1.1 Information Is Bits + Context

Our `hello` program begins life as a *source program* (or *source file*) that the programmer creates with an editor and saves in a text file called `hello.c`. The source program is a sequence of bits, each with a value of 0 or 1, organized in 8-bit chunks called *bytes*. Each byte represents some text character in the program.

Most computer systems represent text characters using the ASCII standard that represents each character with a unique byte-size integer value.[1] For example, **Figure 1.2** shows the ASCII representation of the `hello.c` program.

1. Other encoding methods are used to represent text in non-English languages. See the aside on page 50 for a discussion on this.

The `hello.c` program is stored in a file as a sequence of bytes. Each byte has an integer value that corresponds to some character. For example, the first byte has the integer value 35, which corresponds to the character `'#'`. The second byte has the integer value 105, which corresponds to the character `'i'`, and so on. Notice that each text line is terminated by the invisible *newline* character `'\n'`, which is represented by the integer value 10. Files such as `hello.c` that consist exclusively of ASCII characters are known as *text files*. All other files are known as *binary files*.

The representation of `hello.c` illustrates a fundamental idea: All information in a system—including disk files, programs stored in memory, user data stored in memory, and data transferred across a network—is represented as a bunch of bits. The only thing that distinguishes different data objects is the context in which we view them. For example, in different contexts, the same sequence of bytes might represent an integer, floating-point number, character string, or machine instruction.

As programmers, we need to understand machine representations of numbers because they are not the same as integers and real numbers. They are finite

> ## Aside Origins of the C programming language
>
> C was developed from 1969 to 1973 by Dennis Ritchie of Bell Laboratories. The American National Standards Institute (ANSI) ratified the ANSI C standard in 1989, and this standardization later became the responsibility of the International Standards Organization (ISO). The standards define the C language and a set of library functions known as the *C standard library*. Kernighan and Ritchie describe ANSI C in their classic book, which is known affectionately as "K&R" [61]. In Ritchie's words [92], C is "quirky, flawed, and an enormous success." So why the success?
>
> - **C was closely tied with the Unix operating system.** C was developed from the beginning as the system programming language for Unix. Most of the Unix kernel (the core part of the operating system), and all of its supporting tools and libraries, were written in C. As Unix became popular in universities in the late 1970s and early 1980s, many people were exposed to C and found that they liked it. Since Unix was written almost entirely in C, it could be easily ported to new machines, which created an even wider audience for both C and Unix.
> - **C is a small, simple language.** The design was controlled by a single person, rather than a committee, and the result was a clean, consistent design with little baggage. The K&R book describes the complete language and standard library, with numerous examples and exercises, in only 261 pages. The simplicity of C made it relatively easy to learn and to port to different computers.
> - **C was designed for a practical purpose.** C was designed to implement the Unix operating system. Later, other people found that they could write the programs they wanted, without the language getting in the way.
>
> C is the language of choice for system-level programming, and there is a huge installed base of application-level programs as well. However, it is not perfect for all programmers and all situations. C pointers are a common source of confusion and programming errors. C also lacks explicit support for useful abstractions such as classes, objects, and exceptions. Newer languages such as C++ and Java address these issues for application-level programs.

approximations that can behave in unexpected ways. This fundamental idea is explored in detail in **Chapter 2**.

1.2 Programs Are Translated by Other Programs into Different Forms

The `hello` program begins life as a high-level C program because it can be read and understood by human beings in that form. However, in order to run `hello.c` on the system, the individual C statements must be translated by other programs into a sequence of low-level *machine-language* instructions. These instructions are then packaged in a form called an *executable object program* and stored as a binary disk file. Object programs are also referred to as *executable object files*.

On a Unix system, the translation from source file to object file is performed by a *compiler driver*:

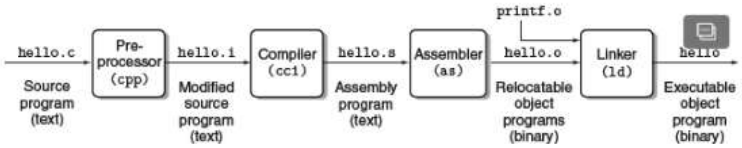


Figure 1.3 The compilation system.

```
linux> gcc -o hello hello.c
```

Here, the GCC compiler driver reads the source file `hello.c` and translates it into an executable object file `hello`. The translation is performed in the sequence of four phases shown in **Figure 1.3**. The programs that perform the four phases (*preprocessor*, *compiler*, *assembler*, and *linker*) are known collectively as the *compilation system*.

- **Preprocessing phase.** The preprocessor (cpp) modifies the original C program according to directives that begin with the '`#`' character. For example, the `#include <stdio.h>` command in line 1 of `hello.c` tells the preprocessor to read the contents of the system header file `stdio.h` and insert it directly into the program text. The result is another C program, typically with the `.i` suffix.
- **Compilation phase.** The compiler (`cc1`) translates the text file `hello.i` into the text file `hello.s`, which contains an *assembly-language program*. This program includes the following definition of `function main`:

```
1       main:
2         subq        $8, %rsp
3         movl        $.LC0, %edi
4         call        puts
5         movl        $0, %eax
6         addq        $8, %rsp
7         ret
```

  Each of lines 2-7 in this definition describes one low-level machine-language instruction in a textual form. Assembly language is useful because it provides a common output language for different compilers for different high-level languages. For example, C compilers and Fortran compilers both generate output files in the same assembly language.
- **Assembly phase.** Next, the assembler (`as`) translates `hello.s` into machine-language instructions, packages them in a form known as a *relocatable object program*, and stores the result in the object file `hello.o`. This file is a binary file containing 17 bytes to encode the instructions for function main. If we were to view `hello.o` with a text editor, it would appear to be gibberish.

> ## Aside The GNU project
>
> GCC is one of many useful tools developed by the GNU (short for GNU's Not Unix) project. The GNU project is a tax-exempt charity started by Richard Stallman in 1984, with the ambitious goal of developing a complete Unix-like system whose source code is unencumbered by restrictions on how it can be modified or distributed. The GNU project has developed an environment with all the major components of a Unix operating system, except for the kernel, which was developed separately by the Linux project. The GNU environment includes the EMACS editor, GCC compiler, GDB debugger, assembler, linker, utilities for manipulating binaries, and other components. The GCC compiler has grown to support many different languages, with the ability to generate code for many different machines. Supported languages include C, C++, Fortran, Java, Pascal, Objective-C, and Ada.
>
> The GNU project is a remarkable achievement, and yet it is often overlooked. The modern open-source movement (commonly associated with Linux) owes its intellectual origins to the GNU project's notion of *free software* ("free" as in "free speech," not "free beer"). Further, Linux owes much of its popularity to the GNU tools, which provide the environment for the Linux kernel.

- **Linking phase.** Notice Notice that our `hello` program calls the `printf` function, which is part of the *standard C library* provided by every C compiler. The `printf` function resides in a separate precompiled object file called `printf.o`, which must somehow be merged with our `hello.o` program. The linker (`ld`) handles this merging. The result is the `hello` file, which is an executable object file (or simply *executable*) that is ready to be loaded into memory and executed by the system.

1.3 It Pays to Understand How Compilation Systems Work

For simple programs such as `hello.c`, we can rely on the compilation system to produce correct and efficient machine code. However, there are some important reasons why programmers need to understand how compilation systems work:

- **Optimizing program performance.** Modern compilers are sophisticated tools that usually produce good code. As programmers, we do not need to know the inner workings of the compiler in order to write efficient code. However, in order to make good coding decisions in our C programs, we do need a basic understanding of machine-level code and how the compiler translates different C statements into machine code. For example, is a `switch` statement always more efficient than a sequence of `if-else` statements? How much overhead is incurred by a function call? Is a `while` loop more efficient than a `for` loop? Are pointer references more efficient than array indexes? Why does our loop run so much faster if we sum into a local variable instead of an argument that is passed by reference? How can a function run faster when we simply rearrange the parentheses in an arithmetic expression?

  In **Chapter 3**, we introduce x86-64, the machine language of recent generations of Linux, Macintosh, and Windows computers. We describe how compilers translate different C constructs into this language. In **Chapter 5**, you will learn how to tune the performance of your C programs by making simple transformations to the C code that help the compiler do its job better. In **Chapter 6**, you will learn about the hierarchical nature of the memory system, how C compilers store data arrays in memory, and how your C programs can exploit this knowledge to run more efficiently.
- **Understanding link-time errors.** In our experience, some of the most perplexing programming errors are related to the operation of the linker, especially when you are trying to build large software systems. For example, what does it mean when the linker reports that it cannot resolve a reference? What is the difference between a static variable and a global variable? What happens if you define two global variables in different C files with the same name? What is the difference between a static library and a dynamic library? Why does it matter what order we list libraries on the command line? And scariest of all, why do some linker-related errors not appear until run time? You will learn the answers to these kinds of questions in **Chapter 7**.
- **Avoiding security holes.** For many years, *buffer overflow vulnerabilities* have accounted for many of the security holes in network and Internet servers. These vulnerabilities exist because too few programmers understand the need to carefully restrict the quantity and forms of data they accept from untrusted sources. A first step in learning secure programming is to understand the consequences of the way data and control information are stored on the program stack. We cover the stack discipline and buffer overflow vulnerabilities in **Chapter 3** as part of our study of assembly language. We will also learn about methods that can be used by the programmer, compiler, and operating system to reduce the threat of attack.

1.4 Processors Read and Interpret Instructions Stored in Memory

At this point, our `hello.c` source program has been translated by the compilation system into an executable object file called `hello` that is stored on disk. To run the executable file on a Unix system, we type its name to an application program known as a *shell*:

```
linux> ./hello
hello, world
linux>
```

The shell is a command-line interpreter that prints a prompt, waits for you to type a command line, and then performs the command. If the first word of the command line does not correspond to a built-in shell command, then the shell

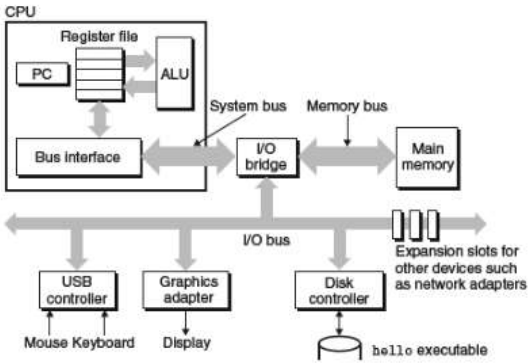


Figure 1.4 Hardware organization of a typical system.
CPU: central processing unit, ALU: arithmetic/logic unit, PC: program counter, USB: Universal Serial Bus.

assumes that it is the name of an executable file that it should load and run. So in this case, the shell loads and runs the `hello` program and then waits for it to terminate. The `hello` program prints its message to the screen and then terminates. The shell then prints a prompt and waits for the next input command line.

1.4.1 Hardware Organization of a System

To understand what happens to our `hello` program when we run it, we need to understand the hardware organization of a typical system, which is shown in **Figure 1.4**. This particular picture is modeled after the family of recent Intel systems, but all systems have a similar look and feel. Don't worry about the complexity of this figure just now. We will get to its various details in stages throughout the course of the book.

Buses

Running throughout the system is a collection of electrical conduits called *buses* that carry bytes of information back and forth between the components. Buses are typically designed to transfer fixed-size chunks of bytes known as *words*. The number of bytes in a word (the *word size*) is a fundamental system parameter that varies across systems. Most machines today have word sizes of either 4 bytes (32 bits) or 8 bytes (64 bits). In this book, we do not assume any fixed definition of word size. Instead, we will specify what we mean by a "word" in any context that requires this to be defined.

I/O Devices

Input/output (I/O) devices are the system's connection to the external world. Our example system has four I/O devices: a keyboard and mouse for user input, a display for user output, and a disk drive (or simply disk) for long-term storage of data and programs. Initially, the executable `hello` program resides on the disk.

Each I/O device is connected to the I/O bus by either a *controller* or an *adapter*. The distinction between the two is mainly one of packaging. Controllers are chip sets in the device itself or on the system's main printed circuit board (often called the *motherboard*). An adapter is a card that plugs into a slot on the motherboard. Regardless, the purpose of each is to transfer information back and forth between the I/O bus and an I/O device.

Chapter 6 has more to say about how I/O devices such as disks work. In **Chapter 10**, you will learn how to use the Unix I/O interface to access devices from your application programs. We focus on the especially interesting class of devices known as networks, but the techniques generalize to other kinds of devices as well.

