds on the numeric ranges of the different data types, as will be covered later, but there are no upper bounds. Since 32-bit machines have been the standard for the last 20 years, many programs have been written assuming the allocations listed as "typical 32-bit" in Figure 2.2. Given the increasing prominence of 64-bit machines in the near future, many hidden word size dependencies will show up as bugs in migrating these programs to new machines. For example, many programmers assume that a program object declared as type `int` can be used to store a pointer. This works fine for most 32-bit machines but leads to problems on an Alpha.

2.1.4 Addressing and Byte Ordering

For program objects that span multiple bytes, we must establish two conventions: what will be the address of the object, and how will we order the bytes in memory. In virtually all machines, a multibyte 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 the four 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 eight, 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*. This convention is followed by most machines from the former Digital Equipment Corporation (now part of Compaq Corporation), as well as by Intel. The latter convention—where the most significant byte comes first—is referred to as *big endian*. This convention is followed by most machines from IBM, Motorola, and Sun Microsystems. Note that we said "most." The conventions do not split precisely along corporate boundaries. For example, personal computers manufactured by IBM use Intel-compatible processors and hence are little endian. Many microprocessor chips, including Alpha and the PowerPC by Motorola, can be run in either mode, with the byte ordering convention determined when the chip is powered up.

Continuing our earlier example, 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.

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 by which end a soft-boiled egg should be opened—the little end or 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.

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.

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 [17], and the terminology has been widely adopted.

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 is 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 12.

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 processor:

```
80483bd:  01 05 64 94 04 08        add    %eax,0x8049464
```

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 these tools and how to interpret lines such as this in the next chapter. For now, we simply note that this line states that the hexadecimal byte sequence `01 05 64 94 04 08` is the byte-level representation of an instruction that adds `0x8049464` to some program value. If we take the final four bytes of the sequence: `64 94 04 08`, and write them in reverse order, we have `08 04 94 64`. Dropping the leading 0, we have the value `0x8049464`, 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* 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 2.3 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. It prints the individual bytes in hexadecimal. The C formatting directive "`%.2x`" indicates that an integer should be printed in hexadecimal with at least two digits.

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 2.3 has the same form as would the declaration of a variable to type "`unsigned char`."

code/data/show-bytes.c

```
#include <stdio.h>

typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, int 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 *));
}
```

code/data/show-bytes.c

Figure 2.3 Code to print the byte representation of program objects. This code uses casting to circumvent the type system.

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 and '%f' to print a floating-point number, and '%c' to print a character having the character code given by the argument.

New to C?: Pointers and arrays.

In function `show_bytes` (Figure 2.3) 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 9. 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`.

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 used by the object.

New to C?: Pointer creation and dereferencing.

In lines 15, 20, and 25 of Figure 2.3 we see uses of two operations that are unique to C and C++. The C "address of" operator `&` creates a pointer. On all three lines, the expression `&x` creates a pointer to the location holding 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, it now is a pointer to data of type `unsigned char`.

These procedures use the C operator `sizeof` 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 2.4 on several different machines, giving the results shown in Figure 2.5. The following machines were used:

Linux:	Intel Pentium II running Linux.
NT:	Intel Pentium II running Windows-NT.
Sun:	Sun Microsystems UltraSPARC running Solaris.
Alpha:	Compaq Alpha 21164 running Tru64 Unix.

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

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, NT, and Alpha, 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 and Sun machines use four-byte addresses, while the Alpha uses eight-byte addresses.

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

Machine	Value	Type	Bytes (hex)
Linux	12,345	int	39 30 00 00
NT	12,345	int	39 30 00 00
Sun	12,345	int	00 00 30 39
Alpha	12,345	int	39 30 00 00
Linux	12,345.0	float	00 e4 40 46
NT	12,345.0	float	00 e4 40 46
Sun	12,345.0	float	46 40 e4 00
Alpha	12,345.0	float	00 e4 40 46
Linux	&ival	int *	3c fa ff bf
NT	&ival	int *	1c ff 44 02
Sun	&ival	int *	ef ff fc e4
Alpha	&ival	int *	80 fc ff 1f 01 00 00 00

Figure 2.5 Byte representations of different data values. Results for int and float are identical, except for byte ordering. Pointer values are machine-dependent.

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.

Practice Problem 2.5

Consider the following three calls to `show_bytes`:

```
int val = 0x12345678;
byte_pointer valp = (byte_pointer) &val;
show_bytes(valp, 1); /* A. */
show_bytes(valp, 2); /* B. */
show_bytes(valp, 3); /* C. */
```

Indicate which of the following values would 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

Using `show_int` and `show_float`, we determine that the integer 3490593 has hexadecimal representation `0x00354321`, while the floating-point number 3490593.0 has hexadecimal representation `0x4A550C84`.

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.

C. How many bits match? What parts of the strings do not match?

2.1.5 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 is more platform-independent than binary data.

Aside: Generating an ASCII table.

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

Practice Problem 2.7

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

```
char *s = "ABCDEF";
show_bytes(s, strlen(s));
```

Note that letters '`A`' through '`Z`' have ASCII codes `0x41` through `0x5A`.

Aside: The Unicode character set.

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. Recently, the 16-bit *Unicode* character set has been adopted to support documents in all languages. This doubling of the character set representation enables a very large number of different characters to be represented. The Java programming language uses Unicode when representing character strings. Program libraries are also available for C that provide Unicode versions of the standard string functions such as `strlen` and `strcpy`.

2.1.6 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:    55 89 e5 8b 45 0c 03 45 08 89 ec 5d c3
NT:       55 89 e5 8b 45 0c 03 45 08 89 ec 5d c3
Sun:      81 C3 E0 08 90 02 00 09
Alpha:    00 00 30 42 01 80 FA 6B
```

Here we find that the instruction codings are different, except for the NT and Linux machines. Different machine types use different and incompatible instructions and encodings. The NT and Linux machines both have Intel processors and hence support the same machine-level instructions. In general, however, the structure of an executable NT program differs from a Linux program, and hence the machines are not fully 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.

2.1.7 Boolean Algebras and Rings

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 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 properties of propositional logic.

There is an infinite number of different Boolean algebras, where the simplest is defined over the two-element set $\{0, 1\}$. Figure 2.6 defines several operations in this Boolean algebra. Our symbols for representing these operations are chosen to match those used by the C bit-level operations, as will be discussed later. The Boolean operation `~` corresponds to the logical operation NOT, denoted in propositional logic as $\neg$. That is, we say that $\neg P$ is true when P is not true, and vice-versa. Correspondingly, $\text{\~{}}p$ equals 1 when p equals 0, and vice-versa. Boolean operation `&` corresponds to the logical operation AND, denoted in propositional logic as $\wedge$. We say that $P \wedge Q$ holds when both P and Q are true. Correspondingly, $p \,\&\, q$ equals 1 only when $p = 1$ and $q = 1$. Boolean operation `|` corresponds to the logical operation OR, denoted in propositional logic as $\vee$. We say that $P \vee Q$ holds when either P or Q are true. Correspondingly, $p \mid q$ equals 1 when either $p = 1$ or $q = 1$. Boolean operation `^` corresponds to the logical operation EXCLUSIVE-OR, denoted in propositional logic as $\oplus$. We say that $P \oplus Q$ holds when either P or Q are true, but not both. Correspondingly, $p \,\hat{}\, q$ equals 1 when either $p = 1$ and $q = 0$, or $p = 0$ and $q = 1$.

Claude Shannon, who later founded the field of information theory, first made the connection between Boolean algebra and digital logic. In his 1937 master's

Figure 2.6
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.

~	
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

Shared properties

Property	Integer ring	Boolean algebra
Commutativity	$a + b = b + a$	$a \mid b = b \mid a$
	$a \times b = b \times a$	$a \,\&\, b = b \,\&\, a$
Associativity	$(a + b) + c = a + (b + c)$	$(a \mid b) \mid c = a \mid (b \mid c)$
	$(a \times b) \times c = a \times (b \times c)$	$(a \,\&\, b) \,\&\, c = a \,\&\, (b \,\&\, c)$
Distributivity	$a \times (b + c) = (a \times b) + (a \times c)$	$a \,\&\, (b \mid c) = (a \,\&\, b) \mid (a \,\&\, c)$
Identities	$a + 0 = a$	$a \mid 0 = a$
	$a \times 1 = a$	$a \,\&\, 1 = a$
Annihilator	$a \times 0 = 0$	$a \,\&\, 0 = 0$
Cancellation	$-(-a) = a$	$\sim(\sim a) = a$

Unique to Rings

Inverse	$a + -a = 0$	—

Unique to Boolean Algebras

Distributivity	—	$a \mid (b \,\&\, c) = (a \mid b) \,\&\, (a \mid c)$
Complement	—	$a \mid \sim a = 1$
	—	$a \,\&\, \sim a = 0$
Idempotency	—	$a \,\&\, a = a$
	—	$a \mid a = a$
Absorption	—	$a \mid (a \,\&\, b) = a$
	—	$a \,\&\, (a \mid b) = a$
DeMorgan's laws	—	$\sim(a \,\&\, b) = \sim a \mid \sim b$
	—	$\sim(a \mid b) = \sim a \,\&\, \sim b$

Figure 2.7 Comparison of integer ring and Boolean algebra. The two mathematical structures share many properties, but there are key differences, particularly between $-$ and $\sim$.

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.

There are many parallels between integer arithmetic and Boolean algebra, as well as several important differences. In particular, the set of integers, denoted $\mathbb{Z}$, forms a mathematical structure known as a *ring*, denoted $\langle \mathbb{Z}, +, \times, -, 0, 1 \rangle$, with addition serving as the *sum* operation, multiplication as the *product* operation, negation as the additive inverse, and elements 0 and 1 serving as the additive and multiplicative identities. The Boolean algebra $\langle \{0, 1\}, \mid, \&, \sim, 0, 1 \rangle$ has similar properties. Figure 2.7 highlights properties of these two structures, showing the properties that are common to both and those that are unique to one or the other. One important difference is that $\sim a$ is not an inverse for a under $\mid$.

Aside: What good is abstract algebra?

Abstract algebra involves identifying and analyzing the common properties of mathematical operations in different domains. Typically, an algebra is characterized by a set of elements, some of its key operations, and some important elements. As an example, modular arithmetic also forms a ring. For modulus n, the algebra is denoted $\langle \mathbb{Z}_n, +_n, \times_n, -_n, 0, 1\rangle$, with components defined as follows:

$$\begin{aligned}
\mathbb{Z}_n &= \{0, 1, \ldots, n-1\} \\
a +_n b &= a + b \bmod n \\
a \times_n b &= a \times b \bmod n \\
-_n a &= \begin{cases} 0, & a = 0 \\ n - a, & a > 0 \end{cases}
\end{aligned}$$

Even though modular arithmetic yields different results from integer arithmetic, it has many of the same mathematical properties. Other well-known rings include rational and real numbers.

If we replace the OR operation of Boolean algebra by the EXCLUSIVE-OR operation, and the complement operation ~ with the identity operation I—where $I(a) = a$ for all a—we have a structure $\langle\{0, 1\}, \hat{}, \&, I, 0, 1\rangle$. This structure is no longer a Boolean algebra—in fact it's a ring. It can be seen to be a particularly simple form of the ring consisting of all integers $\{0, 1, \ldots, n-1\}$ with both addition and multiplication performed modulo n. In this case, we have $n = 2$. That is, the Boolean AND and EXCLUSIVE-OR operations correspond to multiplication and addition modulo 2, respectively. One curious property of this algebra is that every element is its own additive inverse: $a \mathbin{\hat{}} I(a) = a \mathbin{\hat{}} a = 0$.

Aside: Who, besides mathematicians, care about Boolean rings?

Every time you enjoy the clarity of music recorded on a CD or the quality of video recorded on a DVD, you are taking advantage of Boolean rings. These technologies rely on *error-correcting codes* to reliably retrieve the bits from a disk even when dirt and scratches are present. The mathematical basis for these error-correcting codes is a linear algebra based on Boolean rings.

We can extend the four Boolean operations to also operate on bit vectors, i.e., strings of 0s and 1s of some fixed length w. We define the operations over bit vectors according their applications to the matching elements of the arguments. For example, we define $[a_{w-1}, a_{w-2}, \ldots, a_0] \,\&\, [b_{w-1}, b_{w-2}, \ldots, b_0]$ to be $[a_{w-1} \,\&\, b_{w-1}, a_{w-2} \,\&\, b_{w-2}, \ldots, a_0 \,\&\, b_0]$, and similarly for operations ~, |, and ^. Letting $\{0, 1\}^w$ denote the set of all strings of 0s and 1s having length w, and a^w denote the string consisting of w repetitions of symbol a, then one can see that the resulting algebras: $\langle\{0, 1\}^w, |, \&, \sim, 0^w, 1^w\rangle$ and $\langle\{0, 1\}^w, \hat{}, \&, I, 0^w, 1^w\rangle$

form Boolean algebras and rings, respectively. Each value of w defines a different Boolean algebra and a different Boolean ring.

Aside: Are Boolean rings the same as modular arithmetic?

The two-element Boolean ring $\langle \{0, 1\}, \hat{}, \&, I, 0, 1 \rangle$ is identical to the ring of integers modulo two $\langle \mathbb{Z}_2, +_2, \times_2, -_2, 0, 1 \rangle$. The generalization to bit vectors of length w, however, yields a very different ring from modular arithmetic.

Practice Problem 2.8

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

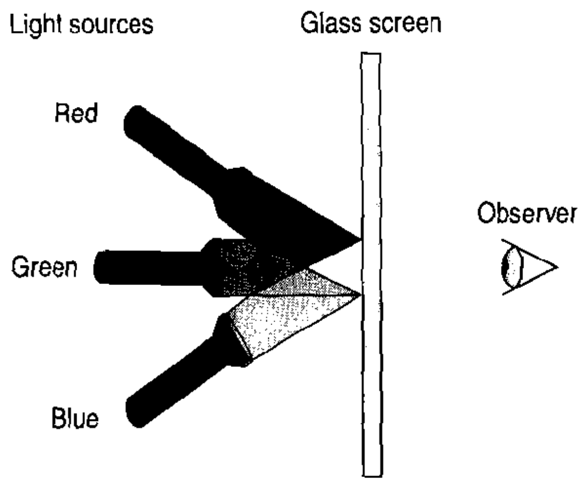
Operation	Result
a	[01101001]
b	[01010101]
~a	
~b	
a & b	
a | b	
a ^ b	

One useful application of bit vectors is to represent finite sets. For example, we can denote any subset $A \subseteq \{0, 1, \ldots, w-1\}$ as 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), we have a = [01101001] representing the set $A = \{0, 3, 5, 6\}$, and b = [01010101] representing the set $B = \{0, 2, 4, 6\}$. Under this interpretation, Boolean operations | and & correspond to set union and intersection, respectively, and ~ corresponds to set complement. For example, the operation a & b yields bit vector [01000001], while $A \cap B = \{0, 6\}$.

In fact, for any set S, the structure $\langle \mathcal{P}(S), \cup, \cap, \overline{}, \emptyset, S \rangle$ forms a Boolean algebra, where $\mathcal{P}(S)$ denotes the set of all subsets of S, and $\overline{}$ denotes the set complement operator. That is, for any set A, its complement is the set $\overline{A} = \{a \in S | a \notin A\}$. The ability to represent and manipulate finite sets using bit vector operations is a practical outcome of a deep mathematical principle.

Practice Problem 2.9

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

This set of colors forms an eight-element Boolean algebra.

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 complements of the eight colors listed above?

B. What colors correspond to Boolean values 0^w and 1^w for this algebra?

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

Blue	|	Red	=
Magenta	&	Cyan	=
Green	^	White	=

2.1.8 Bit-Level Operations in C

One useful feature of C is that it supports bit-wise 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, that is, one declared as type `char` or `int`, with or without qualifiers such as `short`, `long`, or `unsigned`. Here are some examples of expression evaluation:

C expression	Binary expression	Binary result	C result
`~0x41`	~[01000001]	[10111110]	`0xBE`
`~0x00`	~[00000000]	[11111111]	`0xFF`
`0x69 & 0x55`	[01101001] & [01010101]	[01000001]	`0x41`
`0x69 | 0x55`	[01101001] | [01010101]	[01111101]	`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

To show how the ring properties of ^ can be useful, consider the following program:

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

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 1s for the least significant eight 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 1s, regardless of the word size of the machine. Although the same mask can be written `0xFFFFFFFF` for a 32-bit machine, such code is not as portable.

Practice Problem 2.11

Write C expressions for the following values, with the results for x = `0x98FDECBA` and a 32-bit word size shown in square brackets:

A. The least significant byte of x, with all other bits set to 1 [`0xFFFFFFBA`].

B. The complement of the least significant byte of x, with all other bytes left unchanged [`0x98FDEC45`].

C. All but the least significant byte of x, with the least significant byte set to 0 [`0x98FDEC00`].

Although our examples assume a 32-bit word size, your code should work for any word size $w \geq 8$.

Practice Problem 2.12

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.

We would like to write C functions `bis` and `bic` to compute the effect of these two instructions. Fill in the missing expressions in the following code, using the bit-level operations of C:

```
/* Bit Set */
int bis(int x, int m)
{
  /* Write an expression in C that computes the effect of bit set */
  int result = ___________;
  return result;
}
```

