1

Computer Abstractions and Technology

Civilization advances by extending the number of important operations which we can perform without thinking about them.

Alfred North Whitehead
An Introduction to Mathematics, 1911

Computer Organization and Design. DOI: 10.1016/B978-0-12-374750-1.00001-3

1.1 Introduction

Welcome to this book! We're delighted to have this opportunity to convey the excitement of the world of computer systems. This is not a dry and dreary field, where progress is glacial and where new ideas atrophy from neglect. No! Computers are the product of the incredibly vibrant information technology industry, all aspects of which are responsible for almost 10% of the gross national product of the United States, and whose economy has become dependent in part on the rapid improvements in information technology promised by Moore's law. This unusual industry embraces innovation at a breathtaking rate. In the last 25 years, there have been a number of new computers whose introduction appeared to revolutionize the computing industry; these revolutions were cut short only because someone else built an even better computer.

This race to innovate has led to unprecedented progress since the inception of electronic computing in the late 1940s. Had the transportation industry kept pace with the computer industry, for example, today we could travel from New York to London in about a second for roughly a few cents. Take just a moment to contemplate how such an improvement would change society—living in Tahiti while working in San Francisco, going to Moscow for an evening at the Bolshoi Ballet—and you can appreciate the implications of such a change.

Computers have led to a third revolution for civilization, with the information revolution taking its place alongside the agricultural and the industrial revolutions. The resulting multiplication of humankind's intellectual strength and reach naturally has affected our everyday lives profoundly and changed the ways in which the search for new knowledge is carried out. There is now a new vein of scientific investigation, with computational scientists joining theoretical and experimental scientists in the exploration of new frontiers in astronomy, biology, chemistry, and physics, among others.

The computer revolution continues. Each time the cost of computing improves by another factor of 10, the opportunities for computers multiply. Applications that were economically infeasible suddenly become practical. In the recent past, the following applications were "computer science fiction."

- *Computers in automobiles:* Until microprocessors improved dramatically in price and performance in the early 1980s, computer control of cars was ludicrous. Today, computers reduce pollution, improve fuel efficiency via engine controls, and increase safety through the prevention of dangerous skids and through the inflation of air bags to protect occupants in a crash.
- *Cell phones:* Who would have dreamed that advances in computer systems would lead to mobile phones, allowing person-to-person communication almost anywhere in the world?
- *Human genome project:* The cost of computer equipment to map and analyze human DNA sequences is hundreds of millions of dollars. It's unlikely that anyone would have considered this project had the computer costs been 10 to 100 times higher, as they would have been 10 to 20 years ago. Moreover, costs continue to drop; you may be able to acquire your own genome, allowing medical care to be tailored to you.
- *World Wide Web:* Not in existence at the time of the first edition of this book, the World Wide Web has transformed our society. For many, the WWW has replaced libraries.
- *Search engines:* As the content of the WWW grew in size and in value, finding relevant information became increasingly important. Today, many people rely on search engines for such a large part of their lives that it would be a hardship to go without them.

Clearly, advances in this technology now affect almost every aspect of our society. Hardware advances have allowed programmers to create wonderfully useful software, which explains why computers are omnipresent. Today's science fiction suggests tomorrow's killer applications: already on their way are virtual worlds, practical speech recognition, and personalized health care.

Classes of Computing Applications and Their Characteristics

Although a common set of hardware technologies (see Sections 1.3 and 1.7) is used in computers ranging from smart home appliances to cell phones to the largest supercomputers, these different applications have different design requirements and employ the core hardware technologies in different ways. Broadly speaking, computers are used in three different classes of applications.

Desktop computers are possibly the best-known form of computing and are characterized by the personal computer, which readers of this book have likely used extensively. Desktop computers emphasize delivery of good performance to single users at low cost and usually execute third-party software. The evolution of many computing technologies is driven by this class of computing, which is only about 30 years old!

Servers are the modern form of what were once mainframes, minicomputers, and supercomputers, and are usually accessed only via a network. Servers are oriented to carrying large workloads, which may consist of either single complex applications—usually a scientific or engineering application—or handling many small jobs, such as would occur in building a large Web server. These applications are usually based on software from another source (such as a database or simulation system), but are often modified or customized for a particular function. Servers are built from the same basic technology as desktop computers, but provide for greater expandability of both computing and input/output capacity. In general, servers also place a greater emphasis on dependability, since a crash is usually more costly than it would be on a single-user desktop computer.

Servers span the widest range in cost and capability. At the low end, a server may be little more than a desktop computer without a screen or keyboard and cost a thousand dollars. These low-end servers are typically used for file storage, small business applications, or simple Web serving (see Section 6.10). At the other extreme are **supercomputers**, which at the present consist of hundreds to thousands of processors and usually **terabytes** of memory and **petabytes** of storage, and cost millions to hundreds of millions of dollars. Supercomputers are usually used for high-end scientific and engineering calculations, such as weather forecasting, oil exploration, protein structure determination, and other large-scale problems. Although such supercomputers represent the peak of computing capability, they represent a relatively small fraction of the servers and a relatively small fraction of the overall computer market in terms of total revenue.

Although not called supercomputers, Internet **datacenters** used by companies like eBay and Google also contain thousands of processors, terabytes of memory, and petabytes of storage. These are usually considered as large clusters of computers (see Chapter 7).

Embedded computers are the largest class of computers and span the widest range of applications and performance. Embedded computers include the

desktop computer A computer designed for use by an individual, usually incorporating a graphics display, a keyboard, and a mouse.

server A computer used for running larger programs for multiple users, often simultaneously, and typically accessed only via a network.

supercomputer A class of computers with the highest performance and cost; they are configured as servers and typically cost millions of dollars.

terabyte Originally 1,099,511,627,776 (2^{40}) bytes, although some communications and secondary storage systems have redefined it to mean 1,000,000,000,000 (10^{12}) bytes.

petabyte Depending on the situation, either 1000 or 1024 terabytes.

datacenter A room or building designed to handle the power, cooling, and networking needs of a large number of servers.

embedded computer A computer inside another device used for running one predetermined application or collection of software.

microprocessors found in your car, the computers in a cell phone, the computers in a video game or television, and the networks of processors that control a modern airplane or cargo ship. Embedded computing systems are designed to run one application or one set of related applications, that are normally integrated with the hardware and delivered as a single system; thus, despite the large number of embedded computers, most users never really see that they are using a computer!

Figure 1.1 shows that during the last several years, the growth in cell phones that rely on embedded computers has been much faster than the growth rate of desktop computers. Note that the embedded computers are also found in digital TVs and set-top boxes, automobiles, digital cameras, music players, video games, and a variety of other such consumer devices, which further increases the gap between the number of embedded computers and desktop computers.

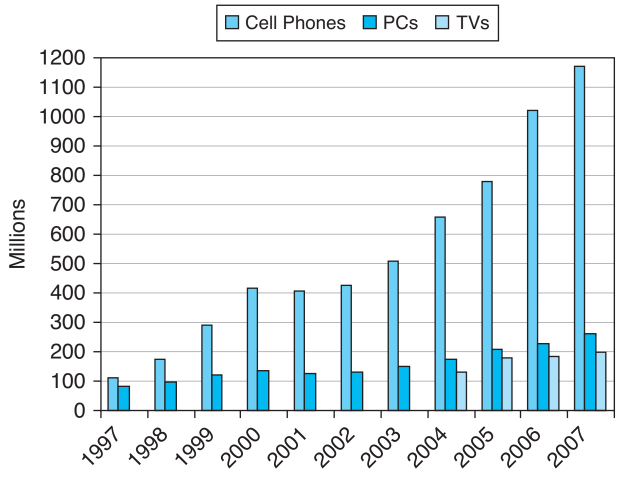


FIGURE 1.1 The number of cell phones, personal computers, and televisions manufactured per year between 1997 and 2007. (We have television data only from 2004.) More than a billion new cell phones were shipped in 2006. Cell phones sales exceeded PCs by only a factor of 1.4 in 1997, but the ratio grew to 4.5 in 2007. The total number in use in 2004 is estimated to be about 2.0B televisions, 1.8B cell phones, and 0.8B PCs. As the world population was about 6.4B in 2004, there were approximately one PC, 2.2 cell phones, and 2.5 televisions for every eight people on the planet. A 2006 survey of U.S. families found that they owned on average 12 gadgets, including three TVs, 2 PCs, and other devices such as game consoles, MP3 players, and cell phones.

Embedded applications often have unique application requirements that combine a minimum performance with stringent limitations on cost or power. For example, consider a music player: the processor need only be as fast as necessary to handle its limited function, and beyond that, minimizing cost and power are the most important objectives. Despite their low cost, embedded computers often have lower tolerance for failure, since the results can vary from upsetting (when your new television crashes) to devastating (such as might occur when the computer in a plane or cargo ship crashes). In consumer-oriented embedded applications, such as a digital home appliance, dependability is achieved primarily through simplicity—the emphasis is on doing one function as perfectly as possible. In large embedded systems, techniques of redundancy from the server world are often employed (see Section 6.9). Although this book focuses on general-purpose computers, most concepts apply directly, or with slight modifications, to embedded computers.

Elaboration: Elaborations are short sections used throughout the text to provide more detail on a particular subject that may be of interest. Disinterested readers may skip over an elaboration, since the subsequent material will never depend on the contents of the elaboration.

Many embedded processors are designed using *processor cores*, a version of a processor written in a hardware description language, such as Verilog or VHDL (see Chapter 4). The core allows a designer to integrate other application-specific hardware with the processor core for fabrication on a single chip.

What You Can Learn in This Book

Successful programmers have always been concerned about the performance of their programs, because getting results to the user quickly is critical in creating successful software. In the 1960s and 1970s, a primary constraint on computer performance was the size of the computer's memory. Thus, programmers 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 in most applications other than those in embedded computing systems.

Programmers interested in performance now need to understand the issues that have replaced the simple memory model of the 1960s: the parallel nature of processors and the hierarchical nature of memories. 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 complete this book, we believe you will be able to answer the following questions:

- How are programs written in a high-level language, such as C or Java, translated into the language of the hardware, and how does the hardware execute the resulting program? Comprehending these concepts forms the basis of understanding the aspects of both the hardware and software that affect program performance.
- What is the interface between the software and the hardware, and how does software instruct the hardware to perform needed functions? These concepts are vital to understanding how to write many kinds of software.
- What determines the performance of a program, and how can a programmer improve the performance? As we will see, this depends on the original program, the software translation of that program into the computer's language, and the effectiveness of the hardware in executing the program.
- What techniques can be used by hardware designers to improve performance? This book will introduce the basic concepts of modern computer design. The interested reader will find much more material on this topic in our advanced book, *Computer Architecture: A Quantitative Approach*.
- What are the reasons for and the consequences of the recent switch from sequential processing to parallel processing? This book gives the motivation, describes the current hardware mechanisms to support parallelism, and surveys the new generation of **"multicore" microprocessors** (see Chapter 7).

multicore microprocessor A microprocessor containing multiple processors ("cores") in a single integrated circuit.

Without understanding the answers to these questions, improving the performance of your program on a modern computer, or evaluating what features might make one computer better than another for a particular application, will be a complex process of trial and error, rather than a scientific procedure driven by insight and analysis.

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, shows how to evaluate performance and power, introduces integrated circuits (the technology that fuels the computer revolution), and explains the shift to multicores.

In this chapter and later ones, you will likely see many new words, or words that you may have heard but are not sure what they mean. Don't panic! Yes, there is a lot of special terminology used in describing modern computers, but the terminology actually helps, since it enables us to describe precisely a function or capability. In addition, computer designers (including your authors) *love* using **acronyms**, which are *easy* to understand once you know what the letters stand for! To help you remember and locate terms, we have included a **highlighted** definition of every term in the margins the first time it appears in the text. After a short time of working with the terminology, you will be fluent, and your friends will be impressed as you correctly use acronyms such as BIOS, CPU, DIMM, DRAM, PCIE, SATA, and many others.

acronym A word constructed by taking the initial letters of a string of words. For example: **RAM** is an acronym for Random Access Memory, and **CPU** is an acronym for Central Processing Unit.

To reinforce how the software and hardware systems used to run a program will affect performance, we use a special section, *Understanding Program Performance*, throughout the book to summarize important insights into program performance. The first one appears below.

Understanding Program Performance

The performance of a program depends on a combination of the effectiveness of the algorithms used in the program, the software systems used to create and translate the program into machine instructions, and the effectiveness of the computer in executing those instructions, which may include input/output (I/O) operations. This table summarizes how the hardware and software affect performance.

Hardware or software component	How this component affects performance	Where is this topic covered?
Algorithm	Determines both the number of source-level statements and the number of I/O operations executed	Other books!
Programming language, compiler, and architecture	Determines the number of computer instructions for each source-level statement	Chapters 2 and 3
Processor and memory system	Determines how fast instructions can be executed	Chapters 4, 5, and 7
I/O system (hardware and operating system)	Determines how fast I/O operations may be executed	Chapter 6

Check Yourself

Check Yourself sections are designed to help readers assess whether they comprehend the major concepts introduced in a chapter and understand the implications of those concepts. Some *Check Yourself* questions have simple answers; others are for discussion among a group. Answers to the specific questions can be found at the end of the chapter. *Check Yourself* questions appear only at the end of a section, making it easy to skip them if you are sure you understand the material.

1. Section 1.1 showed that the number of embedded processors sold every year greatly outnumbers the number of desktop processors. Can you confirm or deny this insight based on your own experience? Try to count the number of embedded processors in your home. How does it compare with the number of desktop computers in your home?
2. As mentioned earlier, both the software and hardware affect the performance of a program. Can you think of examples where each of the following is the right place to look for a performance bottleneck?
   - The algorithm chosen
   - The programming language or compiler
   - The operating system
   - The processor
   - The I/O system and devices

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

1.2 Below Your Program

A typical application, such as a word processor or a large database system, may consist of millions of lines of code and rely on sophisticated software libraries that implement complex functions in support of the application. As we will see, the hardware in a computer can only execute extremely simple low-level instructions. To go from a complex application to the simple instructions involves several layers of software that interpret or translate high-level operations into simple computer instructions.

Figure 1.2 shows that these layers of software are organized primarily in a hierarchical fashion, with applications being the outermost ring and a variety of **systems software** sitting between the hardware and applications software.

systems software Software that provides services that are commonly useful, including operating systems, compilers, loaders, and assemblers.

There are many types of systems software, but two types of systems software are central to every computer system today: an operating system and a compiler. An **operating system** interfaces between a user's program and the hardware and provides a variety of services and supervisory functions. Among the most important functions are

operating system Supervising program that manages the resources of a computer for the benefit of the programs that run on that computer.

- Handling basic input and output operations
- Allocating storage and memory
- Providing for protected sharing of the computer among multiple applications using it simultaneously.

Examples of operating systems in use today are Linux, MacOS, and Windows.

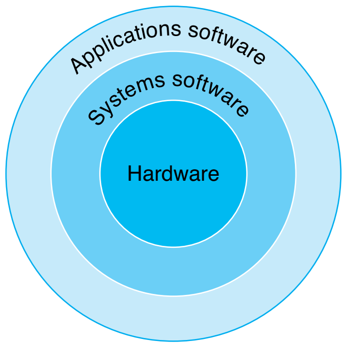


FIGURE 1.2 A simplified view of hardware and software as hierarchical layers, shown as concentric circles with hardware in the center and applications software outermost. In complex applications, there are often multiple layers of application software as well. For example, a database system may run on top of the systems software hosting an application, which in turn runs on top of the database.

Compilers perform another vital function: the translation of a program written in a high-level language, such as C, C++, Java, or Visual Basic into instructions that the hardware can execute. Given the sophistication of modern programming languages and the simplicity of the instructions executed by the hardware, the translation from a high-level language program to hardware instructions is complex. We give a brief overview of the process here and then go into more depth in Chapter 2 and Appendix B.

compiler A program that translates high-level language statements into assembly language statements.