Main Memory

The *main memory* is a temporary storage device that holds both a program and the data it manipulates while the processor is executing the program. Physically, main memory consists of a collection of *dynamic random access memory*(DRAM) chips. Logically, memory is organized as a linear array of bytes, each with its own unique address (array index) starting at zero. In general, each of the machine instructions that constitute a program can consist of a variable number of bytes. The sizes of data items that correspond to C program variables vary according to type. For example, on an x86-64 machine running Linux, data of type `short` require 2 bytes, types `int` and `float` 4 bytes, and types `long` and `double` 8 bytes.

Chapter 6 has more to say about how memory technologies such as DRAM chips work, and how they are combined to form main memory.

Processor

The *central processing unit* (CPU), or simply *processor*, is the engine that interprets (or *executes*) instructions stored in main memory. At its core is a word-size storage device (or *register*) called the *program counter* (PC). At any point in time, the PC points at (contains the address of) some machine-language instruction in main memory.[2]

2. PC is also a commonly used acronym for "personal computer." However, the distinction between the two should be clear from the context.

From the time that power is applied to the system until the time that the power is shut off, a processor repeatedly executes the instruction pointed at by the program counter and updates the program counter to point to the next instruction. A processor *appears* to operate according to a very simple instruction execution model, defined by its *instruction set architecture*. In this model, instructions execute in strict sequence, and executing a single instruction involves performing a series of steps. The processor reads the instruction from memory pointed at by the program counter (PC), interprets the bits in the instruction, performs some simple operation dictated by the instruction, and then updates the PC to point to the next instruction, which may or may not be contiguous in memory to the instruction that was just executed.

There are only a few of these simple operations, and they revolve around main memory, the *register file*, and the *arithmetic/logic unit* (ALU). The register file is a small storage device that consists of a collection of word-size registers, each with its own unique name. The ALU computes new data and address values. Here are some examples of the simple operations that the CPU might carry out at the request of an instruction:

- **Load:** Copy a byte or a word from main memory into a register, overwriting the previous contents of the register.
- **Store:** Copy a byte or a word from a register to a location in main memory, overwriting the previous contents of that location.
- **Operate:** Copy the contents of two registers to the ALU, perform an arithmetic operation on the two words, and store the result in a register, overwriting the previous contents of that register.
- **Jump:** Extract a word from the instruction itself and copy that word into the program counter (PC), overwriting the previous value of the PC.

We say that a processor appears to be a simple implementation of its instruction set architecture, but in fact modern processors use far more complex mechanisms to speed up program execution. Thus, we can distinguish the processor's instruction set architecture, describing the effect of each machine-code instruction, from its *microarchitecture*, describing how the processor is actually implemented. When we study machine code in **Chapter 3**, we will consider the abstraction provided by the machine's instruction set architecture. **Chapter 4** has more to say about how processors are actually implemented. **Chapter 5** describes a model of how modern processors work that enables predicting and optimizing the performance of machine-language programs.

1.4.2 Running the `hello` Program

Given this simple view of a system's hardware organization and operation, we can begin to understand what happens when we run our example program. We must omit a lot of details here that will be filled in later, but for now we will be content with the big picture.

Initially, the shell program is executing its instructions, waiting for us to type a command. As we type the characters `./hello` at the keyboard, the shell program reads each one into a register and then stores it in memory, as shown in **Figure 1.5**.

When we hit the enter key on the keyboard, the shell knows that we have finished typing the command. The shell then loads the executable `hello` file by executing a sequence of instructions that copies the code and data in the `hello`

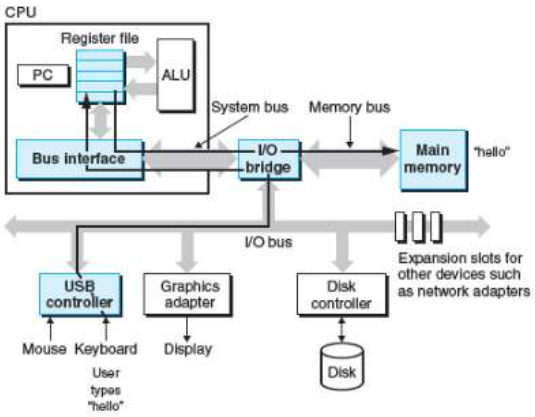


Figure 1.5 Reading the `hello` command from the keyboard.

object file from disk to main memory. The data includes the string of characters `hello, world\n` that will eventually be printed out.

Using a technique known as *direct memory access* (DMA, discussed in **Chapter 6**), the data travel directly from disk to main memory, without passing through the processor. This step is shown in **Figure 1.6**.

Once the code and data in the `hello` object file are loaded into memory, the processor begins executing the machine-language instructions in the `hello` program's `main` routine. These instructions copy the bytes in the `hello, world\n` string from memory to the register file, and from there to the display device, where they are displayed on the screen. This step is shown in **Figure 1.7**.

1.5 Caches Matter

An important lesson from this simple example is that a system spends a lot of time moving information from one place to another. The machine instructions in the `hello` program are originally stored on disk. When the program is loaded, they are copied to main memory. As the processor runs the program, instructions are copied from main memory into the processor. Similarly, the data string `hello, world\n`, originally on disk, is copied to main memory and then copied from main memory to the display device. From a programmer's perspective, much of this copying is overhead that slows down the "real work" of the program. Thus, a major goal for system designers is to make these copy operations run as fast as possible.

Because of physical laws, larger storage devices are slower than smaller storage devices. And faster devices are more expensive to build than their slower

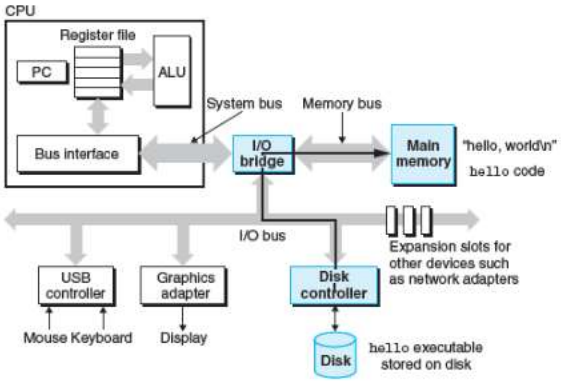


Figure 1.6 Loading the executable from disk into main memory.

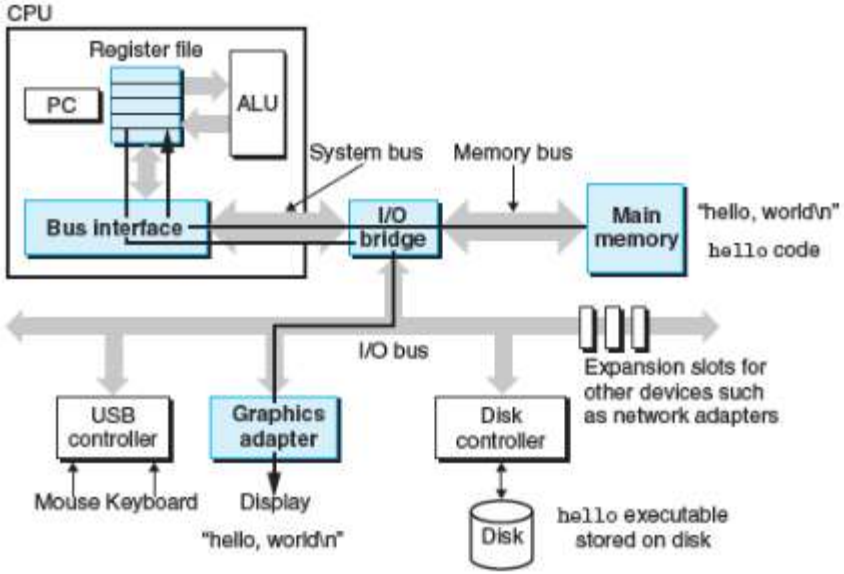


Figure 1.7 Writing the output string from memory to the display.

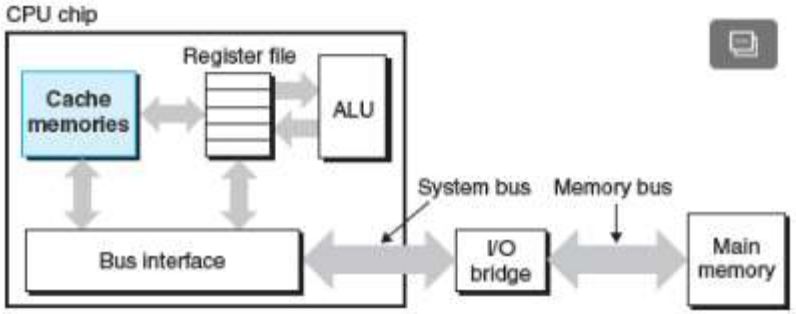


Figure 1.8 Cache memories.

Figure 1.8 Cache memories.

counterparts. For example, the disk drive on a typical system might be 1,000 times larger than the main memory, but it might take the processor 10,000,000 times longer to read a word from disk than from memory.

Similarly, a typical register file stores only a few hundred bytes of information, as opposed to billions of bytes in the main memory. However, the processor can read data from the register file almost 100 times faster than from memory. Even more troublesome, as semiconductor technology progresses over the years, this *processor-memory gap* continues to increase. It is easier and cheaper to make processors run faster than it is to make main memory run faster.

To deal with the processor-memory gap, system designers include smaller, faster storage devices called *cache memories* (or simply caches) that serve as temporary staging areas for information that the processor is likely to need in the near future. **Figure 1.8** shows the cache memories in a typical system. An *L1 cache* on the processor chip holds tens of thousands of bytes and can be accessed nearly as fast as the register file. A larger *L2 cache* with hundreds of thousands to millions of bytes is connected to the processor by a special bus. It might take 5 times longer for the processor to access the L2 cache than the L1 cache, but this is still 5 to 10 times faster than accessing the main memory. The L1 and L2 caches are implemented with a hardware technology known as *static random access memory* (SRAM). Newer and more powerful systems even have three levels of cache: L1, L2, and L3. The idea behind caching is that a system can get the effect of both a very large memory and a very fast one by exploiting *locality*, the tendency for programs to access data and code in localized regions. By setting up caches to hold data that are likely to be accessed often, we can perform most memory operations using the fast caches.

One of the most important lessons in this book is that application programmers who are aware of cache memories can exploit them to improve the performance of their programs by an order of magnitude. You will learn more about these important devices and how to exploit them in **Chapter 6**.

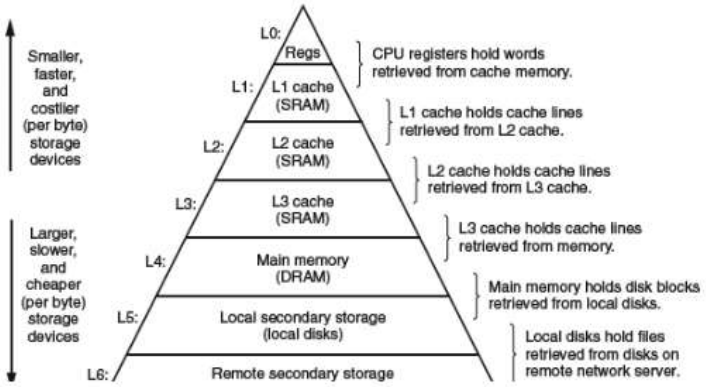


Figure 1.9 An example of a memory hierarchy.

1.6 Storage Devices Form a Hierarchy

This notion of inserting a smaller, faster storage device (e.g., cache memory) between the processor and a larger, slower device (e.g., main memory) turns out to be a general idea. In fact, the storage devices in every computer system are organized as a *memory hierarchy* similar to **Figure 1.9**. As we move from the top of the hierarchy to the bottom, the devices become slower, larger, and less costly per byte. The register file occupies the top level in the hierarchy, which is known as level 0 or L0. We show three levels of caching L1 to L3, occupying memory hierarchy levels 1 to 3. Main memory occupies level 4, and so on.

The main idea of a memory hierarchy is that storage at one level serves as a cache for storage at the next lower level. Thus, the register file is a cache for the L1 cache. Caches L1 and L2 are caches for L2 and L3, respectively. The L3 cache is a cache for the main memory, which is a cache for the disk. On some networked systems with distributed file systems, the local disk serves as a cache for data stored on the disks of other systems.

Just as programmers can exploit knowledge of the different caches to improve performance, programmers can exploit their understanding of the entire memory hierarchy. **Chapter 6** will have much more to say about this.