```
/* Bit Clear */
int bic(int x, int m)
{
  /* Write an expression in C that computes the effect of bit clear */
  int result = __________;
  return result;
}
```

2.1.9 Logical Operations in C

C also provides a set of *logical* operators `||`, `&&`, and `!`, which correspond to the OR, AND, and NOT operations of propositional logic. These can easily be confused with the bit-level operations, but their function 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 bit-wise 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 2.13

Suppose that `x` and `y` have byte values `0x66` and `0x93`, 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 2.14

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.

2.1.10 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 $[x_{n-1}, x_{n-2}, \ldots, x_0]$, the C expression `x << k` yields a value with bit representation $[x_{n-k-1}, x_{n-k-2}, \ldots, x_0, 0, \ldots 0]$. That is, x is shifted k bits to the left, dropping off the k most significant bits and filling the right end with k 0s. The shift amount should be a value between 0 and $n-1$. Shift operations group from left to right, so `x << j << k` is equivalent to `(x << j) << k`. Be careful about operator precedence: `1<<5 - 1` is evaluated as `1 << (5-1)`, not as `(1<<5) - 1`.

There is a corresponding right shift operation `x >> k`, but it has a slightly subtle behavior. Generally, machines support two forms of right shift: *logical* and *arithmetic*. A logical right shift fills the left end with k 0s, giving a result $[0, \ldots, 0, x_{n-1}, x_{n-2}, \ldots x_k]$. An arithmetic right shift fills the left end with k repetitions of the most significant bit, giving a result $[x_{n-1}, \ldots, x_{n-1}, x_{n-1}, x_{n-2}, \ldots x_k]$. This convention might seem peculiar, but as we will see it is useful for operating on signed integer data.

The C standard does not precisely define which type of right shift should be used. For unsigned data (i.e., integral objects declared with the qualifier `unsigned`), right shifts must be logical. For signed data (the default), 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.

Practice Problem 2.15

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`		`x >> 2` (Logical)		`x >> 2` (Arithmetic)	
Hex	Binary	Binary	Hex	Binary	Hex	Binary	Hex
`0xF0`							
`0x0F`							
`0xCC`							
`0x55`							

C declaration	Guaranteed		Typical 32-bit	
	Minimum	Maximum	Minimum	Maximum
`char`	−127	127	−128	127
`unsigned char`	0	255	0	255
`short [int]`	−32,767	32,767	−32,768	32,767
`unsigned short [int]`	0	63,535	0	63,535
`int`	−32,767	32,767	−2,147,483,648	2,147,483,647
`unsigned [int]`	0	65,535	0	4,294,967,295
`long [int]`	−2,147,483,647	2,147,483,647	−2,147,483,648	2,147,483,647
`unsigned long [int]`	0	4,294,967,295	0	4,294,967,295

Figure 2.8 C Integral data types. Text in square brackets is optional.

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.

2.2.1 Integral Data Types

C supports a variety of *integral* data types—ones that represent a finite range of integers. These are shown in Figure 2.8. Each type has a size designator: `char`, `short`, `int`, and `long`, as well as an indication of whether the represented number is nonnegative (declared as `unsigned`), or possibly negative (the default). The typical allocations for these different sizes were given in Figure 2.2. As indicated in Figure 2.8, these different sizes allow different ranges of values to be represented. The C standard defines a minimum range of values each data type must be able to represent. As shown in the figure, a typical 32-bit machine uses a 32-bit representation for data types `int` and `unsigned`, even though the C standard allows 16-bit representations. As described in Figure 2.2, the Compaq Alpha uses a 64-bit word to represent `long` integers, giving an upper limit of over 1.84×10^{19} for unsigned values, and a range of over $\pm 9.22 \times 10^{18}$ for signed values.

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.

2.2.2 Unsigned and Two's-Complement Encodings

Assume we have 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 $\vec{x}$. We express this interpretation as a function $B2U_w$ (for "binary to unsigned," length w):

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

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 0s and 1s of length w to nonnegative integers. The least value is given by bit vector $[00\cdots0]$ having integer value 0, and the greatest value is given by bit vector $[11\cdots1]$ having integer value $UMax_w \doteq \sum_{i=0}^{w-1} 2^i = 2^w - 1$. Thus, the function $B2U_w$ can be defined as a mapping $B2U_w\colon \{0,1\}^w \rightarrow \{0,\ldots,2^w-1\}$. Note that $B2U_w$ is a *bijection*—it associates a unique value to each bit vector of length w; conversely, each integer between 0 and $2^w - 1$ has a unique binary representation as a bit vector of length w.

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):

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \tag{2.2}$$

The most significant bit is also called the *sign bit*. When set to 1, the represented value is negative, and when set to 0 the value is nonnegative. The least representable value is given by bit vector $[10\cdots0]$ (i.e., 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 $[01\cdots1]$, having integer value $TMax_w \doteq \sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$. Again, one can see that $B2T_w$ is a bijection $B2T_w\colon \{0,1\}^w \rightarrow \{-2^{w-1},\ldots,2^{w-1}-1\}$, associating a unique integer in the representable range for each bit pattern.

Practice Problem 2.16

Assuming $w = 4$, we can assign a numeric value to each possible hexadecimal digit, assuming either an unsigned or two's-complement interpretation. Fill in the following table according to these interpretations by writing out the nonzero powers of two in the summations shown in Equations 2.1 and 2.2:

$\vec{x}$		$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
Hexadecimal	Binary		
A	[1010]	$2^3+2^1=10$	$-2^3+2^1=-6$
0			
3			
8			
C			
F			

Figure 2.9 shows the bit patterns and numeric values for several "interesting" numbers for different word sizes. The first three give the ranges of representable integers. A few points are worth highlighting. First, the two's-complement range is asymmetric: $|TMin_w| = |TMax_w| + 1$, that is, there is no positive counterpart to $TMin_w$. As we shall see, this leads to some peculiar properties of two's-complement arithmetic and can be the source of subtle program bugs. Second, the maximum unsigned value is just over twice the maximum two's-complement value: $UMax_w = 2TMax_w + 1$. This follows from the fact that two's-complement notation reserves half of the bit patterns to represent negative values. The other cases are the constants -1 and 0. Note that -1 has the same bit representation as $UMax_w$—a string of all 1s. Numeric value 0 is represented as a string of all 0s in both representations.

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

Figure 2.9 "Interesting" numbers. Both numeric values and hexadecimal representations are shown.

The C standard does not require signed integers to be represented in two's-complement form, but nearly all machines do so. To keep code portable, one should not assume any particular range of representable values or how they are represented, beyond the ranges indicated in Figure 2.2. The file `<limits.h>` in the C library defines a set of constants 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 $UMax_w$.

Aside: Alternative representations of signed numbers.

There are two other standard representations for signed numbers:

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

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

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

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

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

Note the different position of apostrophes: *Two's*-complement versus *Ones'* complement.

As an example, consider the following code:

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

show_bytes((byte_pointer) &x, sizeof(short int));
show_bytes((byte_pointer) &mx, sizeof(short int));
```

Weight	12,345		−12,345		53,191	
	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	4096	0	0	0	0
8,192	1	8192	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 2.10 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 2.10 shows, Equation 2.2 yields values 12,345 and −12,345 for these two bit patterns.

Practice Problem 2.17

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–K (on the right) in the following listing, convert the hexadecimal values shown to the right of the instruction names (`sub`, `push`, `mov`, and `add`) into their decimal equivalents.

```
80483b7:  81 ec 84 01 00 00   sub   $0x184,%esp               A.
80483bd:  53                  push  %ebx
80483be:  8b 55 08            mov   0x8(%ebp),%edx            B.
80483c1:  8b 5d 0c            mov   0xc(%ebp),%ebx            C.
80483c4:  8b 4d 10            mov   0x10(%ebp),%ecx           D.
80483c7:  8b 85 94 fe ff ff   mov   0xfffffe94(%ebp),%eax     E.
80483cd:  01 cb               add   %ecx,%ebx
80483cf:  03 42 10            add   0x10(%edx),%eax           F.
80483d2:  89 85 a0 fe ff ff   mov   %eax,0xfffffea0(%ebp)     G.
80483d8:  8b 85 10 ff ff ff   mov   0xffffff10(%ebp),%eax     H.
80483de:  89 42 1c            mov   %eax,0x1c(%edx)           I.
80483e1:  89 9d 7c ff ff ff   mov   %ebx,0xffffff7c(%ebp)     J.
80483e7:  8b 42 18            mov   0x18(%edx),%eax           K.
```

2.2.3 Conversions Between Signed and Unsigned

Since both $B2U_w$ and $B2T_w$ are bijections, they have well-defined inverses. Define $U2B_w$ to be $B2U_w^{-1}$, and $T2B_w$ to be $B2T_2^{-1}$. These functions give the unsigned or two's-complement bit patterns for a numeric value. Given an integer x in the range $0 \leq x < 2^w$, the function $U2B_w(x)$ gives the unique w-bit unsigned representation of x. Similarly, when x is in the range $-2^{w-1} \leq x < 2^{w-1}$, the function $T2B_w(x)$ gives the unique w-bit two's-complement representation of x. Observe that for values in the range $0 \leq x < 2^{w-1}$, both of these functions will yield the same bit representation—the most significant bit will be 0, and hence it does not matter whether this bit has positive or negative weight.

Consider the function $U2T_w(x) \doteq B2T_w(U2B_w(x))$, which takes a number between 0 and $2^w - 1$ and yields a number between -2^{w-1} and $2^{w-1} - 1$, where the two numbers have identical bit representations, except that the argument is unsigned, while the result has a two's-complement representation. Conversely, the function $T2U_w(x) \doteq B2U_w(T2B_w(x))$ yields the unsigned number having the same bit representation as the two's-complement value of x. For example, as Figure 2.10 indicates, the 16-bit, two's-complement representation of $-12{,}345$ is identical to the 16-bit, unsigned representation of 53,191. Therefore, $T2U_{16}(-12{,}345) = 53{,}191$, and $U2T_{16}(53{,}191) = -12{,}345$.

These two functions might seem to be of only academic interest, but they actually have great practical importance—they formally define the effect of casting between signed and unsigned values in C. For example, consider executing the following code on a two's-complement machine:

```
int x = -1;
unsigned ux = (unsigned) x;
```

This code will set ux to $UMax_w$, where w is the number of bits in data type `int`, since by Figure 2.9 we can see that the w-bit two's-complement representation of -1 has the same bit representation as $UMax_w$. In general, casting from a signed value x to unsigned value `(unsigned)` x is equivalent to applying function $T2U$. The cast does not change the bit representation of the argument, just how these bits are interpreted as a number. Similarly, casting from unsigned value `u` to signed value `(int) u` is equivalent to applying function *U2T*.

Practice Problem 2.18

Using the table you filled in when solving Problem 2.16, fill in the following table describing the function $T2U_4$:

x	$T2U_4(x)$
−8	
−6	
−4	
−1	
0	
3	

To get a better understanding of the relation between a signed number x and its unsigned counterpart $T2U_w(x)$, we can use the fact that they have identical bit representations to derive a numerical relationship. Comparing Equations 2.1 and 2.2, 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 $B2U_w(\vec{x}) = x_{w-1}2^w + B2T_w(\vec{x})$. If we let $x = B2T_w(\vec{x})$, we then have

$$B2U_w(T2B_w(x)) = T2U_w(x) = x_{w-1}2^w + x \qquad (2.3)$$

This relationship is useful for proving relationships between unsigned and two's-complement arithmetic. In the two's-complement representation of x, bit x_{w-1} determines whether or not x is negative, giving

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases} \qquad (2.4)$$

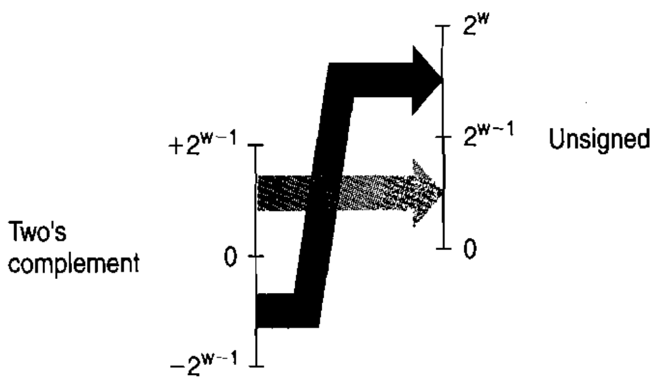


Figure 2.11
Conversion from two's-complement to unsigned. Function *T2U* converts negative numbers to large positive numbers.

Figure 2.11 illustrates the behavior of function $T2U$. As it illustrates, when mapping a signed number to its unsigned counterpart, negative numbers are converted to large positive numbers, while nonnegative numbers remain unchanged.

Practice Problem 2.19

Explain how Equation 2.4 applies to the entries in the table you generated when solving Problem 2.18.

Going in the other direction, we wish to derive the relationship between an unsigned number x and its signed counterpart $U2T_w(x)$. If we let $x = B2U_w(\vec{x})$, we have

$$B2T_w(U2B_w(x)) \quad = \quad U2T_w(x) \quad = \quad -x_{w-1}2^w + x \qquad (2.5)$$

In the unsigned representation of x, bit x_{w-1} determines whether or not x is greater than or equal to 2^{w-1}, giving

$$U2T_w(x) \quad = \quad \begin{cases} x, & x < 2^{w-1} \\ x - 2^w, & x \geq 2^{w-1} \end{cases} \qquad (2.6)$$

This behavior is illustrated in Figure 2.12. For small ($< 2^{w-1}$) numbers, the conversion from unsigned to signed preserves the numeric value. For large ($\geq 2^{w-1}$) the number is converted to a negative value.

To summarize, we can consider the effects of converting in both directions between unsigned and two's-complement representations. For values in the range $0 \leq x < 2^{w-1}$, 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 0 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 2.10, we can see that $T2U_{16}(-12{,}345) = 65{,}536 + -12{,}345 = 53{,}191$.

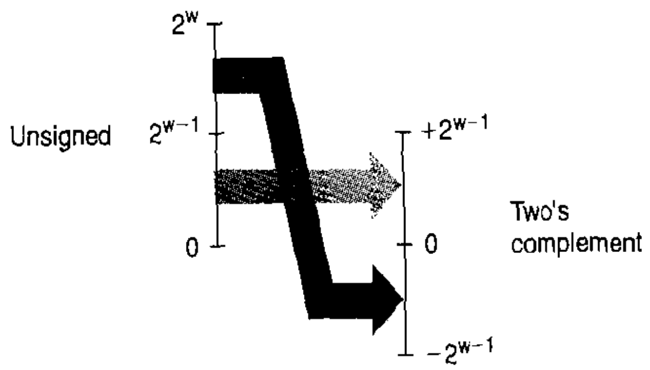


Figure 2.12
Conversion from unsigned to two's-complement. Function $U2T$ converts numbers greater than $2^{w-1} - 1$ to negative values.

2.2.4 Signed vs. Unsigned in C

As indicated in Figure 2.8, 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. To create an unsigned constant, the character '`U`' or '`u`' must be added as suffix (e.g., `12345U` or `0x1A2Bu`).

C allows conversion between unsigned and signed. The rule is that the underlying bit representation is not changed. Thus, on a two's-complement machine, the effect is to apply 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` should be 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 run on a 32-bit machine, it prints the following:

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

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 2.13 Effects of C promotion rules on 32-bit machine. Nonintuitive cases marked by '*.' We must write $TMin_{32}$ in C as `-2147483647-1`, rather than `-2147483648`, to avoid overflow problems. The compiler processes an expression of the form $-X$ by first reading the expression X and then negating it, but `2147483648` is too large to represent as a 32-bit, two's-complement number.

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: $T2U_{32}(-1) = UMax_{32} = 4{,}294{,}967{,}295$ and $U2T_{32}(2^{31}) = 2^{31} - 2^{32} = -2^{31} = TMin_{32}$.

Some peculiar 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 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 2.13 shows some sample relational expressions and their resulting evaluations, assuming a 32-bit machine using 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 2.20

Assuming the expressions are evaluated on a 32-bit 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 2.13:

Expression	Type	Evaluation
`-2147483647-1 == 2147483648U`		
`-2147483647-1 < -2147483647`		
`(unsigned) (-2147483647-1) < -2147483647`		
`-2147483647-1 < 2147483647`		
`(unsigned) (-2147483647-1) < 2147483647`		

2.2.5 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. To convert an unsigned number to a larger data type, we can simply add leading 0s to the representation. this operation is known as *zero extension*. 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. Thus, if our original value has bit representation $[x_{w-1}, x_{w-2}, \ldots, x_0]$, the expanded representation would be $[x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]$.

As an example, consider the following code:

```
short sx = val;            /* -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 on a 32-bit big-endian machine using two's-complement representations, this code prints the following 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 0s, having hexadecimal representation 0x0000.

Can we justify that sign extension works? What we want to prove is that

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

where in the expression on the lefthand side, we have made k additional copies of bit x_{w-1}. The proof follows by induction on k. That is, if we can prove that sign extending by one 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 lefthand expression with Equation 2.2 gives the following:

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

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.

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 additional code for our previous example:

```
unsigned  uy = x;          /* Mystery! */

printf("uy  = %u:\t", uy);
show_bytes((byte_pointer) &uy, sizeof(unsigned));
```

This portion of the code causes the following output to be printed:

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

This shows that the expressions:

```
(unsigned) (int) sx            /* 4294954951 */
```

and

```
(unsigned) (unsigned short) sx      /* 53191 */
```

produce different values, even though the original and the final data types are the same. In the former expression, we first sign extend the 16-bit `short` to a 32-bit `int`, whereas zero extension is performed in the latter expression.

Practice Problem 2.21

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 on a machine with a 32-bit word size 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:

w	fun1(w)	fun2(w)
127		
128		
255		
256		

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

2.2.6 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 */
```

On a typical 32-bit machine, when we cast x to be `short`, we truncate the 32-bit `int` to be a 16-bit `short int`. As we saw before, this 16-bit pattern is the two's-complement representation of $-12{,}345$. When we cast this back to `int`, sign extension will set the high-order 16 bits to 1s, 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. We now investigate what numeric value will result. For an unsigned number x, the result of truncating it to k bits is equivalent to computing $x \bmod 2^k$. This can be seen by applying the modulus operation to Equation 2.1:

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

In this derivation, we make use of the property that $2^i \bmod 2^k = 0$ for any $i \geq k$, and that $\sum_{i=0}^{k-1} x_i 2^i \leq \sum_{i=0}^{k-1} 2^i = 2^k - 1 < 2^k$.

For a two's-complement number x, a similar argument shows that $B2T_w([x_w, x_{w-1}, \ldots, x_0]) \bmod 2^k = B2U_k([x_k, x_{k-1}, \ldots, x_0])$. That is, $x \bmod 2^k$ can be represented by an unsigned number having bit-level representation $[x_{k-1}, \ldots, x_0]$. In general, however, we treat the truncated number as being signed. This will have numeric value $U2T_k(x \bmod 2^k)$.

Summarizing, the effects of truncation are as follows:

$$B2U_k([x_k, x_{k-1}, \ldots, x_0]) = B2U_w([x_w, x_{w-1}, \ldots, x_0]) \bmod 2^k \qquad (2.7)$$

$$B2T_k([x_k, x_{k-1}, \ldots, x_0]) = U2T_k(B2T_w([x_w, x_{w-1}, \ldots, x_0]) \bmod 2^k) \qquad (2.8)$$

Practice Problem 2.22

Suppose we truncate a four-bit value (represented by hex digits 0 through F) to a three-bit value (represented as hex digits 0 through 7). Fill in the following table 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	
3	3	3		3	
8	0	8		−8	
A	2	10		−6	
F	7	15		−1	

Explain how Equations 2.7 and 2.8 apply to these cases.

2.2.7 Advice on Signed vs. 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 is invisible, we can often overlook its effects.

Practice Problem 2.23

Consider the following code, which 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, and show how the code can be corrected.

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 $0 \le x, y \le 2^w - 1$. Each of these numbers can be represented by w-bit unsigned numbers. If we compute their sum, however, we have a possible range $0 \le x + y \le 2^{w+1} - 2$. Representing this sum could require $w + 1$ bits. For example, Figure 2.14 shows a plot of the function $x + y$ when x and y have four-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. 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 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 *infinite precision* arithmetic to allow arbitrary (within the memory limits of the machine, of course) integer arithmetic. More commonly, programming languages support fixed-precision arithmetic, and hence operations such as "addition" and "multiplication" differ from their counterpart operations over integers.