From a High-Level Language to the Language of Hardware

To actually speak to electronic hardware, you need to send electrical signals. The easiest signals for computers to understand are *on* and *off*, and so the computer 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 computer 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, which are called **instructions**. Instructions, which are just collections of bits that the computer understands and obeys, can be thought of as numbers. For example, the bits

binary digit Also called a **bit**. One of the two numbers in base 2 (0 or 1) that are the components of information.

instruction A command that computer hardware understands and obeys.

```
1000110010100000
```

tell one computer to add two numbers. Chapter 2 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 computer to help program the computer, 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

assembler A program that translates a symbolic version of instructions into the binary version.

```
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**. In contrast, the binary language that the machine understands is the **machine language**.

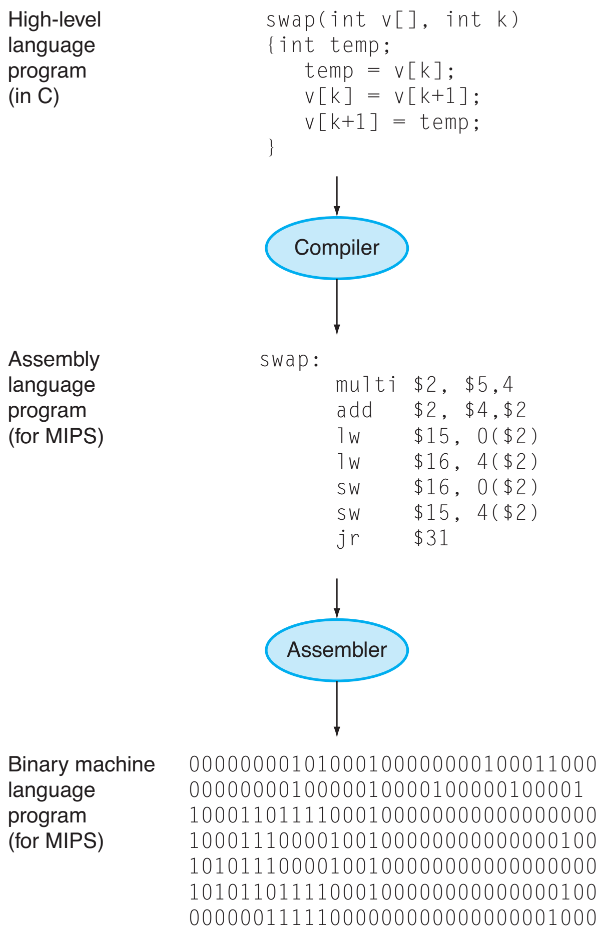
assembly language A symbolic representation of machine instructions.

machine language A binary representation of machine instructions.

Although a tremendous improvement, assembly language is still far from the notations 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 computer will follow, forcing the programmer to think like the computer.

The recognition that a program could be written to translate a more powerful language into computer instructions was one of the great breakthroughs in the early days of computing. Programmers today owe their productivity—and their sanity—to the creation of **high-level programming languages** and compilers that translate programs in such languages into instructions. Figure 1.3 shows the relationships among these programs and languages.

high-level programming language A portable language such as C, C++, Java, or Visual Basic that is composed of words and algebraic notation that can be translated by a compiler into assembly language.


High-level language program (in C)
Compiler
Assembly language program (for MIPS)
Assembler
Binary machine language program (for MIPS)

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

```
swap:
      multi $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
```

```
00000000101000100000000100011000
0000000010000010000100000100001
10001101111000100000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

FIGURE 1.3 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 2.

A compiler enables a programmer to write this high-level language expression:

```
A + B
```

The compiler would compile it into this assembly language statement:

```
add A,B
```

As shown above, the assembler would translate this statement into the binary instructions that tell the computer to add the two numbers `A` and `B`.

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 1.3). 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. There are also domain-specific languages for even narrower groups of users, such as those interested in simulation of fluids, for example.

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 computer. These three advantages are so strong that today little programming is done in assembly language.

1.3 Under the Covers

Now that we have looked below your program to uncover the underlying software, let's open the covers of your computer to learn about the underlying hardware. The underlying hardware in any computer performs the same basic functions: inputting data, outputting data, processing data, and storing data. How these functions are performed is the primary topic of this book, and subsequent chapters deal with different parts of these four tasks.

When we come to an important point in this book, a point so important that we hope you will remember it forever, we emphasize it by identifying it as a *Big Picture* item. We have about a dozen Big Pictures in this book, the first being the five components of a computer that perform the tasks of inputting, outputting, processing, and storing data.

The BIG Picture

The five classic components of a computer are input, output, memory, datapath, and control, with the last two sometimes combined and called the processor. Figure 1.4 shows the standard organization of a computer. This organization is independent of hardware technology: you can place every piece of every computer, past and present, into one of these five categories. To help you keep all this in perspective, the five components of a computer are shown on the front page of each of the following chapters, with the portion of interest to that chapter highlighted.

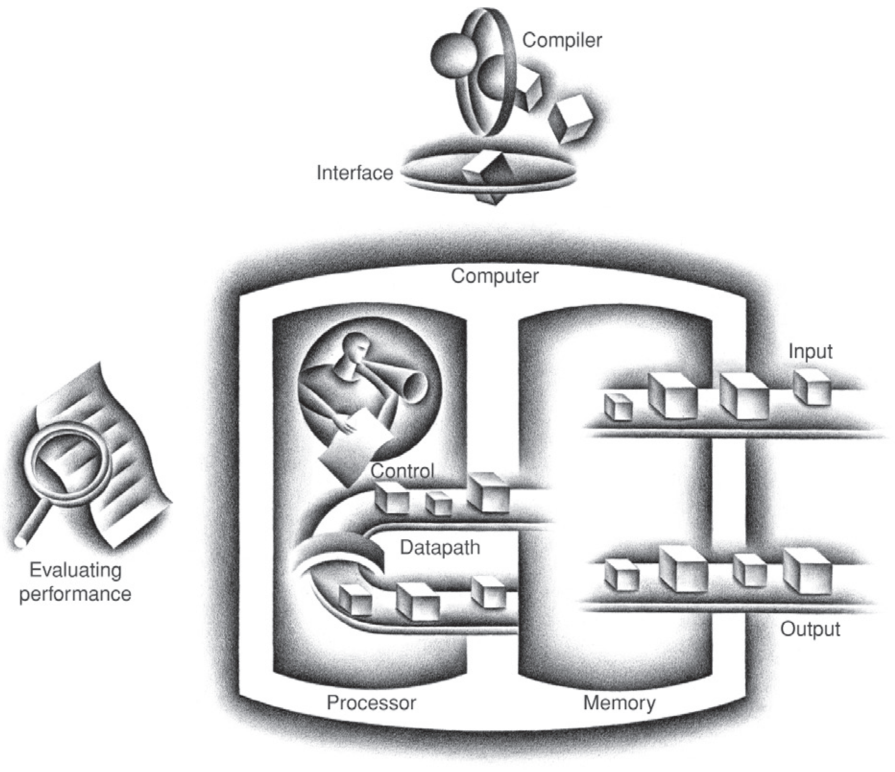


FIGURE 1.4 The organization of a computer, showing the five classic components. The processor gets instructions and data from memory. Input writes data to memory, and output reads data from memory. Control sends the signals that determine the operations of the datapath, memory, input, and output.

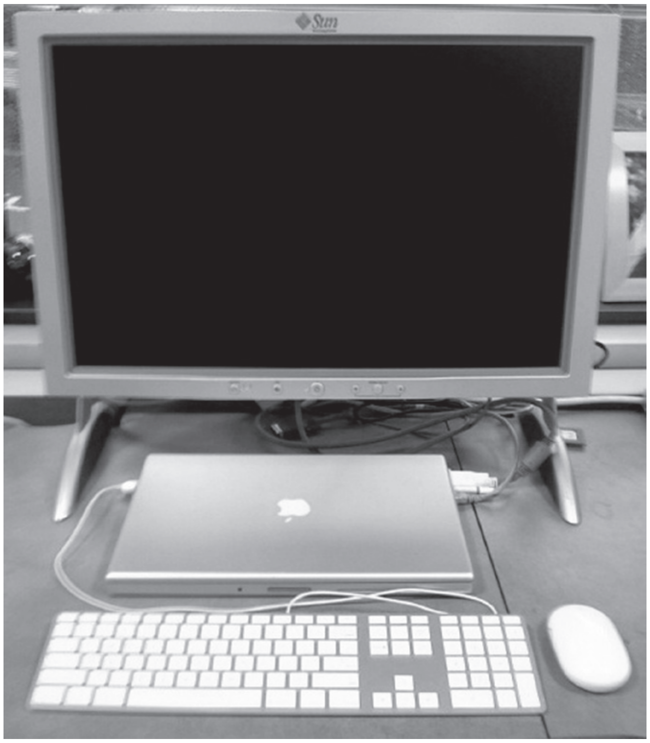


FIGURE 1.5 A desktop computer. The liquid crystal display (LCD) screen is the primary output device, and the keyboard and mouse are the primary input devices. On the right side is an Ethernet cable that connected the laptop to the network and the Web. The laptop contains the processor, memory, and additional I/O devices. This system is a Macbook Pro 15" laptop connected to an external display.

Figure 1.5 shows a computer with keyboard, wireless mouse, and screen. This photograph reveals two of the key components of computers: **input devices**, such as the keyboard and mouse, and **output devices**, such as the screen. As the names suggest, input feeds the computer, and output is the result of computation sent to the user. Some devices, such as networks and disks, provide both input and output to the computer.

input device
A mechanism through which the computer is fed information, such as the keyboard or mouse.

output device
A mechanism that conveys the result of a computation to a user or another computer.

Chapter 6 describes input/output (I/O) devices in more detail, but let's take an introductory tour through the computer hardware, starting with the external I/O devices.

I got the idea for the mouse while attending a talk at a computer conference. The speaker was so boring that I started daydreaming and hit upon the idea.

Doug Engelbart

Through computer displays I have landed an airplane on the deck of a moving carrier, observed a nuclear particle hit a potential well, flown in a rocket at nearly the speed of light and watched a computer reveal its innermost workings.

Ivan Sutherland, the "father" of computer graphics, *Scientific American*, 1984

Anatomy of a Mouse

Although many users now take mice for granted, the idea of a pointing device such as a mouse was first shown by Doug Engelbart using a research prototype in 1967. The Alto, which was the inspiration for all workstations as well as for the Macintosh and Windows OS, included a mouse as its pointing device in 1973. By the 1990s, all desktop computers included this device, and new user interfaces based on graphics displays and mice became the norm.

The original mouse was electromechanical and used a large ball that when rolled across a surface would cause an *x* and *y* counter to be incremented. The amount of increase in each counter told how far the mouse had been moved.

The electromechanical mouse has largely been replaced by the newer all-optical mouse. The optical mouse is actually a miniature optical processor including an LED to provide lighting, a tiny black-and-white camera, and a simple optical processor. The LED illuminates the surface underneath the mouse; the camera takes 1500 sample pictures a second under the illumination. Successive pictures are sent to a simple optical processor that compares the images and determines whether the mouse has moved and how far. The replacement of the electromechanical mouse by the electro-optical mouse is an illustration of a common phenomenon where the decreasing costs and higher reliability of electronics cause an electronic solution to replace the older electromechanical technology. On page 22 we'll see another example: flash memory.

Through the Looking Glass

The most fascinating I/O device is probably the graphics display. All laptop and handheld computers, calculators, cellular phones, and almost all desktop computers now use **liquid crystal displays (LCDs)** to get a thin, low-power display. The LCD is not the source of light; instead, it controls the transmission of light. A typical LCD includes rod-shaped molecules in a liquid that form a twisting helix that bends light entering the display, from either a light source behind the display or less often from reflected light. The rods straighten out when a current is applied and no longer bend the light. Since the liquid crystal material is between two screens polarized at 90 degrees, the light cannot pass through unless it is bent. Today, most LCD displays use an **active matrix** that has a tiny transistor switch at each pixel to precisely control current and make sharper images. A red-green-blue mask associated with each dot on the display determines the intensity of the three color components in the final image; in a color active matrix LCD, there are three transistor switches at each point.

liquid crystal display A display technology using a thin layer of liquid polymers that can be used to transmit or block light according to whether a charge is applied.

active matrix display A liquid crystal display using a transistor to control the transmission of light at each individual pixel.

The image is composed of a matrix of picture elements, or **pixels**, which can be represented as a matrix of bits, called a *bit map*. Depending on the size of the screen and the resolution, the display matrix ranges in size from 640 × 480 to 2560 × 1600 pixels in 2008. A color display might use 8 bits for each of the three colors (red, blue, and green), for 24 bits per pixel, permitting millions of different colors to be displayed.

pixel The smallest individual picture element. Screens are composed of hundreds of thousands to millions of pixels, organized in a matrix.

The computer hardware support for graphics consists mainly of a *raster refresh buffer*, or *frame buffer*, to store the bit map. The image to be represented onscreen is stored in the frame buffer, and the bit pattern per pixel is read out to the graphics display at the refresh rate. Figure 1.6 shows a frame buffer with a simplified design of just 4 bits per pixel.

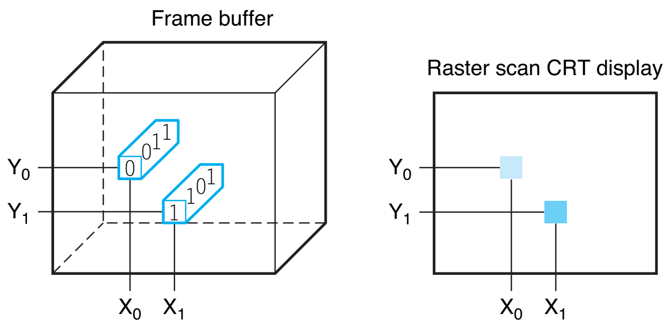


FIGURE 1.6 Each coordinate in the frame buffer on the left determines the shade of the corresponding coordinate for the raster scan CRT display on the right. Pixel (X_0, Y_0) contains the bit pattern 0011, which is a lighter shade on the screen than the bit pattern 1101 in pixel (X_1, Y_1).

The goal of the bit map is to faithfully represent what is on the screen. The challenges in graphics systems arise because the human eye is very good at detecting even subtle changes on the screen.

Opening the Box

If we open the box containing the computer, we see a fascinating board of thin plastic, covered with dozens of small gray or black rectangles. Figure 1.7 shows the contents of the laptop computer in Figure 1.5. The **motherboard** is shown in the upper part of the photo. Two disk drives are in front—the hard drive on the left and a DVD drive on the right. The hole in the middle is for the laptop battery.

The small rectangles on the motherboard contain the devices that drive our advancing technology, called **integrated circuits** and nicknamed **chips**. The board is composed of three pieces: the piece connecting to the I/O devices mentioned earlier, the memory, and the processor.

The **memory** is where the programs are kept when they are running; it also contains the data needed by the running programs. Figure 1.8 shows that memory is found on the two small boards, and each small memory board contains eight integrated circuits. The memory in Figure 1.8 is built from DRAM chips. *DRAM*

motherboard A plastic board containing packages of integrated circuits or chips, including processor, cache, memory, and connectors for I/O devices such as networks and disks.

integrated circuit Also called a **chip**. A device combining dozens to millions of transistors.

memory The storage area in which programs are kept when they are running and that contains the data needed by the running programs.