1.7 The Operating System Manages the Hardware

Back to our `hello` example. When the shell loaded and ran the `hello` program, and when the `hello` program printed its message, neither program accessed the

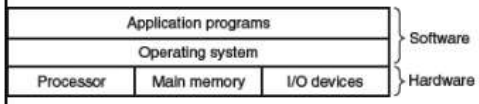


Figure 1.10 Layered view of a computer system.

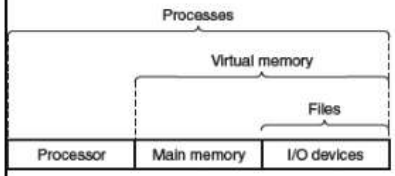


Figure 1.11 Abstractions provided by an operating system.

keyboard, display, disk, or main memory directly. Rather, they relied on the services provided by the *operating system*. We can think of the operating system as a layer of software interposed between the application progra and the hardware, as shown in **Figure 1.10**. All attempts by an application program to manipulate the hardware must go through the operating system.

The operating system has two primary purposes: (1) to protect the hardware from misuse by runaway applications and (2) to provide applications with simple and uniform mechanisms for manipulating complicated and ofte wildly different low-level hardware devices. The operating system achieves both goals via the fundamental abstractions shown in **Figure 1.11**: *processes*, *virtual memory*, and *files*. As this figure suggests, files are abstractions for I/O devices, virtual memory is an abstraction for both the main memory and disk I/O devices, and processes are abstractions for the processor, main memory, and I/O devices. We will discuss each in turn.

1.7.1 Processes

When a program such as `hello` runs on a modern system, the operating system provides the illusion that the program is the only one running on the system. The program appears to have exclusive use of both the process main memory, and I/O devices. The processor appears to execute the instructions in the program, one after the other, without interruption. And the code and data of the program appear to be the only objects in the system's memory. These illusions are provided by the notion of a process, one of the most important and successful ideas in computer science.

A *process* is the operating system's abstraction for a running program. Multiple processes can run concurrently on the same system, and each process appears to have exclusive use of the hardware. By *concurrently*, we mean that the instructions of one process are interleaved with the instructions of another process. In most systems, there are more processes to run than there are CPUs to run them.

Aside Unix, Posix, and the Standard Unix Specification

Z

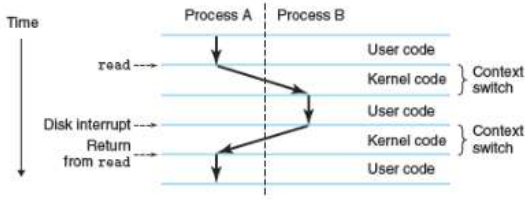


Figure 1.12 Process context switching.

and then passing control to the new process. The new process picks up exactly where it left off. **Figure 1.12** shows the basic idea for our example `hello` scenario.

There are two concurrent processes in our example scenario: the shell process and the `hello` process. Initially, the shell process is running alone, waiting for input on the command line. When we ask it to run the `hello` program, the shell carries out our request by invoking a special function known as a *system call* that passes control to the operating system. The operating system saves the shell's context, creates a new `hello` process and its context, and then passes control to the new `hello` process. After `hello` terminates, the operating system restores the context of the shell process and passes control back to it, where it waits for the next command-line input.

As **Figure 1.12** indicates, the transition from one process to another is managed by the operating system *kernel*. The kernel is the portion of the operating system code that is always resident in memory. When an application program requires some action by the operating system, such as to read or write a file, it executes a special *system call* instruction, transferring control to the kernel. The kernel then performs the requested operation and returns back to the application program. Note that the kernel is not a separate process. Instead, it is a collection of code and data structures that the system uses to manage all the processes.

Implementing the process abstraction requires close cooperation between both the low-level hardware and the operating system software. We will explore how this works, and how applications can create and control their own processes, in **Chapter 8**.

1.7.2 Threads

Although we normally think of a process as having a single control flow, in modern systems a process can actually consist of multiple execution units, called *threads*, each running in the context of the process and sharing the same code and global data. Threads are an increasingly important programming model because of the requirement for concurrency in network servers, because it is easier to share data between multiple threads than between multiple processes, and because threads are typically more efficient than processes. Multi-threading is also one way to make programs run faster when multiple processors are available, as we will discuss in

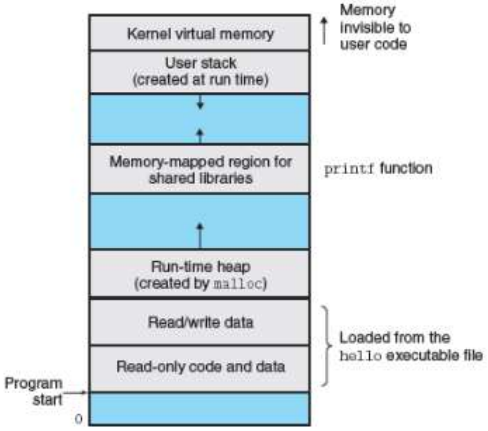


Figure 1.13 Process virtual address space.
(The regions are not drawn to scale.)

Section 1.9.2. You will learn the basic concepts of concurrency, including how to write threaded programs, in Chapter 12.

1.7.3 Virtual Memory

Virtual memory is an abstraction that provides each process with the illusion that it has exclusive use of the main memory. Each process has the same uniform view of memory, which is known as its *virtual address space*. The virtual address space for Linux processes is shown in **Figure 1.13**. (Other Unix systems use a similar layout.) In Linux, the topmost region of the address space is reserved for code and data in the operating system that is common to all processes. The lower region of the address space holds the code and data defined by the user's process. Note that addresses in the figure increase from the bottom to the top.

The virtual address space seen by each process consists of a number of well-defined areas, each with a specific purpose. You will learn more about these areas later in the book, but it will be helpful to look briefly at each, starting with the lowest addresses and working our way up:

- **Program code and data.** Code begins at the same fixed address for all processes, followed by data locations that correspond to global C variables. The code and data areas are initialized directly from the contents of an executable object file—in our case, the `hello` executable. You will learn more about this part of the address space when we study linking and loading in **Chapter 7**.
- **Heap.** The code and data areas are followed immediately by the run-time *heap*. Unlike the code and data areas, which are fixed in size once the process begins running, the heap expands and contracts dynamically at run time as a result of calls to C standard library routines such as `malloc` and `free`. We will study heaps in detail when we learn about managing virtual memory in **Chapter 9**.
- **Shared libraries.** Near the middle of the address space is an area that holds the code and data for *shared libraries* such as the C standard library and the math library. The notion of a shared library is a powerful but somewhat difficult concept. You will learn how they work when we study dynamic linking in **Chapter 7**.
- **Stack.** At the top of the user's virtual address space is the *user stack* that the compiler uses to implement function calls. Like the heap, the user stack expands and contracts dynamically during the execution of the program. In particular, each time we call a function, the stack grows. Each time we return from a function, it contracts. You will learn how the compiler uses the stack in **Chapter 3**.
- **Kernel virtual memory.** The top region of the address space is reserved for the kernel. Application programs are not allowed to read or write the contents of this area or to directly call functions defined in the kernel code. Instead, they must invoke the kernel to perform these operations.

For virtual memory to work, a sophisticated interaction is required between the hardware and the operating system software, including a hardware translation of every address generated by the processor. The basic idea is to store the contents of a process's virtual memory on disk and then use the main memory as a cache for the disk. **Chapter 9** explains how this works and why it is so important to the operation of modern systems.

1.7.4 Files

A *file* is a sequence of bytes, nothing more and nothing less. Every I/O device, including disks, keyboards, displays, and even networks, is modeled as a file. All input and output in the system is performed by reading and writing files, using a small set of system calls known as *Unix I/O*.

This simple and elegant notion of a file is nonetheless very powerful because it provides applications with a uniform view of all the varied I/O devices that might be contained in the system. For example, application programmers who manipulate the contents of a disk file are blissfully unaware of the specific disk technology. Further, the same program will run on different systems that use different disk technologies. You will learn about Unix I/O in **Chapter 10**.

1.8 Systems Communicate with Other Systems Using Networks

Up to this point in our tour of systems, we have treated a system as an isolated collection of hardware and software. In practice, modern systems are often linked to other systems by networks. From the point of view of an individual system, the network can be viewed as just another I/O device, as shown in **Figure 1.14**. When the system copies a sequence of bytes from main memory to the network adapter, the data flow across the network to another machine instead of, say, to a local disk drive. Similarly, the system can read data sent from other machines and copy these data to its main memory.

Aside The Linux project

In August 1991, a Finnish graduate student named Linus Torvalds modestly announced a new Unix-like operating system kernel:

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Date: 25 Aug 91 20:57:08 GMT