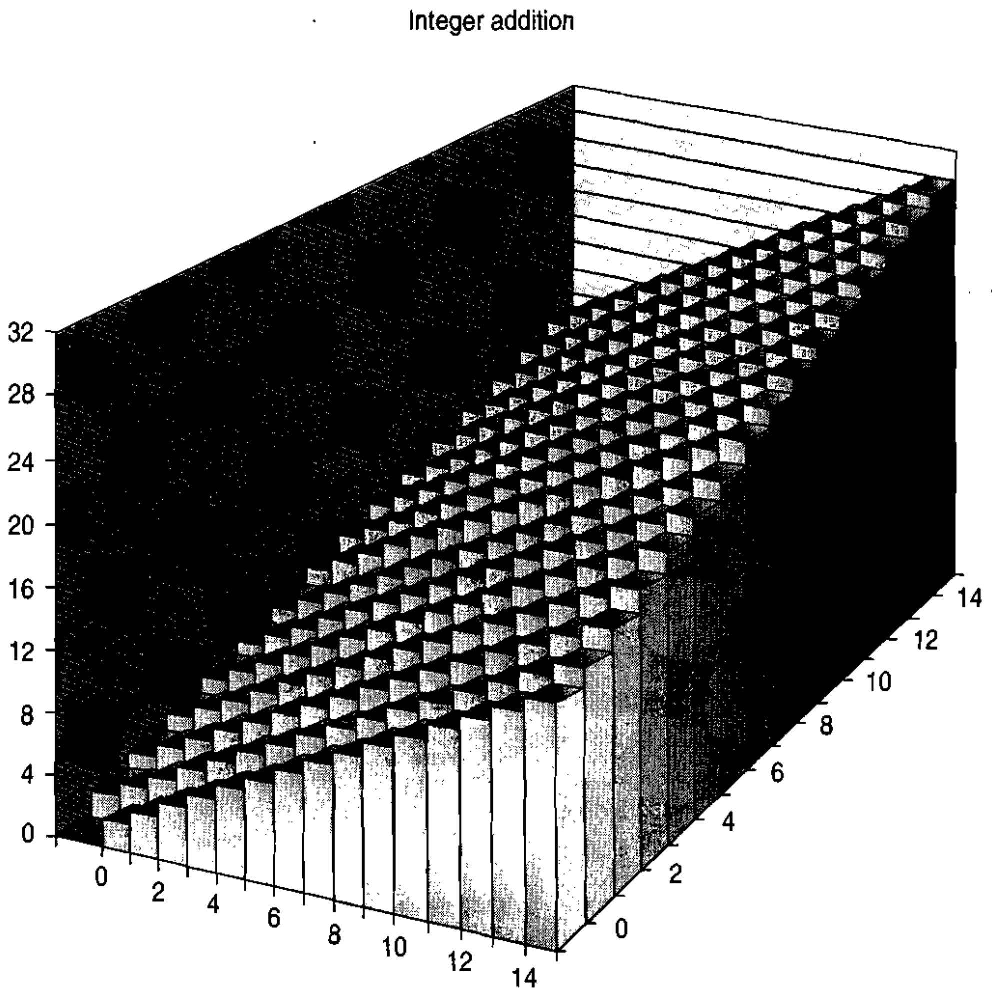


Figure 2.14 Integer addition. With a four-bit word size, the sum could require 5 bits.

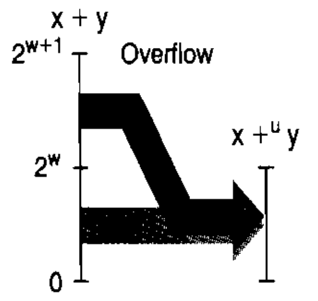


Figure 2.15
Relation between integer addition and unsigned addition. When $x + y$ is greater than $2^w - 1$, the sum overflows.

Unsigned arithmetic can be viewed as a form of modular arithmetic. Unsigned addition is equivalent to computing the sum modulo 2^w. This value can be computed by simply discarding the high-order bit in the $w+1$-bit representation of $x + y$. For example, consider a four-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.

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. These two cases are illustrated in Figure 2.15. This will give us a value in the range $0 \leq x+y-2^w < 2^{w+1}-2^w = 2^w$, which is precisely the modulo 2^w sum of x and y. Let us define the operation $+^{\mathrm{u}}_{w}$ for arguments x and y such that $0 \leq x, y < 2^w$, as follows:

$$x +^{\mathrm{u}}_{w} y \quad = \quad \begin{cases} x+y, & x+y<2^w \\ x+y-2^w, & 2^w \leq x+y < 2^{w+1} \end{cases} \tag{2.9}$$

This is precisely the result we get in C when performing addition on two w-bit unsigned values.

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 2.9 indicates, overflow occurs when the two operands sum to 2^w or more. Figure 2.16 shows a plot of the unsigned addition function for word size $w = 4$. The sum is computed modulo $2^4 = 16$. When $x+y < 16$, there is no overflow, and $x +^{\mathrm{u}}_{4} y$ is simply $x + y$. This is shown as the region forming a sloping plane labeled "Normal." When $x + y \geq 16$, the addition overflows, having the effect of decrementing the sum by 16. This is shown as the region forming a sloping plane labeled "Overflow."

When executing C programs, overflows are not signaled as errors. At times, however, we might wish to determine whether overflow has occurred. For example, suppose we compute $s \doteq x +^{\mathrm{u}}_{w} y$, and we wish to determine whether s equals $x + y$. We claim that overflow has occurred if and only if $s < x$ (or equivalently $s < y$.) To see this, 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 $s = x + y - 2^w < x$. In our

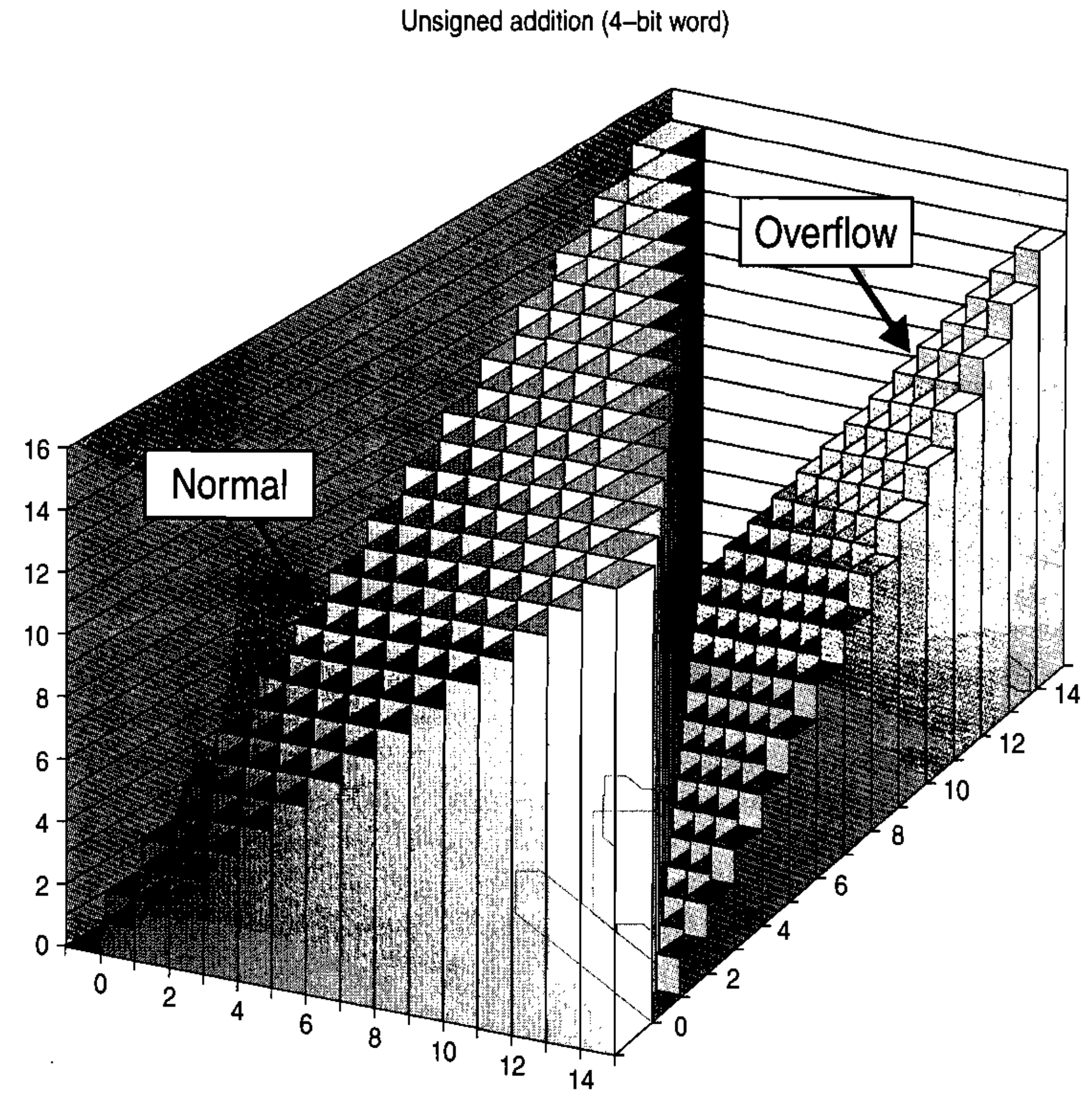


Figure 2.16 Unsigned addition. With a four-bit word size, addition is performed modulo 16.

earlier example, we saw that $9 +^{\text{u}}_{4} 12 = 5$. We can see that overflow occurred, since $5 < 9$.

Modular addition forms a mathematical structure known as an *Abelian group*, named after the Danish 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 $+^{\text{u}}_{w}$. For every value x, there must be some value $-^{\text{u}}_{w} x$ such that $-^{\text{u}}_{w} x +^{\text{u}}_{w} x = 0$. 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 $0 \leq 2^w - x < 2^w$, and $(x + 2^w - x) \bmod 2^w = 2^w \bmod 2^w = 0$. Hence, it is the inverse of x under $+^{\text{u}}_{w}$. These two cases lead to the following equation for $0 \leq x < 2^w$:

$$-^{\text{u}}_{w} x = \begin{cases} x, & x = 0 \\ 2^w - x, & x > 0 \end{cases} \tag{2.10}$$

Practice Problem 2.24

We can represent a bit pattern of length $w = 4$ with a single hex digit. For an unsigned interpretation of these digits, use Equation 2.10 fill in the following table giving the values and the bit representations (in hex) of the unsigned additive inverses of the digits shown.

x		$-^{\mathrm{u}}_{4}\, x$	
Hex	Decimal	Decimal	Hex
0			
3			
8			
A			
F			

2.3.2 Two's-Complement Addition

A similar problem arises for two's-complement addition. 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.

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. Thus, we can define two's-complement addition for word size w, denoted as $+^{\mathrm{t}}_{w}$ on operands x and y such that $-2^{w-1} \leq x, y < 2^{w-1}$ as

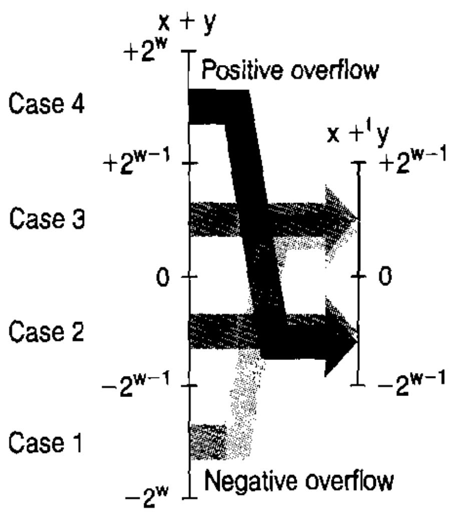
$$x +^{\mathrm{t}}_{w} y \doteq U2T_{w}(T2U_{w}(x) +^{\mathrm{u}}_{w} T2U_{w}(y)) \qquad (2.11)$$

By Equation 2.3 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}.

Figure 2.17 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 $2^{w-1} + 1$, there is a positive overflow.



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 $x +^t_w y$. We can divide the analysis into four cases as illustrated in Figure 2.17:

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 2.6, we see that z' is in the range such that $z'' = z'$. This case is referred to as *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 2.6, 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 case is referred to as *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 $z'' = x + y - 2^w$.

By the preceding analysis, we have shown that when operation $+^t_w$ is applied to values x and y in the range $-2^{w-1} \le x, y \le 2^{w-1} - 1$, we have the following:

$$x +^t_w y = \begin{cases} x + y - 2^w, & 2^{w-1} \le x + y & \text{Positive Overflow} \\ x + y, & -2^{w-1} \le x + y < 2^{w-1} & \text{Normal} \\ x + y + 2^w, & x + y < -2^{w-1} & \text{Negative Overflow} \end{cases} \tag{2.12}$$

x	y	$x+y$	$x +^{t}_{4} y$	Case
−8 [1000]	−5 [1011]	−13	3 [0011]	1
−8 [1000]	−8 [1000]	−16	0 [0000]	1
−8 [1000]	5 [0101]	−3	−3 [1101]	2
2 [0010]	5 [0101]	7	7 [0111]	3
5 [0101]	5 [0101]	10	−6 [1010]	4

Figure 2.18 Two's-complement addition examples. The bit-level representation of the four-bit two's-complement sum can be obtained by performing binary addition of the operands and truncating the result to four bits.

As an illustration, Figure 2.18 shows some examples of four-bit two's-complement addition. Each example is labeled by the case to which it corresponds in the derivation of Equation 2.12. Note that $2^4 = 16$, and thus 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 four bits.

Figure 2.19 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 underflow, 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 2.12 also lets us identify the cases where overflow has occurred. When both x and y are negative, but $x +^{t}_{w} y \geq 0$, we have negative overflow. When both x and y are positive, but $x +^{t}_{w} y < 0$, we have positive overflow.

Practice Problem 2.25

Fill in the table that follows in the style of Figure 2.18. 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 2.12.

x	y	$x+y$	$x +^{\mathrm{t}}_{5} y$	Case
[10000]	[10101]			
[10000]	[10000]			
[11000]	[00111]			
[11110]	[00101]			
[01000]	[01000]			

2.3.3 Two's-Complement Negation

We can see that every number x in the range $-2^{w-1} \leq x < 2^{w-1}$ has an additive inverse under $+^{\mathrm{t}}_{w}$: First, for $x \neq -2^{w-1}$, we can see that its additive inverse is simply $-x$. That is, we have $-2^{w-1} < -x < 2^{w-1}$ and $-x +^{\mathrm{t}}_{w} x = -x + x = 0$. For $x = -2^{w-1} = TMin_w$, on the other hand, $-x = 2^{w-1}$ cannot be represented as a w-bit number. We claim that this special value has itself as the additive inverse

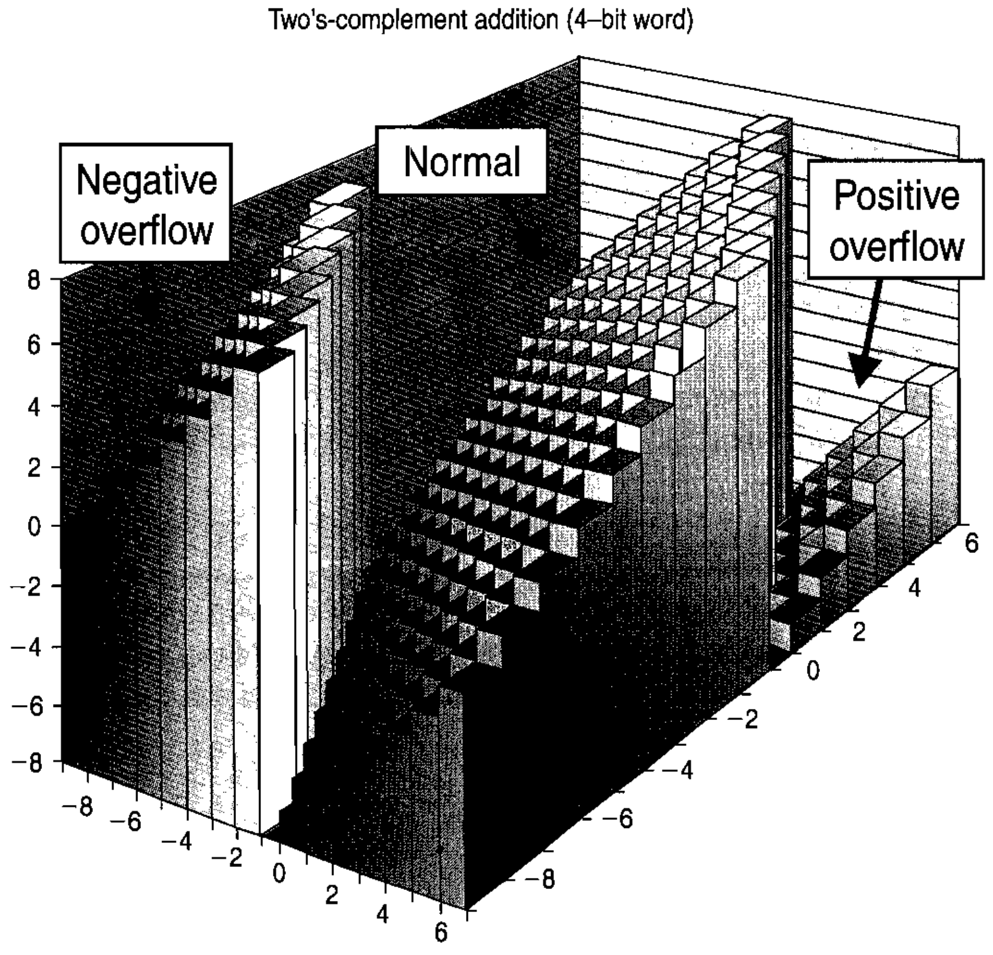


Figure 2.19 Two's-complement addition. With a four-bit word size, addition can have a negative overflow when $x + y < -8$ and a positive overflow when $x + y \geq 8$.

under $+^t_w$. The value of $-2^{w+1} +^t_w -2^{w+1}$ is given by the third case of Equation 2.12, since $-2^{w-1} + -2^{w-1} = -2^w$. This gives $-2^{w+1} +^t_w -2^{w+1} = -2^w + 2^w = 0$. From this analysis, we can define the two's-complement negation operation $-^t_w$ for x in the range $-2^{2-1} \leq x < 2^{w-1}$ as follows:

$$-^t_w x = \begin{cases} -2^{w-1}, & x = -2^{w-1} \\ -x, & x > -2^{w-1} \end{cases} \qquad (2.13)$$

Practice Problem 2.26

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			
3			
8			
A			
F			

What do you observe about the bit patterns generated by two's-complement and unsigned (Problem 2.24) negation?