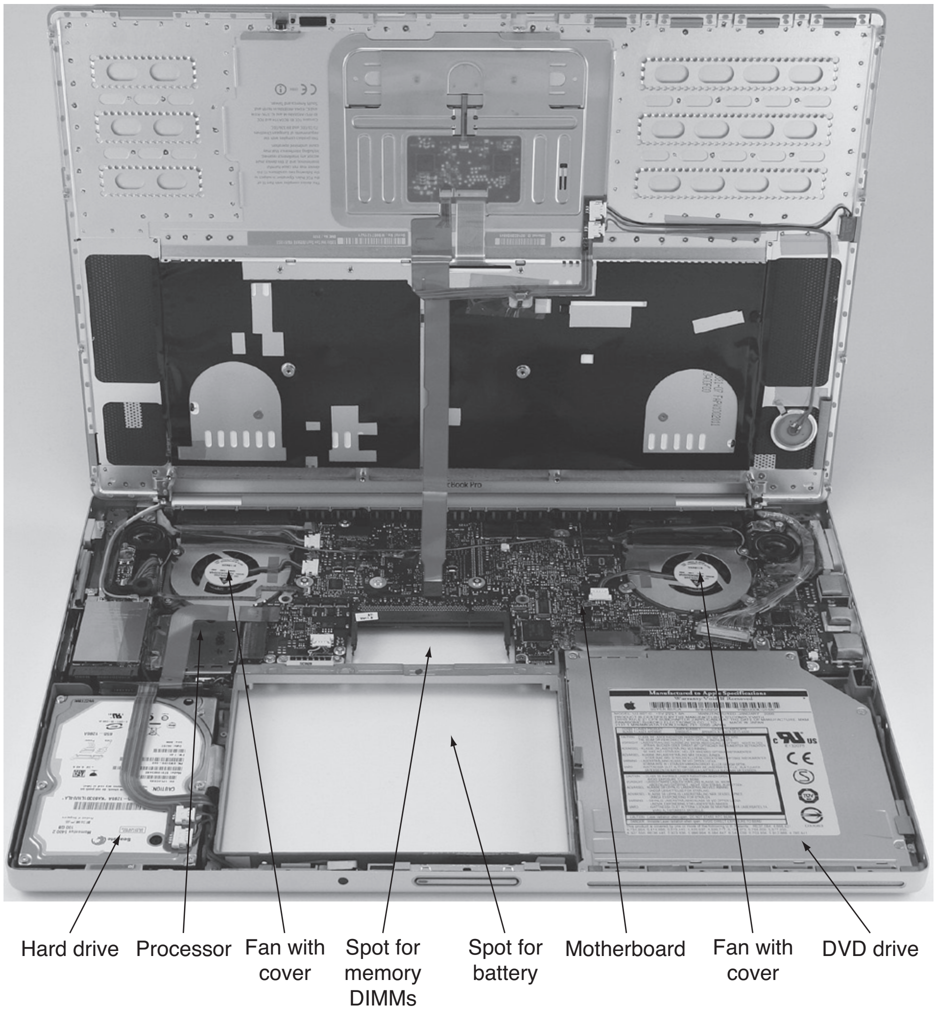


FIGURE 1.7 Inside the laptop computer of Figure 1.5. The shiny box with the white label on the lower left is a 100 GB SATA hard disk drive, and the shiny metal box on the lower right side is the DVD drive. The hole between them is where the laptop battery would be located. The small hole above the battery hole is for memory DIMMs. Figure 1.8 is a close-up of the DIMMs, which are inserted from the bottom in this laptop. Above the battery hole and DVD drive is a printed circuit board (PC board), called the *motherboard*, which contains most of the electronics of the computer. The two shiny circles in the upper half of the picture are two fans with covers. The processor is the large raised rectangle just below the left fan. Photo courtesy of OtherWorldComputing.com.

stands for **dynamic random access memory**. Several DRAMs are used together to contain the instructions and data of a program. In contrast to sequential access memories, such as magnetic tapes, the *RAM* portion of the term DRAM means that memory accesses take basically the same amount of time no matter what portion of the memory is read.

dynamic random access memory (DRAM) Memory built as an integrated circuit; it provides random access to any location.

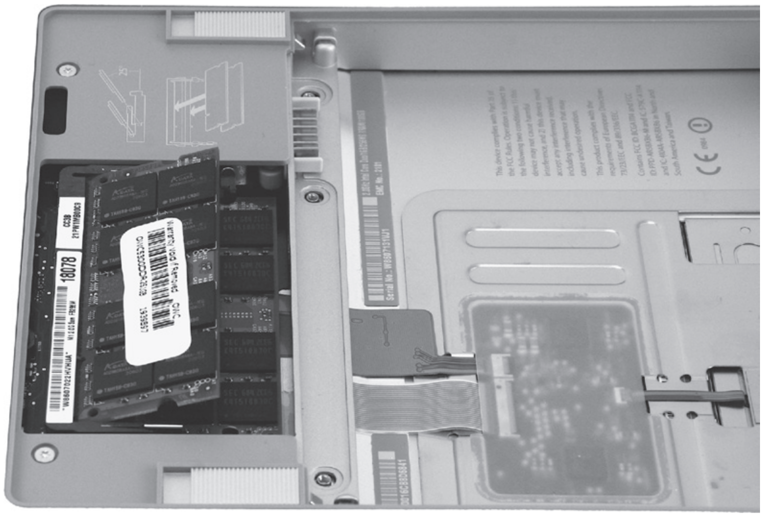


FIGURE 1.8 Close-up of the bottom of the laptop reveals the memory. The main memory is contained on one or more small boards shown on the left. The hole for the battery is to the right. The DRAM chips are mounted on these boards (called **DIMMs**, for dual inline memory modules) and then plugged into the connectors. Photo courtesy of OtherWorldComputing.com.

dual inline memory module (DIMM) A small board that contains DRAM chips on both sides. (SIMMs have DRAMs on only one side.)

central processor unit (CPU) Also called processor. The active part of the computer, which contains the datapath and control and which adds numbers, tests numbers, signals I/O devices to activate, and so on.

The *processor* is the active part of the board, following the instructions of a program to the letter. It adds numbers, tests numbers, signals I/O devices to activate, and so on. The processor is under the fan and covered by a heat sink on the left side of Figure 1.7. Occasionally, people call the processor the **CPU**, for the more bureaucratic-sounding **central processor unit**.

Descending even lower into the hardware, Figure 1.9 reveals details of a microprocessor. The processor logically comprises two main components: datapath and control, the respective brawn and brain of the processor. The **datapath** performs the arithmetic operations, and **control** tells the datapath, memory, and I/O devices what to do according to the wishes of the instructions of the program. Chapter 4 explains the datapath and control for a higher-performance design.

datapath The component of the processor that performs arithmetic operations

control The component of the processor that commands the datapath, memory, and I/O devices according to the instructions of the program.

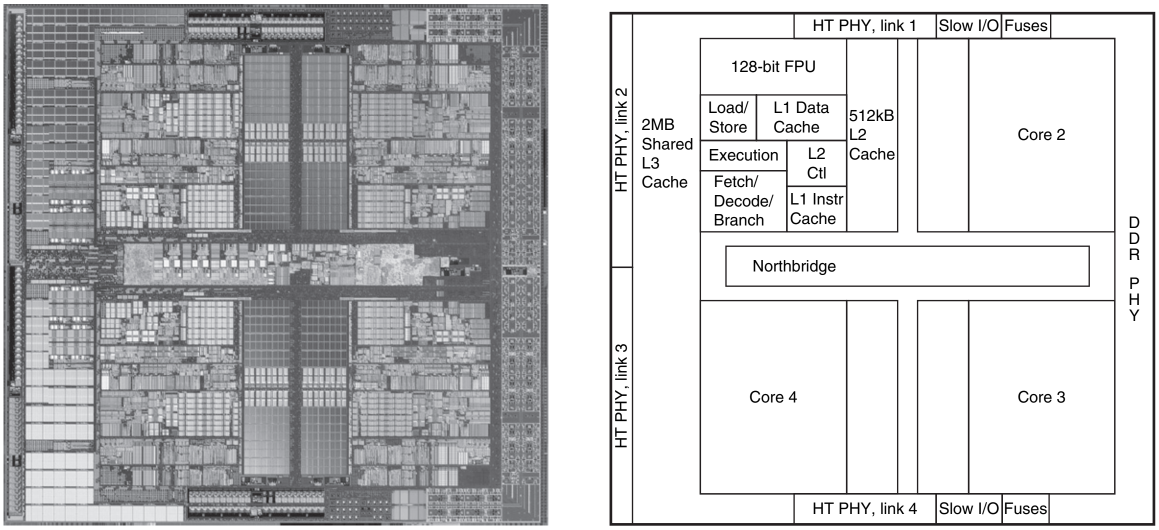


FIGURE 1.9 Inside the AMD Barcelona microprocessor. The left-hand side is a microphotograph of the AMD Barcelona processor chip, and the right-hand side shows the major blocks in the processor. This chip has four processors or "cores". The microprocessor in the laptop in Figure 1.7 has two cores per chip, called an Intel Core 2 Duo.

cache memory A small, fast memory that acts as a buffer for a slower, larger memory.

static random access memory (SRAM) Also memory built as an integrated circuit, but faster and less dense than DRAM.

abstraction A model that renders lower-level details of computer systems temporarily invisible to facilitate design of sophisticated systems.

Descending into the depths of any component of the hardware reveals insights into the computer. Inside the processor is another type of memory—cache memory. **Cache memory** consists of a small, fast memory that acts as a buffer for the DRAM memory. (The nontechnical definition of *cache* is a safe place for hiding things.) Cache is built using a different memory technology, **static random access memory** (**SRAM**). SRAM is faster but less dense, and hence more expensive, than DRAM (see Chapter 5).

You may have noticed a common theme in both the software and the hardware descriptions: delving into the depths of hardware or software reveals more information or, conversely, lower-level details are hidden to offer a simpler model at higher levels. The use of such layers, or **abstractions**, is a principal technique for designing very sophisticated computer systems.

One of the most important abstractions is the interface between the hardware and the lowest-level software. Because of its importance, it is given a special name: the **instruction set architecture**, or simply **architecture**, of a computer. The instruction set architecture includes anything programmers need to know to make a binary machine language program work correctly, including instructions, I/O devices, and so on. Typically, the operating system will encapsulate the details of doing I/O, allocating memory, and other low-level system functions so that application programmers do not need to worry about such details. The combination of the basic instruction set and the operating system interface provided for application programmers is called the **application binary interface** (**ABI**).

An instruction set architecture allows computer designers to talk about functions independently from the hardware that performs them. For example, we can talk about the functions of a digital clock (keeping time, displaying the time, setting the alarm) independently from the clock hardware (quartz crystal, LED displays, plastic buttons). Computer designers distinguish architecture from an **implementation** of an architecture along the same lines: an implementation is hardware that obeys the architecture abstraction. These ideas bring us to another Big Picture.

instruction set architecture Also called **architecture**. An abstract interface between the hardware and the lowest-level software that encompasses all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, ...

application binary interface (ABI) The user portion of the instruction set plus the operating system interfaces used by application programmers. Defines a standard for binary portability across computers.

implementation Hardware that obeys the architecture abstraction.

The BIG Picture

Both hardware and software consist of hierarchical layers, with each lower layer hiding details from the level above. This principle of *abstraction* is the way both hardware designers and software designers cope with the complexity of computer systems. One key interface between the levels of abstraction is the *instruction set architecture*—the interface between the hardware and low-level software. This abstract interface enables many *implementations* of varying cost and performance to run identical software.

A Safe Place for Data

Thus far, we have seen how to input data, compute using the data, and display data. If we were to lose power to the computer, however, everything would be lost because the memory inside the computer is **volatile**—that is, when it loses power, it forgets. In contrast, a DVD doesn't forget the recorded film when you turn off the power to the DVD player and is thus a **nonvolatile memory** technology.

To distinguish between the volatile memory used to hold data and programs while they are running and this nonvolatile memory used to store data and programs between runs, the term **main memory** or **primary memory** is used for the

volatile memory Storage, such as DRAM, that retains data only if it is receiving power.

nonvolatile memory A form of memory that retains data even in the absence of a power source and that is used to store programs between runs. Magnetic disk is nonvolatile.

main memory Also called **primary memory**. Memory used to hold programs while they are running; typically consists of DRAM in today's computers.

former, and **secondary memory** for the latter. DRAMs have dominated main memory since 1975, but **magnetic disks** have dominated secondary memory since 1965. The primary nonvolatile storage used in all server computers and workstations is the magnetic **hard disk**. **Flash memory**, a nonvolatile semiconductor memory, is used instead of disks in mobile devices such as cell phones and is increasingly replacing disks in music players and even laptops.

secondary memory Nonvolatile memory used to store programs and data between runs; typically consists of magnetic disks in today's computers.

magnetic disk Also called **hard disk**. A form of nonvolatile secondary memory composed of rotating platters coated with a magnetic recording material.

flash memory A nonvolatile semiconductor memory. It is cheaper and slower than DRAM but more expensive and faster than magnetic disks.

As Figure 1.10 shows, a magnetic hard disk consists of a collection of platters, which rotate on a spindle at 5400 to 15,000 revolutions per minute. The metal platters are covered with magnetic recording material on both sides, similar to the material found on a cassette or videotape. To read and write information on a hard disk, a movable *arm* containing a small electromagnetic coil called a *read-write head* is located just above each surface. The entire drive is permanently sealed to control the environment inside the drive, which, in turn, allows the disk heads to be much closer to the drive surface.

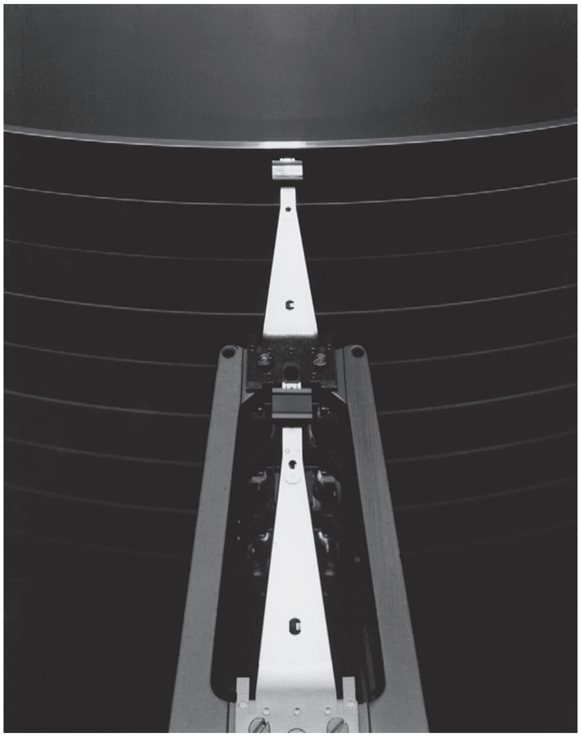


FIGURE 1.10 A disk showing 10 disk platters and the read/write heads.

Diameters of hard disks vary by more than a factor of 3 today, from 1 inch to 3.5 inches, and have been shrunk over the years to fit into new products; workstation servers, personal computers, laptops, palmtops, and digital cameras have all inspired new disk form factors. Traditionally, the widest disks have the highest performance and the smallest disks have the lowest unit cost. The best cost per **gigabyte** varies. Although most hard drives appear inside computers, as in Figure 1.7, hard drives can also be attached using external interfaces such as universal serial bus (USB).

gigabyte Traditionally 1,073,741,824 (2^{30}) bytes, although some communications and secondary storage systems have redefined it to mean 1,000,000,000 (10^9) bytes. Similarly, depending on the context, megabyte is either 2^{20} or 10^6 bytes.

The use of mechanical components means that access times for magnetic disks are much slower than for DRAMs: disks typically take 5–20 milliseconds, while DRAMs take 50–70 nanoseconds—making DRAMs about 100,000 times faster. Yet disks have much lower costs than DRAM for the same storage capacity, because the production costs for a given amount of disk storage are lower than for the same amount of integrated circuit. In 2008, the cost per gigabyte of disk is 30 to 100 times less expensive than DRAM.

Thus, there are three primary differences between magnetic disks and main memory: disks are nonvolatile because they are magnetic; they have a slower access time because they are mechanical devices; and they are cheaper per gigabyte because they have very high storage capacity at a modest cost.

Many have tried to invent a technology cheaper than DRAM but faster than disk to fill that gap, but many have failed. Challengers have never had a product to market at the right time. By the time a new product would ship, DRAMs and disks had continued to make rapid advances, costs had dropped accordingly, and the challenging product was immediately obsolete.

Flash memory, however, is a serious challenger. This semiconductor memory is nonvolatile like disks and has about the same bandwidth, but latency is 100 to 1000 times faster than disk. Flash is popular in cameras and portable music players because it comes in much smaller capacities, it is more rugged, and it is more power efficient than disks, despite the cost per gigabyte in 2008 being about 6 to 10 times higher than disk. Unlike disks and DRAM, flash memory bits wear out after 100,000 to 1,000,000 writes. Thus, file systems must keep track of the number of writes and have a strategy to avoid wearing out storage, such as by moving popular data. Chapter 6 describes flash in more detail.

Although hard drives are not removable, there are several storage technologies in use that include the following:

- Optical disks, including both compact disks (CDs) and digital video disks (DVDs), constitute the most common form of removable storage. The Blu-Ray (BD) optical disk standard is the heir-apparent to DVD.
- Flash-based removable memory cards typically attach to a USB connection and are often used to transfer files.
- Magnetic tape provides only slow serial access and has been used to back up disks, a role now often replaced by duplicate hard drives.

Optical disk technology works differently than magnetic disk technology. In a CD, data is recorded in a spiral fashion, with individual bits being recorded by burning small pits—approximately 1 micron (10^{-6} meters) in diameter—into the disk surface. The disk is read by shining a laser at the CD surface and determining by examining the reflected light whether there is a pit or flat (reflective) surface. DVDs use the same approach of bouncing a laser beam off a series of pits and flat surfaces. In addition, there are multiple layers that the laser beam can focus on, and the size of each bit is much smaller, which together increase capacity significantly. Blu-Ray uses shorter wavelength lasers that shrink the size of the bits and thereby increase capacity.

Optical disk writers in personal computers use a laser to make the pits in the recording layer on the CD or DVD surface. This writing process is relatively slow, taking from minutes (for a full CD) to tens of minutes (for a full DVD). Thus, for large quantities a different technique called *pressing* is used, which costs only pennies per optical disk.

Rewritable CDs and DVDs use a different recording surface that has a crystalline, reflective material; pits are formed that are not reflective in a manner similar to that for a write-once CD or DVD. To erase the CD or DVD, the surface is heated and cooled slowly, allowing an annealing process to restore the surface recording layer to its crystalline structure. These rewritable disks are the most expensive, with write-once being cheaper; for read-only disks—used to distribute software, music, or movies—both the disk cost and recording cost are much lower.

Communicating with Other Computers

We've explained how we can input, compute, display, and save data, but there is still one missing item found in today's computers: computer networks. Just as the processor shown in Figure 1.4 is connected to memory and I/O devices, networks interconnect whole computers, allowing computer users to extend the power of computing by including communication. Networks have become so popular that they are the backbone of current computer systems; a new computer without an optional network interface would be ridiculed. Networked computers have several major advantages:

- *Communication:* Information is exchanged between computers at high speeds.
- *Resource sharing:* Rather than each computer having its own I/O devices, devices can be shared by computers on the network.
- *Nonlocal access:* By connecting computers over long distances, users need not be near the computer they are using.

Networks vary in length and performance, with the cost of communication increasing according to both the speed of communication and the distance that information travels. Perhaps the most popular type of network is *Ethernet*. It can be up to a kilometer long and transfer at upto 10 gigabits per second. Its length and speed make Ethernet useful to connect computers on the same floor of a building; hence, it is an example of what is generically called a **local area network**. Local area networks are interconnected with switches that can also provide routing services and security. **Wide area networks** cross continents and are the backbone of the Internet, which supports the World Wide Web. They are typically based on optical fibers and are leased from telecommunication companies.

local area network (LAN) A network designed to carry data within a geographically confined area, typically within a single building.

wide area network (WAN) A network extended over hundreds of kilometers that can span a continent.

Networks have changed the face of computing in the last 25 years, both by becoming much more ubiquitous and by making dramatic increases in performance. In the 1970s, very few individuals had access to electronic mail, the Internet and Web did not exist, and physically mailing magnetic tapes was the primary way to transfer large amounts of data between two locations. Local area networks were almost nonexistent, and the few existing wide area networks had limited capacity and restricted access.

As networking technology improved, it became much cheaper and had a much higher capacity. For example, the first standardized local area network technology, developed about 25 years ago, was a version of Ethernet that had a maximum capacity (also called bandwidth) of 10 million bits per second, typically shared by tens of, if not a hundred, computers. Today, local area network technology offers a capacity of from 100 million bits per second to 10 gigabits per second, usually shared by at most a few computers. Optical communications technology has allowed similar growth in the capacity of wide area networks, from hundreds of kilobits to gigabits and from hundreds of computers connected to a worldwide network to millions of computers connected. This combination of dramatic rise in deployment of networking combined with increases in capacity have made network technology central to the information revolution of the last 25 years.

For the last decade another innovation in networking is reshaping the way computers communicate. Wireless technology is widespread, and laptops now incorporate this technology. The ability to make a radio in the same low-cost semiconductor technology (CMOS) used for memory and microprocessors enabled a significant improvement in price, leading to an explosion in deployment. Currently available wireless technologies, called by the IEEE standard name 802.11, allow for transmission rates from 1 to nearly 100 million bits per second. Wireless technology is quite a bit different from wire-based networks, since all users in an immediate area share the airwaves.

Check Yourself

- Semiconductor DRAM and disk storage differ significantly. Describe the fundamental difference for each of the following: volatility, access time, and cost.

Technologies for Building Processors and Memory

Processors and memory have improved at an incredible rate, because computer designers have long embraced the latest in electronic technology to try to win the race to design a better computer. Figure 1.11 shows the technologies that have been used over time, with an estimate of the relative performance per unit cost for each technology. Section 1.7 explores the technology that has fueled the computer industry since 1975 and will continue to do so for the foreseeable future. Since this technology shapes what computers will be able to do and how quickly they will evolve, we believe all computer professionals should be familiar with the basics of integrated circuits.

Year	Technology used in computers	Relative performance/unit cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit	900
1995	Very large-scale integrated circuit	2,400,000
2005	Ultra large-scale integrated circuit	6,200,000,000

FIGURE 1.11 Relative performance per unit cost of technologies used in computers over time. Source: Computer Museum, Boston, with 2005 extrapolated by the authors. See Section 1.10 on the CD.

vacuum tube An electronic component, predecessor of the transistor, that consists of a hollow glass tube about 5 to 10 cm long from which as much air has been removed as possible and that uses an electron beam to transfer data.

transistor An on/off switch controlled by an electric signal.

very large-scale integrated (VLSI) circuit A device containing hundreds of thousands to millions of transistors.

A **transistor** is simply an on/off switch controlled by electricity. The *integrated circuit* (IC) combined dozens to hundreds of transistors into a single chip. To describe the tremendous increase in the number of transistors from hundreds to millions, the adjective *very large scale* is added to the term, creating the abbreviation *VLSI,* for **very large-scale integrated circuit**.

This rate of increasing integration has been remarkably stable. Figure 1.12 shows the growth in DRAM capacity since 1977. For 20 years, the industry has consistently quadrupled capacity every 3 years, resulting in an increase in excess of 16,000 times! This increase in transistor count for an integrated circuit is popularly known as Moore's law, which states that transistor capacity doubles every 18–24 months. Moore's law resulted from a prediction of such growth in IC capacity made by Gordon Moore, one of the founders of Intel during the 1960s.

Sustaining this rate of progress for almost 40 years has required incredible innovation in manufacturing techniques. In Section 1.7, we discuss how to manufacture integrated circuits.

1.4 Performance

Assessing the performance of computers can be quite challenging. The scale and intricacy of modern software systems, together with the wide range of performance improvement techniques employed by hardware designers, have made performance assessment much more difficult.

When trying to choose among different computers, performance is an important attribute. Accurately measuring and comparing different computers is critical to

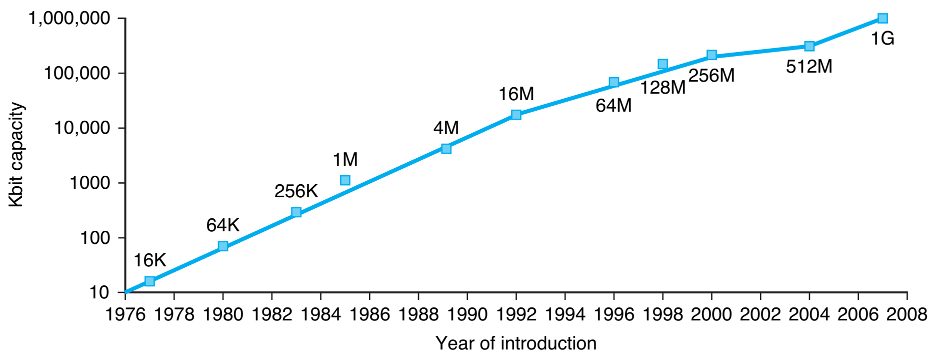


FIGURE 1.12 Growth of capacity per DRAM chip over time. The *y*-axis is measured in Kilobits, where K = 1024 (2^{10}). The DRAM industry quadrupled capacity almost every three years, a 60% increase per year, for 20 years. In recent years, the rate has slowed down and is somewhat closer to doubling every two years to three years.

purchasers and therefore to designers. The people selling computers know this as well. Often, salespeople would like you to see their computer in the best possible light, whether or not this light accurately reflects the needs of the purchaser's application. Hence, understanding how best to measure performance and the limitations of performance measurements is important in selecting a computer.

The rest of this section describes different ways in which performance can be determined; then, we describe the metrics for measuring performance from the viewpoint of both a computer user and a designer. We also look at how these metrics are related and present the classical processor performance equation, which we will use throughout the text.

Defining Performance

When we say one computer has better performance than another, what do we mean? Although this question might seem simple, an analogy with passenger airplanes shows how subtle the question of performance can be. Figure 1.13 shows some typical passenger airplanes, together with their cruising speed, range, and capacity. If we wanted to know which of the planes in this table had the best performance, we would first need to define performance. For example, considering different measures of performance, we see that the plane with the highest cruising speed is the Concorde, the plane with the longest range is the DC-8, and the plane with the largest capacity is the 747.

Let's suppose we define performance in terms of speed. This still leaves two possible definitions. You could define the fastest plane as the one with the highest cruising speed, taking a single passenger from one point to another in the least time. If you

Airplane	Passenger capacity	Cruising range (miles)	Cruising speed (m.p.h.)	Passenger throughput (passengers × m.p.h.)
Boeing 777	375	4630	610	228,750
Boeing 747	470	4150	610	286,700
BAC/Sud Concorde	132	4000	1350	178,200
Douglas DC-8-50	146	8720	544	79,424

FIGURE 1.13 The capacity, range, and speed for a number of commercial airplanes. The last column shows the rate at which the airplane transports passengers, which is the capacity times the cruising speed (ignoring range and takeoff and landing times).

response time Also called **execution time**. The total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time, and so on.

throughput Also called **bandwidth**. Another measure of performance, it is the number of tasks completed per unit time.

were interested in transporting 450 passengers from one point to another, however, the 747 would clearly be the fastest, as the last column of the figure shows. Similarly, we can define computer performance in several different ways.

If you were running a program on two different desktop computers, you'd say that the faster one is the desktop computer that gets the job done first. If you were running a datacenter that had several servers running jobs submitted by many users, you'd say that the faster computer was the one that completed the most jobs during a day. As an individual computer user, you are interested in reducing **response time**—the time between the start and completion of a task—also referred to as **execution time**. Datacenter managers are often interested in increasing **throughput** or **bandwidth**—the total amount of work done in a given time. Hence, in most cases, we will need different performance metrics as well as different sets of applications to benchmark embedded and desktop computers, which are more focused on response time, versus servers, which are more focused on throughput.

Throughput and Response Time

EXAMPLE

Do the following changes to a computer system increase throughput, decrease response time, or both?

1. Replacing the processor in a computer with a faster version
2. Adding additional processors to a system that uses multiple processors for separate tasks—for example, searching the World Wide Web

ANSWER

Decreasing response time almost always improves throughput. Hence, in case 1, both response time and throughput are improved. In case 2, no one task gets work done faster, so only throughput increases.

If, however, the demand for processing in the second case was almost as large as the throughput, the system might force requests to queue up. In this case, increasing the throughput could also improve response time, since it would reduce the waiting time in the queue. Thus, in many real computer systems, changing either execution time or throughput often affects the other.

In discussing the performance of computers, we will be primarily concerned with response time for the first few chapters. To maximize performance, we want to minimize response time or execution time for some task. Thus, we can relate performance and execution time for a computer X:

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\text{Performance}_X > \text{Performance}_Y$$

$$\frac{1}{\text{Execution time}_X} > \frac{1}{\text{Execution time}_Y}$$

$$\text{Execution time}_Y > \text{Execution time}_X$$

That is, the execution time on Y is longer than that on X, if X is faster than Y.

In discussing a computer design, we often want to relate the performance of two different computers quantitatively. We will use the phrase "X is *n* times faster than Y"—or equivalently "X is *n* times as fast as Y"—to mean

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

If X is *n* times faster than Y, then the execution time on Y is *n* times longer than it is on X:

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

Relative Performance

EXAMPLE

If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?

ANSWER

We know that A is *n* times faster than B if

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Thus the performance ratio is

$$\frac{15}{10} = 1.5$$

and A is therefore 1.5 times faster than B.

In the above example, we could also say that computer B is 1.5 times *slower than* computer A, since

$$\frac{\text{Performance}_A}{\text{Performance}_B} = 1.5$$

means that

$$\frac{\text{Performance}_A}{1.5} = \text{Performance}_B$$

For simplicity, we will normally use the terminology *faster than* when we try to compare computers quantitatively. Because performance and execution time are reciprocals, increasing performance requires decreasing execution time. To avoid the potential confusion between the terms *increasing* and *decreasing*, we usually say "improve performance" or "improve execution time" when we mean "increase performance" and "decrease execution time."

Measuring Performance

Time is the measure of computer performance: the computer that performs the same amount of work in the least time is the fastest. Program *execution time* is measured in seconds per program. However, time can be defined in different ways, depending on what we count. The most straightforward definition of time is called *wall clock time*, *response time*, or *elapsed time*. These terms mean the total time to complete a task, including disk accesses, memory accesses, input/output (I/O) activities, operating system overhead—everything.

Computers are often shared, however, and a processor may work on several programs simultaneously. In such cases, the system may try to optimize throughput rather than attempt to minimize the elapsed time for one program. Hence, we often want to distinguish between the elapsed time and the time that the processor is working on our behalf. **CPU execution time** or simply **CPU time**, which recognizes this distinction, is the time the CPU spends computing for this task and does not include time spent waiting for I/O or running other programs. (Remember, though, that the response time experienced by the user will be the elapsed time of the program, not the CPU time.) CPU time can be further divided into the CPU time spent in the program, called **user CPU time**, and the CPU time spent in the operating system performing tasks on behalf of the program, called **system CPU time**. Differentiating between system and user CPU time is difficult to

CPU execution time Also called **CPU time**. The actual time the CPU spends computing for a specific task.

user CPU time The CPU time spent in a program itself.

system CPU time The CPU time spent in the operating system performing tasks on behalf of the program.

do accurately, because it is often hard to assign responsibility for operating system activities to one user program rather than another and because of the functionality differences among operating systems.