Hello everybody out there using minix -
I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since April, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torvalds@kruuna.helsinki.fi)
```

As Torvalds indicates, his starting point for creating Linux was Minix, an operating system developed by Andrew S. Tanenbaum for educational purposes [113].

The rest, as they say, is history. Linux has evolved into a technical and cultural phenomenon. By combining forces with the GNU project, the Linux project has developed a complete, Posix-compliant version of the Unix operating system, including the kernel and all of the supporting infrastructure. Linux is available on a wide array of computers, from handheld devices to mainframe computers. A group at IBM has even ported Linux to a wristwatch!

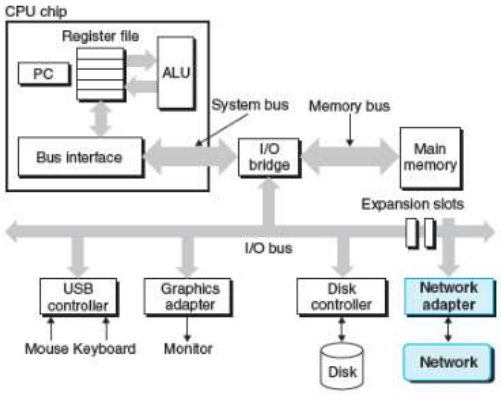


Figure 1.14 A network is another I/O device.

With the advent of global networks such as the Internet, copying information from one machine to another has become one of the most important uses of computer systems. For example, applications such as email, instant messaging, the World Wide Web, FTP, and telnet are all based on the ability to copy information over a network.

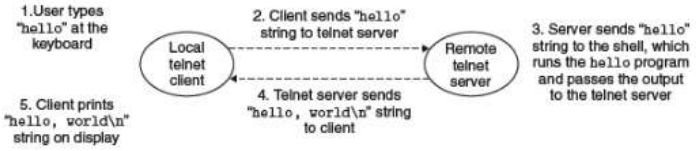


Figure 1.15 Using telnet to run `hello` remotely over a network.

Returning to our `hello` example, we could use the familiar telnet application to run `hello` on a remote machine. Suppose we use a telnet *client* running on our local machine to connect to a telnet *server* on a remote machine. After we log in to the remote machine and run a shell, the remote shell is waiting to receive an input command. From this point, running the `hello` program remotely involves the five basic steps shown in **Figure 1.15**.

After we type in the `hello` string to the telnet client and hit the enter key, the client sends the string to the telnet server. After the telnet server receives the string from the network, it passes it along to the remote shell program. Next, the remote shell runs the `hello` program and passes the output line back to the telnet server. Finally, the telnet server forwards the output string across the network to the telnet client, which prints the output string on our local terminal.

1.9.1 Amdahl's Law

Gene Amdahl, one of the early pioneers in computing, made a simple but insightful observation about the effectiveness of improving the performance of one part of a system. This observation has come to be known as *Amdahl's law*. The main idea is that when we speed up one part of a system, the effect on the overall system performance depends on both how significant this part was and how much it sped up. Consider a system in which executing some application requires time T_{old}. Suppose some part of the system requires a fraction α of this time, and that we improve its performance by a factor of k. That is, the component originally required time αT_{old}, and it now requires time $(\alpha T_{old})/k$. The overall execution time would thus be

$$\begin{aligned} T_{\text{new}} &= (1-\alpha)T_{\text{old}} + (\alpha T_{\text{old}})/k \\ &= T_{old}[(1-\alpha) + \alpha/k] \end{aligned}$$

From this, we can compute the speedup $S = T_{old}/T_{new}$ as

$$S = \frac{1}{(1-\alpha) + \alpha/k} \tag{1.1}$$

As an example, consider the case where a part of the system that initially consumed 60% of the time ($\alpha = 0.6$) is sped up by a factor of 3 ($k = 3$). Then we get a speedup of 1/[0.4 + 0.6/3] = 1.67×. Even though we made a substantial improvement to a major part of the system, our net speedup was significantly less than the speedup for the one part. This is the major insight of Amdahl's law—to significantly speed up the entire system, we must improve the speed of a very large fraction of the overall system.

Practice Problem 1.1 (solution page 28)

Suppose you work as a truck driver, and you have been hired to carry a load of potatoes from Boise, Idaho, to Minneapolis, Minnesota, a total distance of 2,500 kilometers. You estimate you can average 100 km/hr driving within the speed limits, requiring a total of 25 hours for the trip.

Aside Expressing relative performance

The best way to express a performance improvement is as a ratio of the form T_{old}/T_{new}, where T_{old} is the time required for the original version and T_{new} is the time required by the modified version. This will be a number greater than 1.0 if any real improvement occurred. We use the suffix `×' to indicate such a ratio, where the factor "2.2×" is expressed verbally as "2.2 times."

The more traditional way of expressing relative change as a percentage works well when the change is small, but its definition is ambiguous. Should it be $100 \cdot (T_{old} - T_{new})/T_{new}$, or possibly $100 \cdot (T_{old} - T_{new})/T_{old}$, or something else? In addition, it is less instructive for large changes. Saying that "performance improved by 120%" is more difficult to comprehend than simply saying that the performance improved by 2.2×.

A. You hear on the news that Montana has just abolished its speed limit, which constitutes 1,500 km of the trip. Your truck can travel at 150 km/hr. What will be your speedup for the trip?

B. You can buy a new turbocharger for your truck at www.fasttrucks.com. They stock a variety of models, but the faster you want to go, the more it will cost. How fast must you travel through Montana to get an overall speedup for your trip of 1.67×?

Practice Problem 1.2 (solution page 28)

> The marketing department at your company has promised your customers that the next software release will show a 2× performance improvement. You have been assigned the task of delivering on that promise. You have determined that only 80% of the system can be improved. How much (i.e., what value of *k*) would you need to improve this part to meet the overall performance target?

One interesting special case of Amdahl's law is to consider the effect of setting k to ∞. That is, we are able to take some part of the system and speed it up to the point at which it takes a negligible amount of time. We then get

$$S_\infty = \frac{1}{(1-\alpha)} \tag{1.2}$$

So, for example, if we can speed up 60% of the system to the point where it requires close to no time, our net speedup will still only be 1/0.4 = 2.5×.

Amdahl's law describes a general principle for improving any process. In addition to its application to speeding up computer systems, it can guide a company trying to reduce the cost of manufacturing razor blades, or a student trying to improve his or her grade point average. Perhaps it is most meaningful in the world of computers, where we routinely improve performance by factors of 2 or more. Such high factors can only be achieved by optimizing large parts of a system.

1.9.2 Concurrency and Parallelism

Throughout the history of digital computers, two demands have been constant forces in driving improvements: we want them to do more, and we want them to run faster. Both of these factors improve when the processor does more things at once. We use the term *concurrency* to refer to the general concept of a system with multiple, simultaneous activities, and the term *parallelism* to refer to the use of concurrency to make a system run faster. Parallelism can be exploited at multiple levels of abstraction in a computer system. We highlight three levels here, working from the highest to the lowest level in the system hierarchy.

Thread-Level Concurrency

Building on the process abstraction, we are able to devise systems where multiple programs execute at the same time, leading to *concurrency*. With threads, we can even have multiple control flows executing within a single process. Support for concurrent execution has been found in computer systems since the advent of time-sharing in the early 1960s. Traditionally, this concurrent execution was only *simulated*, by having a single computer rapidly switch among its executing processes, much as a juggler keeps multiple balls flying through the air. This form of concurrency allows multiple users to interact with a system at the same time, such as when many people want to get pages from a single Web server. It also allows a single user to engage in multiple tasks concurrently, such as having a Web browser in one window, a word processor in another, and streaming music playing at the same time. Until recently, most actual computing was done by a single processor, even if that processor had to switch among multiple tasks. This configuration is known as a *uniprocessor system*.

When we construct a system consisting of multiple processors all under the control of a single operating system kernel, we have a *multiprocessor system*. Such systems have been available for large-scale computing since the 1980s, but they have more recently become commonplace with the advent of *multi-core* processors and *hyperthreading*. **Figure 1.16** shows a taxonomy of these different processor types.

Multi-core processors have several CPUs (referred to as "cores") integrated onto a single integrated-circuit chip. **Figure 1.17** illustrates the organization of a

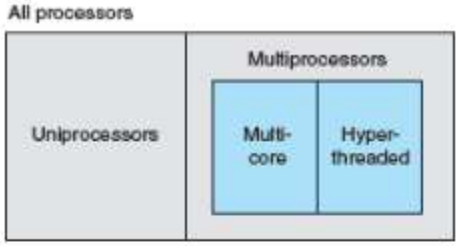


Figure 1.16 Categorizing different processor configurations.

Multiprocessors are becoming prevalent with the advent of multi-core processors and hyperthreading.

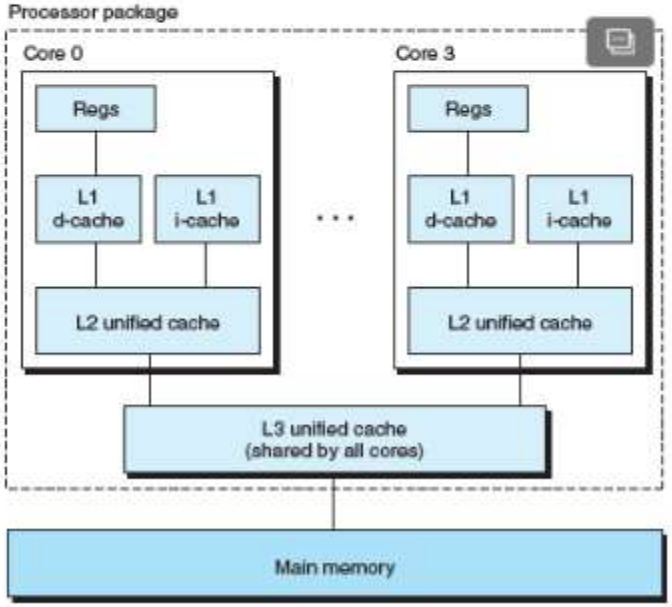


Figure 1.17 Multi-core processor organization.
Four processor cores are integrated onto a single chip.

typical multi-core processor, where the chip has four CPU cores, each with its own L1 and L2 caches, and with each L1 cache split into two parts—one to hold recently fetched instructions and one to hold data. The cores share higher levels of cache as well as the interface to main memory. Industry experts predict that they will be able to have dozens, and ultimately hundreds, of cores on a single chip.

Hyperthreading, sometimes called *simultaneous multi-threading*, is a technique that allows a single CPU to execute multiple flows of control. It involves having multiple copies of some of the CPU hardware, such as program counters and register files, while having only single copies of other parts of the hardware, such as the units that perform floating-point arithmetic. Whereas a conventional processor requires around 20,000 clock cycles to shif between different threads, a hyper threaded processor decides which of its threads to execute on a cycle-by-cycle basis. It enables the CPU to take better advantage of its processing resources. For example, if one thread must wait for some data to be loaded into a cache, the CPU can proceed with the execution of a different thread. As an example, the Intel Core i7 processor can have each core executing two threads, and so a four-core system can actually execute eight threads in parallel.

The use of multiprocessing can improve system performance in two ways. First, it reduces the need to simulate concurrency when performing multiple tasks. As mentioned, even a personal computer being used by a single person is expected to perform many activities concurrently. Second, it can run a single application program faster, but only if that program is expressed in terms of multiple threads that can effectively execute in parallel. Thus although the principles of concurrency have been formulated and studied for over 50 years, the advent of multi-core and hyperthreaded systems has greatly increased the desire to find ways to write application programs tha can exploit the thread-level parallelism available with the hardware. **Chapter 12** will look much more deeply into concurrency and its use to provide a sharing of processing resources and to enable more parallelism in program execution.

Instruction-Level Parallelism

At a much lower level of abstraction, modern processors can execute multiple instructions at one time, a property known as *instruction-level parallelism*. For example, early microprocessors, such as the 1978-vintage Intel 8086, required multiple (typically 3-10) clock cycles to execute a single instruction. More recent processors can sustain execution rates of 2-4 instructions per clock cycle. Any given instruction requires much longer from start to finish, perhaps 20 cycles or more, but the processor uses a number of clever tricks to process as many as 100 instructions at a time. In **Chapter 4**, we will explore the use of *pipelining*, where the actions required to execute an instruction are partitioned into different steps and the processor hardware is organized as a series of stages, each performing one of these steps. The stages can operate in parallel, working on different parts of different instructions. We will see that a fairly simple hardware design can sustain an execution rate close to 1 instruction per clock cycle.

Processors that can sustain execution rates faster than 1 instruction per cycle are known as *superscalar* processors. Most modern processors support superscalar operation. In **Chapter 5**, we will describe a high-level model of such processors. We will see that application programmers can use this model to understand the performance of their programs. They can then write programs such that the generated code achieves higher degrees of instruction-level parallelism and therefore runs faster.

Single-Instruction, Multiple-Data (SIMD) Parallelism

At the lowest level, many modern processors have special hardware that allows a single instruction to cause multiple operations to be performed in parallel, a mode known as *single-instruction, multiple-data*(SIMD) parallelism. For example, recent generations of Intel and AMD processors have instructions that can add 8 pairs of single-precision floating-point numbers (C data type `float`) in parallel.

These SIMD instructions are provided mostly to speed up applications that process image, sound, and video data. Although some compilers attempt to automatically extract SIMD parallelism from C programs, a more reliable method is to write programs using special *vector* data types supported in compilers such as GCC. We describe this style of programming in Web Aside OPT:SIMD, as a supplement to the more general presentation on program optimization found in **Chapter 5**.

1.9.3 The Importance of Abstractions in Computer Systems

The use of *abstractions* is one of the most important concepts in computer science. For example, one aspect of good programming practice is to formulate a simple application program interface (API) for a set of functions that allow programmers to use the code without having to delve into its inner workings. Different programming

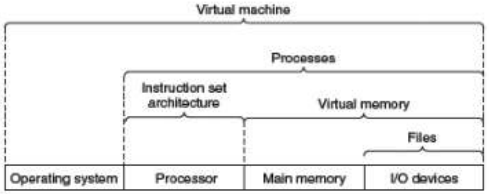


Figure 1.18 Some abstractions provided by a computer system.
A major theme in computer systems is to provide abstract representations at different levels to hide the complexity of the actual implementations.

languages provide different forms and levels of support for abstraction, such as Java class declarations and C function prototypes.

We have already been introduced to several of the abstractions seen in computer systems, as indicated in **Figure 1.18**. On the processor side, the *instruction set architecture* provides an abstraction of the actual processor hardware. With this abstraction, a machine-code program behaves as if it were executed on a processor that performs just one instruction at a time. The underlying hardware is far more elaborate, executing multiple instructions in parallel, but always in a way that is consistent with the simple, sequential model. By keeping the same execution model, different processor implementations can execute the same machine code while offering a range of cost and performance.

On the operating system side, we have introduced three abstractions: *files* as an abstraction of I/O devices, *virtual memory* as an abstraction of program memory, and *processes* as an abstraction of a running program. To these abstractions we add a new one: the *virtual machine*, providing an abstraction of the entire computer, including the operating system, the processor, and the programs. The idea of a virtual machine was introduced by IBM in the 1960s, but it has become more prominent recently as a way to manage computers that must be able to run programs designed for multiple operating systems (such as Microsoft Windows, Mac OS X, and Linux) or different versions of the same operating system.

We will return to these abstractions in subsequent sections of the book.

1.10 Summary

A computer system consists of hardware and systems software that cooperate to run application programs. Information inside the computer is represented as groups of bits that are interpreted in different ways, depending on t context. Programs are translated by other programs into different forms, beginning as ASCII text and then translated by compilers and linkers into binary executable files.

Processors read and interpret binary instructions that are stored in main memory. Since computers spend most of their time copying data between memory, I/O devices, and the CPU registers, the storage devices in a system are arranged in a hierarchy, with the CPU registers at the top, followed by multiple levels of hardware cache memories, DRAM main memory, and disk storage. Storage devices that are higher in the hierarchy are faster and m costly per bit than those lower in the hierarchy. Storage devices that are higher in the hierarchy serve as caches for devices that are lower in the hierarchy. Programmers can optimize the performance of their C programs by understanding and exploiting the memory hierarchy.

The operating system kernel serves as an intermediary between the application and the hardware. It provides three fundamental abstractions: (1) Files are abstractions for I/O devices. (2) Virtual memory is an abstraction for both main memory and disks. (3) Processes are abstractions for the processor, main memory, and I/O devices.

Finally, networks provide ways for computer systems to communicate with one another. From the viewpoint of a particular system, the network is just another I/O device.

2.1 Information Storage

Rather than accessing individual bits in memory, most computers use blocks of 8 bits, or *bytes*, as the smallest addressable unit of memory. A machine-level program views memory as a very large array of bytes, referred to *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 just a conceptual image presented to the machine-level program. The actual implementation (presented in **Chapter 9**) uses a combination of dynamic random access memory (DRAM), flash memory, disk storage, specia hardware, and operating system software to provide the program with what appears to be a monolithic byte array.

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

Aside The evolution of the C programming language

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

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

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

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

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

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

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

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

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

Figure 2.1 Specifying different versions of C to `gcc`.

New to C? The role of pointers in C

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

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

2.1.1 Hexadecimal Notation

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

In C, numeric constants starting with `0x` or `0X` are interpreted as being in hexadecimal. The characters 'A' through 'F' may be written in either upper- or lowercase. For example, we could write the number $FA1D37B_{16}$ as `0xFA1D37B`, as `0xfa1d37b`, or even mixing upper- and 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 2.2**. One simple trick for doing the conversion in your head is to memorize the decimal equivalents of hex digits `A, C`, and `F`.

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

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

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

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

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

This gives the binary representation 000101110011101001001100.

Conversely, given a binary number 11110010101101101100 11, you convert it to hexadecimal by first splitting it into groups of 4 bits each. Note, however, that if the total number of bits is not a multiple of 4, you should make the *leftmost* group be the one with fewer than 4 bits, effectively padding the number with leading zeros. Then you translate each group of bits into the corresponding hexadecimal digit:

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

Practice Problem 2.1 (solution page **143**)

Perform the following number conversions:

A. `0x39A7F8` to binary

B. binary 1100100101111011 to hexadecimal

C. `0xD5E4C` to binary

D. binary 100110111001111010101 to hexadecimal

When a value x is a power of 2, that is, $x = 2^n$ for some nonnegative integer n, we can readily write x in hexadecimal form by remembering that the binary representation of x is simply 1 followed by n zeros. The hexadecimal digit 0 represents 4 binary zeros. So, for n written in the form $i + 4j$, where $0 \le i \le 3$, 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 $x = 2{,}048 = 211$, we have $n = 11 = 3 + 4{\cdot}2$, giving hexadecimal representation `0x800`.

Practice Problem 2.2 (solution page 143)

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

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

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

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

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

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

Practice Problem 2.3 (solution page 144)

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

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

Aside Converting between decimal and hexadecimal

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

> Convert `0xabcd` to decimal

or

> 123 in hex

Decimal	Binary	Hexadecimal
_____	_____	`0x52`
_____	_____	`0xAC`
_____	_____	`0xE7`

Practice Problem 2.4 (solution page **144**)

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

A. `0x503c + 0x8 =` _____

B. `0x503c - 0x40 =` _____

C. `0x503c + 64 =` _____

D. `0x50ea - 0x503c =` _____

2.1.2 Data Sizes

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

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

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

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

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

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

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

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

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

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

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

New to C? Declaring pointers

For any data type *T*, the declaration