A well-known technique for performing two's-complement negation at the bit level is to complement the bits and then increment the result. In C, this can be written as `~x + 1`. To justify the correctness of this technique, observe that for any single bit x_i, we have $\tilde{}x_i = 1 - x_i$. Let $\vec{x}$ be a bit vector of length w and $x \doteq B2T_w(\vec{x})$ be the two's-complement number it represents. By Equation 2.2, the complemented bit vector $\tilde{}\vec{x}$ has the following numeric value:

$$\begin{aligned} B2T_w(\tilde{}\vec{x}) &= -(1 - x_{w-1})2^{w-1} + \sum_{i=0}^{w-2}(1 - x_i)2^i \\ &= \left[-2^{w-1} + \sum_{i=0}^{w-2} 2^i\right] - \left[-x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i\right] \\ &= [-2^{w-1} + 2^{w-1} - 1] - B2T_w(\vec{x}) \\ &= -1 - x \end{aligned}$$

The key simplification in the preceding derivation is that $\sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$. It follows that by incrementing $\tilde{}\vec{x}$ we obtain $-x$.

$\vec{x}$		$\sim\vec{x}$		$incr(\sim\vec{x})$	
[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

Figure 2.20 Examples of complementing and incrementing four-bit numbers. The effect is to compute the two's-value negation.

To increment a number x represented at the bit-level as $\vec{x} \doteq [x_{w-1}, x_{w-2}, \ldots, x_0]$, define the operation *incr* as follows: Let k be the position of the rightmost zero, such that $\vec{x}$ is of the form $[x_{w-1}, x_{w-2}, \ldots, x_{k+1}, 0, 1, \ldots, 1]$. We then define $incr(\vec{x})$ to be $[x_{w-1}, x_{w-2}, \ldots, x_{k+1}, 1, 0, \ldots, 0]$. For the special case in which the bit-level representation of x is $[1, 1, \ldots, 1]$, define $incr(\vec{x})$ to be $[0, \ldots, 0]$. To show that $incr(\vec{x})$ yields the bit-level representation of $x +^{\mathrm{t}}_{w} 1$, consider the following cases:

1. When $\vec{x} = [1, 1, \ldots, 1]$, we have $x = -1$. The incremented value $incr(\vec{x}) \doteq [0, \ldots, 0]$ has numeric value 0.
2. When $k = w-1$, i.e., $\vec{x} = [0, 1, \ldots, 1]$, we have $x = TMax_w$. The incremented value $incr(\vec{x}) = [1, 0, \ldots, 0]$ has numeric value $TMin_w$. From Equation 2.12, we can see that $TMax_w +^{\mathrm{t}}_{w} 1$ is one of the positive overflow cases, yielding $TMin_w$.
3. When $k < w - 1$, i.e., $x \neq TMax_w$ and $x \neq -1$, we can see that the low-order $k+1$ bits of $incr(\vec{x})$ has numeric value 2^k, while the low-order $k+1$ bits of $\vec{x}$ has numeric value $\sum_{i=0}^{k-1} 2^i = 2^k - 1$. The high-order $w - k + 1$ bits have matching numeric values. Thus, $incr(\vec{x})$ has numeric value $x + 1$. In addition, for $x \neq TMax_w$, adding 1 to x will not cause an overflow, and hence $x +^{\mathrm{t}}_{w} 1$ has numeric value $x + 1$ as well.

As illustrations, Figure 2.20 shows how complementing and incrementing affect the numeric values of several four-bit vectors.

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. By Equation 2.7, this can be seen to be equivalent to computing the product modulo 2^w. Thus, the effect of the w-bit unsigned multiplication operation $*^{\mathrm{u}}_{w}$ is

$$x *^{\mathrm{u}}_{w} y = (x \cdot y) \bmod 2^w \tag{2.14}$$

It is well known that modular arithmetic forms a ring. We can therefore deduce that unsigned arithmetic over w-bit numbers forms a ring $\langle\{0, \ldots, 2^w - 1\}, +^{\mathrm{u}}_w, *^{\mathrm{u}}_w, -^{\mathrm{u}}_w, 0, 1\rangle$.

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—most cases would fit into $2w - 1$ bits, but the special case of 2^{2w-2} requires the full $2w$ bits (to include a sign bit of 0). Instead, signed multiplication in C generally is performed by truncating the $2w$-bit product to w bits. By Equation 2.8, the effect of the w-bit two's-complement multiplication operation $*^{\mathrm{t}}_w$ is

$$x *^{\mathrm{t}}_w y \quad = \quad U2T_w((x \cdot y) \bmod 2^w) \qquad (2.15)$$

We claim that the bit-level representation of the product operation is identical for both unsigned and two's-complement multiplication. That is, given bit vectors $\vec{x}$ and $\vec{y}$ of length w, the bit-level representation of the unsigned product $B2U_w(\vec{x}) *^{\mathrm{u}}_w B2U_w(\vec{y})$ is identical to the bit-level representation of the two's-complement product $B2T_w(\vec{x}) *^{\mathrm{t}}_w B2T_w(\vec{x})$. This implies that the machine can use a single type of multiply instruction to multiply both signed and unsigned integers.

To see this, let $x = B2T_w(\vec{x})$ and $y = B2T_w(\vec{y})$ be the two's-complement values denoted by these bit patterns, and let $x' = B2U_w(\vec{x})$ and $y' = B2U_w(\vec{y})$ be the unsigned values. From Equation 2.3, 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:

$$\begin{aligned}
(x' \cdot y') \bmod 2^w &= [(x + x_{w-1}2^w) \cdot (y + y_{w-1}2^w)] \bmod 2^w \\
&= [x \cdot y + (x_{w-1}y + y_{w-1}x)2^w + x_{w-1}y_{w-1}2^{2w}] \bmod 2^w \\
&= (x \cdot y) \bmod 2^w \qquad (2.16)
\end{aligned}$$

Thus, the low-order w bits of $x \cdot y$ and $x' \cdot y'$ are identical.

As illustrations, Figure 2.21 shows the results of multiplying different three-bit numbers. For each pair of bit-level operands, we perform both unsigned and two's-complement multiplication. Note that the unsigned truncated product always equals $x \cdot y \bmod 8$, and that the bit-level representations of both truncated products are identical.

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

Figure 2.21 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.

Practice Problem 2.27

Fill in the following table showing the results of multiplying different three-bit numbers, in the style of Figure 2.21:

Mode	x	y	$x \cdot y$	Truncated $x \cdot y$
Unsigned	[110]	[010]		
Two's-Comp.	[110]	[010]		
Unsigned	[001]	[111]		
Two's-Comp.	[001]	[111]		
Unsigned	[111]	[111]		
Two's-Comp.	[111]	[111]		

We can see that unsigned arithmetic and two's-complement arithmetic over w-bit numbers are isomorphic—the operations $+^{\mathrm{u}}_{w}$, $-^{\mathrm{u}}_{w}$, and $*^{\mathrm{u}}_{w}$ have the exact same effect at the bit level as do $+^{\mathrm{t}}_{w}$, $-^{\mathrm{t}}_{w}$, and $*^{\mathrm{t}}_{w}$. From this, we can deduce that two's-complement arithmetic forms a ring $\langle\{-2^{w-1}, \ldots, 2^{w-1}-1\}, +^{\mathrm{t}}_{w}, *^{\mathrm{t}}_{w}, -^{\mathrm{t}}_{w}, 0, 1\rangle$.

2.3.6 Multiplying by Powers of Two

On most machines, the integer multiply instruction is fairly slow, requiring 12 or more clock cycles, whereas other integer operations—such as addition, subtraction, bit-level operations, and shifting—require only one clock cycle. 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.

Let x be the unsigned integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$. Then for any $k \geq 0$, we claim the bit-level representation of $x2^k$ is given by $[x_{w-1}, x_{w-2}, \ldots, x_0, 0, \ldots, 0]$, where k 0s have been added to the right. This property can be derived using Equation 2.1:

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

For $k < w$, we can truncate the shifted bit vector to be of length w, giving $[x_{w-k-1}, x_{w-k-2}, \ldots, x_0, 0, \ldots, 0]$. By Equation 2.7, this bit-vector has numeric value $x2^k \bmod 2^w = x *^{\mathrm{u}}_w 2^k$. Thus, for unsigned variable x, the C expression x << k is equivalent to x * pwr2k, where pwr2k equals 2^{k}. In particular, we can compute pwr2k as 1U << k.

By similar reasoning, we can show that for a two's-complement number x having bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and any k in the range $0 \leq k < w$, bit pattern $[x_{w-k-1}, \ldots, x_0, 0, \ldots, 0]$ will be the two's-complement representation of $x *^{\mathrm{t}}_w 2^k$. Therefore, for signed variable x, the C expression x << k is equivalent to x * pwr2k, where pwr2k equals 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.

Practice Problem 2.28

As we will see in Chapter 3, the leal instruction on an Intel-compatible processor can perform computations of the form a<<k + b, where k is either 0, 1, or 2, 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.

What multiples of a can be computed with this instruction?

2.3.7 Dividing by Powers of Two

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

Integer division always rounds toward zero. For $x \geq 0$ and $y > 0$, the result should be $\lfloor x/y \rfloor$, where for any real number a, $\lfloor a \rfloor$ is defined 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 $\lfloor 3 \rfloor = 3$.

Consider the effect of performing a logical right shift on an unsigned number. Let x be the unsigned integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and 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 x'' be the unsigned number with k-bit representation $[x_{k-1}, \ldots, x_0]$. We claim that $x' = \lfloor x/2^k \rfloor$. To see this, by Equation 2.1, we have $x = \sum_{i=0}^{w-1} x_i 2^i$, $x' = \sum_{i=k}^{w-k-1} x_i 2^{i-k}$ and $x'' = \sum_{i=0}^{k-1} x_i 2^i$. We can therefore write x as $x = 2^k x' + x''$. Observe that $0 \le x'' \le \sum_{i=0}^{k-1} 2^i = 2^k - 1$, and hence $0 \le x'' < 2^k$, implying that $\lfloor x''/2^k \rfloor = 0$. Therefore, $\lfloor x/2^k \rfloor = \lfloor x' + x''/2^k \rfloor = x' + \lfloor x''/2^k \rfloor = x'$.

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

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

This bit vector has numeric value x'. That is, logically right shifting an unsigned number by k is equivalent to dividing it by 2^k. Therefore, for unsigned variable x, the C expression `x >> k` is equivalent to `x / pwr2k`, where pwr2k equals 2^k.

Now consider the effect of performing an arithmetic right shift on a two's-complement number. Let x be the two's-complement integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and k be in the range $0 \le k < w$. Let x' be the two's-complement number represented by the $w-k$ bits $[x_{w-1}, x_{w-2}, \ldots, x_k]$, and 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 $0 \le x'' < 2^k$, 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 $\lfloor x/y \rfloor$.

For $x \ge 0$, our analysis shows that this shifted result is the desired value. For $x < 0$ and $y > 0$, however, the result of integer division should be $\lceil x/y \rceil$, where for any real number a, $\lceil a \rceil$ is defined to be the unique integer a' such that $a' - 1 < a \le a'$. That is, integer division should round negative results upward toward zero. For example, the C expression `-5/2` yields `-2`. Thus, right shifting a negative number by k is not equivalent to dividing it by 2^k when rounding occurs. For example, the four-bit representation of -5 is [1011]. If we shift it right by one arithmetically we get [1101], which is the two's-complement representation of -3.

We can correct for this improper rounding by "biasing" the value before shifting. This technique exploits the property that $\lceil x/y \rceil = \lfloor (x + y - 1)/y \rfloor$ for integers x and y such that $y > 0$. Thus, for $x < 0$, if we first add $2^k - 1$ to x before right shifting, we will get a correctly rounded result. This analysis shows that for a two's-complement machine using arithmetic right shifts, the C expression

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

is equivalent to `x/pwr2k`, where pwr2k equals 2^k. For example, to divide -5 by 2, we first add bias $2 - 1 = 1$ giving bit pattern [1100]. Right shifting this by one arithmetically gives bit pattern [1110], which is the two's-complement representation of -2.