For consistency, we maintain a distinction between performance based on elapsed time and that based on CPU execution time. We will use the term *system performance* to refer to elapsed time on an unloaded system and *CPU performance* to refer to user CPU time. We will focus on CPU performance in this chapter, although our discussions of how to summarize performance can be applied to either elapsed time or CPU time measurements.

Understanding Program Performance

Different applications are sensitive to different aspects of the performance of a computer system. Many applications, especially those running on servers, depend as much on I/O performance, which, in turn, relies on both hardware and software. Total elapsed time measured by a wall clock is the measurement of interest. In some application environments, the user may care about throughput, response time, or a complex combination of the two (e.g., maximum throughput with a worst-case response time). To improve the performance of a program, one must have a clear definition of what performance metric matters and then proceed to look for performance bottlenecks by measuring program execution and looking for the likely bottlenecks. In the following chapters, we will describe how to search for bottlenecks and improve performance in various parts of the system.

Although as computer users we care about time, when we examine the details of a computer it's convenient to think about performance in other metrics. In particular, computer designers may want to think about a computer by using a measure that relates to how fast the hardware can perform basic functions. Almost all computers are constructed using a clock that determines when events take place in the hardware. These discrete time intervals are called **clock cycles** (or **ticks**, **clock ticks**, **clock periods**, **clocks**, **cycles**). Designers refer to the length of a **clock period** both as the time for a complete *clock cycle* (e.g., 250 picoseconds, or 250 ps) and as the *clock rate* (e.g., 4 gigahertz, or 4 GHz), which is the inverse of the clock period. In the next subsection, we will formalize the relationship between the clock cycles of the hardware designer and the seconds of the computer user.

clock cycle Also called **tick**, **clock tick**, **clock period**, **clock**, **cycle**. The time for one clock period, usually of the processor clock, which runs at a constant rate.

clock period The length of each clock cycle.

Check Yourself

1. Suppose we know that an application that uses both a desktop client and a remote server is limited by network performance. For the following changes, state whether only the throughput improves, both response time and throughput improve, or neither improves.

   a. An extra network channel is added between the client and the server, increasing the total network throughput and reducing the delay to obtain network access (since there are now two channels).

   b. The networking software is improved, thereby reducing the network communication delay, but not increasing throughput.
   c. More memory is added to the computer.
2. Computer C's performance is 4 times faster than the performance of computer B, which runs a given application in 28 seconds. How long will computer C take to run that application?

CPU Performance and Its Factors

Users and designers often examine performance using different metrics. If we could relate these different metrics, we could determine the effect of a design change on the performance as experienced by the user. Since we are confining ourselves to CPU performance at this point, the bottom-line performance measure is CPU execution time. A simple formula relates the most basic metrics (clock cycles and clock cycle time) to CPU time:

$$\begin{matrix}\text{CPU execution time} \\ \text{for a program}\end{matrix} = \begin{matrix}\text{CPU clock cycles} \\ \text{for a program}\end{matrix} \times \text{Clock cycle time}$$

Alternatively, because clock rate and clock cycle time are inverses,

$$\begin{matrix}\text{CPU execution time} \\ \text{for a program}\end{matrix} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

This formula makes it clear that the hardware designer can improve performance by reducing the number of clock cycles required for a program or the length of the clock cycle. As we will see in later chapters, the designer often faces a trade-off between the number of clock cycles needed for a program and the length of each cycle. Many techniques that decrease the number of clock cycles may also increase the clock cycle time.

EXAMPLE

Improving Performance

Our favorite program runs in 10 seconds on computer A, which has a 2 GHz clock. We are trying to help a computer designer build a computer, B, which will run this program in 6 seconds. The designer has determined that a substantial increase in the clock rate is possible, but this increase will affect the rest of the CPU design, causing computer B to require 1.2 times as many clock cycles as computer A for this program. What clock rate should we tell the designer to target?

ANSWER

Let's first find the number of clock cycles required for the program on A:

$$\text{CPU time}_A = \frac{\text{CPU clock cycles}_A}{\text{Clock rate}_A}$$

$$10 \text{ seconds} = \frac{\text{CPU clock cycles}_A}{2 \times 10^9 \dfrac{\text{cycles}}{\text{second}}}$$

$$\text{CPU clock cycles}_A = 10 \text{ seconds} \times 2 \times 10^9 \frac{\text{cycles}}{\text{second}} = 20 \times 10^9 \text{ cycles}$$

CPU time for B can be found using this equation:

$$\text{CPU time}_B = \frac{1.2 \times \text{CPU clock cycles}_A}{\text{Clock rate}_B}$$

$$6 \text{ seconds} = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{\text{Clock rate}_B}$$

$$\text{Clock rate}_B = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{6 \text{ seconds}} = \frac{0.2 \times 20 \times 10^9 \text{ cycles}}{\text{second}} = \frac{4 \times 10^9 \text{ cycles}}{\text{second}} = 4 \text{ GHz}$$

To run the program in 6 seconds, B must have twice the clock rate of A.

Instruction Performance

The performance equations above did not include any reference to the number of instructions needed for the program. (We'll see what the instructions that make up a program look like in the next chapter.) However, since the compiler clearly generated instructions to execute, and the computer had to execute the instructions to run the program, the execution time must depend on the number of instructions in a program. One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction. Therefore, the number of clock cycles required for a program can be written as

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \begin{array}{c}\text{Average clock cycles}\\ \text{per instruction}\end{array}$$

The term **clock cycles per instruction**, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as **CPI**. Since different

clock cycles per instruction (CPI) Average number of clock cycles per instruction for a program or program fragment.

instructions may take different amounts of time depending on what they do, CPI is an average of all the instructions executed in the program. CPI provides one way of comparing two different implementations of the same instruction set architecture, since the number of instructions executed for a program will, of course, be the same.

Using the Performance Equation

EXAMPLE

Suppose we have two implementations of the same instruction set architecture. Computer A has a clock cycle time of 250 ps and a CPI of 2.0 for some program, and computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program. Which computer is faster for this program and by how much?

ANSWER

We know that each computer executes the same number of instructions for the program; let's call this number *I*. First, find the number of processor clock cycles for each computer:

$$\text{CPU clock cycles}_{\text{A}} = I \times 2.0$$

$$\text{CPU clock cycles}_{\text{B}} = I \times 1.2$$

Now we can compute the CPU time for each computer:

$$\begin{aligned}\text{CPU time}_{\text{A}} &= \text{CPU clock cycles}_{\text{A}} \times \text{Clock cycle time} \\ &= I \times 2.0 \times 250\ \text{ps} = 500 \times I\ \text{ps}\end{aligned}$$

Likewise, for B:

$$\text{CPU time}_{\text{B}} = I \times 1.2 \times 500\ \text{ps} = 600 \times I\ \text{ps}$$

Clearly, computer A is faster. The amount faster is given by the ratio of the execution times:

$$\frac{\text{CPU performance}_{\text{A}}}{\text{CPU performance}_{\text{B}}} = \frac{\text{Execution time}_{\text{B}}}{\text{Execution time}_{\text{A}}} = \frac{600 \times I\ \text{ps}}{500 \times I\ \text{ps}} = 1.2$$

We can conclude that computer A is 1.2 times as fast as computer B for this program.

The Classic CPU Performance Equation

We can now write this basic performance equation in terms of **instruction count** (the number of instructions executed by the program), CPI, and clock cycle time:

instruction count The number of instructions executed by the program.

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

or, since the clock rate is the inverse of clock cycle time:

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

These formulas are particularly useful because they separate the three key factors that affect performance. We can use these formulas to compare two different implementations or to evaluate a design alternative if we know its impact on these three parameters.

Comparing Code Segments

EXAMPLE

A compiler designer is trying to decide between two code sequences for a particular computer. The hardware designers have supplied the following facts:

	CPI for each instruction class		
	A	B	C
CPI	1	2	3

For a particular high-level language statement, the compiler writer is considering two code sequences that require the following instruction counts:

	Instruction counts for each instruction class		
Code sequence	A	B	C
1	2	1	2
2	4	1	1

Which code sequence executes the most instructions? Which will be faster? What is the CPI for each sequence?

ANSWER

Sequence 1 executes $2 + 1 + 2 = 5$ instructions. Sequence 2 executes $4 + 1 + 1 = 6$ instructions. Therefore, sequence 1 executes fewer instructions.

We can use the equation for CPU clock cycles based on instruction count and CPI to find the total number of clock cycles for each sequence:

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{C}_i)$$

This yields

$$\text{CPU clock cycles}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10 \text{ cycles}$$

$$\text{CPU clock cycles}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9 \text{ cycles}$$

So code sequence 2 is faster, even though it executes one extra instruction. Since code sequence 2 takes fewer overall clock cycles but has more instructions, it must have a lower CPI. The CPI values can be computed by

$$\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instruction count}}$$

$$\text{CPI}_1 = \frac{\text{CPU clock cycles}_1}{\text{Instruction count}_1} = \frac{10}{5} = 2.0$$

$$\text{CPI}_2 = \frac{\text{CPU clock cycles}_2}{\text{Instruction count}_2} = \frac{9}{6} = 1.5$$

The BIG Picture

Figure 1.14 shows the basic measurements at different levels in the computer and what is being measured in each case. We can see how these factors are combined to yield execution time measured in seconds per program:

$$\text{Time} = \text{Seconds/Program} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Always bear in mind that the only complete and reliable measure of computer performance is time. For example, changing the instruction set to lower the instruction count may lead to an organization with a slower clock cycle time or higher CPI that offsets the improvement in instruction count. Similarly, because CPI depends on type of instructions executed, the code that executes the fewest number of instructions may not be the fastest.

Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

FIGURE 1.14 The basic components of performance and how each is measured.

How can we determine the value of these factors in the performance equation? We can measure the CPU execution time by running the program, and the clock cycle time is usually published as part of the documentation for a computer. The instruction count and CPI can be more difficult to obtain. Of course, if we know the clock rate and CPU execution time, we need only one of the instruction count or the CPI to determine the other.

We can measure the instruction count by using software tools that profile the execution or by using a simulator of the architecture. Alternatively, we can use hardware counters, which are included in most processors, to record a variety of measurements, including the number of instructions executed, the average CPI, and often, the sources of performance loss. Since the instruction count depends on the architecture, but not on the exact implementation, we can measure the instruction count without knowing all the details of the implementation. The CPI, however, depends on a wide variety of design details in the computer, including both the memory system and the processor structure (as we will see in Chapters 4 and 5), as well as on the mix of instruction types executed in an application. Thus, CPI varies by application, as well as among implementations with the same instruction set.

The above example shows the danger of using only one factor (instruction count) to assess performance. When comparing two computers, you must look at all three components, which combine to form execution time. If some of the factors are identical, like the clock rate in the above example, performance can be determined by comparing all the nonidentical factors. Since CPI varies by **instruction mix**, both instruction count and CPI must be compared, even if clock rates are identical. Several exercises at the end of this chapter ask you to evaluate a series of computer and compiler enhancements that affect clock rate, CPI, and instruction count. In Section 1.8, we'll examine a common performance measurement that does not incorporate all the terms and can thus be misleading.

instruction mix A measure of the dynamic frequency of instructions across one or many programs.

Understanding Program Performance

The performance of a program depends on the algorithm, the language, the compiler, the architecture, and the actual hardware. The following table summarizes how these components affect the factors in the CPU performance equation.

Hardware or software component	Affects what?	How?
Algorithm	Instruction count, possibly CPI	The algorithm determines the number of source program instructions executed and hence the number of processor instructions executed. The algorithm may also affect the CPI, by favoring slower or faster instructions. For example, if the algorithm uses more floating-point operations, it will tend to have a higher CPI.
Programming language	Instruction count, CPI	The programming language certainly affects the instruction count, since statements in the language are translated to processor instructions, which determine instruction count. The language may also affect the CPI because of its features; for example, a language with heavy support for data abstraction (e.g., Java) will require indirect calls, which will use higher CPI instructions.
Compiler	Instruction count, CPI	The efficiency of the compiler affects both the instruction count and average cycles per instruction, since the compiler determines the translation of the source language instructions into computer instructions. The compiler's role can be very complex and affect the CPI in complex ways.
Instruction set architecture	Instruction count, clock rate, CPI	The instruction set architecture affects all three aspects of CPU performance, since it affects the instructions needed for a function, the cost in cycles of each instruction, and the overall clock rate of the processor.

Elaboration: Although you might expect that the minimum CPI is 1.0, as we'll see in Chapter 4, some processors fetch and execute multiple instructions per clock cycle. To reflect that approach, some designers invert CPI to talk about *IPC*, or *instructions per clock cycle*. If a processor executes on average 2 instructions per clock cycle, then it has an IPC of 2 and hence a CPI of 0.5.

Check Yourself

A given application written in Java runs 15 seconds on a desktop processor. A new Java compiler is released that requires only 0.6 as many instructions as the old compiler. Unfortunately, it increases the CPI by 1.1. How fast can we expect the application to run using this new compiler? Pick the right answer from the three choices below

a. $\frac{15 \times 0.6}{1.1} = 8.2 \text{ sec}$

b. $15 \times 0.6 \times 1.1 = 9.9 \text{ sec}$

c. $\frac{15 \times 1.1}{0.6} = 27.5 \text{ sec}$

1.5 The Power Wall

Figure 1.15 shows the increase in clock rate and power of eight generations of Intel microprocessors over 25 years. Both clock rate and power increased rapidly for decades, and then flattened off recently. The reason they grew together is that they are correlated, and the reason for their recent slowing is that we have run into the practical power limit for cooling commodity microprocessors.

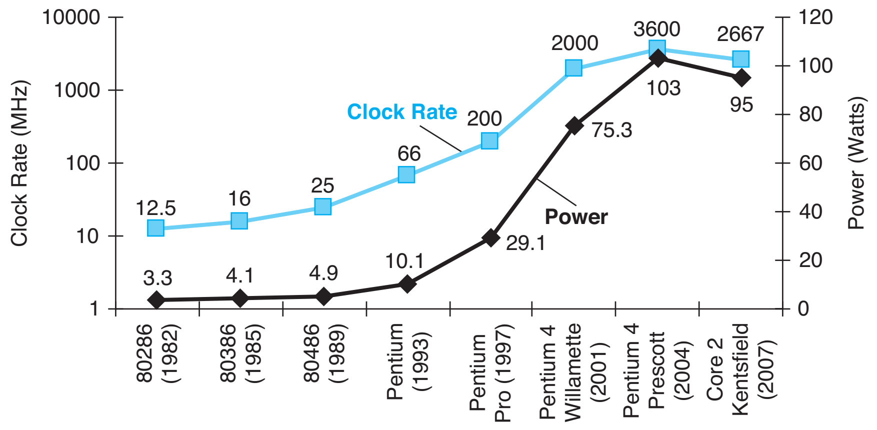


FIGURE 1.15 Clock rate and Power for Intel x86 microprocessors over eight generations and 25 years. The Pentium 4 made a dramatic jump in clock rate and power but less so in performance. The Prescott thermal problems led to the abandonment of the Pentium 4 line. The Core 2 line reverts to a simpler pipeline with lower clock rates and multiple processors per chip.

The dominant technology for integrated circuits is called CMOS (complementary metal oxide semiconductor). For CMOS, the primary source of power dissipation is so-called dynamic power—that is, power that is consumed during switching. The dynamic power dissipation depends on the capacitive loading of each transistor, the voltage applied, and the frequency that the transistor is switched:

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}$$

Frequency switched is a function of the clock rate. The capacitive load per transistor is a function of both the number of transistors connected to an output (called the *fanout*) and the technology, which determines the capacitance of both wires and transistors.

How could clock rates grow by a factor of 1000 while power grew by only a factor of 30? Power can be reduced by lowering the voltage, which occurred with each new generation of technology, and power is a function of the voltage squared. Typically, the voltage was reduced about 15% per generation. In 20 years, voltages have gone from 5V to 1V, which is why the increase in power is only 30 times.

EXAMPLE

Relative Power

Suppose we developed a new, simpler processor that has 85% of the capacitive load of the more complex older processor. Further, assume that it has adjustable voltage so that it can reduce voltage 15% compared to processor B, which results in a 15% shrink in frequency. What is the impact on dynamic power?

ANSWER

$$\frac{\text{Power}_{\text{new}}}{\text{Power}_{\text{old}}} = \frac{\langle \text{Capacitive load} \times 0.85 \rangle \times \langle \text{Voltage} \times 0.85 \rangle^2 \times \langle \text{Frequency switched} \times 0.85 \rangle}{\text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}}$$

Thus the power ratio is

$$0.85^4 = 0.52$$

Hence, the new processor uses about half the power of the old processor.

The problem today is that further lowering of the voltage appears to make the transistors too leaky, like water faucets that cannot be completely shut off. Even today about 40% of the power consumption is due to leakage. If transistors started leaking more, the whole process could become unwieldy.

To try to address the power problem, designers have already attached large devices to increase cooling, and they turn off parts of the chip that are not used in a given clock cycle. Although there are many more expensive ways to cool chips and thereby raise their power to, say, 300 watts, these techniques are too expensive for desktop computers.

Since computer designers slammed into a power wall, they needed a new way forward. They chose a different way from the way they designed microprocessors for their first 30 years.

Elaboration: Although dynamic power is the primary source of power dissipation in CMOS, static power dissipation occurs because of leakage current that flows even when a transistor is off. As mentioned above, leakage is typically responsible for 40% of the power consumption in 2008. Thus, increasing the number of transistors increases power dissipation, even if the transistors are always off. A variety of design techniques and technology innovations are being deployed to control leakage, but it's hard to lower voltage further.

1.6 The Sea Change: The Switch from Uniprocessors to Multiprocessors

"Up to now, most software has been like music written for a solo performer; with the current generation of chips we're getting a little experience with duets and quartets and other small ensembles; but scoring a work for large orchestra and chorus is a different kind of challenge."

Brian Hayes, *Computing in a Parallel Universe*, 2007.

The power limit has forced a dramatic change in the design of microprocessors. Figure 1.16 shows the improvement in response time of programs for desktop microprocessors over time. Since 2002, the rate has slowed from a factor of 1.5 per year to less than a factor of 1.2 per year.

Rather than continuing to decrease the response time of a single program running on the single processor, as of 2006 all desktop and server companies are shipping microprocessors with multiple processors per chip, where the benefit is often more on throughput than on response time. To reduce confusion between the words processor and microprocessor, companies refer to processors as "cores," and such microprocessors are generically called multicore microprocessors. Hence, a "quadcore" microprocessor is a chip that contains four processors or four cores.

Figure 1.17 shows the number of processors (cores), power, and clock rates of recent microprocessors. The official plan of record for many companies is to double the number of cores per microprocessor per semiconductor technology generation, which is about every two years (see Chapter 7).

In the past, programmers could rely on innovations in hardware, architecture, and compilers to double performance of their programs every 18 months without having to change a line of code. Today, for programmers to get significant improvement in response time, they need to rewrite their programs to take advantage of multiple processors. Moreover, to get the historic benefit of running faster on new microprocessors, programmers will have to continue to improve performance of their code as the number of cores doubles.

To reinforce how the software and hardware systems work hand in hand, we use a special section, *Hardware/Software Interface*, throughout the book, with the first one appearing below. These elements summarize important insights at this critical interface.

Hardware/Software Interface

Parallelism has always been critical to performance in computing, but it was often hidden. Chapter 4 will explain pipelining, an elegant technique that runs programs faster by overlapping the execution of instructions. This is one example of *instruction-level parallelism*, where the parallel nature of the hardware is abstracted away so the programmer and compiler can think of the hardware as executing instructions sequentially.

Forcing programmers to be aware of the parallel hardware and to explicitly rewrite their programs to be parallel had been the "third rail" of computer architecture, for companies in the past that depended on such a change in behavior failed (see **Section 7.14** on the CD). From this historical perspective, it's startling that the whole IT industry has bet its future that programmers will finally successfully switch to explicitly parallel programming.

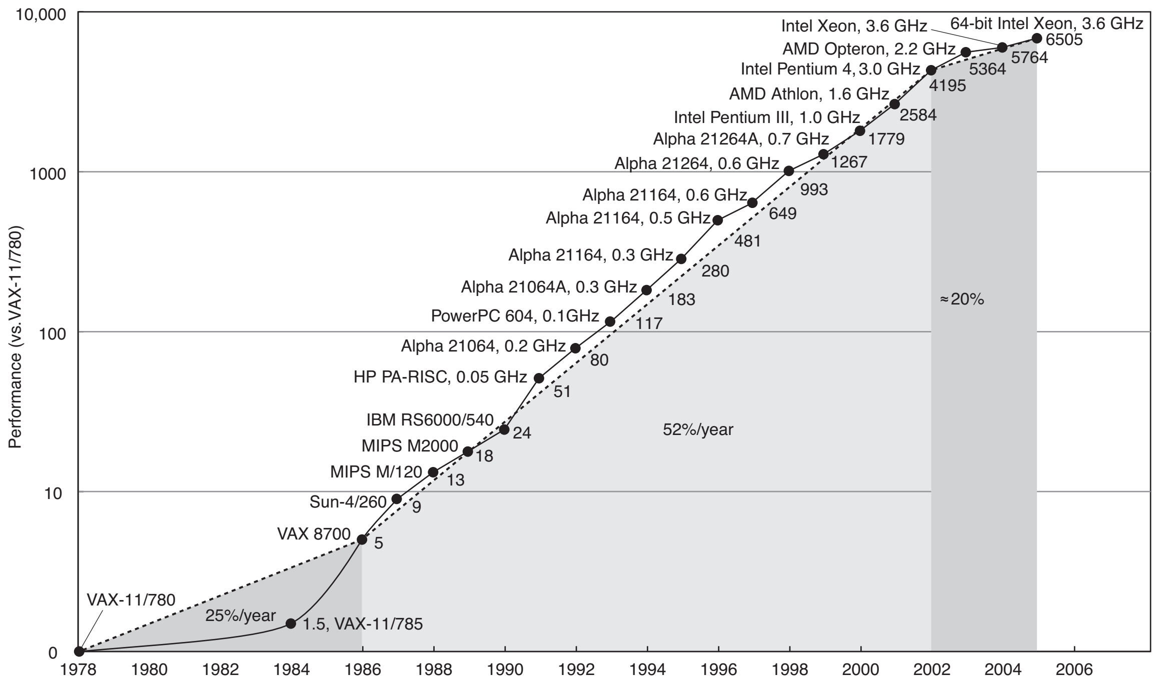


FIGURE 1.16 Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.8). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. By 2002, this growth led to a difference in performance of about a factor of seven. Performance for floating-point-oriented calculations has increased even faster. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 20% per year.

Product	AMD Opteron X4 (Barcelona)	Intel Nehalem	IBM Power 6	Sun Ultra SPARC T2 (Niagara 2)
Cores per chip	4	4	2	8
Clock rate	2.5 GHz	~ 2.5 GHz ?	4.7 GHz	1.4 GHz
Microprocessor power	120 W	~ 100 W ?	~ 100 W ?	94 W

FIGURE 1.17 Number of cores per chip, clock rate, and power for 2008 multicore microprocessors.

Why has it been so hard for programmers to write explicitly parallel programs? The first reason is that parallel programming is by definition performance programming, which increases the difficulty of programming. Not only does the program need to be correct, solve an important problem, and provide a useful interface to the people or other programs that invoke it, the program must also be fast. Otherwise, if you don't need performance, just write a sequential program.

The second reason is that fast for parallel hardware means that the programmer must divide an application so that each processor has roughly the same amount to do at the same time, and that the overhead of scheduling and coordination doesn't fritter away the potential performance benefits of parallelism.

As an analogy, suppose the task was to write a newspaper story. Eight reporters working on the same story could potentially write a story eight times faster. To achieve this increased speed, one would need to break up the task so that each reporter had something to do at the same time. Thus, we must *schedule* the subtasks. If anything went wrong and just one reporter took longer than the seven others did, then the benefits of having eight writers would be diminished. Thus, we must *balance the load* evenly to get the desired speedup. Another danger would be if reporters had to spend a lot of time talking to each other to write their sections. You would also fall short if one part of the story, such as the conclusion, couldn't be written until all of the other parts were completed. Thus, care must be taken to *reduce communication and synchronization overhead*. For both this analogy and parallel programming, the challenges include scheduling, load balancing, time for synchronization, and overhead for communication between the parties. As you might guess, the challenge is stiffer with more reporters for a newspaper story and more processors for parallel programming.

To reflect this sea change in the industry, the next five chapters in this edition of the book each have a section on the implications of the parallel revolution to that chapter:

- *Chapter 2, Section 2.11: Parallelism and Instructions: Synchronization*. Usually independent parallel tasks need to coordinate at times, such as to say when they have completed their work. This chapter explains the instructions used by multicore processors to synchronize tasks.
- *Chapter 3, Section 3.6: Parallelism and Computer Arithmetic: Associativity*. Often parallel programmers start from a working sequential program. A natural question to learn if their parallel version works is, "does it get the same answer?" If not, a logical conclusion is that there are bugs in the new version. This logic assumes that computer arithmetic is associative: you get the same sum when adding a million numbers, no matter what the order. This chapter explains that while this logic holds for integers, it doesn't hold for floating-point numbers.
- *Chapter 4, Section 4.10: Parallelism and Advanced Instruction-Level Parallelism*. Given the difficulty of explicitly parallel programming, tremendous effort was invested in the 1990s in having the hardware and the compiler uncover implicit parallelism. This chapter describes some of these aggressive techniques, including fetching and executing multiple instructions simultaneously and guessing on the outcomes of decisions, and executing instructions speculatively.

- *Chapter 5, Section 5.8: Parallelism and Memory Hierarchies: Cache Coherence.* One way to lower the cost of communication is to have all processors use the same address space, so that any processor can read or write any data. Given that all processors today use caches to keep a temporary copy of the data in faster memory near the processor, it's easy to imagine that parallel programming would be even more difficult if the caches associated with each processor had inconsistent values of the shared data. This chapter describes the mechanisms that keep the data in all caches consistent.
- *Chapter 6, Section 6.9: Parallelism and I/O: Redundant Arrays of Inexpensive Disks.* If you ignore input and output in this parallel revolution, the unintended consequence of parallel programming may be to make your parallel program spend most of its time waiting for I/O. This chapter describes RAID, a technique to accelerate the performance of storage accesses. RAID points out another potential benefit of parallelism: by having many copies of resources, the system can continue to provide service despite a failure of one resource. Hence, RAID can improve both I/O performance and availability.

In addition to these sections, there is a full chapter on parallel processing. Chapter 7 goes into more detail on the challenges of parallel programming; presents the two contrasting approaches to communication of shared addressing and explicit message passing; describes a restricted model of parallelism that is easier to program; discusses the difficulty of benchmarking parallel processors; introduces a new simple performance model for multicore microprocessors and finally describes and evaluates four examples of multicore microprocessors using this model.

Starting with this edition of the book, Appendix A describes an increasingly popular hardware component that is included with desktop computers, the graphics processing unit (GPU). Invented to accelerate graphics, GPUs are becoming programming platforms in their own right. As you might expect, given these times, GPUs are highly parallel. Appendix A describes the NVIDIA GPU and highlights parts of its parallel programming environment.

I thought [computers] would be a universally applicable idea, like a book is. But I didn't think it would develop as fast as it did, because I didn't envision we'd be able to get as many parts on a chip as we finally got. The transistor came along unexpectedly. It all happened much faster than we expected.

J. Presper Eckert, coinventor of ENIAC, speaking in 1991

1.7 Real Stuff: Manufacturing and Benchmarking the AMD Opteron X4

Each chapter has a section entitled "Real Stuff" that ties the concepts in the book with a computer you may use every day. These sections cover the technology underlying modern computers. For this first "Real Stuff" section, we look at how integrated circuits are manufactured and how performance and power are measured, with the AMD Opteron X4 as the example.

Let's start at the beginning. The manufacture of a chip begins with **silicon**, a substance found in sand. Because silicon does not conduct electricity well, it is called a **semiconductor**. With a special chemical process, it is possible to add materials to silicon that allow tiny areas to transform into one of three devices:

- Excellent conductors of electricity (using either microscopic copper or aluminum wire)
- Excellent insulators from electricity (like plastic sheathing or glass)
- Areas that can conduct *or* insulate under special conditions (as a switch)

Transistors fall in the last category. A VLSI circuit, then, is just billions of combinations of conductors, insulators, and switches manufactured in a single small package.

The manufacturing process for integrated circuits is critical to the cost of the chips and hence important to computer designers. Figure 1.18 shows that process. The process starts with a **silicon crystal ingot**, which looks like a giant sausage. Today, ingots are 8–12 inches in diameter and about 12–24 inches long. An ingot is finely sliced into **wafers** no more than 0.1 inch thick. These wafers then go through a series of processing steps, during which patterns of chemicals are placed on

silicon A natural element that is a semiconductor.

semiconductor A substance that does not conduct electricity well.

silicon crystal ingot A rod composed of a silicon crystal that is between 8 and 12 inches in diameter and about 12 to 24 inches long.

wafer A slice from a silicon ingot no more than 0.1 inch thick, used to create chips.

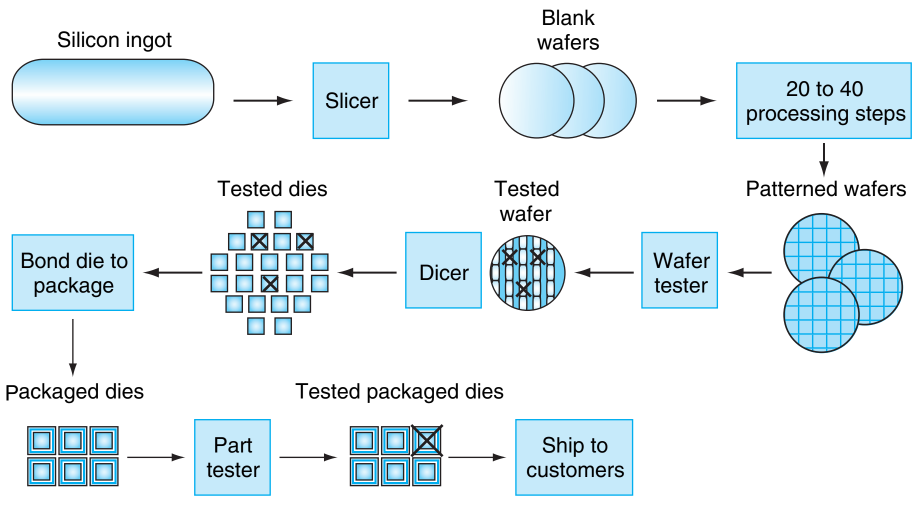


FIGURE 1.18 The chip manufacturing process. After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers (see Figure 1.19). These patterned wafers are then tested with a wafer tester, and a map of the good parts is made. Then, the wafers are diced into dies (see Figure 1.9). In this figure, one wafer produced 20 dies, of which 17 passed testing. (*X* means the die is bad.) The yield of good dies in this case was 17/20, or 85%. These good dies are then bonded into packages and tested one more time before shipping the packaged parts to customers. One bad packaged part was found in this final test.

each wafer, creating the transistors, conductors, and insulators discussed earlier. Today's integrated circuits contain only one layer of transistors but may have from two to eight levels of metal conductor, separated by layers of insulators.

A single microscopic flaw in the wafer itself or in one of the dozens of patterning steps can result in that area of the wafer failing. These **defects**, as they are called, make it virtually impossible to manufacture a perfect wafer. To cope with imperfection, several strategies have been used, but the simplest is to place many independent components on a single wafer. The patterned wafer is then chopped up, or *diced,* into these components, called **dies** and more informally known as **chips**. Figure 1.19 is a photograph of a wafer containing microprocessors before they have been diced; earlier, Figure 1.9 on page 20 shows an individual microprocessor die and its major components.

defect A microscopic flaw in a wafer or in patterning steps that can result in the failure of the die containing that defect.

die The individual rectangular sections that are cut from a wafer, more informally known as **chips**.

yield The percentage of good dies from the total number of dies on the wafer.

Dicing enables you to discard only those dies that were unlucky enough to contain the flaws, rather than the whole wafer. This concept is quantified by the **yield** of a process, which is defined as the percentage of good dies from the total number of dies on the wafer.

The cost of an integrated circuit rises quickly as the die size increases, due both to the lower yield and the smaller number of dies that fit on a wafer. To reduce the cost, a large die is often "shrunk" by using the next generation process, which incorporates smaller sizes for both transistors and wires. This improves the yield and the die count per wafer.

Once you've found good dies, they are connected to the input/output pins of a package, using a process called *bonding*. These packaged parts are tested a final time, since mistakes can occur in packaging, and then they are shipped to customers.

As mentioned above, an increasingly important design constraint is power. Power is a challenge for two reasons. First, power must be brought in and distributed around the chip; modern microprocessors use hundreds of pins just for power and ground! Similarly, multiple levels of interconnect are used solely for power and ground distribution to portions of the chip. Second, power is dissipated as heat and must be removed. An AMD Opteron X4 model 2356 2.0 GHz burns 120 watts in 2008, which must be removed from a chip whose surface area is just over 1 cm^2!

Elaboration: The cost of an integrated circuit can be expressed in three simple equations:

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{yield}}$$

$$\text{Dies per wafer} \approx \frac{\text{Wafer area}}{\text{Die area}}$$

$$\text{Yield} = \frac{1}{(1 + (\text{Defects per area} \times \text{Die area}/2))^2}$$

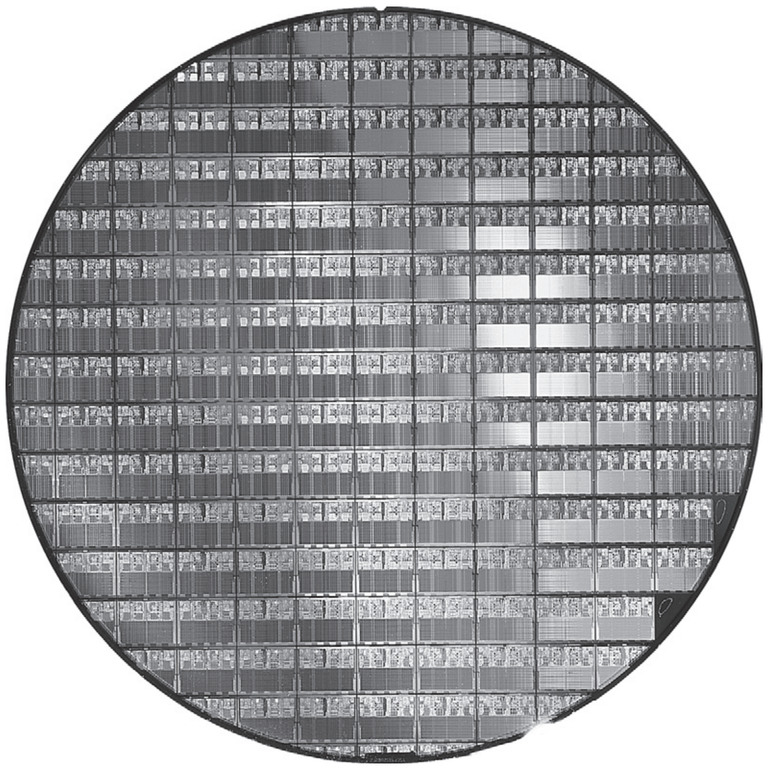


FIGURE 1.19 A 12-inch (300mm) wafer of AMD Opteron X2 chips, the predecessor of Opteron X4 chips (Courtesy AMD). The number of dies per wafer at 100% yield is 117. The several dozen partially rounded chips at the boundaries of the wafer are useless; they are included because it's easier to create the masks used to pattern the silicon. This die uses a 90-nanometer technology, which means that the smallest transistors are approximately 90 nm in size, although they are typically somewhat smaller than the actual feature size, which refers to the size of the transistors as "drawn" versus the final manufactured size.

The first equation is straightforward to derive. The second is an approximation, since it does not subtract the area near the border of the round wafer that cannot accommodate the rectangular dies (see Figure 1.19). The final equation is based on empirical observations of yields at integrated circuit factories, with the exponent related to the number of critical processing steps.

Hence, depending on the defect rate and the size of the die and wafer, costs are generally not linear in die area.

SPEC CPU Benchmark

A computer user who runs the same programs day in and day out would be the perfect candidate to evaluate a new computer. The set of programs run would form a **workload**. To evaluate two computer systems, a user would simply compare the execution time of the workload on the two computers. Most users, however, are not in this situation. Instead, they must rely on other methods that measure the performance of a candidate computer, hoping that the methods will reflect how well the computer will perform with the user's workload. This alternative is usually followed by evaluating the computer using a set of **benchmarks**—programs specifically chosen to measure performance. The benchmarks form a workload that the user hopes will predict the performance of the actual workload.

workload A set of programs run on a computer that is either the actual collection of applications run by a user or constructed from real programs to approximate such a mix. A typical workload specifies both the programs and the relative frequencies.

benchmark A program selected for use in comparing computer performance.

SPEC (System Performance Evaluation Cooperative) is an effort funded and supported by a number of computer vendors to create standard sets of benchmarks for modern computer systems. In 1989, SPEC originally created a benchmark set focusing on processor performance (now called SPEC89), which has evolved through five generations. The latest is SPEC CPU2006, which consists of a set of 12 integer benchmarks (CINT2006) and 17 floating-point benchmarks (CFP2006). The integer benchmarks vary from part of a C compiler to a chess program to a quantum computer simulation. The floating-point benchmarks include structured grid codes for finite element modeling, particle method codes for molecular dynamics, and sparse linear algebra codes for fluid dynamics.

Figure 1.20 describes the SPEC integer benchmarks and their execution time on the Opteron X4 and shows the factors that explain execution time: instruction count, CPI, and clock cycle time. Note that CPI varies by a factor of 13.

To simplify the marketing of computers, SPEC decided to report a single number to summarize all 12 integer benchmarks. The execution time measurements are first normalized by dividing the execution time on a reference processor by the execution time on the measured computer; this normalization yields a measure, called the *SPECratio,* which has the advantage that bigger numeric results indicate faster performance (i.e., the SPECratio is the inverse of execution time). A CINT2006 or CFP2006 summary measurement is obtained by taking the geometric mean of the SPECratios.

Elaboration: When comparing two computers using SPECratios, use the geometric mean so that it gives the same relative answer no matter what computer is used to normalize the results. If we averaged the normalized execution time values with an arithmetic mean, the results would vary depending on the computer we choose as the reference.

Description	Name	Instruction Count × 10^9	CPI	Clock cycle time (seconds × 10^{-9})	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2,118	0.75	0.4	637	9,770	15.3
Block-sorting compression	bzip2	2,389	0.85	0.4	817	9,650	11.8
GNU C compiler	gcc	1,050	1.72	0.4	724	8,050	11.1
Combinatorial optimization	mcf	336	10.00	0.4	1,345	9,120	6.8
Go game (AI)	go	1,658	1.09	0.4	721	10,490	14.6
Search gene sequence	hmmer	2,783	0.80	0.4	890	9,330	10.5
Chess game (AI)	sjeng	2,176	0.96	0.4	837	12,100	14.5
Quantum computer simulation	libquantum	1,623	1.61	0.4	1,047	20,720	19.8
Video compression	h264avc	3,102	0.80	0.4	993	22,130	22.3
Discrete event simulation library	omnetpp	587	2.94	0.4	690	6,250	9.1
Games/path finding	astar	1,082	1.79	0.4	773	7,020	9.1
XML parsing	xalancbmk	1,058	2.70	0.4	1,143	6,900	6.0
Geometric Mean							11.7

FIGURE 1.20 SPECINTC2006 benchmarks running on AMD Opteron X4 model 2356 (Barcelona). As the equation on page 35 explains, execution time is the product of the three factors in this table: instruction count in billions, clocks per instruction (CPI), and clock cycle time in nanoseconds. SPECratio is simply the reference time, which is supplied by SPEC, divided by the measured execution time. The single number quoted as SPECINTC2006 is the geometric mean of the SPECratios. Figure 5.40 on page 542 shows that mcf, libquantum, omnetpp, and xalancbmk have relatively high CPIs because they have high cache miss rates.

The formula for the geometric mean is

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

where Execution time ratio$_i$ is the execution time, normalized to the reference computer, for the ith program of a total of n in the workload, and

$$\prod_{i=1}^{n} a_i \text{ means the product } a_1 \times a_2 \times \ldots \times a_n$$

SPEC Power Benchmark

Today, SPEC offers a dozen different benchmark sets designed to test a wide variety of computing environments using real applications and strictly specified execution rules and reporting requirements. The most recent is SPECpower. It reports power consumption of servers at different workload levels, divided into 10% increments, over a period of time. Figure 1.21 shows the results for a server using Barcelona.

SPECpower started with the SPEC benchmark for Java business applications (SPECJBB2005), which exercises the processors, caches, and main memory as well as the Java virtual machine, compiler, garbage collector, and pieces of the operating

Target Load %	Performance (ssj_ops)	Average Power (Watts)
100%	231,867	295
90%	211,282	286
80%	185,803	275
70%	163,427	265
60%	140,160	256
50%	118,324	246
40%	92,035	233
30%	70,500	222
20%	47,126	206
10%	23,066	180
0%	0	141
Overall Sum	1,283,590	2,605
Σssj_ops / Σpower =		493

FIGURE 1.21 SPECpower_ssj2008 running on dual socket 2.3 GHz AMD Opteron X4 2356 (Barcelona) with 16 GB Of DDR2-667 DRAM and one 500 GB disk.

system. Performance is measured in throughput, and the units are business operations per second. Once again, to simplify the marketing of computers, SPEC boils these numbers down to a single number, called "overall ssj_ops per Watt." The formula for this single summarizing metric is

$$\text{overall ssj\_ops per Watt} = \left(\sum_{i=0}^{10} \text{ssj\_ops}_i\right) / \left(\sum_{i=0}^{10} \text{power}_i\right)$$

where ssj_ops$_i$ is performance at each 10% increment and power$_i$ is power consumed at each performance level.

Check Yourself

A key factor in determining the cost of an integrated circuit is volume. Which of the following are reasons why a chip made in high volume should cost less?

1. With high volumes, the manufacturing process can be tuned to a particular design, increasing the yield.
2. It is less work to design a high-volume part than a low-volume part.
3. The masks used to make the chip are expensive, so the cost per chip is lower for higher volumes.
4. Engineering development costs are high and largely independent of volume; thus, the development cost per die is lower with high-volume parts.
5. High-volume parts usually have smaller die sizes than low-volume parts and therefore have higher yield per wafer.

1.8 Fallacies and Pitfalls

Science must begin with myths, and the criticism of myths.

Sir Karl Popper, *The Philosophy of Science,* 1957

The purpose of a section on fallacies and pitfalls, which will be found in every chapter, is to explain some commonly held misconceptions that you might encounter. We call such misbeliefs *fallacies.* When discussing a fallacy, we try to give a counterexample. We also discuss *pitfalls*, or easily made mistakes. Often pitfalls are generalizations of principles that are true in a limited context. The purpose of these sections is to help you avoid making these mistakes in the computers you may design or use. Cost/performance fallacies and pitfalls have ensnared many a computer architect, including us. Accordingly, this section suffers no shortage of relevant examples. We start with a pitfall that traps many designers and reveals an important relationship in computer design.

Pitfall: Expecting the improvement of one aspect of a computer to increase overall performance by an amount proportional to the size of the improvement.

This pitfall has visited designers of both hardware and software. A simple design problem illustrates it well. Suppose a program runs in 100 seconds on a computer, with multiply operations responsible for 80 seconds of this time. How much do I have to improve the speed of multiplication if I want my program to run five times faster?

The execution time of the program after making the improvement is given by the following simple equation known as **Amdahl's law**:

Amdahl's law A rule stating that the performance enhancement possible with a given improvement is limited by the amount that the improved feature is used. It is a quantitative version of the law of diminishing returns.

$$\text{Execution time after improvement} =$$
$$\frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

For this problem:

$$\text{Execution time after improvement} = \frac{80 \text{ seconds}}{n} + (100 - 80 \text{ seconds})$$

Since we want the performance to be five times faster, the new execution time should be 20 seconds, giving

$$20 \text{ seconds} = \frac{80 \text{ seconds}}{n} + 20 \text{ seconds}$$
$$0 = \frac{80 \text{ seconds}}{n}$$

That is, there is *no amount* by which we can enhance-multiply to achieve a fivefold increase in performance, if multiply accounts for only 80% of the workload.

The performance enhancement possible with a given improvement is limited by the amount that the improved feature is used. This concept also yields what we call the law of diminishing returns in everyday life.

We can use Amdahl's law to estimate performance improvements when we know the time consumed for some function and its potential speedup. Amdahl's law, together with the CPU performance equation, is a handy tool for evaluating potential enhancements. Amdahl's law is explored in more detail in the exercises.

A common theme in hardware design is a corollary of Amdahl's law: *Make the common case fast*. This simple guideline reminds us that in many cases the frequency with which one event occurs may be much higher than the frequency of another. Amdahl's law reminds us that the opportunity for improvement is affected by how much time the event consumes. Thus, making the common case fast will tend to enhance performance better than optimizing the rare case. Ironically, the common case is often simpler than the rare case and hence is often easier to enhance.

Amdahl's law is also used to argue for practical limits to the number of parallel processors. We examine this argument in the Fallacies and Pitfalls section of Chapter 7.

Fallacy: Computers at low utilization use little power.

Power efficiency matters at low utilizations because server workloads vary. CPU utilization for servers at Google, for example, is between 10% and 50% most of the time and at 100% less than 1% of the time. Figure 1.22 shows power for servers with the best SPECpower results at 100% load, 50% load, 10% load, and idle. Even servers that are only 10% utilized burn about two-thirds of their peak power.

Since servers' workloads vary but use a large fraction of peak power, Luiz Barroso and Urs Hölzle [2007] argue that we should redesign hardware to achieve "energy-proportional computing." If future servers used, say, 10% of peak power at 10% workload, we could reduce the electricity bill of datacenters and become good corporate citizens in an era of increasing concern about CO_2 emissions.

Server Manufacturer	Micro-processor	Total Cores/ Sockets	Clock Rate	Peak Performance (ssj_ops)	100% Load Power	50% Load Power	50% Load/ 100% Power	10% Load Power	10% Load/ 100% Power	Active Idle Power	Active Idle/ 100% Power
HP	Xeon E5440	8/2	3.0 GHz	308,022	269 W	227 W	84%	174 W	65%	160 W	59%
Dell	Xeon E5440	8/2	2.8 GHz	305,413	276 W	230 W	83%	173 W	63%	157 W	57%
Fujitsu Seimens	Xeon X3220	4/1	2.4 GHz	143,742	132 W	110 W	83%	85 W	65%	80 W	60%

FIGURE 1.22 SPECPower results for three servers with the best overall ssj_ops per watt in the fourth quarter of 2007. The overall ssj_ops per watt of the three servers are 698, 682, and 667, respectively. The memory of the top two servers is 16 GB and the bottom is 8 GB.

Pitfall: Using a subset of the performance equation as a performance metric.

We have already shown the fallacy of predicting performance based on simply one of clock rate, instruction count, or CPI. Another common mistake is to use only two of the three factors to compare performance. Although using two of the three factors may be valid in a limited context, the concept is also easily misused. Indeed, nearly all proposed alternatives to the use of time as the performance metric have led eventually to misleading claims, distorted results, or incorrect interpretations.

One alternative to time is **MIPS (million instructions per second)**. For a given program, MIPS is simply

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

million instructions per second (MIPS) A measurement of program execution speed based on the number of millions of instructions. MIPS is computed as the instruction count divided by the product of the execution time and 10^6.

Since MIPS is an instruction execution rate, MIPS specifies performance inversely to execution time; faster computers have a higher MIPS rating. The good news about MIPS is that it is easy to understand, and faster computers mean bigger MIPS, which matches intuition.

There are three problems with using MIPS as a measure for comparing computers. First, MIPS specifies the instruction execution rate but does not take into account the capabilities of the instructions. We cannot compare computers with different instruction sets using MIPS, since the instruction counts will certainly differ. Second, MIPS varies between programs on the same computer; thus, a computer cannot have a single MIPS rating. For example, by substituting for execution time, we see the relationship between MIPS, clock rate, and CPI:

$$\text{MIPS} = \frac{\text{Instruction count}}{\dfrac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

Recall that CPI varied by 13× for SPEC CPU2006 on Opteron X4, so MIPS does as well. Finally, and most importantly, if a new program executes more instructions but each instruction is faster, MIPS can vary independently from performance!

Check Yourself

Consider the following performance measurements for a program:

Measurement	Computer A	Computer B
Instruction count	10 billion	8 billion
Clock rate	4 GHz	4 GHz
CPI	1.0	1.1

a. Which computer has the higher MIPS rating?

b. Which computer is faster?

Where . . . the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have 1,000 vacuum tubes and perhaps weigh just 1½ tons.

Popular Mechanics, March 1949

1.9 Concluding Remarks

Although it is difficult to predict exactly what level of cost/performance computers will have in the future, it's a safe bet that they will be much better than they are today. To participate in these advances, computer designers and programmers must understand a wider variety of issues.

Both hardware and software designers construct computer systems in hierarchical layers, with each lower layer hiding details from the level above. This principle of abstraction is fundamental to understanding today's computer systems, but it does not mean that designers can limit themselves to knowing a single abstraction. Perhaps the most important example of abstraction is the interface between hardware and low-level software, called the *instruction set architecture*. Maintaining the instruction set architecture as a constant enables many implementations of that architecture—presumably varying in cost and performance—to run identical software. On the downside, the architecture may preclude introducing innovations that require the interface to change.

There is a reliable method of determining and reporting performance by using the execution time of real programs as the metric. This execution time is related to other important measurements we can make by the following equation:

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

We will use this equation and its constituent factors many times. Remember, though, that individually the factors do not determine performance: only the product, which equals execution time, is a reliable measure of performance.

The BIG Picture

Execution time is the only valid and unimpeachable measure of performance. Many other metrics have been proposed and found wanting. Sometimes these metrics are flawed from the start by not reflecting execution time; other times a metric that is valid in a limited context is extended and used beyond that context or without the additional clarification needed to make it valid.

The key hardware technology for modern processors is silicon. Equal in importance to an understanding of integrated circuit technology is an understanding of the expected rates of technological change. While silicon fuels the rapid advance of hardware, new ideas in the organization of computers have improved price/performance. Two of the key ideas are exploiting parallelism in the program, typically today via multiple processors, and exploiting locality of accesses to a memory hierarchy, typically via caches.

Power has replaced die area as the most critical resource of microprocessor design. Conserving power while trying to increase performance has forced the hardware industry to switch to multicore microprocessors, thereby forcing the software industry to switch to programming parallel hardware.

Computer designs have always been measured by cost and performance, as well as other important factors such as power, reliability, cost of ownership, and scalability. Although this chapter has focused on cost, performance, and power, the best designs will strike the appropriate balance for a given market among all the factors.

Road Map for This Book

At the bottom of these abstractions are the five classic components of a computer: datapath, control, memory, input, and output (refer to Figure 1.4). These five components also serve as the framework for the rest of the chapters in this book:

- *Datapath:* Chapters 3, 4, 7, and Appendix A
- *Control:* Chapters 4, 7, and Appendix A
- *Memory:* Chapter 5
- *Input:* Chapter 6
- *Output:* Chapter 6

As mentioned above, Chapter 4 describes how processors exploit implicit parallelism, Chapter 7 describes the explicitly parallel multicore microprocessors that are at the heart of the parallel revolution, and Appendix A describes the highly parallel graphics processor chip. Chapter 5 describes how a memory hierarchy exploits locality. Chapter 2 describes instruction sets—the interface between compilers and the computer—and emphasizes the role of compilers and programming languages in using the features of the instruction set. Appendix B provides a reference for the instruction set of Chapter 2. Chapter 3 describes how computers handle arithmetic data. **Appendix C**, on the CD, introduces logic design.

1.10 Historical Perspective and Further Reading

An active field of science is like an immense anthill; the individual almost vanishes into the mass of minds tumbling over each other, carrying information from place to place, passing it around at the speed of light.

Lewis Thomas, "Natural Science," in *The Lives of a Cell*, 1974

For each chapter in the text, a section devoted to a historical perspective can be found on the CD that accompanies this book. We may trace the development of an idea through a series of computers or describe some important projects, and we provide references in case you are interested in probing further.

The historical perspective for this chapter provides a background for some of the key ideas presented in this opening chapter. Its purpose is to give you the human story behind the technological advances and to place achievements in their historical context. By understanding the past, you may be better able to understand the forces that will shape computing in the future. Each historical perspectives section on the CD ends with suggestions for further reading, which are also collected separately on the CD under the section "**Further Reading**." The rest of **Section 1.10** is found on the CD.

1.11 Exercises

Contributed by Javier Bruguera of Universidade de Santiago de Compostela

Most of the exercises in this edition are designed so that they feature a qualitative description supported by a table that provides alternative quantitative parameters. These parameters are needed to solve the questions that comprise the exercise. Individual questions can be solved using any or all of the parameters—you decide how many of the parameters should be considered for any given exercise question. For example, it is possible to say "complete Question 4.1.1 using the parameters given in row A of the table." Alternately, instructors can customize these exercises to create novel solutions by replacing the given parameters with your own unique values.

The number of quantitative exercises varies from chapter to chapter and depends largely on the topics covered. More conventional exercises are provided where the quantitative approach does not fit.

The relative time ratings of exercises are shown in square brackets after each exercise number. On average, an exercise rated [10] will take you twice as long as one rated [5]. Sections of the text that should be read before attempting an exercise will be given in angled brackets; for example, <1.3> means you should have read Section 1.3, Under the Covers, to help you solve this exercise.

Exercise 1.1

Find the word or phrase from the list below that best matches the description in the following questions. Use the numbers to the left of words in the answer. Each answer should be used only once.

1.	virtual worlds	**14.**	operating system
2.	desktop computers	**15.**	compiler
3.	servers	**16.**	bit
4.	low-end servers	**17.**	instruction
5.	supercomputers	**18.**	assembly language
6.	terabyte	**19.**	machine language
7.	petabyte	**20.**	C
8.	data centers	**21.**	assembler
9.	embedded computers	**22.**	high-level language
10.	multicore processors	**23.**	system software
11.	VHDL	**24.**	application software
12.	RAM	**25.**	Cobol
13.	CPU	**26.**	Fortran

1.1.1 [2] <1.1> Computer used to run large problems and usually accessed via a network

1.1.2 [2] <1.1> 10^{15} or 2^{50} bytes

1.1.3 [2] <1.1> A class of computers composed of hundred to thousand processors and terabytes of memory and having the highest performance and cost

1.1.4 [2] <1.1> Today's science fiction application that probably will be available in the near future

1.1.5 [2] <1.1> A kind of memory called random access memory

1.1.6 [2] <1.1> Part of a computer called central processor unit

1.1.7 [2] <1.1> Thousands of processors forming a large cluster

1.1.8 [2] <1.1> Microprocessors containing several processors in the same chip

1.1.9 [2] <1.1> Desktop computer without a screen or keyboard usually accessed via a network

1.1.10 [2] <1.1> A computer used to running one predetermined application or collection of software

1.1.11 [2] <1.1> Special language used to describe hardware components

1.1.12 [2] <1.1> Personal computer delivering good performance to single users at low cost

1.1.13 [2] <1.2> Program that translates statements in high-level language to assembly language

1.1.14 [2] <1.2> Program that translates symbolic instructions to binary instructions

1.1.15 [2] <1.2> High-level language for business data processing

1.1.16 [2] <1.2> Binary language that the processor can understand

1.1.17 [2] <1.2> Commands that the processors understand

1.1.18 [2] <1.2> High-level language for scientific computation

1.1.19 [2] <1.2> Symbolic representation of machine instructions

1.1.20 [2] <1.2> Interface between user's program and hardware providing a variety of services and supervision functions

1.1.21 [2] <1.2> Software/programs developed by the users

1.1.22 [2] <1.2> Binary digit (value 0 or 1)

1.1.23 [2] <1.2> Software layer between the application software and the hardware that includes the operating system and the compilers

1.1.24 [2] <1.2> High-level language used to write application and system software

1.1.25 [2] <1.2> Portable language composed of words and algebraic expressions that must be translated into assembly language before run in a computer

1.1.26 [2] <1.2> 10^{12} or 2^{40} bytes

Exercise 1.2

Consider the different configurations shown in the table

	Configuration	Resolution	Main Memory	Ethernet Network
a.	1	640 × 480	2 Gbytes	100 Mbit
	2	1280 × 1024	4 Gbytes	1 Gbit
b.	1	1024 × 768	2 Gbytes	100 Mbit
	2	2560 × 1600	4 Gbytes	1Gbit

1.2.1 [10] <1.3> For a color display using 8 bits for each of the primary colors (red, green, blue) per pixel, what should be the minimum size in bytes of the frame buffer to store a frame?

1.2.2 [5] <1.3> How many frames could it store, assuming the memory contains no other information?

1.2.3 [5] <1.3> If a 256 Kbytes file is sent through the Ethernet connection, how long it would take?

For problems below, use the information about access time for every type of memory in the following table.

	Cache	DRAM	Flash Memory	Magnetic Disk
a.	5 ns	50 ns	5 µs	5 ms
b.	7 ns	70 ns	15 µs	20 ms

1.2.4 [5] <1.3> Find how long it takes to read a file from a DRAM if it takes 2 microseconds from the cache memory.

1.2.5 [5] <1.3> Find how long it takes to read a file from a disk if it takes 2 microseconds from the cache memory.

1.2.6 [5] <1.3> Find how long it takes to read a file from a flash memory if it takes 2 microseconds from the cache memory.

Exercise 1.3

Consider three different processors P1, P2, and P3 executing the same instruction set with the clock rates and CPIs given in the following table.

	Processor	Clock Rate	CPI
a.	P1	3 GHz	1.5
	P2	2.5 GHz	1.0
	P3	4 GHz	2.2
b.	P1	2 GHz	1.2
	P2	3 GHz	0.8
	P3	4 GHz	2.0

1.3.1 [5] <1.4> Which processor has the highest performance expressed in instructions per second?

1.3.2 [10] <1.4> If the processors each execute a program in 10 seconds, find the number of cycles and the number of instructions.

1.3.3 [10] <1.4> We are trying to reduce the time by 30% but this leads to an increase of 20% in the CPI. What clock rate should we have to get this time reduction?

For problems below, use the information in the following table.

	Processor	Clock Rate	No. Instructions	Time
a.	P1	3 GHz	20.00E+09	7 s
	P2	2.5 GHz	30.00E+09	10 s
	P3	4 GHz	90.00E+09	9 s
b.	P1	2 GHz	20.00E+09	5 s
	P2	3 GHz	30.00E+09	8 s
	P3	4 GHz	25.00E+09	7 s

1.3.4 [10] <1.4> Find the IPC (instructions per cycle) for each processor.

1.3.5 [5] <1.4> Find the clock rate for P2 that reduces its execution time to that of P1.

1.3.6 [5] <1.4> Find the number of instructions for P2 that reduces its execution time to that of P3.

Exercise 1.4

Consider two different implementations of the same instruction set architecture. There are four classes of instructions, A, B, C, and D. The clock rate and CPI of each implementation are given in the following table.

		Clock Rate	CPI Class A	CPI Class B	CPI Class C	CPI Class D
a.	P1	2.5 GHz	1	2	3	3
	P2	3 GHz	2	2	2	2
b.	P1	2.5 GHz	2	1.5	2	1
	P2	3 GHz	1	2	1	1

1.4.1 [10] <1.4> Given a program with 10^6 instructions divided into classes as follows: 10% class A, 20% class B, 50% class C, and 20% class D, which implementation is faster?

1.4.2 [5] <1.4> What is the global CPI for each implementation?

1.4.3 [5] <1.4> Find the clock cycles required in both cases.

The following table shows the number of instructions for a program.

	Arith	Store	Load	Branch	Total
a.	650	100	600	50	1400
b.	750	250	500	500	2000

1.4.4 [5] <1.4> Assuming that arith instructions take 1 cycle, load and store 5 cycles, and branches 2 cycles, what is the execution time of the program in a 2 GHz processor?

1.4.5 [5] <1.4> Find the CPI for the program.

1.4.6 [10] <1.4> If the number of load instructions can be reduced by one half, what is the speedup and the CPI?

Exercise 1.5

Consider two different implementations, P1 and P2, of the same instruction set. There are five classes of instructions (A, B, C, D, and E) in the instruction set. The clock rate and CPI of each class is given below.

		Clock Rate	CPI Class A	CPI Class B	CPI Class C	CPI Class D	CPI Class E
a.	P1	2.0 GHz	1	2	3	4	3
	P2	4.0 GHz	2	2	2	4	4
b.	P1	2.0 GHz	1	1	2	3	2
	P2	3.0 GHz	1	2	3	4	3

1.5.1 [5] <1.4> Assume that peak performance is defined as the fastest rate that a computer can execute any instruction sequence. What are the peak performances of P1 and P2 expressed in instructions per second?

1.5.2 [10] <1.4> If the number of instructions executed in a certain program is divided equally among the classes of instructions except for class A, which occurs twice as often as each of the others, which computer is faster? How much faster is it?

1.5.3 [10] <1.4> If the number of instructions executed in a certain program is divided equally among the classes of instructions except for class E, which occurs twice as often as each of the others, which computer is faster? How much faster is it?

The table below shows instruction-type breakdown for different programs. Using this data, you will be exploring the performance trade-offs for different changes made to an MIPS processor.

		No. Instructions				
		Compute	Load	Store	Branch	Total
a.	Program1	600	600	200	50	1450
b.	Program 2	900	500	100	200	1700

1.5.4 [5] <1.4> Assuming that computes take 1 cycle, loads and store instructions take 10 cycles, and branches take 3 cycles, find the execution time on a 3 GHz MIPS processor.

1.5.5 [5] <1.4> Assuming that computes take 1 cycle, loads and store instructions take 2 cycles, and branches take 3 cycles, find the execution time on a 3 GHz MIPS processor.

1.5.6 [5] <1.4> Assuming that computes take 1 cycle, loads and store instructions take 2 cycles, and branches take 3 cycles, what is the speedup if the number of compute instruction can be reduced by one-half?

Exercise 1.6

Compilers can have a profound impact on the performance of an application on given a processor. This problem will explore the impact compilers have on execution time.

	Compiler A		Compiler B	
	No. Instructions	Execution Time	No. