

mers often followed a simple credo: Minimize memory space to make programs fast. In the last decade, advances in computer design and memory technology have greatly reduced the importance of small memory size. Programmers interested in performance now need to understand the issues that have replaced the simple memory model of the 1960s: the hierarchical nature of memories and the parallel nature of processors. Programmers who seek to build competitive versions of compilers, operating systems, databases, and even applications will therefore need to increase their knowledge of computer organization.

We are honored to have the opportunity to explain what's inside this revolutionary machine, unraveling the software below your program and the hardware under the covers of your computer. By the time you finish this book, you will understand the secrets of programming a computer in its native tongue, the internal organization of computers and how it affects performance of your programs, and even how you could go about designing a computer of your very own.

This first chapter lays the foundation for the rest of the book. It introduces the basic ideas and definitions, places the major components of software and hardware in perspective, and introduces integrated circuits, the technology that fuels the computer revolution.

1.2

Below Your Program

In Paris they simply stared when I spoke to them in French; I never did succeed in making those idiots understand their own language.

Mark Twain, *The Innocents Abroad*, 1869

To actually speak to an electronic machine, you need to send electrical signals. The easiest signals for machines to understand are *on* and *off*, and so the machine alphabet is just two letters. Just as the 26 letters of the English alphabet do not limit how much can be written, the two letters of the computer alphabet do not limit what computers can do. The two symbols for these two letters are the numbers 0 and 1, and we commonly think of the machine language as numbers in base 2, or *binary numbers*. We refer to each "letter" as a *binary digit* or *bit*. Computers are slaves to our commands; hence, the name for an individual command is *instruction*. Instructions, which are just collections of bits that the computer understands, can be thought of as numbers. For example, the bits

1000110010100000

tell one computer to add two numbers. Chapter 3 explains why we use numbers for instructions *and* data; we don't want to steal that chapter's thunder, but using numbers for both instructions and data is a foundation of computing.

The first programmers communicated to computers in binary numbers, but this was so tedious that they quickly invented new notations that were closer to the way humans think. At first these notations were translated to binary by hand, but this process was still tiresome. Using the machine to help program the machine, the pioneers invented programs to translate from symbolic notation to binary. The first of these programs was named an *assembler*. This program translates a symbolic version of an instruction into the binary version. For example, the programmer would write

```
add A,B
```

and the assembler would translate this notation into

```
1000110010100000
```

This instruction tells the computer to add the two numbers A and B. The name coined for this symbolic language, still used today, is *assembly language*.

Although a tremendous improvement, assembly language is still far from the notation a scientist might like to use to simulate fluid flow or that an accountant might use to balance the books. Assembly language requires the programmer to write one line for every instruction that the machine will follow, forcing the programmer to think like the machine.

Such low-level thinking inspired a simple question: If we can write a program to translate from assembly language to binary instructions to simplify programming, what prevents us from writing a program that translates from some higher-level notation down to assembly language?

The answer was: nothing. Although more challenging to create than an assembler, this higher-level translator was plausible.

Programmers today owe their productivity—and their sanity—to this observation. Programs that accept this more natural notation are called *compilers*, and the languages they *compile* are called *high-level programming languages*. They enable a programmer to write this high-level language expression:

```
A + B
```

The compiler would compile it into this assembly language statement:

```
add A,B
```

The assembler would translate this statement into the binary instruction that tells the computer to add the two numbers A and B:

```
1000110010100000
```

Figure 1.1 shows the relationships among these programs and languages.

High-level
language
program
(in C)

```
swap(int v[], int k)
{int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

C compiler

Assembly
language
program
(for MIPS)

```
swap:
  muli $2, $5, 4
  add  $2, $4, $2
  lw   $15, 0($2)
  lw   $16, 4($2)
  sw   $16, 0($2)
  sw   $15, 4($2)
  jr   $31
```

Assembler

Binary machine
language
program
(for MIPS)