Practice Problem 2.29

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 <<= 4;
  x -= t;
  if (y < 0) y += 3;
  y >>= 2;  /* Arithmetic shift */
  return x+y;
}
```

What are the values of M and N?

Practice Problem 2.30

Assume we are running code on a 32-bit machine using two's-complement arithmetic 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) || ((2*x) < 0)`

B. `(x & 7) != 7 || (x<<30 < 0)`

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

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

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

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

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

2.4 Floating Point

Floating-point representation encodes rational numbers of the form $V = x \times 2^y$. It is useful for performing computations involving very large numbers ($|V| \gg 0$), numbers very close to 0 ($|V| \ll 1$), and more generally as an approximation to real arithmetic.

Up until the 1980s, every computer manufacturer devised its own conventions for how floating-point numbers were represented and the details of the operations performed on them. In addition, they often did not worry too much about the accuracy of the operations, viewing speed and ease of implementation as being more critical than numerical precision.

All of this changed around 1985 with the advent of IEEE Standard 754, a carefully crafted standard for representing floating-point numbers and the operations performed on them. This effort started in 1976 under Intel's sponsorship with the design of the 8087, a chip that provided floating-point support for the 8086 processor. They hired William Kahan, a professor at the University of California, Berkeley, as a consultant to help design a floating-point standard for its future processors. They allowed Kahan to join forces with a committee generating an industry-wide standard under the auspices of the Institute of Electrical and Electronics Engineers (IEEE). The committee ultimately adopted a standard close to the one Kahan had devised for Intel. Nowadays virtually all computers support what has become known as *IEEE floating point*. This has greatly improved the portability of scientific application programs across different machines.

Aside: The IEEE.

The Institute of Electrical and Electronic Engineers (IEEE—pronounced "I-Triple-E") is a professional society that encompasses all of electronic and computer technology. It publishes journals, sponsors conferences, and sets up committees to define standards on topics ranging from power transmission to software engineering.

In this section, we will see how numbers are represented in the IEEE floating-point format. We will also explore issues of *rounding*, when a number cannot be represented exactly in the format and hence must be adjusted upward or downward. We will then explore the mathematical properties of addition, multiplication, and relational operators. Many programmers consider floating point to be at best uninteresting and at worst arcane and incomprehensible. We will see that since the IEEE format is based on a small and consistent set of principles, it is really quite elegant and understandable.

2.4.1 Fractional Binary Numbers

A first step in understanding floating-point numbers is to consider binary numbers having fractional values. Let us first examine the more familiar decimal notation. Decimal notation uses a representation of the form: $d_m d_{m-1} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots d_{-n}$, where each decimal digit d_i ranges between 0 and 9. This notation represents a number d defined as

$$d = \sum_{i=-n}^{m} 10^i \times d_i$$

The weighting of the digits is defined relative to the decimal point symbol '.,' meaning that digits to the left are weighted by positive powers of 10, giving integral values, while digits to the right are weighted by negative powers of 10, giving fractional values. For example, 12.34_{10} represents the number $1 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} = 12\frac{34}{100}$.

By analogy, consider a notation of the form $b_m b_{m-1} \cdots b_1 b_0 . b_{-1} b_{-2} \cdots b_{-n}$, where each binary digit, or bit, b_i ranges between 0 and 1. This notation represents a number b defined as

$$b = \sum_{i=-n}^{m} 2^i \times b_i \qquad (2.17)$$

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

One can readily see from Equation 2.17 that shifting the binary point one position to the left has the effect of dividing the number by two. For example, while 101.11_2 represents the number $5\frac{3}{4}$, 10.111_2 represents the number $2 + 0 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = 2\frac{7}{8}$. Similarly, shifting the binary point one position to the right has the effect of multiplying the number by two. For example, 1011.1_2 represents the number $8 + 0 + 2 + 1 + \frac{1}{2} = 11\frac{1}{2}$.

Note that numbers of the form $0.11 \cdots 1_2$ represent numbers just below 1. For example, 0.111111_2 represents $\frac{63}{64}$. We will use the shorthand notation $1.0 - \epsilon$ to represent such values.

Assuming we consider only finite-length encodings, decimal notation cannot represent numbers such as $\frac{1}{3}$ and $\frac{5}{7}$ exactly. Similarly, fractional binary notation can only represent numbers that can be written $x \times 2^y$. Other values can only be approximated. For example, although the number $\frac{1}{5}$ can be approximated with increasing accuracy by lengthening the binary representation, we cannot represent it exactly as a fractional binary number:

Representation	Value	Decimal
0.0_2	0	0.0_{10}
0.01_2	$\frac{1}{4}$	0.25_{10}
0.010_2	$\frac{2}{8}$	0.25_{10}
0.0011_2	$\frac{3}{16}$	0.1875_{10}
0.00110_2	$\frac{6}{32}$	0.1875_{10}
0.001101_2	$\frac{13}{64}$	0.203125_{10}
0.0011010_2	$\frac{26}{128}$	0.203125_{10}
0.00110011_2	$\frac{51}{256}$	0.19921875_{10}

Practice Problem 2.31

Fill in the missing information in the following table:

Fractional value	Binary representation	Decimal representation
$\frac{1}{4}$	0.01	0.25
$\frac{3}{8}$		
$\frac{23}{16}$		
	10.1101	
	1.011	
		5.625
		3.0625

Practice Problem 2.32

The imprecision of floating-point arithmetic can have disastrous effects. On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks and killed 28 soldiers. The U. S. General Accounting Office (GAO) conducted a detailed analysis of the failure [52] and determined that the underlying cause was an imprecision in a numeric calculation. In this exercise, you will reproduce part of the GAO's analysis.

The Patriot system contains an internal clock, implemented as a counter that is incremented every 0.1 seconds. To determine the time in seconds, the program would multiply the value of this counter by a 24-bit quantity that was a fractional binary approximation to $\frac{1}{10}$. In particular, the binary representation of $\frac{1}{10}$ is the nonterminating sequence

$$0.000110011[0011]\cdots_2$$

where the portion in brackets is repeated indefinitely. The computer approximated 0.1 using just the leading bit plus the first 23 bits of this sequence to the right of the binary point. Let us call this number x.

A. What is the binary representation of $x - 0.1$?

B. What is the approximate decimal value of $x - 0.1$?

C. The clock starts at 0 when the system is first powered up and keeps counting up from there. In this case, the system had been running for around 100 hours. What was the difference between the time computed by the software and the actual time?

D. The system predicts where an incoming missile will appear based on its velocity and the time of the last radar detection. Given that a Scud travels at around 2,000 meters per second, how far off was its prediction?

Normally, a slight error in the absolute time reported by a clock reading would not affect a tracking computation. Instead, it should depend on the relative time between two successive readings. The problem was that the Patriot software had been upgraded to use a more accurate function for reading time, but not all of the function calls had been replaced by the new code. As a result, the tracking software used the accurate time for one reading and the inaccurate time for the other [71].

2.4.2 IEEE Floating-Point Representation

Positional notation such as considered in the previous section would not be efficient for representing very large numbers. For example, the representation of 5×2^{100} would consist of the bit pattern 101 followed by 100 zeros. Instead, we would like to represent numbers in a form $x \times 2^y$ by giving the values of x and y.

The IEEE floating-point standard represents a number in a form $V = (-1)^s \times M \times 2^E$:

- The *sign* s determines whether the number is negative ($s = 1$) or positive ($s = 0$), where the interpretation of the sign bit for numeric value 0 is handled as a special case.
- The *significand* M is a fractional binary number that ranges either between 1 and $2 - \epsilon$ or between 0 and $1 - \epsilon$.
- The *exponent* E weights the value by a (possibly negative) power of 2.

The bit representation of a floating-point number is divided into three fields to encode these values:

- The single sign bit `s` directly encodes the sign s.
- The k-bit exponent field `exp` $= e_{k-1} \cdots e_1 e_0$ encodes the exponent E.
- The n-bit fraction field `frac` $= f_{n-1} \cdots f_1 f_0$ encodes the significand M, but the value encoded also depends on whether or not the exponent field equals 0.

In the single-precision floating-point format (a `float` in C), fields `s`, `exp`, and `frac` are 1, $k = 8$, and $n = 23$ bits each, yielding a 32-bit representation.

In the double-precision floating-point format (a `double` in C), fields `s`, `exp`, and `frac` are 1, $k = 11$, and $n = 52$ bits each, yielding a 64-bit representation.

The value encoded by a given bit representation can be divided into three different cases, depending on the value of `exp`:

Normalized Values

This is the most common case. These kinds occur when the bit pattern of `exp` is neither all 0s (numeric value 0) nor all 1s (numeric value 255 for single precision, 2047 for double). In this case, the exponent field is interpreted as representing a signed integer in *biased* form. That is, the exponent value is $E = e - Bias$ where e is the unsigned number having bit representation $e_{k-1} \cdots e_1 e_0$, and $Bias$ is a bias value equal to $2^{k-1} - 1$ (127 for single precision and 1023 for double). This yields exponent ranges from -126 to $+127$ for single precision and -1022 to $+1023$ for double precision.

The fraction field `frac` is interpreted as representing the fractional value f, where $0 \leq f < 1$, having binary representation $0.f_{n-1} \cdots f_1 f_0$, that is, with the binary point to the left of the most significant bit. The significand is defined to be $M = 1 + f$. This is sometimes called an *implied leading 1* representation, because we can view M to be the number with binary representation $1.f_{n-1} f_{n-2} \cdots f_0$. This representation is a trick for getting an additional bit of precision for free, since we can always adjust the exponent E so that significand M is in the range $1 \leq M < 2$ (assuming there is no overflow). We therefore do not need to explicitly represent the leading bit, since it always equals 1.

Denormalized Values

When the exponent field is all 0s, the represented number is in *denormalized* form. In this case, the exponent value is $E = 1 - Bias$, and the significand value is $M = f$, that is, the value of the fraction field without an implied leading 1.

Aside: Why set the bias this way for denormalized values?

Having the exponent value be $1 - Bias$ rather than simply $-Bias$ might seem counterintuitive. We will see shortly that it provides for smooth transition from denormalized to normalized values.

Denormalized numbers serve two purposes. First, they provide a way to represent numeric value 0, since with a normalized number we must always have $M \geq 1$, and hence we cannot represent 0. In fact the floating-point representation of $+0.0$ has a bit pattern of all 0s: the sign bit is 0, the exponent field is all 0s (indicating a denormalized value), and the fraction field is all 0s, giving $M = f = 0$. Curiously, when the sign bit is 1, but the other fields are all 0s, we get the value -0.0. With IEEE floating-point format, the values -0.0 and $+0.0$ are considered different in some ways and the same in others.

A second function of denormalized numbers is to represent numbers that are very close to 0.0. They provide a property known as *gradual underflow* in which possible numeric values are spaced evenly near 0.0.

Special Values

A final category of values occurs when the exponent field is all 1s. When the fraction field is all 0s, the resulting values represent infinity, either $+\infty$ when $s = 0$, or $-\infty$ when $s = 1$. Infinity can represent results that *overflow*, as when we multiply two very large numbers, or when we divide by zero. When the fraction field is nonzero, the resulting value is called a "*NaN*," short for "Not a Number." Such values are returned as the result of an operation where the result cannot be given as a real number or as infinity, as when computing $\sqrt{-1}$ or $\infty - \infty$. They can also be useful in some applications for representing uninitialized data.

2.4.3 Example Numbers

Figure 2.22 shows the set of values that can be represented in a hypothetical 6-bit format having $k = 3$ exponent bits and $n = 2$ significand bits. The bias is $2^{3-1} - 1 = 3$. Part A of the figure shows all representable values (other than *NaN*). The two infinities are at the extreme ends. The normalized numbers with maximum magnitude are ± 14. The denormalized numbers are clustered around 0. These can be seen more clearly in part B of the figure, where we show just the numbers between -1.0 and $+1.0$. The two zeros are special cases of denormalized numbers. Observe that the representable numbers are not uniformly distributed—they are denser nearer the origin.

Figure 2.23 shows some examples for a hypothetical eight-bit floating-point format having $k = 4$ exponent bits and $n = 3$ fraction bits. The bias is $2^{4-1} - 1 = 7$. The figure is divided into three regions representing the three classes of numbers. Closest to 0 are the denormalized numbers, starting with 0 itself. Denormalized numbers in this format have $E = 1 - 7 = -6$, giving a weight $2^E = \frac{1}{64}$. The fractions f range over the values $0, \frac{1}{8}, \ldots, \frac{7}{8}$, giving numbers V in the range 0 to $\frac{7}{8\times 64} = \frac{7}{512}$.

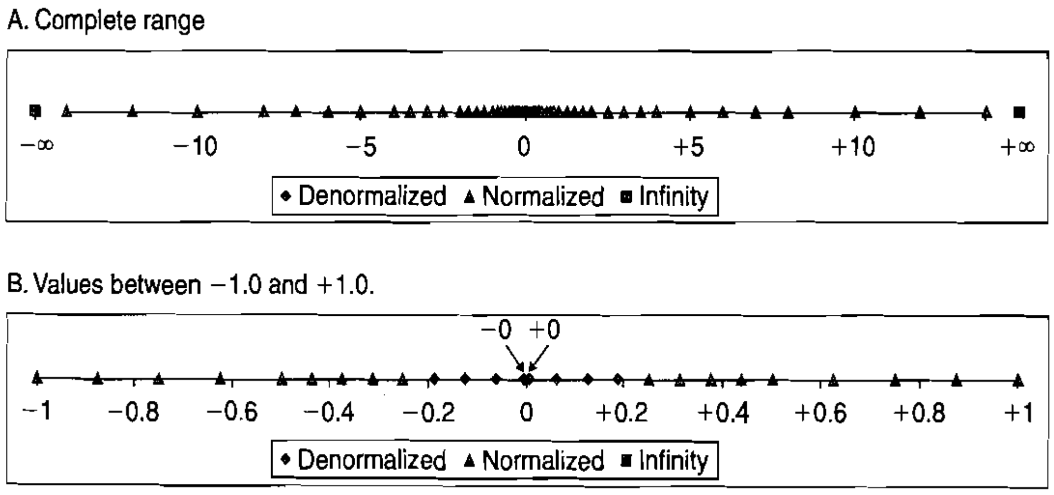


Figure 2.22 Representable values for six-bit floating-point format. There are $k = 3$ exponent bits and $n = 2$ significand bits. The bias is 3.

Description	Bit representation	e	E	f	M	V
Zero	`0 0000 000`	0	−6	0	0	0
Smallest pos.	`0 0000 001`	0	−6	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$
	`0 0000 010`	0	−6	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$
	`0 0000 011`	0	−6	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$
	…					
	`0 0000 110`	0	−6	$\frac{6}{8}$	$\frac{6}{8}$	$\frac{6}{512}$
Largest denorm.	`0 0000 111`	0	−6	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$
Smallest norm.	`0 0001 000`	1	−6	0	$\frac{8}{8}$	$\frac{8}{512}$
	`0 0001 001`	1	−6	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$
	…					
	`0 0110 110`	6	−1	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$
	`0 0110 111`	6	−1	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$
One	`0 0111 000`	7	0	0	$\frac{8}{8}$	1
	`0 0111 001`	7	0	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$
	`0 0111 010`	7	0	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$
	…					
	`0 1110 110`	14	7	$\frac{6}{8}$	$\frac{14}{8}$	224
Largest norm.	`0 1110 111`	14	7	$\frac{7}{8}$	$\frac{15}{8}$	240
Infinity	`0 1111 000`	–	–	–	–	$+\infty$

Figure 2.23 Example nonnegative values for eight-bit floating-point format. There are $k = 4$ exponent bits and $n = 3$ significand bits. The bias is 7.

The smallest normalized numbers in this format also have $E = 1 - 7 = -6$, and the fractions also range over the values $0, \frac{1}{8}, \ldots \frac{7}{8}$. However, the significands then range from $1 + 0 = 1$ to $1 + \frac{7}{8} = \frac{15}{8}$, giving numbers V in the range $\frac{8}{512}$ to $\frac{15}{512}$.

Observe the smooth transition between the largest denormalized number $\frac{7}{512}$ and the smallest normalized number $\frac{8}{512}$. This smoothness is due to our definition of E for denormalized values. By making it $1 - Bias$ rather than $-Bias$, we compensate for the fact that the significand of a denormalized number does not have an implied leading 1.

As we increase the exponent, we get successively larger normalized values, passing through 1.0 and then to the largest normalized number. This number has exponent $E = 7$, giving a weight $2^E = 128$. The fraction equals $\frac{7}{8}$ giving a significand $M = \frac{15}{8}$. Thus, the numeric value is $V = 240$. Going beyond this overflows to $+\infty$.

One interesting property of this representation is that if we interpret the bit representations of the values in Figure 2.23 as unsigned integers, they occur in ascending order, as do the values they represent as floating-point numbers. This is no accident—the IEEE format was designed so that floating-point numbers could be sorted using an integer-sorting routine. A minor difficulty occurs when dealing with negative numbers, since they have a leading 1, and they occur in descending order, but this can be overcome without requiring floating-point operations to perform comparisons (see Problem 2.56).

Practice Problem 2.33

Consider a five-bit floating-point representation based on the IEEE floating-point format, with one sign bit, two exponent bits ($k = 2$), and two fraction bits ($n = 2$). The exponent bias is $2^{2-1} - 1 = 1$.

The table that follows enumerates the entire nonnegative range for this five-bit floating-point representation. Fill in the blank table entries using the following directions:

- e: The value represented by considering the exponent field to be an unsigned integer.
- E: The value of the exponent after biasing.
- f: The value of the fraction.
- M: The value of the significand.
- V: The numeric value represented.

Express the values of f, M and V as fractions of the form $\frac{x}{4}$. You need not fill in entries marked "—".

Bits	e	E	f	M	V
0 00 00					
0 00 01					
0 00 10					
0 00 11					
0 01 00					
0 01 01					
0 01 10					
0 01 11					
0 10 00	2	1	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{8}{4}$
0 10 01					
0 10 10					
0 10 11					
0 11 00	—	—	—	—	$+\infty$
0 11 01	—	—	—	—	*NaN*
0 11 10	—	—	—	—	*NaN*
0 11 11	—	—	—	—	*NaN*

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

Figure 2.24 Examples of nonnegative floating-point numbers.

Figure 2.24 shows the representations and numeric values of some important single and double-precision floating-point numbers. As with the eight-bit format shown in Figure 2.23 we can see some general properties for a floating-point representation with a k-bit exponent and an n-bit fraction:

- The value +0.0 always has a bit representation of all 0s.
- The smallest positive denormalized value has a bit representation consisting of a 1 in the least significant bit position and otherwise all 0s. It has a fraction (and significand) value $M = f = 2^{-n}$ and an exponent value $E = -2^{k-1}+2$. The numeric value is therefore $V = 2^{-n-2^{k-1}+2}$.
- The largest denormalized value has a bit representation consisting of an exponent field of all 0s and a fraction field of all 1s. It has a fraction (and significand) value $M = f = 1 - 2^{-n}$ (which we have written $1 - \epsilon$) and an exponent value $E = -2^{k-1} + 2$. The numeric value is therefore $V = (1-2^{-n}) \times 2^{-2^{k-1}+2}$, which is just slightly smaller than the smallest normalized value.
- The smallest positive normalized value has a bit representation with a 1 in the least significant bit of the exponent field and otherwise all 0s. It has a significand value $M = 1$ and an exponent value $E = -2^{k-1}+2$. The numeric value is therefore $V = 2^{-2^{k-1}+2}$.
- The value 1.0 has a bit representation with all but the most significant bit of the exponent field equal to 1 and all other bits equal to 0. Its significand value is $M = 1$ and its exponent value is $E = 0$.
- The largest normalized value has a bit representation with a sign bit of 0, the least significant bit of the exponent equal to 0, and all other bits equal to 1. It has a fraction value of $f = 1 - 2^{-n}$, giving a significand $M = 2 - 2^{-n}$ (which we have written $2 - \epsilon$). It has an exponent value $E = 2^{k-1} - 1$, giving a numeric value $V = (2 - 2^{-n}) \times 2^{2^{k-1}-1} = (1 - 2^{-n-1}) \times 2^{2^{k-1}}$.

One useful exercise for understanding floating-point representations is to convert sample integer values into floating-point form. For example, we saw in Figure 2.10 that 12,345 has binary representation [11000000111001]. We create a normalized representation of this by shifting 13 positions to the right of a binary point, giving $12345 = 1.1000000111001_2 \times 2^{13}$. To encode this in IEEE single precision format, we construct the fraction field by dropping the leading 1 and adding 10 0s to the end, giving binary representation [10000001110010000000000]. To construct the exponent field, we add bias 127 to 13, giving 140, which has binary representation [10001100]. We combine this with a sign bit of 0 to get the floating-point representation in binary of [01000110010000001110010000000000]. Recall from Section 2.1.4 that we observed the following correlation in the bit-level representations of the integer value 12345 (0x3039) and the single-precision floating-point value 12345.0 (0x4640E400):

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

We can now see that the region of correlation corresponds to the low-order bits of the integer, stopping just before the most significant bit equal to 1 (this bit forms the implied leading 1), matching the high-order bits in the fraction part of the floating-point representation.

Practice Problem 2.34

As mentioned in Practice Problem 2.6, the integer 3490593 has hexadecimal representation 0x354321, while the single-precision, floating-point number 3490593.0 has hexadecimal representation 0x4A550C84. Derive this floating-point representation and explain the correlation between the bits of the integer and floating-point representations.

Practice Problem 2.35

A. For a floating-point format with a k-bit exponent and an n-bit fraction, give a formula for the smallest positive integer that cannot be represented exactly (because it would require an $n+1$-bit fraction to be exact).

B. What is the numeric value of this integer for single-precision format ($k = 8$, $n = 23$)?

2.4.4 Rounding

Floating-point arithmetic can only approximate real arithmetic, since the representation has limited range and precision. Thus, for a value x, we generally want a systematic method of finding the "closest" matching value x' that can be represented in the desired floating-point format. This is the task of the *rounding* operation. The key problem is to define the direction to round a value that is halfway between two possibilities. For example, if I have \$1.50 and want to round it to the nearest dollar, should the result be \$1 or \$2? An alternative approach is to maintain a lower and an upper bound on the actual number. For example, we could determine representable values x^- and x^+ such that the value x is guaran-

Mode	\$1.40	\$1.60	\$1.50	\$2.50	\$–1.50
Round-to-even	\$1	\$2	\$2	\$2	\$–2
Round-toward-zero	\$1	\$1	\$1	\$2	\$–1
Round-down	\$1	\$1	\$1	\$2	\$–2
Round-up	\$2	\$2	\$2	\$3	\$–1

Figure 2.25 Illustration of rounding modes for dollar rounding. The first rounds to a nearest value, while the other three bound the result above or below.

teed to lie between them: $x^- \leq x \leq x^+$. The IEEE floating-point format defines four different *rounding modes*. The default method finds a closest match, while the other three can be used for computing upper and lower bounds.

Figure 2.25 illustrates the four rounding modes applied to the problem of rounding a monetary amount to the nearest whole dollar. Round-to-even (also called round-to-nearest) is the default mode. It attempts to find a closest match. Thus, it rounds \$1.40 to \$1 and \$1.60 to \$2, since these are the closest whole dollar values. The only design decision is to determine the effect of rounding values that are halfway between two possible results. Round-to-even mode adopts the convention that it rounds the number either upward or downward such that the least significant digit of the result is even. Thus, it rounds both \$1.50 and \$2.50 to \$2.

The other three modes produce guaranteed bounds on the actual value. These can be useful in some numerical applications. Round-toward-zero mode rounds positive numbers downward and negative numbers upward, giving a value $\hat{x}$ such that $|\hat{x}| \leq |x|$. Round-down mode rounds both positive and negative numbers downward, giving a value x^- such that $x^- \leq x$. Round-up mode rounds both positive and negative numbers upward, giving a value x^+ such that $x \leq x^+$.

Round-to-even at first seems like it has a rather arbitrary goal—why is there any reason to prefer even numbers? Why not consistently round values halfway between two representable values upward? The problem with such a convention is that one can easily imagine scenarios in which rounding a set of data values would then introduce a statistical bias into the computation of an average of the values. The average of a set of numbers that we rounded by this means would be slightly higher than the average of the numbers themselves. Conversely, if we always rounded numbers halfway between downward, the average of a set of rounded numbers would be slightly lower than the average of the numbers themselves. Rounding toward even numbers avoids this statistical bias in most real-life situations. It will round upward about 50% of the time and round downward about 50% of the time.

Round-to-even rounding can be applied even when we are not rounding to a whole number. We simply consider whether the least significant digit is even or odd. For example, suppose we want to round decimal numbers to the nearest hundredth. We would round 1.2349999 to 1.23 and 1.2350001 to 1.24, regardless of rounding mode, since they are not halfway between 1.23 and 1.24. On the other hand, we would round both 1.2350000 and 1.2450000 to 1.24, since four is even.

Similarly, round-to-even rounding can be applied to binary fractional numbers. We consider least significant bit value 0 to be even and 1 to be odd. In general, the rounding mode is only significant when we have a bit pattern of the form $XX \cdots X.YY \cdots Y100 \cdots$, where X and Y denote arbitrary bit values with the rightmost Y being the position to which we wish to round. Only bit patterns of this form denote values that are halfway between two possible results. As examples, consider the problem of rounding values to the nearest quarter (i.e., 2 bits to the right of the binary point). We would round 10.00011_2 ($2\frac{3}{32}$) down to 10.00_2 (2), and 10.00110_2 ($2\frac{3}{16}$) up to 10.01_2 ($2\frac{1}{4}$), because these values are not halfway between two possible values. We would round 10.11100_2 ($2\frac{7}{8}$) up to 11.00_2 (3) and 10.10100_2 down to 10.10_2 ($2\frac{1}{2}$), since these values are halfway between two possible results, and we prefer to have the least significant bit equal to zero.

2.4.5 Floating-Point Operations

The IEEE standard specifies a simple rule for determining the result of an arithmetic operation such as addition or multiplication. Viewing floating-point values x and y as real numbers, and some operation $\odot$ defined over real numbers, the computation should yield $Round(x \odot y)$, the result of applying rounding to the exact result of the real operation. In practice, there are clever tricks floating-point unit designers use to avoid performing this exact computation, since the computation need only be sufficiently precise to guarantee a correctly rounded result. When one of the arguments is a special value such as -0, ∞, or NaN, the standard specifies conventions that attempt to be reasonable. For example $1/-0$ is defined to yield $-\infty$, while $1/+0$ is defined to yield $+\infty$.

One strength of the IEEE standard's method of specifying the behavior of floating-point operations is that it is independent of any particular hardware or software realization. Thus, we can examine its abstract mathematical properties without considering how it is actually implemented.

We saw earlier that integer addition, both unsigned and two's*complement*, forms an Abelian group. Addition over real numbers also forms an Abelian group, but we must consider what effect rounding has on these properties. Let us define $x +^f y$ to be $Round(x + y)$. This operation is defined for all values of x and y, although it may yield infinity even when both x and y are real numbers due to overflow. The operation is commutative, with $x +^f y = y +^f x$ for all values of x and y. On the other hand, the operation is not associative. For example, with single-precision floating point the expression `(3.14+1e10)-1e10` would evaluate to `0.0`—the value 3.14 would be lost due to rounding. On the other hand, the expression `3.14+(1e10-1e10)` would evaluate to `3.14`. As with an Abelian group, most values have inverses under floating-point addition, that is, $x +^f -x = 0$. The exceptions are infinities (since $+\infty - \infty = NaN$), and NaN's, since $NaN +^f x = NaN$ for any x.

The lack of associativity in floating-point addition is the most important group property that is lacking. It has important implications for scientific programmers and compiler writers. For example, suppose a compiler is given the following code fragment:

```
x = a + b + c;
y = b + c + d;
```

The compiler might be tempted to save one floating-point addition by generating the following code:

```
t = b + c;
x = a + t;
y = t + d;
```

However, this computation might yield a different value for `x` than would the original, since it uses a different association of the addition operations. In most applications, the difference would be so small as to be inconsequential. Unfortunately, compilers have no way of knowing what trade-offs the user is willing to make between efficiency and faithfulness to the exact behavior of the original program. As a result, they tend to be very conservative, avoiding any optimizations that could have even the slightest effect on functionality.

On the other hand, floating-point addition satisfies the following monotonicity property: if $a \geq b$ then $x + a \geq x + b$ for any values of a, b, and x other than NaN. This property of real (and integer) addition is not obeyed by unsigned or two's-complement addition.

Floating-point multiplication also obeys many of the properties one normally associates with multiplication, namely those of a ring. Let us define $x *^{\mathrm{f}} y$ to be $Round(x \times y)$. This operation is closed under multiplication (although possibly yielding infinity or NaN), it is commutative, and it has 1.0 as a multiplicative identity. On the other hand, it is not associative due to the possibility of overflow or the loss of precision due to rounding. For example, with single-precision floating point, the expression `(1e20*1e20)*1e-20` will evaluate to $+\infty$, while `1e20*(1e20*1e-20)` will evaluate to `1e20`. In addition, floating-point multiplication does not distribute over addition. For example, with single-precision floating point, the expression `1e20*(1e20-1e20)` will evaluate to `0.0`, while `1e20*1e20-1e20*1e20` will evaluate to `NaN`.

On the other hand, floating-point multiplication satisfies the following monotonicity properties for any values of a, b, and c other than NaN:

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

In addition, we are also guaranteed that $a *^{\mathrm{f}} a \geq 0$, as long as $a \neq NaN$. As we saw earlier, none of these monotonicity properties hold for unsigned or two's-complement multiplication.

This lack of associativity and distributivity is of serious concern to scientific programmers and to compiler writers. Even such a seemingly simple task as writing code to determine whether two lines intersect in three-dimensional space can be a major challenge.

2.4.6 Floating Point in C

C provides two different floating-point data types: `float` and `double`. On machines that support IEEE floating point, these data types correspond to single-

and double-precision floating point. In addition, the machines use the round-to-even rounding mode. Unfortunately, since the C standard does require the machine use IEEE floating point, there are no standard methods to change the rounding mode or to get special values such as -0, $+\infty$, $-\infty$, or *NaN*. Most systems provide a combination of include ('`.h`') files and procedure libraries to provide access to these features, but the details vary from one system to another. For example, the GNU compiler GCC defines macros `INFINITY` (for $+\infty$) and `NAN` (for *NaN*) when the following sequence occurs in the program file:

```
#define _GNU_SOURCE 1
#include <math.h>
```

Practice Problem 2.36

Fill in the following macro definitions to generate the double-precision values $+\infty$, $-\infty$, and 0.

```
#define POS_INFINITY
#define NEG_INFINITY
#define NEG_ZERO
#endif
```

You cannot use any include files (such as `math.h`), but you can make use of the fact that the largest finite number that can be represented with double precision is around 1.8×10^{308}.

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

- From `int` to `float`, the number cannot overflow, but it may be rounded.
- From `int` or `float` to `double`, the exact numeric value can be preserved because `double` has both greater range (i.e., the range of representable values), as well as greater precision (i.e., the number of significant bits).
- From `double` to `float`, the value can overflow to $+\infty$ or $-\infty$, since the range is smaller. Otherwise, it may be rounded, because the precision is smaller.
- From `float` or `double` to `int` the value will be truncated toward zero. For example, 1.999 will be converted to 1, while -1.999 will be converted to -1. Note that this behavior is very different from rounding. Furthermore, the value may overflow. The C standard does not specify a fixed result for this case, but on most machines the result will either be $TMax_w$ or $TMin_w$, where w is the number of bits in an `int`.

*Intel IA32 Floating-Point Arithmetic

In the next chapter, we will begin an in-depth study of Intel IA32 processors, the processor found in most of today's personal computers. Here we highlight an idiosyncrasy of these machines that can seriously affect the behavior of programs operating on floating-point numbers when compiled with GCC.

IA32 processors, like most other processors, have special memory elements called *registers* for holding floating-point values as they are being computed and used. Values held in registers can be read and written more quickly than those held in the main memory. The unusual feature of IA32 is that the floating-point registers use a special 80-bit *extended-precision* format to provide a greater range and precision than the normal 32-bit single-precision and 64-bit double-precision formats used for values held in memory. As described in Homework Problem 2.58, the extended-precision representation is similar to an IEEE floating-point format with a 15-bit exponent (i.e., $k = 15$) and a 63-bit fraction (i.e., $n = 63$). All single and double-precision numbers are converted to this format as they are loaded from memory into floating-point registers. The arithmetic is always performed in extended precision. Numbers are converted from extended precision to single or double-precision format as they are stored in memory.

This extension to 80 bits for all register data and then contraction to a smaller format for all memory data has some undesirable consequences for programmers. It means that storing a value in memory and then retrieving it can change its value, due to rounding, underflow, or overflow. This storing and retrieving is not always visible to the C programmer, leading to some very peculiar results.

The following example illustrates this property:

code/data/fcomp.c

```c
double recip(int denom)
{
  return 1.0/(double) denom;
}

void do_nothing() {} /* Just like the name says */

void test1(int denom)
{
  double r1, r2;
  int t1, t2;

  r1 = recip(denom);  /* Stored in memory */
  r2 = recip(denom);  /* Stored in register */
  t1 = r1 == r2;      /* Compares register to memory */
  do_nothing();       /* Forces register save to memory */
  t2 = r1 == r2;      /* Compares memory to memory */
  printf("test1 t1: r1 %f %c= r2 %f\n", r1, t1 ? '=' : '!', r2);
  printf("test1 t2: r1 %f %c= r2 %f\n", r1, t2 ? '=' : '!', r2);
}
```

code/data/fcomp.c

Variables r1 and r2 are computed by the same function with the same argument. One would expect them to be identical. Furthermore, both variables t1 and t2 are computing by evaluating the expression r1 == r2, and so we would expect them both to equal 1. There are no apparent hidden side effects—function recip does a straightforward reciprocal computation, and, as the name suggests, function do_nothing does nothing. When the file is compiled with optimization flag '-O2' and run with argument 10, however, we get the following result:

```
test1 t1: r1 0.100000 != r2 0.100000
test1 t2: r1 0.100000 == r2 0.100000
```

The first test indicates the two reciprocals are different, while the second indicates they are the same! This is certainly not what we expect, nor what we want. Understanding all of the details of this example requires studying the machine-level floating-point code generated by GCC (see Section 3.14), but the comments in the code provide a clue as to why this outcome occurs. The value computed by function recip returns its result in a floating-point register. Whenever procedure test1 calls some function, it must store any value currently in a floating-point register onto the main program stack, where local variables for a function are stored. In performing this store, the processor converts the extended-precision register values to double-precision memory values. Thus, before making the second call to recip (line 14), variable r1 is converted and stored as a double-precision number. After the second call, variable r2 has the extended-precision value returned by the function. In computing t1 (line 15), the double-precision number r1 is compared to the extended-precision number r2. Since 0.1 cannot be represented exactly in either format, the outcome of the test is false. Before calling function do_nothing (line 16), r2 is converted and stored as a double-precision number. In computing t2 (line 17), two double-precision numbers are compared, yielding true.

This example demonstrates a deficiency of GCC on IA32 machines (the same result occurs for both Linux and Microsoft Windows). The value associated with a variable changes due to operations that are not visible to the programmer, such as the saving and restoring of floating-point registers. Our experiments with the Microsoft Visual C++ compiler indicate that it does not have this problem.

Aside: Why should we be concerned about these inconsistencies?

As we will discuss in Chapter 5, one of the fundamental principles of optimizing compilers is that programs should produce the exact same results whether or not optimization is enabled. Unfortunately, GCC does not satisfy this requirement for floating-point code on IA32 machines.

There are several ways to overcome this problem, although none are ideal. The simplest is to invoke GCC with the command-line option "-ffloat-store" indicating that the result of every floating-point computation should be stored to memory and read back before using, rather than simply held in a register. This will force every computed value to be converted to the lower-precision form. This slows down the program somewhat but makes the behavior more predictable. Unfortunately, we have found that GCC does not follow this write-then-read convention strictly, even when given the command-line option. For example, consider the following function:

code/data/fcomp.c

```
void test2(int denom)
{
  double r1;
  int t1;
  r1 = recip(denom);          /* Default: register, Forced store: memory */
  t1 = r1 == 1.0/(double) denom; /* Compares register or memory to register */
  printf("test2 t1: r1 %f %c= 1.0/10.0\n", r1, t1 ? '=' : '!');
}
```

code/data/fcomp.c

When compiled with just the "`-O2`" option, `t1` gets value 1—the comparison is made between two register values. When compiled with the "`-ffloat-store`" flag, `t1` gets value 0! Although the result of the call to `recip` is written to memory and read back into a register, the computed value `1.0/(double) denom` is kept in a register. Overall, we have found that seemingly minor changes in a program can cause these tests to succeed or fail in unpredictable ways.

As an alternative, we can have GCC use extended precision in all of its computations by declaring all of the variables to be `long double` as shown in the following code:

code/data/fcomp.c

```
long double recip_l(int denom)
{
  return 1.0/(long double) denom;
}

void test3(int denom)
{
  long double r1, r2;
  int t1, t2, t3;

  r1 = recip_l(denom); /* Stored in memory */
  r2 = recip_l(denom); /* Stored in register */
  t1 = r1 == r2;       /* Compares register to memory */
  do_nothing();        /* Forces register save to memory */
  t2 = r1 == r2;       /* Compares memory to memory */
  t3 = r1 == 1.0/(long double) denom; /* Compare memory to register */
  printf("test3 t1: r1 %f %c= r2 %f\n",
         (double) r1, t1 ? '=' : '!', (double) r2);
  printf("test3 t2: r1 %f %c= r2 %f\n",
         (double) r1, t2 ? '=' : '!', (double) r2);
  printf("test3 t3: r1 %f %c= 1.0/10.0\n",
         (double) r1, t2 ? '=' : '!');
}
```

code/data/fcomp.c

The declaration `long double` is allowed as part of the ANSI C standard, although for most machines and compilers, this declaration is equivalent to an ordinary `double`. For GCC on IA32 machines, however, it uses the extended-precision format for memory data as well as for floating point register data. This allows us to take full advantage of the wider range and greater precision provided by the extended-precision format while avoiding the anomalies we have seen in our earlier examples. Unfortunately, this solution comes at a price. GCC uses 12 bytes to store a long double, increasing memory consumption by 50%. (Although 10 bytes would suffice, it rounds this up to 12 to give a better memory performance. The same allocation is used on both Linux and Windows machines.) Transferring these longer data between registers and memory takes more time, too. Still, this is the best option for programs that want to get the most accurate and predictable results.

Aside: Ariane 5: the high cost of floating-point overflow.

Converting large floating-point numbers to integers is a common source of programming errors. Such an error had disastrous consequences for the maiden voyage of the Ariane 5 rocket, on June 4, 1996. Just 37 seconds after liftoff, the rocket veered off its flight path, broke up, and exploded. Communication satellites valued at $500 million were on board the rocket.

A later investigation [49] showed that the computer controlling the inertial navigation system had sent invalid data to the computer controlling the engine nozzles. Instead of sending flight control information, it had sent a diagnostic bit pattern indicating that an overflow had occurred during the conversion of a 64-bit floating-point number to a 16-bit signed integer.

The value that overflowed measured the horizontal velocity of the rocket, which could be more than five times higher than that achieved by the earlier Ariane 4 rocket. In the design of the Ariane 4 software, they had carefully analyzed the numeric values and determined that the horizontal velocity would never overflow a 16-bit number. Unfortunately, they simply reused this part of the software in the Ariane 5 without checking the assumptions on which it had been based.

Practice Problem 2.37

Assume variables `x`, `f`, and `d` are of type `int`, `float`, and `double`, respectively. Their values are arbitrary, except that neither `f` nor `d` equals $+\infty$, $-\infty$, or *NaN*. For each of the following C expressions, either argue that it will always be true (i.e., evaluate to 1) or give a value for the variables such that it is not true (i.e., evaluates to 0).

A. `x == (int)(float) x`

B. `x == (int)(double) x`

C. `f == (float)(double) f`

D. `d == (float) d`

E. `f == -(-f)`

F. `2/3 == 2/3.0`

G. `(d >= 0.0) || ((d*2) < 0.0)`

H. `(d+f)-d == f`

2.5 Summary

Computers encode information as bits, generally organized as sequences of bytes. Different encodings are used for representing integers, real numbers, and character strings. Different models of computers use different conventions for encoding numbers and for ordering the bytes within multibyte data.

The C language is designed to accommodate a wide range of different implementations in terms of word sizes and numeric encodings. Most current machines have 32-bit word sizes, although high-end machines increasingly have 64-bit words. Most machines use two's-complement encoding of integers and IEEE encoding of floating point. Understanding these encodings at the bit level, as well as understanding the mathematical characteristics of the arithmetic operations, is important for writing programs that operate correctly over the full range of numeric values.

The C standard dictates that when casting between signed and unsigned integers, the underlying bit pattern should not change. On a two's-complement machine, this behavior is characterized by functions $T2U_w$ and $U2T_w$, for a w-bit value. The implicit casting of C gives results that many programmers do not anticipate, often leading to program bugs.

Due to the finite lengths of the encodings, computer arithmetic has properties quite different from conventional integer and real arithmetic. The finite length can cause numbers to overflow, when they exceed the range of the representation. Floating-point values can also underflow, when they are so close to 0.0 that they are changed to zero.

The finite integer arithmetic implemented by C, as well as most other programming languages, has some peculiar properties compared to true integer arithmetic. For example, the expression `x*x` can evaluate to a negative number due to overflow. Nonetheless, both unsigned and two's-complement arithmetic satisfies the properties of a ring. This allows compilers to do many optimizations. For example, in replacing the expression `7*x` by `(x<<3)-x`, we make use of the associative, commutative and distributive properties, along with the relationship between shifting and multiplying by powers of two.

We have seen several clever ways to exploit combinations of bit-level operations and arithmetic operations. For example, we saw that with two's-complement arithmetic, `~x+1` is equivalent to `-x`. As another example, suppose we want a bit pattern of the form $[0, \ldots, 0, 1, \ldots, 1]$, consisting of $w - k$ 0s followed by k 1s. Such bit patterns are useful for masking operations. This pattern can be generated by the C expression `(1<<k)-1`, exploiting the property that the desired bit pattern has numeric value $2^k - 1$. For example, the expression `(1<<8)-1` will generate the bit pattern `0xFF`.

Floating-point representations approximate real numbers by encoding numbers of the form $x \times 2^y$. The most common floating-point representation was defined by IEEE Standard 754. It provides for several different precisions, with the most common being single (32 bits) and double (64 bits). IEEE floating point also has representations for special values ∞ and not-a-number.

Floating-point arithmetic must be used very carefully because it has only limited range and precision, and because it does not obey common mathematical properties such as associativity.

Bibliographic Notes

Reference books on C [40, 32] discuss properties of the different data types and operations. The C standard does not specify details such as precise word sizes or numeric encodings. Such details are intentionally omitted to make it possible to implement C on a wide range of different machines. Several books have been written giving advice to C programmers [41, 50] that warn about problems with overflow, implicit casting to unsigned, and some of the other pitfalls we have covered in this chapter. These books also provide helpful advice on variable naming, coding styles, and code testing. Books on Java (we recommend the one coauthored by James Gosling, the creator of the language [1]) describe the data formats and arithmetic operations supported by Java.

Most books on logic design [86, 39] have a section on encodings and arithmetic operations. Such books describe different ways of implementing arithmetic circuits. Overton's book on IEEE floating point [56] provides a detailed description of the format as well as the properties from the perspective of a numerical applications programmer.

Homework Problems

♦	= *quick problem to try out the idea*
♦♦	= *5–15 minutes to complete and may involve writing/running programs*
♦♦♦	= *sustained problems that may require hours to complete*
♦♦♦♦	= *laboratory assignment that may take one or two weeks to complete*

2.38 ♦

Compile and run the sample code that uses `show_bytes` (file `show-bytes.c`) on different machines to which you have access. Determine the byte orderings used by these machines.

2.39 ◆
Try running the code for `show_bytes` for different sample values.

2.40 ◆
Write procedures `show_short`, `show_long`, and `show_double` that print the byte representations of C objects of types `short int`, `long int`, and `double`, respectively. Try these out on several machines.

2.41 ◆◆
Write a procedure `is_little_endian` that will return 1 when compiled and run on a little-endian machine, and will return 0 when compiled and run on a big-endian machine. This program should run on any machine, regardless of its word size.

2.42 ◆◆
Write a C expression that will yield a word consisting of the least significant byte of x, and the remaining bytes of y. For operands x = `0x89ABCDEF` and y = `0x76543210`, this would give `0x765432EF`.

2.43 ◆◆
Using only bit-level and logical operations, write C expressions that yield 1 for the described condition and 0 otherwise. Your code should work on a machine with any word size. Assume x is an integer.

A. Any bit of x equals 1.

B. Any bit of x equals 0.

C. Any bit in the least significant byte of x equals 1.

D. Any bit in the least significant byte of x equals 0.

2.44 ◆◆◆
Write a function `int_shifts_are_arithmetic()` that yields 1 when run on a machine that uses arithmetic right shifts for `int`'s and 0 otherwise. Your code should work on a machine with any word size. Test your code on several machines. Write and test a procedure `unsigned_shifts_are_arithmetic()` that determines the form of shifts used for `unsigned int`'s.

2.45 ◆◆
You are given the task of writing a procedure `int_size_is_32()` that yields 1 when run on a machine for which an `int` is 32 bits, and yields 0 otherwise. Here is a first attempt:

```
/* The following code does not run properly on some machines */
int bad_int_size_is_32()
{
    /* Set most significant bit (msb) of 32-bit machine */
    int set_msb = 1 << 31;
    /* Shift past msb of 32-bit word */
    int beyond_msb = 1 << 32;

    /* set_msb is nonzero when word size >= 32
       beyond_msb is zero when word size <= 32 */
    return set_msb && !beyond_msb;
}
```

When compiled and run on a 32-bit SUN SPARC, however, this procedure returns 0. The following compiler message gives us an indication of the problem:

```
warning: left shift count >= width of type
```

A. In what way does our code fail to comply with the C standard?

B. Modify the code to run properly on any machine for which `int`'s are at least 32 bits.

C. Modify the code to run properly on any machine for which `int`'s are at least 16 bits.

2.46 ♦

You just started working for a company that is implementing a set of procedures to operate on a data structure where four signed bytes are packed into a 32-bit `unsigned`. Bytes within the word are numbered from 0 (least significant) to 3 (most significant). You have been assigned the task of implementing a function for a machine using two's-complement arithmetic and arithmetic right shifts with the following prototype:

```
/* Declaration of data type where 4 bytes are packed
   into an unsigned */
typedef unsigned packed_t;

/* Extract byte from word.  Return as signed integer */
int xbyte(packed_t word, int bytenum);
```

That is, the function will extract the designated byte and sign extend it to be a 32-bit `int`.

Your predecessor (who was fired for his incompetence) wrote the following code:

```
/* Failed attempt at xbyte */
int xbyte(packed_t word, int bytenum)
{
  return
    (word >> (bytenum << 3)) & 0xFF;
}
```

A. What is wrong with this code?

B. Give a correct implementation of the function that uses only left and right shifts, along with one subtraction.

2.47 ◆

Fill in the following table showing the effects of complementing and incrementing several five-bit vectors in the style of Figure 2.20. Show both the bit vectors and the numeric values.

$\vec{x}$	~$\vec{x}$	*incr*(~$\vec{x}$)
[01101]		
[01111]		
[11000]		
[11111]		
[10000]		

2.48 ◆◆

Show that first decrementing and then complementing is equivalent to complementing and then incrementing. That is, for any signed value x, the C expressions `-x`, `~x+1`, and `~(x-1)` yield identical results. What mathematical properties of two's-complement addition does your derivation rely on?

2.49 ◆◆◆

Suppose we want to compute the complete $2w$-bit representation of $x \cdot y$, where both x and y are unsigned, on a machine for which data type `unsigned` is w bits. The low-order w bits of the product can be computed with the expression `x*y`, so we only require a procedure with prototype

```
unsigned int unsigned_high_prod(unsigned x, unsigned y);
```

that computes the high-order w bits of $x \cdot y$ for unsigned variables.

We have access to a library function with prototype:

```
int signed_high_prod(int x, int y);
```

that computes the high-order w bits of $x \cdot y$ for the case where x and y are in two's-complement form. Write code calling this procedure to implement the function for unsigned arguments. Justify the correctness of your solution.

Hint: Look at the relationship between the signed product $x \cdot y$ and the unsigned product $x' \cdot y'$ in the derivation of Equation 2.16.

2.50 ◆◆

Suppose we are given the task of generating code to multiply integer variable x by various different constant factors K. To be efficient we want to use only the operations +, -, and <<. For the following values of K, write C expressions to perform the multiplication using at most three operations per expression.

A. $K = 5$:

B. $K = 9$:

C. $K = 14$:

D. $K = -56$:

2.51 ◆◆

Write C expressions to generate the bit patterns that follow, where a^k represents k repetitions of symbol a. Assume a w-bit data type. Your code may contain references to parameters `j` and `k`, representing the values of j and k, but not a parameter representing w.

A. $1^{w-k}0^k$.

B. $0^{w-k-j}1^k0^j$.

2.52 ◆◆

Suppose we number the bytes in a w-bit word from 0 (least significant) to $w/8-1$ (most significant). Write code for the following C function, which will return an unsigned value in which byte `i` of argument `x` has been replaced by byte `b`:

```
unsigned replace_byte (unsigned x, int i, unsigned char b);
```

Here are some examples showing how the function should work:

```
replace_byte(0x12345678, 2, 0xAB) --> 0x12AB5678
replace_byte(0x12345678, 0, 0xAB) --> 0x123456AB
```

2.53 ◆◆◆

Fill in code for the following C functions. Function `srl` performs a logical right shift using an arithmetic right shift (given by value `xsra`), followed by other operations not including right shifts or division. Function `sra` performs an arithmetic right shift using a logical right shift (given by value `xsrl`), followed by other operations not including right shifts or division. You may assume that `int`'s are 32-bits long. The shift amount `k` can range from 0 to 31.

```
unsigned srl(unsigned x, int k)
{
    /* Perform shift arithmetically */
    unsigned xsra = (int) x >> k;

    /* ... */

}
```

```
int sra(int x, int k)
{
    /* Perform shift logically */
    int xsrl = (unsigned) x >> k;

    /* ... */

}
```

2.54 ◆

We are running programs on a machine where values of type `int` are 32 bits. They are represented in two's-complement, and they are right shifted arithmetically. Values of type `unsigned` are also 32 bits.

We generate arbitrary values x and y, and convert them to other unsigned as follows:

```
/* Create some arbitrary values */
int x = random();
int y = random();
/* Convert to unsigned */
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
```

For each of the following C expressions, you are to indicate whether or not the expression *always* yields 1. If it always yields 1, describe the underlying mathematical principles. Otherwise, give an example of arguments that make it yield 0.

A. `(x<y) == (-x>-y)`.

B. `((x+y)<<4) + y-x == 17*y+15*x`.

C. `~x+~y == ~(x+y)`.

D. `(int) (ux-uy) == -(y-x)`.

E. `((x >> 1) << 1) <= x`.

2.55 ◆◆

Consider numbers having a binary representation consisting of an infinite string of the form $0.y\,y\,y\,y\,y\,y\cdots$, where y is a k-bit sequence. For example, the binary representation of $\frac{1}{3}$ is $0.01010101\cdots$ ($y = 01$), while the representation of $\frac{1}{5}$ is $0.001100110011\cdots$ ($y = 0011$).

A. Let $Y = B2U_k(y)$, that is, the number having binary representation y. Give a formula in terms of Y and k for the value represented by the infinite string. **Hint:** Consider the effect of shifting the binary point k positions to the right.

B. What is the numeric value of the string for the following values of y?

(a) 001

(b) 1001

(c) 000111

2.56 ◆

Fill in the return value for the following procedure that tests whether its first argument is greater than or equal to its second. Assume the function `f2u` returns an unsigned 32-bit number having the same bit representation as its floating-point argument. You can assume that neither argument is *NaN*. The two flavors of zero: +0 and −0 are considered equal.

```
int float_ge(float x, float y)
{
    unsigned ux = f2u(x);
    unsigned uy = f2u(y);

    /* Get the sign bits */
    unsigned sx = ux >> 31;
    unsigned sy = uy >> 31;

    /* Give an expression using only ux, uy, sx, and sy */
    return /* ... */ ;
}
```

2.57 ◆

Given a floating-point format with a k-bit exponent and an n-bit fraction, write formulas for the exponent E, significand M, the fraction f, and the value V for the quantities that follow. In addition, describe the bit representation.

A. The number 5.0.

B. The largest odd integer that can be represented exactly.

C. The reciprocal of the smallest positive normalized value.

2.58 ◆

Intel-compatible processors also support an "extended precision" floating-point format with an 80-bit word divided into a sign bit, $k = 15$ exponent bits, a single *integer* bit, and $n = 63$ fraction bits. The integer bit is an explicit copy of the implied bit in the IEEE floating-point representation. That is, it equals 1 for normalized values and 0 for denormalized values. Fill in the following table giving the approximate values of some "interesting" numbers in this format:

Description	Extended precision	
	Value	Decimal
Smallest denormalized		
Smallest normalized		
Largest normalized		

2.59 ◆

Consider a 16-bit floating-point representation based on the IEEE floating-point format, with one sign bit, seven exponent bits ($k = 7$), and eight fraction bits ($n = 8$). The exponent bias is $2^{7-1} - 1 = 63$.

Fill in the table that follows for each of the numbers given, with the following instructions for each column:

Hex: The four hexadecimal digits describing the encoded form.

M: The value of the significand. This should be a number of the form x or $\frac{x}{y}$, where x is an integer, and y is an integral power of 2. Examples include: 0, $\frac{67}{64}$, and $\frac{1}{256}$.

E: The integer value of the exponent.

V: The numeric value represented. Use the notation x or $x \times 2^z$, where x and z are integers.

As an example, to represent the number $\frac{7}{2}$, we would have $s = 0$, $M = \frac{7}{4}$, and $E = 1$. Our number would therefore have an exponent field of `0x40` (decimal value $63 + 1 = 64$) and a significand field `0xC0` (binary 11000000_2), giving a hex representation `40C0`.

You need not fill in entries marked "—".

Description	Hex	M	E	V
−0				—
Smallest value > 1				
256				—
Largest denormalized				
$-\infty$		—	—	—
Number with hex representation `3AA0`	—			

2.60 ◆

We are running programs on a machine where values of type `int` have a 32-bit two's-complement representation. Values of type `float` use the 32-bit IEEE format, and values of type `double` use the 64-bit IEEE format.

We generate arbitrary integer values x, y, and z, and convert them to other `double` as follows:

```
/* Create some arbitrary values */
int x = random();
int y = random();
int z = random();
/* Convert to double */
double   dx = (double) x;
double   dy = (double) y;
double   dz = (double) z;
```

For each of the following C expressions, you are to indicate whether or not the expression *always* yields 1. If it always yields 1, describe the underlying mathematical principles. Otherwise, give an example of arguments that make it yield 0. Note that you cannot use a IA32 machine running GCC to test your answers, since it would use the 80-bit extended-precision representation for both `float` and `double`.

A. `(double)(float) x == dx`.

B. `dx + dy == (double) (y+x)`.

C. `dx + dy + dz == dz + dy + dx`.

D. `dx * dy * dz == dz * dy * dx`.

E. `dx / dx == dy / dy`.

2.61 ◆

You have been assigned the task of writing a C function to compute a floating-point representation of 2^x. You realize that the best way to do this is to directly construct the IEEE single-precision representation of the result. When x is too small, your routine will return 0.0. When x is too large, it will return $+\infty$. Fill in the blank portions of the code that follows to compute the correct result. Assume the function `u2f` returns a floating-point value having an identical bit representation as its unsigned argument.

```
float fpwr2(int x)
{
    /* Result exponent and significand */
    unsigned exp, sig;
    unsigned u;

    if (x < ______) {              /* Too small. Return 0.0 */
        exp = ____________;
        sig = ____________;
    } else if (x < ______) {       /* Denormalized result. */
        exp = ____________;
        sig = ____________;
    } else if (x < ______) {       /* Normalized result. */
        exp = ____________;
        sig = ____________;
    } else {                       /* Too big. Return +oo */
        exp = ____________;
        sig = ____________;
    }

    /* Pack exp and sig into 32 bits */
    u = exp << 23 | sig;
    /* Return as float */
    return u2f(u);
}
```

2.62 ◆

Around 250 B.C., the Greek mathematician Archimedes proved that $\frac{223}{71} < \pi < \frac{22}{7}$. Had he had access to a computer and the standard library `<math.h>`, he would have been able to determine that the single-precision floating-point approximation of π has the hexadecimal representation `0x40490FDB`. Of course, all of these are just approximations, since π is not rational.

A. What is the fractional binary number denoted by this floating-point value?

B. What is the fractional binary representation of $\frac{22}{7}$? **Hint:** See Problem 2.55.

C. At what bit position (relative to the binary point) do these two approximations to π diverge?

Solution to Practice Problems

Problem 2.1 Solution [Pg. 29]

Understanding the relation between hexadecimal and binary formats will be important once we start looking at machine-level programs. The method for doing these conversions is in the text, but it takes a little practice for it to become familiar

A. `0x8F7A93` to binary:

Hexadecimal	8	F	7	A	9	3
Binary	1000	1111	0111	1010	1001	0011

B. Binary 1011011110011100 to hexadecimal:

Binary	1011	0111	1001	1100
Hexadecimal	B	7	9	C

C. `0xC4E5D` to binary:

Hexadecimal	C	4	E	5	D
Binary	1100	0100	1110	0101	1101

D. Binary 1101011011011111100110 to hexadecimal:

Binary	11	0101	1011	0111	1110	0110
Hexadecimal	3	5	B	7	E	6

Problem 2.2 Solution [Pg. 30]

This problem gives you a chance to think about powers of two and their hexadecimal representations.

n	2^n (Decimal)	2^n (Hexadecimal)
11	2048	`0x800`
7	128	`0x80`
13	8192	`0x2000`
17	131072	`0x20000`
16	65536	`0x10000`
8	256	`0x100`
5	32	`0x20`

Problem 2.3 Solution [Pg. 31]

This problem gives you a chance to try out conversions between hexadecimal and decimal representations for some smaller numbers. For larger ones, it becomes much more convenient and reliable to use a calculator or conversion program.

Decimal	Binary	Hexadecimal
0	00000000	00
$55 = 3 \cdot 16 + 7$	0011 0111	37
$136 = 8 \cdot 16 + 8$	1000 1000	88
$243 = 15 \cdot 16 + 3$	1111 0011	F3
$5 \cdot 16 + 2 = 82$	0101 0010	52
$10 \cdot 16 + 12 = 172$	1010 1100	AC
$14 \cdot 16 + 7 = 231$	1110 0111	E7
$10 \cdot 16 + 7 = 167$	1010 0111	A7
$3 \cdot 16 + 14 = 62$	0011 1110	3E
$11 \cdot 16 + 12 = 188$	1011 1100	BC

Problem 2.4 Solution [Pg. 32]

When you begin debugging machine-level programs, you will find many cases where some simple hexadecimal arithmetic would be useful. You can always convert numbers to decimal, perform the arithmetic, and convert them back, but being able to work directly in hexadecimal is more efficient and informative.

A. `0x502c` + `0x8` = `0x5034`. Adding `8` to hex `c` gives 4 with a carry of `1`.

B. `0x502c`−`0x30` = `0x4ffc`. Subtracting `3` from `2` in the second digit position requires a borrow from the third. Since this digit is `0`, we must also borrow from the fourth position.

C. `0x502c` + 64 = `0x506c`. Decimal 64 (2^6) equals hexadecimal `0x40`.

D. `0x51da` − `0x502c` = `0xae`. To subtract hex `c` (decimal 12) from hex `a` (decimal 10), we borrow 16 from the second digit, giving hex `e` (decimal 14). In the second digit, we now subtract `2` from hex `c` (decimal 12), giving hex `a` (decimal 10).

Problem 2.5 Solution [Pg. 40]

This problem tests your understanding of the byte representation of data and the two different byte orderings.

A. Little endian: `78`	Big endian: `12`
B. Little endian: `78 56`	Big endian: `12 34`
C. Little endian: `78 56 34`	Big endian: `12 34 56`

Recall that `show_bytes` enumerates a series of bytes starting from the one with lowest address and working toward the one with highest address. On a little-endian machine it would list the bytes from least significant to most. On a big-endian machine, it would list bytes from the most significant byte to the least.

Problem 2.6 Solution [Pg. 40]

This problem is another chance to practice hexadecimal to binary conversion. It also gets you thinking about integer and floating-point representations. We will explore these representations in more detail later in this chapter.

A. Using the notation of the example in the text, we write the two strings as follows:

```
   0   0   3   5   4   3   2   1
00000000001101010100001100100001
          *********************
     4   A   5   5   0   C   8   4
  01001010010101010000110010000100
```

B. With the second word shifted two positions relative to the first, we find a sequence with 21 matching bits.

C. We find all bits of the integer embedded in the floating-point number, except for the most significant bit having value 1. Such is the case for the example in the text as well. In addition, the floating-point number has some nonzero high-order bits that do not match those of the integer.

Problem 2.7 Solution [Pg. 41]

It prints `41 42 43 44 45 46`. Recall also that the library routine `strlen` does not count the terminating null character, and so `show_bytes` printed only through the character '`F`.'

Problem 2.8 Solution [Pg. 45]

This problem is a drill to help you become more familiar with Boolean operations.

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

Problem 2.9 Solution [Pg. 45]

This problem illustrates how Boolean algebra can be used to describe and reason about real-world systems. We can see that this color algebra is identical to the Boolean algebra over bit vectors of length 3.

A. Colors are complemented by complementing the values of R, G, and B. From this we can see that White is the complement of Black, Yellow is the complement of Blue, Magenta is the complement of Green, and Cyan is the complement of Red.

B. Black is 0, and White is 1.

C. We perform Boolean operations based on a bit-vector representation of the colors. From this we get the following:

Blue (001)	|	Red (100)	=	Magenta (101)
Magenta (101)	&	Cyan (011)	=	Blue (001)
Green (010)	^	White (111)	=	Magenta (101)

Problem 2.10 Solution [Pg. 47]

This procedure relies on the fact that EXCLUSIVE-OR is commutative and associative, and that $a \mathbin{\hat{}} a = 0$ for any a. We will see in Chapter 5 that the code does not work correctly when the two pointers x and y are equal (that is, they point to the same location).

Step	*x	*y
Initially	a	b
Step 1	$a \mathbin{\hat{}} b$	b
Step 2	$a \mathbin{\hat{}} b$	$(a \mathbin{\hat{}} b) \mathbin{\hat{}} b = (b \mathbin{\hat{}} b) \mathbin{\hat{}} a = a$
Step 3	$(a \mathbin{\hat{}} b) \mathbin{\hat{}} a = (a \mathbin{\hat{}} a) \mathbin{\hat{}} b = b$	a

Problem 2.11 Solution [Pg. 48]

Here are the expressions:

A. `x | ~0xFF`

B. `x ^ 0xFF`

C. `x & ~0xFF`

These expressions are typical of the kind commonly found in performing low-level bit operations. The expression `~0xFF` creates a mask where the 8 least-significant bits equal 0 and the rest equal 1. Observe that such a mask will be generated regardless of the word size. By contrast, the expression `0xFFFFFF00` would only work on a 32-bit machine.

Problem 2.12 Solution [Pg. 48]

These problems help you think about the relation between Boolean operations and typical masking operations. Here is the code:

```
/* Bit Set */
int bis(int x, int m)
{
  int result = x | m;
  return result;
}

/* Bit Clear */
int bic(int x, int m)
{
  int result = x & ~m;
  return result;
}
```

It is easy to see that `bis` is equivalent to Boolean OR—a bit is set in `z` if either this bit is set in `x` or it is set in `m`.

The `bic` operation is a bit more subtle. We want to set a bit of `z` to 0 if the corresponding bit of `m` equals 1. If we complement the mask giving `~m`, then we want to set a bit of `z` to 0 if the corresponding bit of the complemented mask equals 0. We can do this with the AND operation.

Problem 2.13 Solution [Pg. 49]

This problem highlights the relation between bit-level Boolean operations and logic operations in C:

Expression	Value	Expression	Value
`x & y`	`0x02`	`x && y`	`0x01`
`x | y`	`0xF7`	`x || y`	`0x01`
`~x | ~y`	`0xFD`	`!x || !y`	`0x00`
`x & !y`	`0x00`	`x && ~y`	`0x01`

Problem 2.14 Solution [Pg. 50]

The expression is `!(x ^ y)`.

That is `x^y` will be zero if and only if every bit of `x` matches the corresponding bit of `y`. We then exploit the ability of `!` to determine whether a word contains any nonzero bit.

There is no real reason to use this expression rather than simply writing `x == y`, but it demonstrates some of the nuances of bit-level and logical operations.

Problem 2.15 Solution [Pg. 50]

This problem is a drill to help you understand the different shift operations.

`x`		`x << 3`		`x >> 2` (Logical)		`x >> 2` (Arithmetic)	
Hex	Binary	Binary	Hex	Binary	Hex	Binary	Hex
`0xF0`	[11110000]	[10000000]	`0x80`	[00111100]	`0x3C`	[11111100]	`0xFC`
`0x0F`	[00001111]	[01111000]	`0x78`	[00000011]	`0x03`	[00000011]	`0x03`
`0xCC`	[11001100]	[01100000]	`0x60`	[00110011]	`0x33`	[11110011]	`0xF3`
`0x55`	[01010101]	[10101000]	`0xA8`	[00010101]	`0x15`	[00010101]	`0x15`

Problem 2.16 Solution [Pg. 52]

In general, working through examples for very small word sizes is a very good way to understand computer arithmetic.

The unsigned values correspond to those in Figure 2.1. For the two's-complement values, hex digits 0 through 7 have a most significant bit of 0, yielding nonnegative values, while hex digits 8 through F, have a most significant bit of 1, yielding a negative value.

$\vec{x}$		$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
Hexadecimal	Binary		
A	[1010]	$2^3+2^1=10$	$-2^3+2^1=-6$
0	[0000]	0	0
3	[0011]	$2^1+2^0=3$	$2^1+2^0=3$
8	[1000]	$2^3=8$	$-2^3=-8$
C	[1100]	$2^3+2^2=12$	$-2^3+2^2=-4$
F	[1111]	$2^3+2^2+2^1+2^0=15$	$-2^3+2^2+2^1+2^0=-1$

Problem 2.17 Solution [Pg. 55]

For a 32-bit machine, any value consisting of eight hexadecimal digits beginning with one of the digits 8 through f represents a negative number. It is quite common to see numbers beginning with a string of f's, since the leading bits of a negative number are all 1s. You must look carefully, though. For example, the number 0x80483b7 has only seven digits. Filling this out with a leading zero gives 0x080483b7, a positive number.

```
80483b7: 81 ec 84 01 00 00   sub   $0x184,%esp              A.   388
80483bd: 53                  push  %ebx
80483be: 8b 55 08            mov   0x8(%ebp),%edx           B.     8
80483c1: 8b 5d 0c            mov   0xc(%ebp),%ebx           C.    12
80483c4: 8b 4d 10            mov   0x10(%ebp),%ecx          D.    16
80483c7: 8b 85 94 fe ff ff   mov   0xfffffe94(%ebp),%eax    E.  -364
80483cd: 01 cb               add   %ecx,%ebx
80483cf: 03 42 10            add   0x10(%edx),%eax          F.    16
80483d2: 89 85 a0 fe ff ff   mov   %eax,0xfffffea0(%ebp)    G.  -352
80483d8: 8b 85 10 ff ff ff   mov   0xffffff10(%ebp),%eax    H.  -240
80483de: 89 42 1c            mov   %eax,0x1c(%edx)          I.    28
80483e1: 89 9d 7c ff ff ff   mov   %ebx,0xffffff7c(%ebp)    J.  -132
80483e7: 8b 42 18            mov   0x18(%edx),%eax          K.    24
```

Problem 2.18 Solution [Pg. 57]

The functions *T2U* and *U2T* are very peculiar from a mathematical perspective. It is important to understand how they behave.

We solve this problem by reordering the rows in the solution of Practice Problem 2.16 according to the two's-complement value and then listing the unsigned value as the result of the function application. We show the hexadecimal values to make this process more concrete.

$\vec{x}$ (hex)	x	$T2U_4(x)$
8	−8	8
A	−6	10
C	−4	12
F	−1	15
0	0	0
3	3	3

Problem 2.19 Solution [Pg. 58]

This exercise tests your understanding of Equation 2.4.

For the first four entries, the values of x are negative and $T2U_4(x) = x + 2^4$. For the remaining two entries, the values of x are nonnegative and $T2U_4(x) = x$.

Problem 2.20 Solution [Pg. 60]

This problem reinforces your understanding of the relation between two's-complement and unsigned representations, and the effects of the C promotion rules. Recall that $TMin_{32}$ is −2147483648, and when cast to unsigned it becomes 2147483648. In addition, if either operand is unsigned, then the other operand will be cast to unsigned before comparing.

Expression	Type	Evaluation
`-2147483647-1 == 2147483648U`	unsigned	1
`-2147483647-1 < -2147483647`	signed	1
`(unsigned) (-2147483647-1) < -2147483647`	unsigned	1
`-2147483647-1 < 2147483647`	signed	1
`(unsigned) (-2147483647-1) < 2147483647`	unsigned	0

Problem 2.21 Solution [Pg. 63]

The expressions in these functions are common program "idioms" for extracting values from a word in which multiple bit fields have been packed. They exploit the zero-filling and sign-extending properties of the different shift operations. Note carefully the ordering of the cast and shift operations. In `fun1`, the shifts are performed on unsigned `word` and hence are logical. In `fun2`, shifts are performed after casting `word` to `int` and hence are arithmetic.

A.

`w`	`fun1(w)`	`fun2(w)`
127	127	127
128	128	−128
255	255	−1
256	0	0

B. Function `fun1` extracts a value from the low-order 8 bits of the argument, giving an integer ranging between 0 and 255. Function `fun2` also extracts a value from the low-order 8 bits of the argument, but it also performs sign extension. The result will be a number between −128 and 127.

Problem 2.22 Solution [Pg. 64]

The effect of truncation is fairly intuitive for unsigned numbers, but not for two's-complement numbers. This exercise lets you explore its properties using very small word sizes.

Hex		Unsigned		Two's-complement	
Original	Truncated	Original	Truncated	Original	Truncated
0	0	0	0	0	0
3	3	3	3	3	3
8	0	8	0	−8	0
A	2	10	2	−6	2
F	7	15	7	−1	−1

As Equation 2.7 states, the effect of this truncation on unsigned values is to simply to find their residue, modulo 8. The effect of the truncation on signed values is a bit more complex. According to Equation 2.8, we first compute the modulo 8 residue of the argument. This will give values 0 to 7 for arguments 0 to 7, and also for arguments −8 to −1. Then we apply function $U2T_3$ to these residues, giving two repetitions of the sequences 0 to 3, and −4 to −1.

Problem 2.23 Solution [Pg. 65]

This problem was designed to demonstrate how easily bugs can arise due to the implicit casting from signed to unsigned. It seems quite natural to pass parameter `length` as an unsigned, since one would never want to use a negative length. The stopping criterion `i <= length-1` also seems quite natural. But combining these two yields an unexpected outcome!

Since parameter `length` is unsigned, the computation $0 - 1$ is performed using unsigned arithmetic, which is equivalent to modular addition. The result is then $UMax_{32}$ (assuming a 32-bit machine). The $\leq$ comparison is also performed using an unsigned comparison, and since any 32-bit number is less than or equal to $UMax_{32}$, the comparison always holds! Thus, the code attempts to access invalid elements of array `a`.

The code can be fixed by either declaring `length` to be an `int`, or by changing the test of the `for` loop to be `i < length`.

Problem 2.24 Solution [Pg. 69]

This problem is a simple demonstration of arithmetic modulo 16. The easiest way to solve it is to convert the hex pattern into its unsigned decimal value. For nonzero values of x, we must have $(\overset{u}{-}_4\, x) + x = 16$. Then we convert the complemented value back to hex.

x		$-^{\mathrm{u}}_{4}\, x$	
Hex	Decimal	Decimal	Hex
0	0	0	0
3	3	13	D
8	8	8	8
A	10	6	6
F	15	1	1

Problem 2.25 Solution [Pg. 71]

This problem is an exercise to make sure you understand two's-complement addition.

x	y	$x + y$	$x +^{\mathrm{t}}_{5} y$	Case
−16 [10000]	−11 [10101]	−27	5 [00101]	1
−16 [10000]	−16 [10000]	−32	0 [00000]	1
−8 [11000]	7 [00111]	−1	−1 [11111]	2
−2 [11110]	5 [00101]	3	3 [00011]	3
8 [01000]	8 [01000]	16	−16 [10000]	4

Problem 2.26 Solution [Pg. 73]

This problem helps you understand two's-complement negation using a very small word size.

For $w = 4$, we have $TMin_4 = -8$. So -8 is its own additive inverse, while other values are negated by integer negation.

x		$-^{\mathrm{t}}_{4}\, x$	
Hex	Decimal	Decimal	Hex
0	0	0	0
3	3	−3	D
8	−8	−8	8
A	−6	6	6
F	−1	1	1

The bit patterns are the same as for unsigned negation.

Problem 2.27 Solution [Pg. 76]

This problem is an exercise to make sure you understand two's-complement multiplication.

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	6	[110]	2	[010]	12	[001100]	4	[100]
Two's-Comp.	−2	[110]	2	[010]	−4	[111100]	−4	[100]
Unsigned	1	[001]	7	[111]	7	[000111]	7	[111]
Two's-Comp.	1	[001]	−1	[111]	−1	[111111]	−1	[111]
Unsigned	7	[111]	7	[111]	49	[110001]	1	[001]
Two's-Comp.	−1	[111]	−1	[111]	1	[000001]	1	[001]

Problem 2.28 Solution [Pg. 77]

In Chapter 3, we will see many examples of the `leal` instruction in action. The instruction is provided to support pointer arithmetic, but the C compiler often uses it as a way to perform multiplication by small constants.

For each value of k, we can compute two multiples: 2^k (when `b` is 0) and 2^k+1 (when `b` is `a`). Thus, we can compute multiples 1, 2, 3, 4, 5, 8, and 9.

Problem 2.29 Solution [Pg. 79]

We have found that people have difficulty with this exercise when working directly with assembly code. It becomes more clear when put in the form shown in `optarith`.

We can see that `M` is 15; `x*M` is computed as `(x<<4)-x`.

We can see that `N` is 4; a bias value of 3 is added when `y` is negative, and the right shift is by 2.

Problem 2.30 Solution [Pg. 79]

These "C puzzle" problems provide a clear demonstration that programmers must understand the properties of computer arithmetic:

A. `(x >= 0) || ((2*x) < 0)`.
False. Let `x` be −2147483648 ($TMin_{32}$). We will then have `2*x` equal to 0.

B. `(x & 7) != 7 || (x«30 < 0)`.
True. If `(x & 7) != 7` evaluates to 0, then we must have bit x_2 equal to 1. When shifted left by 30, this will become the sign bit.

C. `(x * x) >= 0`.
False. When `x` is 65535 (`0xFFFF`), `x*x` is −131071 (`0xFFFE0001`).

D. `x < 0 || -x <= 0`.
True. If `x` is nonnegative, then `-x` is nonpositive.

E. `x > 0 || -x >= 0`.
False. Let `x` be −2147483648 ($TMin_{32}$). Then both `x` and `-x` are negative.

F. `x*y == ux*uy`.
True. Two's-complement and unsigned multiplication have the same bit-level behavior.

G. `~x*y + uy*ux == -y`.
True. `~x` equals `-x-1`. `uy*ux` equals `x*y`. Thus, the left hand side is equivalent to `-x*y-y+x*y`.

Problem 2.31 Solution [Pg. 82]

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

Fractional value	Binary representation	Decimal representation
$\frac{1}{4}$	0.01	0.25
$\frac{3}{8}$	0.011	0.375
$\frac{23}{16}$	1.0111	1.4375
$\frac{45}{16}$	10.1101	2.8125
$\frac{11}{8}$	1.011	1.375
$\frac{45}{8}$	101.101	5.625
$\frac{49}{16}$	11.0001	3.0625

One simple way to think about fractional binary representations is to represent a number as a fraction of the form $\frac{x}{2^k}$. We can write this in binary using the binary representation of x, with the binary point inserted k positions from the right. As an example, for $\frac{23}{16}$, we have $23_{10} = 10111_2$. We then put the binary point 4 positions from the right to get 1.0111_2.

Problem 2.32 Solution [Pg. 82]

In most cases, the limited precision of floating-point numbers is not a major problem, because the *relative* error of the computation is still fairly low. In this example, however, the system was sensitive to the *absolute* error.

A. We can see that $x - 0.1$ has binary representation:

$$0.000000000000000000000001100[1100]\cdots_2$$

Comparing this to the binary representation of $\frac{1}{10}$, we can see that it is simply $2^{-20} \times \frac{1}{10}$, which is around 9.54×10^{-8}.

B. $9.54 \times 10^{-8} \times 100 \times 60 \times 60 \times 10 \approx 0.343$.

C. $0.343 \times 2000 \approx 687$.

Problem 2.33 Solution [Pg. 87]

Working through floating point representations for very small word sizes helps clarify how IEEE floating point works. Note especially the transition between denormalized and normalized values.

Bits	e	E	f	M	V
`0 00 00`	0	0	0	0	0
`0 00 01`	0	0	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
`0 00 10`	0	0	$\frac{2}{4}$	$\frac{2}{4}$	$\frac{2}{4}$
`0 00 11`	0	0	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$
`0 01 00`	1	0	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{4}{4}$
`0 01 01`	1	0	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{5}{4}$
`0 01 10`	1	0	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{6}{4}$
`0 01 11`	1	0	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{7}{4}$
`0 10 00`	2	1	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{8}{4}$
`0 10 01`	2	1	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{10}{4}$
`0 10 10`	2	1	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{12}{4}$
`0 10 11`	2	1	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{14}{4}$
`0 11 00`	—	—	—	—	$+\infty$
`0 11 01`	—	—	—	—	*NaN*
`0 11 10`	—	—	—	—	*NaN*
`0 11 11`	—	—	—	—	*NaN*

Problem 2.34 Solution [Pg. 89]

Hexadecimal `0x354321` is equivalent to binary [1101010100001100100001]. Shifting this right 21 places gives $1.101010100001100100001_2 \times 2^{21}$. We form the fraction field by dropping the leading 1 and adding 2 0s, giving [10101010000110010000100]. The exponent is formed by adding bias 127 to 21, giving 148 (binary [10010100]). We combine this with a sign field of 0 to give a binary representation

$$[01001010010101010000110010000100].$$

We see that the correlation between the two representations correspond to the low-order bits of the integer, up to the most significant bit equal to 1 matching the high-order 21 bits of the fraction:

```
   0   0   3   5   4   3   2   1
 00000000001101010100001100100001
                *********************
     4   A   5   5   0   C   8   4
   01001010010101010000110010000100
```

Problem 2.35 Solution [Pg. 89]

This exercise helps you think about what numbers cannot be represented exactly in floating point.

The number has binary representation 1 followed by n 0's followed by 1, giving value $2^{n+1}+1$.

When $n=23$, the value is $2^{24}+1=16{,}777{,}217$.

Problem 2.36 Solution [Pg. 93]

In general it is better to use a library macro rather than inventing your own code. This code seems to work on a variety of machines, however.

We assume that the value `1e400` overflows to infinity.

code/data/ieee.c

```
#define POS_INFINITY 1e400
#define NEG_INFINITY (-POS_INFINITY)
#define NEG_ZERO (-1.0/POS_INFINITY)
```

code/data/ieee.c

Problem 2.37 Solution [Pg. 97]

Exercises such as this one help you develop your ability to reason about floating point operations from a programmer's perspective. Make sure you understand each of the answers.

A. `x == (int)(float) x`
No. For example, when `x` is *TMax*.

B. `x == (int)(double) x`
Yes, since `double` has greater precision and range than `int`.

C. `f == (float)(double) f`
Yes, since `double` has greater precision and range than `float`.

D. `d == (float) d`
No. For example, when `d` is `1e40`, we will get $+\infty$ on the right.

E. `f == -(-f)`
Yes, since a floating-point number is negated by simply inverting its sign bit.

F. `2/3 == 2/3.0`
No, the lefthand value will be the integer value `0`, while the righthand value will be the floating-point approximation of $\frac{2}{3}$.

G. `(d >= 0.0) || ((d*2) < 0.0)`
Yes, since multiplication is monotonic.

H. `(d+f)-d == f`
No, for example when `d` is $+\infty$ and `f` is 1, the lefthand side will be *NaN*, while the righthand side will be 1.