```
T *p;
```

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

```
char *p;
```

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

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

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

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

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

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

We will consistently use the forms found in **Figure 2.3**.

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

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

2.1.3 Addressing and Byte Ordering

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

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

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

Big endian

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

Little endian

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

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

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

Aside Origin of “endian”

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

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

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

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

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

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

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

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

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

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

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

```
1 #include <stdio.h>
2
3 typedef unsigned char *byte_pointer;
4
5 void show_bytes(byte_pointer start, size_t len) {
6     int i;
7     for (i = 0; i < len; i++)
8         printf(" %.2x", start[i]);
9     printf("\n");
10 }
11
12 void show_int(int x) {
13     show_bytes((byte_pointer) &x, sizeof(int));
14 }
15
16 void show_float(float x) {
17     show_bytes((byte_pointer) &, sizeof(float));
18 }
19
20 void show_pointer(void *x) {
21     show_bytes((byte_pointer) &, sizeof(void *));
22 }
```

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

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

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

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

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

----------*code/data/show-bytes.c*

```
1 void test_show_bytes(int val) {
2     int ival = val;
3     float fval = (float) ival;
4     int *pval = &ival;
5     show_int(ival);
6     show_float(fval);
7     show_pointer(pval);
8 }
```

----------*code/data/show-bytes.c*

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

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

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

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

New to C? Naming data types with `typedef`

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

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

For example, the declaration

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

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

```
int *ip;
```

New to C? Formatted printing with `printf`

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

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

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

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

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

New to C? Pointers and arrays

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

New to C? Pointer creation and dereferencing

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

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

Aside Generating an ASCII table

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

2.1.5 Representing Code

Consider the following C function:

```
1  int sum(int x, int y) {
2      return x + y;
3  }
```

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

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

Aside The Unicode standard for text encoding

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

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

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

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

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

2.1.6 Introduction to Boolean Algebra

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

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

~	
0	1
1	0

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

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

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

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

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

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

$$\begin{array}{rr} & 0110 \\ \& & 1100 \\ \hline & 0100 \end{array} \qquad \begin{array}{rr} & 0110 \\ | & 1100 \\ \hline & 1110 \end{array} \qquad \begin{array}{rr} & 0110 \\ \hat{} & 1100 \\ \hline & 1010 \end{array} \qquad \begin{array}{rr} \sim & 1100 \\ \hline & 0011 \end{array}$$

Practice Problem 2.8 (solution page 145)

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

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

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

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

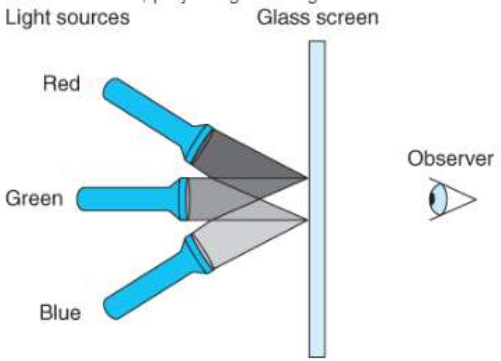
Operation	Result
a	[01101001]
b	[01010101]
$\sim a$	_____
$\sim b$	_____
$a \,\&\, b$	_____
$a \mid b$	_____
$a \,\hat{}\, b$	_____

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

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

Practice Problem 2.9 (solution page **146**)

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



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

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

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

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

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

Blue | Green =_____

Yellow & Cyan =_____

Red ^ Magenta =_____

2.1.7 Bit-Level Operations in C

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

C expression	Binary expression	Binary result	Hexadecimal result
`~0x41`	~[0100 0001]	[1011 1110]	`0xBE`
`~0x00`	~[0000 0000]	[1111 1111]	`0xFF`
`0x69 & 0x55`	[0110 1001] & [0101 0101]	[0100 0001]	`0x41`
`0x69 | 0x55`	[0110 1001] | [01010101]	[0111 1101]	`0x7D`

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

Practice Problem 2.10 (solution page **146**)

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

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

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

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

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

Practice Problem 2.11 (solution page 146)

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

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

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

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

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

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

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

Practice Problem 2.12 (solution page 146)

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

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

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

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

Practice Problem 2.13 (solution page 147)

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

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

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

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

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

2.1.8 Logical Operations in C

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

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

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

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

Practice Problem 2.14 (solution page 147)

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

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

Practice Problem 2.15 (solution page 148)

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

2.1.9 Shift Operations in C

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

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

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

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

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

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

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

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

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

Practice Problem 2.16 (solution page 148)

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

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

Aside Shifting by *k*, for large values of *k*

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

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

Z

Aside Shifting by k, for large values of k

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

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

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

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

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

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

Aside Operator precedence issues with shift operations

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

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

2.2 Integer Representations

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

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

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

Z

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

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

2.2.1 Integral Data Types

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

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

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

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

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

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

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

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

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

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

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

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

2.2.2 Unsigned Encodings

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

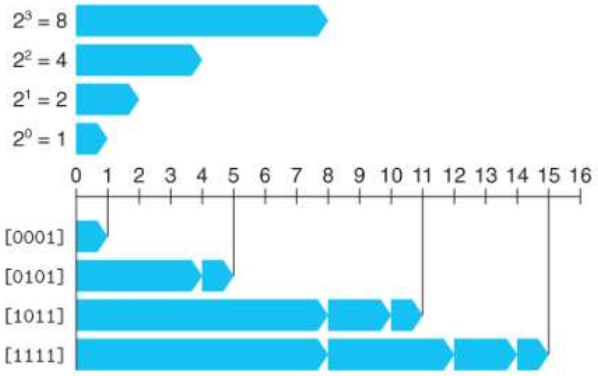


Figure 2.12 Unsigned number examples for
$w = 4$. When bit i in the binary representation has value 1, it contributes 2^i to the value.

Z

Principle:

Definition of unsigned encoding

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

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

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

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

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

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

The unsigned binary representation has the important property that every number between 0 and 2^w — 1 has a unique encoding as a w-bit value. For example, there is only one representation of decimal value 11 as an unsigned 4–bit number—namely, [1011]. We highlight this as a mathematical principle, which we first state and then explain.

Principle:

Uniqueness of unsigned encoding

Function $B2U_w$ is a bijection.

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

2.2.3 Two's-Complement Encodings

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

Principle:

Definition of two's-complement encoding

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

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

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

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

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

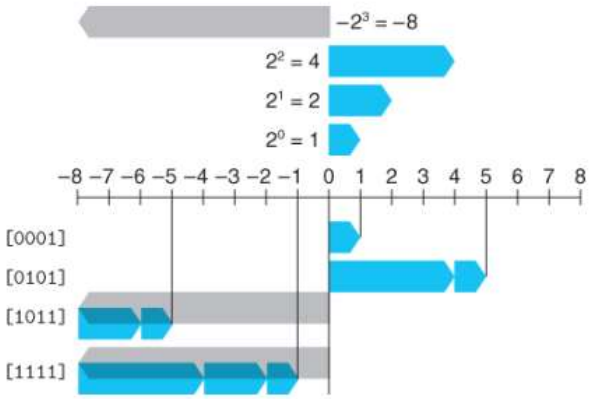


Figure 2.13 Two's-complement number examples for
$w = 4$. Bit 3 serves as a sign bit; when set to 1, it contributes $-2^3 = -8$ to the value. This weighting is shown as a leftward-pointing gray bar.

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

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

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

Principle:

Uniqueness of two's-complement encoding

Function $B2T_w$ is a bijection.

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

Practice Problem 2.17 (solution page **148**)

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

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

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

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

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

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

Aside More on fixed-size integer types

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

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

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

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

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

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

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

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

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

The C standards do not require signed integers to be represented in two's-complement form, but nearly all machines do so. Programmers who are concerned with maximizing portability across all possible machines should no assume any particular range of representable values, beyond the ranges indicated in **Figure 2.11**, nor should they assume any particular representation of signed numbers. On the other hand, many programs are written assuming a two's-complement representation of signed numbers, and the "typical" ranges shown in **Figures 2.9** and **2.10**, and these programs are portable across a broad range of machines and compilers. The file `<limits.h>` in the C library defines a set of constants

Aside Alternative representations of signed numbers

There are two other standard representations for signed numbers:

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

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

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

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

Both of these representations have the curious property that there are two different encodings of the number 0. For both representations, [00 ... 0] is interpreted as +0. The value –0 can be represented in sign-magnitude form as [10 ... 0] and in ones' complement as [11 ... 1]. Although machines based on ones'-complement representations were built in the past, almost all modern machines use two's complement. We will see that sign-magnitude encoding is used with floating-point numbers.

Note the different position of apostrophes: *two's* complement versus *ones'* complement. The term "two's complement" arises from the fact that for nonnegative x we compute a w-bit representation of $-x$ as $2^w - x$ (a single two.) The term "ones' complement" comes from the property that we can compute $-x$ in this notation as [111 ... 1] – x (multiple ones).

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

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

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

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

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

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

Z

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

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

Practice Problem 2.18 (solution page 149)

In Chapter 3, we will look at listings generated by a *disassembler*, a program that converts an executable program file back to a more readable ASCII form. These files contain many hexadecimal numbers, typically representing values in two's-complement form. Being able to recognize these numbers and understand their significance (for example, whether they are negative or positive) is an important skill.
For the lines labeled A–I (on the right) in the following listing, convert the hexadecimal values (in 32-bit two's-complement form) shown to the right of the instruction names (`sub`, `mov`, and `add`) into their decimal equivalents:

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

2.2.4 Conversions between Signed and Unsigned

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

For example, consider the following code:

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

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

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

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

Z

Similarly, consider the following code:

```
1  unsigned u = 4294967295u;    /* UMax */
2  int tu = (int) u;
3  printf("u = %u, tu = %d\n", u, tu);
```

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

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

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

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

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

Pursuing our earlier examples, we see from **Figure 2.15** that $T2U_{16}(-12{,}345) = 53{,}191$, and that $U2T_{16}(53{,}191) = -12{,}345$. That is, the 16-bit pattern written in hexadecimal as `0xCFC7` is both the two's-complement representation of –12,345 and the unsigned representation of 53,191. Note also that $12{,}345 + 53{,}191 = 65{,}536 = 2^{16}$. This property generalizes to a relationship between the two numeric values (two's complement and unsigned) represented by a given bit pattern. Similarly, from **Figure 2.14**, we see that $T2U32(-1) = 4{,}294{,}967{,}295$, and $U2T_{32}(4{,}294{,}967{,}295) = -1$. That is, $UMax$ has the same bit representation in unsigned form as does –1 in two's-complement form. We can also see the relationship between these two numbers: $1 + UMax_w = 2^w$.

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

Practice Problem 2.19 (solution page 149)

Using the table you filled in when solving **Problem 2.17**, fill in the following table describing the function *T2U*$_4$:

x	*T2U*$_4$*(x)*
–8	_____
–3	_____
–2	_____
–1	_____
0	_____
5	_____

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

Principle:

Conversion from two's complement to unsigned

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

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

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

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

Derivation:

Conversion from two's complement to unsigned

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

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

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

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

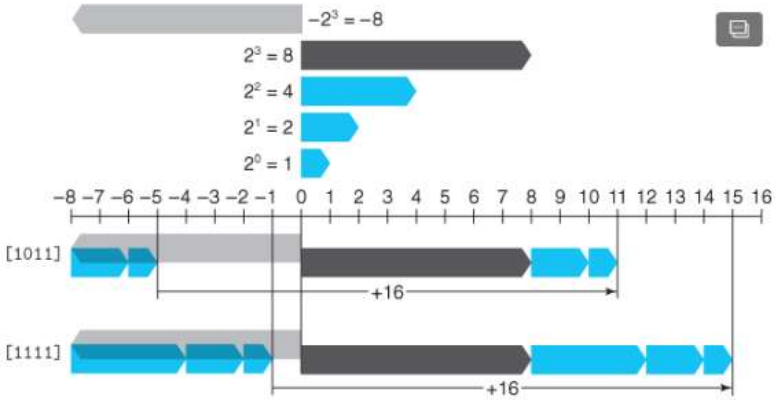


Figure 2.16 Comparing unsigned and two's-complement representations for
$w = 4$. The weight of the most significant bit is –8 for two's complement and +8 for unsigned, yielding a net difference of 16.

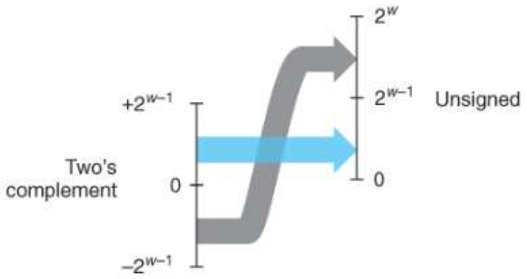


Figure 2.17 Conversion from two's complement to unsigned.
Function *T2U* converts negative numbers to large positive numbers.

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

Practice Problem 2.20 (solution page 149)

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

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

Principle:

Unsigned to two's-complement conversion

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

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

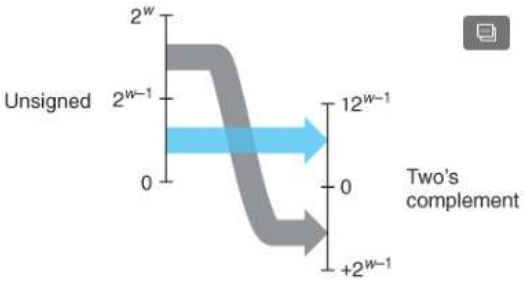


Figure 2.18 Conversion from unsigned to two's complement.
Function $U2T$ converts numbers greater than $TMax_w = 2^{w-1} - 1$ to negative values.

This principle can be justified as follows:

This principle can be justified as follows:

Derivation:

Unsigned to two's-complement conversion

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

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

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

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

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

.

2.2.5 Signed versus Unsigned in C

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

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

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

Z

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

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

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

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

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

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

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

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

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

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

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

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

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

Practice Problem 2.21 (solution page **149**)

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

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

2.2.6 Expanding the Bit Representation of a Number

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

Web Aside DATA:TMIN Writing *TMin* in C

In **Figure 2.19** and in **Problem 2.21**, we carefully wrote the value of $TMin_{32}$ as `-2,147,483,647-1`. Why not simply write it as either `-2,147,483,648` or `0x80000000`? Looking at the C header file `limits.h`, we see that they use a similar method as we have to write $TMin_{32}$ and $TMax_{32}$:

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

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

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

Principle:

Expansion of an unsigned number by zero extension

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

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

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

Principle:

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

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

As an example, consider the following code:

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

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

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

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

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

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

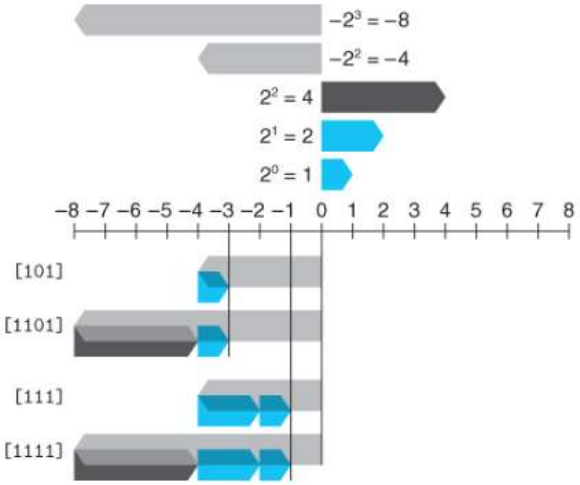


Figure 2.20 Examples of sign extension from $w = 3$ to $w = 4$.
For $w = 4$, the combined weight of the upper 2 bits is $-8 + 4 = -4$, matching that of the sign bit for $w = 3$.

Derivation:

Expansion of a two's-complement number by sign extension Let $w' = w + k$. What we want to prove is that

$$B2T_{w+k}([\underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ times}}, x_{w-2}, \ldots, x_0]) = B2T([x_{w-1}, x_{w-2}, \ldots, x_0])$$

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

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

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

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

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

Practice Problem 2.22 (solution page 150)

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

A. [1011]
B. [11011]
C. [111011]

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

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

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

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

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

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

Practice Problem 2.23 (solution page 150)

Consider the following C functions:

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

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

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

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

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

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

2.2.7 Truncating Numbers

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

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

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

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

Principle:

Truncation of an unsigned number

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

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

Derivation:

Truncation of an unsigned number

Applying the modulus operation to **Equation 2.1** yields

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

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

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

Principle:

Truncation of a two's-complement number

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

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

Derivation:

Truncation of a two's-complement number

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

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

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

Summarizing, the effect of truncation for unsigned numbers is

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

while the effect for two's-complement numbers is

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

Practice Problem 2.24 (solution page 150)

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

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

Explain how **Equations 2.9** and **2.10** apply to these cases.

2.2.8 Advice on Signed versus Unsigned

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

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

Practice Problem 2.25 (solution page 151)

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

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

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

Practice Problem 2.26 (solution page 151)

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

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

Here is your first attempt at the function:

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

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

A. For what cases will this function produce an incorrect result?
B. Explain how this incorrect result comes about.
C. Show how to fix the code so that it will work reliably.

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

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

2.3 Integer Arithmetic

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

2.3.1 Unsigned Addition

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

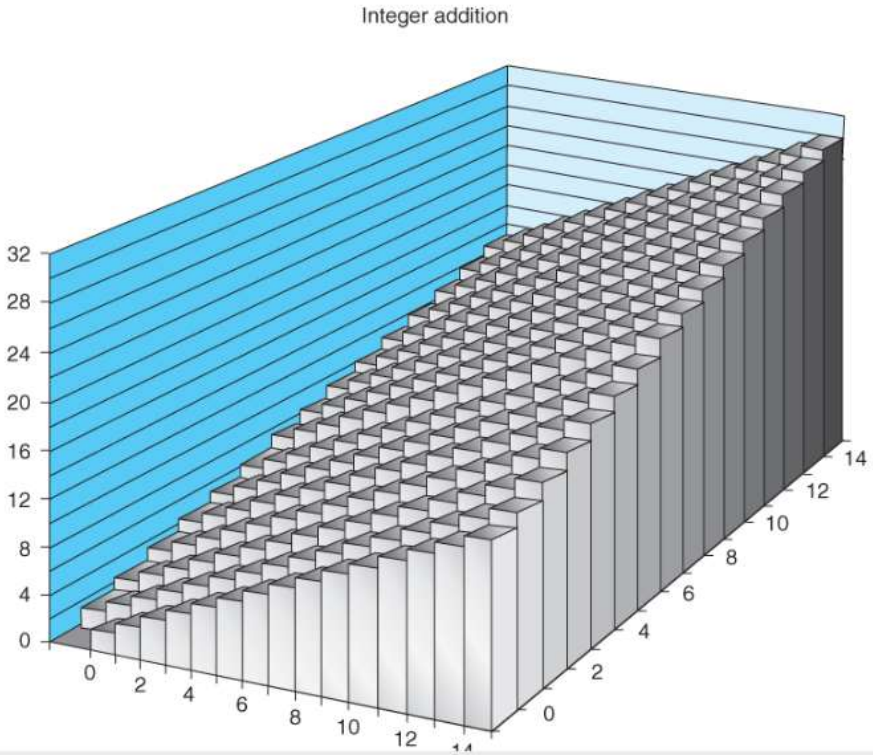


Figure 2.21 Integer addition.
With a 4–bit word size, the sum could require 5 bits.

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

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

Aside Security vulnerability in `getpeername`

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

```
/*
 * Illustration of code vulnerability similar to that found in
 * FreeBSD's implementation of getpeername()
 */

/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);

/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

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

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

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

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

We can characterize operation $+_w^{\mathrm{u}}$ as follows:

Principle:

Unsigned addition

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

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

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

Derivation:

Unsigned addition

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

An arithmetic operation is said to *overflow* when the full integer result cannot fit within the word size limits of the data type. As **Equation 2.11** indicates, overflow occurs when the two operands sum to 2^w or more. **Figure 2.23** shows a plot of the unsigned addition function for word size w = 4. The sum is computed modulo 2^4 = 16. When $x + y < 16$, there is no overflow,

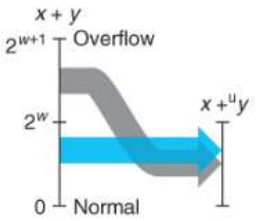


Figure 2.22 Relation between integer addition and unsigned addition.
When $x + y$ is greater than $2^w - 1$, the sum overflows.

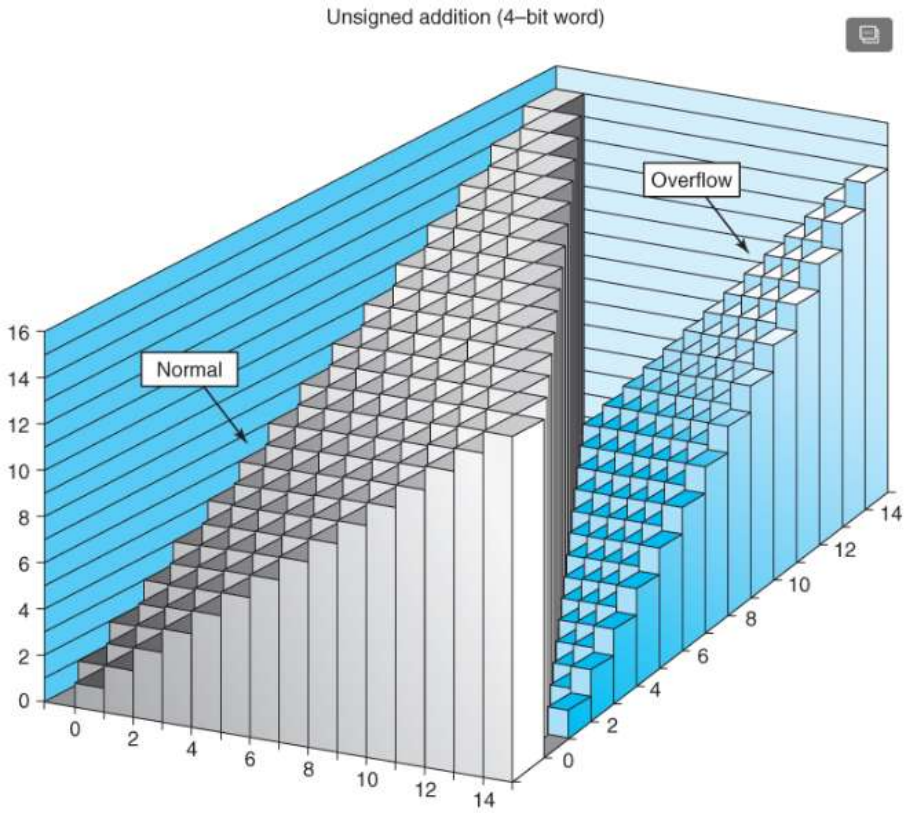


Figure 2.23 Unsigned addition.
With a 4-bit word size, addition is performed modulo 16.

and $x +_4^u y$ is simply $x + y$. This is shown as the region forming a sloping plane labeled "Normal." When $x + y \geq 16$, the addition overflows, having the effect of decrementing the sum by 16. This is shown as the region forming a sloping plane labeled "Overflow."

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

Principle:

Detecting overflow of unsigned addition

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

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

Derivation:

Detecting overflow of unsigned addition

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

Practice Problem 2.27 (solution page 152)

Write a function with the following prototype:

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

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

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

Principle:

Unsigned negation

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

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

This result can readily be derived by case analysis:

Derivation:

Unsigned negation

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

2.3.2 Two's-Complement Addition

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

Principle:

Two's-complement addition

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

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

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

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

Derivation:

Two's-complement addition

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

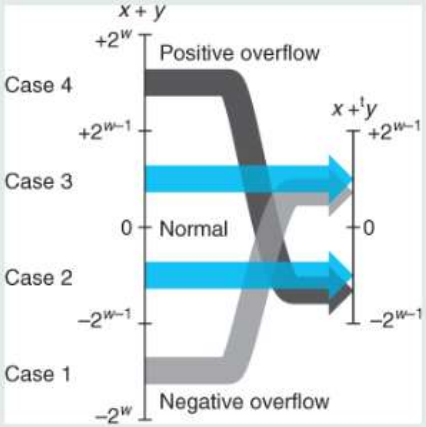


Figure 2.24 Relation between integer and two's-complement addition.

When $x + y$ is less than -2^{w-1}, there is a negative overflow. When it is greater than or equal to 2^{w-1}, there is a positive overflow.

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

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

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

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

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

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

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

Figure 2.25 Two's-complement addition examples.

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

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

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

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

Principle:

Detecting overflow in two's-complement addition

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

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

Derivation:

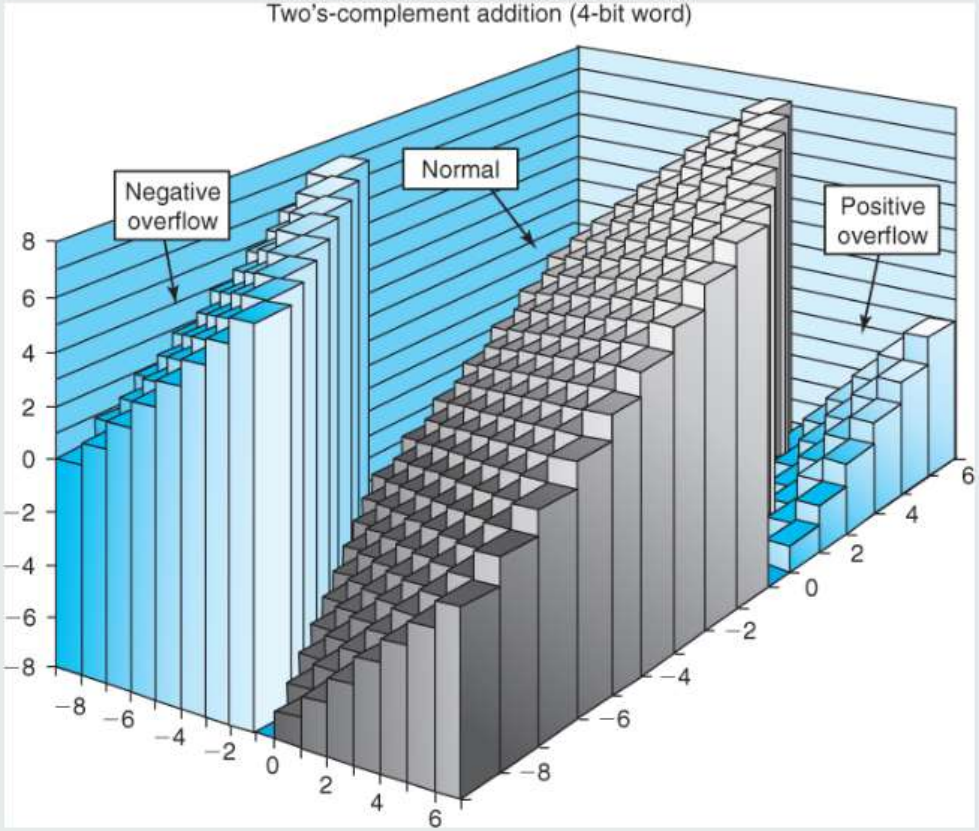


Figure 2.26 Two's-complement addition.

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

Detecting overflow of two's-complement addition

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

2.3.3 Two's-Complement Negation

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

Principle:

Two's-complement negation

For x in the range $TMin_w \leq x \leq TMax_w$, its two's-complement negation $-_w^t x$ is given by the formula

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

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

Derivation:

Two's-complement negation

Observe that $TMin_w + TMin_w = -2\omega - 1 + -2^{w-1} = -2^w$. This would cause negative overflow, and hence $TMin_w +_w^t TMin_w = -2^w + 2^w = 0$. For values of x such that $x > TMin_w$, the value $-x$ can also be represented as a w-bit, two's-complement number, and their sum will be $-x + x = 0$.

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 2*w* 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 2*w*-bit integer product. Let us denote this value as $x \ *_w^{\mathrm{u}} \ y$.

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

Principle:

Unsigned multiplication

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

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

2.3.5 Two's-Complement Multiplication

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

Principle:

Two's-complement multiplication

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

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

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

Principle:

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

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

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

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

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

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

Derivation:

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

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

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

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

$$T2U_w\left(x *_w^t y\right) = T2U_w\left(U2T_w((x \cdot y) \mod 2^w)\right) = (x \cdot y) \mod 2^w$$

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

$$U2B_w\left(T2U_w\left(x *_w^t y\right)\right) = T2B_w\left(x *_w^t y\right) = U2B_w\left(x' *_w^t y'\right)$$

2.3.6 Multiplying by Constants

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

Principle:

Multiplication by a power of 2

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

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

Derivation:

Multiplication by a power of 2

This property can be derived using **Equation 2.1**:

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

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

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

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

Principle:

Unsigned multiplication by a power of 2

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

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

Principle:

Two's-complement multiplication by a power of 2

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

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

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

2.3.7 Dividing by Powers of 2

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

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

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

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

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

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

Principle:

Unsigned division by a power of 2

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

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

Derivation:

Unsigned division by a power of 2

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

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

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

`k`	`>> k` (binary)	Decimal	$-12340/2^k$
0	`1100111111001100`	–12,340	–12,340.0
1	`1110011111100110`	–6,170	–6,170.0
4	`1111110011111100`	–772	–771.25
8	`1111111111001111`	–49	–48.203125

Figure 2.29 Applying arithmetic right shift.

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

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

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

Principle:

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

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

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

Derivation:

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

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

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

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

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

Principle:

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

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

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

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

Derivation:

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

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

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

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

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

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

2.3.8 Final Thoughts on Integer Arithmetic

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

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

2.4 Floating Point

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

> ## Aside The IEEE
>
> The Institute of Electrical and Electronics Engineers (IEEE—pronounced "eye-triple-ee") is a professional society that encompasses all of electronic and computer technology. It publishes journals, sponsors conferences, and sets up committees to define standards on topics ranging from power transmission to software engineering. Another example of an IEEE standard is the 802.11 standard for wireless networking.

numbers very close to 0 ($|V| \ll 1$), and more generally as an approximation to real arithmetic.

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

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

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

2.4.1 Fractional Binary Numbers

A first step in understanding floating-point numbers is to consider binary numbers having fractional values. Let us first examine the more familiar decimal notation. Decimal notation uses a representation of the form

$$d_m d_{m-1} \ldots d_1 d_0 . \; d_{-1} d_{-2} \ldots d_{-n}$$

where each decimal digit d_i ranges between 0 and 9. This notation represents a value d defined as

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

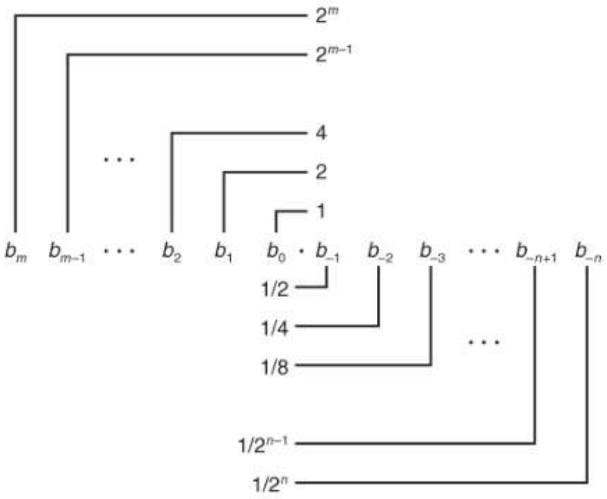


Figure 2.31 Fractional binary representation.
Digits to the left of the binary point have weights of the form 2^i, while those to the right have weights of the form $1/2^i$.

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

By analogy, consider a notation of the form

$$b_m b_{m-1} \ldots b_1 b_0 . \; b_{-1} b_{-2} \ldots b_{-n+1} b_{-n}$$

where each binary digit, or bit, b_i ranges between 0 and 1, as is illustrated in **Figure 2.31**. This notation represents a number b defined as

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

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

One can readily see from **Equation 2.19** that shifting the binary point one position to the left has the effect of dividing the number by 2. For example, while 101.11_2 represents the number $5\frac{3}{4}$, 10.111_2 represents the number $2 + 0 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = 2\frac{7}{8}$. Similarly, shifting the binary point one position to the right has the effect of multiplying the number by 2. For example, 1011.1_2 represents the number $8 + 0 + 2 + 1 + \frac{1}{2} = 11\frac{1}{2}$.

Note that numbers of the form $0.11 \cdots 1_2$ 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, the number $\frac{1}{5}$ can be represented exactly as the fractional decimal number 0.20. As a fractional binary number, however, we cannot represent it exactly and instead must approximate it with increasing accuracy by lengthening the binary representation:

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

2.4.2 IEEE Floating-Point Representation

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

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

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

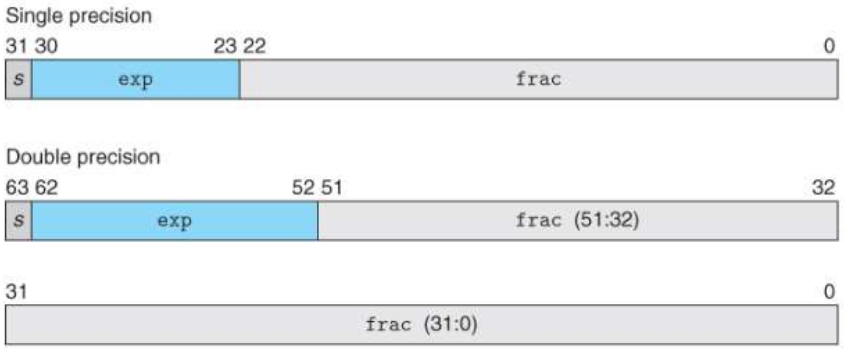


Figure 2.32 Standard floating-point formats.
Floating-point numbers are represented by three fields. For the two most common formats, these are packed in 32-bit (single-precision) or 64-bit (double-precision) words.

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

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

Figure 2.32 shows the packing of these three fields into words for the two most common formats. In the single-precision floating-point format (a `float` in C), fields `s, exp`, and `frac` are 1, $k = 8$, and $n = 23$ bits each, yielding a 32-bit representation. In the double-precision floating-point format (a `double` in C), fields `s, exp`, and `frac` are 1, $k = 11$, and $n = 52$ bits each, yielding a 64-bit representation.

The value encoded by a given bit representation can be divided into three different cases (the latter having two variants), depending on the value of exp. These are illustrated in Figure 2.33 for the single-precision format

Case 1: Normalized Values

This is the most common case. It occurs when the bit pattern of `exp` is neither all zeros (numeric value 0) nor all ones (numeric value 255 for single precision, 2047 for double). In this case, the exponent field is interpreted a 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 precisi 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 $0 \le f < 1$, having binary representation $0.f_{n-1} \cdots f_1 f_0$, that is, with the

> **Aside** Why set the bias this way for denormalized values?
>
> Having the exponent value be $1 - Bias$ rather than simply $-Bias$ might seem counterintuitive. We will see shortly that it provides for smooth transition from denormalized to normalized values.

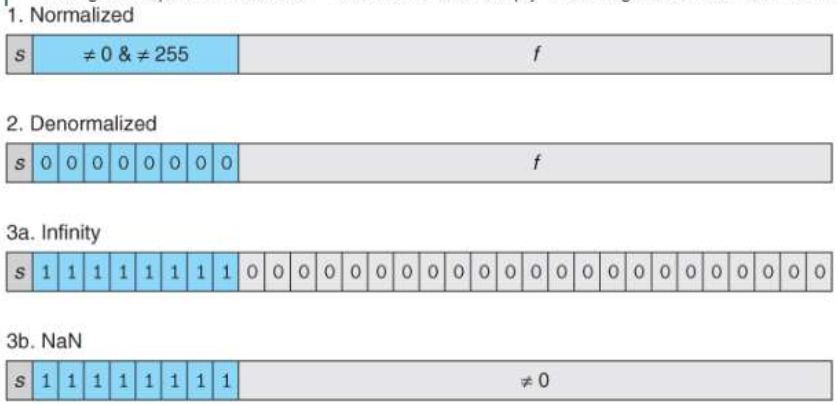


Figure 2.33 Categories of single-precision floating-point values.
The value of the exponent determines whether the number is (1) normalized, (2) denormalized, or (3) a special value.

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

Case 2: Denormalized Values

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

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

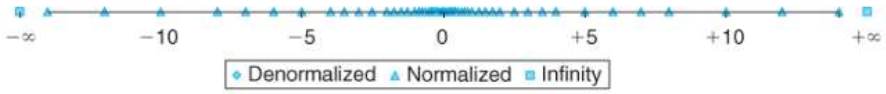
A second function of denormalized numbers is to represent numbers that are very close to 0.0. They provide a property known as *gradual underflow* in which possible numeric values are spaced evenly near 0.0.

Case 3: Special Values

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

2.4.3 Example Numbers

Figure 2.34 shows the set of values that can be represented in a hypothetical 6-bit format having $k = 3$ exponent bits and $n = 2$ fraction bits. The bias is $2^{3-1} - 1 = 3$. Part (a) of the figure shows all representable values (other than *NaN*). The two infinities are at the extreme ends. The normalized numbers with maximum magnitude are ±14. The denormalized numbers are clustered around 0. These can be seen more clearly in part (b) of the figure, where we show just the numbers between –1.0 and +1.0. The two zeros are special cases of denormalized numbers. Observe that the representable numbers are not uniformly distributed—they are denser nearer the origin.



(a) Complete range

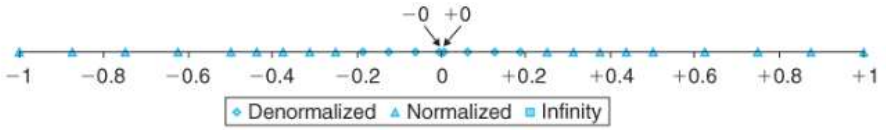


Figure 2.34 Representable values for 6-bit floating-point format.
There are $k = 3$ exponent bits and $n = 2$ fraction bits. The bias is 3.

Figure 2.35 shows some examples for a hypothetical 8-bit floating-point format having $k = 4$ exponent bits and $n = 3$ fraction bits. The bias is $2^{4-1} - 1 = 7$. The figure is divided into three regions representing the three classes of numbers. The different columns show how the exponent field encodes the exponent E, while the fraction field encodes the significand M, and together they form the

		Exponent			Fraction		Value		
Description	Bit representation	e	E	2^E	f	M	$2^E \times M$	V	Decimal
Zero	`0 0000 000`	0	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{0}{8}$	$\frac{0}{512}$	0	0.0
Smallest positive	`0 0000 001`	0	−6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$	$\frac{1}{512}$	0.001953
	`0 0000 010`	0	−6	$\frac{1}{64}$	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$	$\frac{2}{256}$	0.003906
	`0 0000 011`	0	−6	$\frac{1}{64}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$	$\frac{3}{512}$	0.005859
	⋮								
Largest denormalized	`0 0000 111`	0	−6	$\frac{1}{64}$	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$	$\frac{7}{512}$	0.013672
Smallest normalized	`0 0001 000`	1	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{512}$	$\frac{1}{64}$	0.015625
	`0 0001 001`	1	−6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$	$\frac{9}{512}$	0.017578
	⋮								
	`0 0110 110`	6	−1	$\frac{1}{2}$	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$	$\frac{7}{8}$	0.875
	`0 0110 111`	6	−1	$\frac{1}{2}$	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$	$\frac{15}{16}$	0.9375
One	`0 0111 000`	7	0	1	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{8}$	1	1.0
	`0 0111 001`	7	0	1	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	1.125
	`0 0111 010`	7	0	1	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$	$\frac{5}{4}$	1.25
	⋮								
	`0 1110 110`	14	7	128	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{1792}{8}$	224	224.0
Largest normalized	`0 1110 111`	14	7	128	$\frac{6}{8}$	$\frac{15}{8}$	$\frac{1920}{8}$	240	240.0
Infinity	`0 1111 000`	—	—	—	—	—	—	∞	—

Figure 2.35 Example nonnegative values for 8-bit floating-point format.

There are $k = 4$ exponent bits and $n = 3$ fraction bits. The bias is 7.

represented value $V = 2^E \times M$. Closest to 0 are the denormalized numbers, starting with 0 itself. Denormalized numbers in this format have $E = 1 - 7 = -6$, giving a weight $2^E = \frac{1}{64}$. The fractions f and significands M range over the values 0, $\frac{1}{8}, \ldots, \frac{7}{8}$, giving numbers V in the range 0 to $\frac{1}{64} \times \frac{7}{8} = \frac{7}{512}$.

The smallest normalized numbers in this format also have $E = 1 - 7 = -6$, and the fractions also range over the values 0, $\frac{1}{8}, \ldots \frac{7}{8}$. However, the significands then range from 1 + 0 = 1 to $1 + \frac{7}{8} = \frac{15}{8}$, giving numbers V in t range $\frac{8}{512} = \frac{1}{64}$ to $\frac{15}{512}$.

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

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

One interesting property of this representation is that if we interpret the bit representations of the values in **Figure 2.35** as unsigned integers, they occur in ascending order, as do the values they represent as floating-poir numbers. This is no accident—the IEEE format was designed so that floating-point numbers could be sorted using an integer sorting routine. A minor difficulty occurs when dealing with negative numbers, since they have a leading 1 and occur in descending order, but this can be overcome without requiring floating-point operations to perform comparisons (see **Problem 2.84**).

2.4.4 Rounding

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

Figure 2.37 illustrates the four rounding modes applied to the problem of rounding a monetary amount to the nearest whole dollar. Round-to-even (also called round-to-nearest) is the default mode. It attempts to find a closest match. Thus, it rounds $1.40 to $1 and $1.60 to $2, since these are the closest whole dollar values. The only design decision is to determine the effect of rounding values that are halfway between two possible results. Round-to-even mode adopts the convention that it rounds the number either upward or downward such that the least significant digit of the result is even. Thus, it rounds both $1.50 and $2.50 to $2.

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

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

Figure 2.37 Illustration of rounding modes for dollar rounding.
The first rounds to a nearest value, while the other three bound the result above or below.

that $|\widehat{x}| \leq |x|$. Round-down mode rounds both positive and negative numbers downward, giving a value x^- such that $x^- \leq x$. Round-up mode rounds both positive and negative numbers upward, giving a value x^+ such that $x \leq x^+$.

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

Round-to-even rounding can be applied even when we are not rounding to a whole number. We simply consider whether the least significant digit is even or odd. For example, suppose we want to round decimal numbers to the nearest hundredth. We would round 1.2349999 to 1.23 and 1.2350001 to 1.24, regardless of rounding mode, since they are not halfway between 1.23 and 1.24. On the other hand, we would round both 1.2350000 and 1.2450000 to 1.24, since 4 is even.

Similarly, round-to-even rounding can be applied to binary fractional numbers. We consider least significant bit value 0 to be even and 1 to be odd. In general, the rounding mode is only significant when we have a bit pattern o the form $XX\cdots X.YY\cdots Y100\cdots$, where X and Y denote arbitrary bit values with the rightmost Y being the position to which we wish to round. Only bit patterns of this form denote values that are halfway between two possible results. As examples, consider the problem of rounding values to the nearest quarter (i.e., 2 bits to the right of the binary point.) We would round $10.00011_2(2\frac{3}{32})$ down to 10.00_2 (2), and $10.00110_2(2\frac{3}{16})$ up to $10.01_2(2\frac{1}{4})$, because these values are not halfway between two possible values. We would round $10.11100_2(2\frac{7}{8})$ up to 11.00_2 (3) and $10.10100_2(2\frac{5}{8})$ down to $10.10_2(2\frac{1}{2})$, since these values are halfway between two possible results, and we prefer to have the least significant bit equal to zero.

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 -∞, while 1/+0 is defined to yield +∞.

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

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

Floating-point multiplication also obeys many of the properties one normally associates with multiplication. Let us define $x *^{\mathrm{f}} y$ to be *Round(x* × *y)*. This operation is closed under multiplication (although possibly yielding infini 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 floatin point, the expression `(1e20*1e20)*1e-20 evaluates to +∞`, while `1e20*(1e20*1e-20)` evaluates to `1e20`. In addition, floating-point multiplication does not distribute over addition. For example, with single-precision floating point, the expression `1e20*(1e20-1e20)` evaluates to `0.0`, while `1e20*1e20-1e20*1e20` evaluates to `NaN`.

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

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

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

All versions of C provide two different floating-point data types: `float` and `double`. On machines that support IEEE floating point, these data types correspond to single- and double-precision floating point. In addition, the machines use the round-to-even rounding mode. Unfortunately, since the C standards do not require the machine to use IEEE floating point, there are no standard methods to change the rounding mode or to get special values such as –0, +∞, –∞, or *NaN*. Most systems provide a combination of include `(. h)` files and procedure libraries to provide access to these features, but the details vary from one system to another. For example, the GNU compiler GCC defines program constants `INFINITY` (for +∞) and `NAN` (for *NaN*) when the following sequence occurs in the program file:

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

Practice Problem 2.53 (solution page **160**)

Fill in the following macro definitions to generate the double-precision values +∞, –∞, and –0:

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

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

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

- From `int` to `float`, the number cannot overflow, but it may be rounded.
- From `int` or `float` to `double`, the exact numeric value can be preserved because `double` has both greater range (i.e., the range of representable values), as well as greater precision (i.e., the number of significant bits).
- From `double` to `float`, the value can overflow to +∞ or –∞, since the range is smaller. Otherwise, it may be rounded, because the precision is smaller.
- From `float` or `double` to `int`, the value will be rounded toward zero. For example, 1.999 will be converted to 1, while –1.999 will be converted to –1. Furthermore, the value may overflow. The C standards do not specify a fixed result for this case. Intel-compatible microprocessors designate the bit pattern [10 ... 00] ($TMin_w$ for word size w) as an *integer indefinite* value. Any conversion from floating point to integer that cannot assign a reasonable integer approximation yields this value. Thus, the expression `(int) +1e10` yields `-21483648`, generating a negative value from a positive one.