```
000000001010000100000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
100011001111001000000000000000100
101011001111001000000000000000000
101011000110001000000000000000100
00000011111000000000000000001000
```

FIGURE 1.1 C program compiled into assembly language and then assembled into binary machine language. Although the translation from high-level language to binary machine language is shown in two steps, some compilers cut out the middleman and produce binary machine language directly. These languages and this program are examined in more detail in Chapter 3. (Web Extension III, available at www.mkp.com/cod2e.htm, explains C to Pascal programmers.)

WEB
ENHANCED

High-level programming languages offer several important benefits. First, they allow the programmer to think in a more natural language, using English words and algebraic notation, resulting in programs that look much more like text than like tables of cryptic symbols (see Figure 1.1). Moreover, they allow

languages to be designed according to their intended use. Hence, Fortran was designed for scientific computation, Cobol for business data processing, Lisp for symbol manipulation, and so on.

The second advantage of programming languages is improved programmer productivity. One of the few areas of widespread agreement in software development is that it takes less time to develop programs when they are written in languages that require fewer lines to express an idea. Conciseness is a clear advantage of high-level languages over assembly language.

→ The final advantage is that programming languages allow programs to be independent of the computer on which they were developed, since compilers and assemblers can translate high-level language programs to the binary instructions of any machine. These three advantages are so strong that today little programming is done in assembly language.

As programming matured, many of its practitioners saw that reusing programs was much more efficient than writing everything from scratch. Hence programmers began to pool potentially widely used routines into libraries. One of the first of these *subroutine libraries* was for inputting and outputting data, which included, for example, routines to control printers, such as ensuring paper is in the printer before printing can begin. Such software controlled other input/output devices, such as magnetic disks, magnetic tapes, and displays.

It soon became apparent that a set of programs could be run more efficiently if there was a separate program that supervised running those programs. As soon as one program completed, the supervising program would start the next program in the queue, thereby avoiding delays. These supervising programs, which soon included the input/output subroutine libraries, are the basis for what we call *operating systems* today. Operating systems are programs that manage the resources of a computer for the benefit of the programs that run on that machine.

Software came to be categorized by its use. Software that provides services that are commonly useful is called systems software. Operating systems, compilers, and assemblers are examples of systems software. In contrast to programs aimed at programmers, applications software, or just *applications*, is the name given to programs aimed at computer users, such as spreadsheets or text editors. Figure 1.2 shows the classical drawing mapping the hierarchical layers of software and hardware.

This simplified view has some problems. Should we really place compilers in the systems software level in Figure 1.2? Compilers produce programs at both the applications and the systems level, and applications programs don't normally call on the compiler while they are running. A more realistic view of the nature of systems appears in Figure 1.3. It shows that software does not consist of monolithic layers, but is composed of many programs that build on one another. Like the strands of a thick rope, each time you look carefully at what appears to be a single strand, you find it is really composed of many finer components.

systems software:

OS
compilers
assemblers

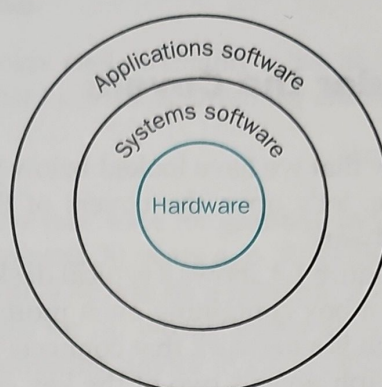


FIGURE 1.2 A simplified view of hardware and software as hierarchical layers, classically shown as concentric rings building up from the core of hardware to the software closest to the user.

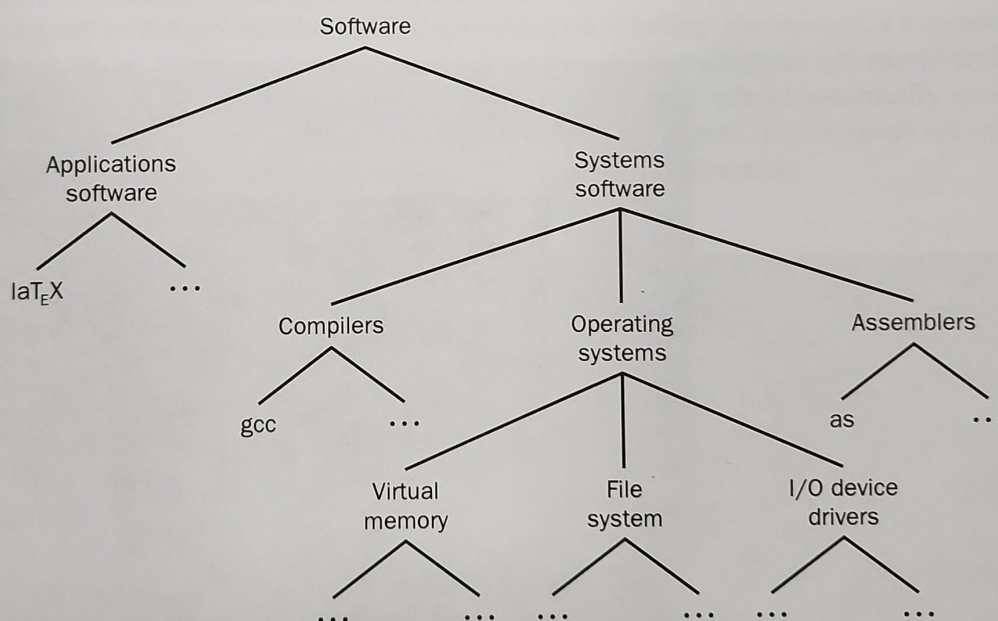


FIGURE 1.3 An example of the decomposability of computer systems. The terms in the middle of the chart, such as `laTeX` and `gcc`, are examples of Unix programs. The terms lower in the chart, such as virtual memory, will be introduced in Chapters 7 and 8.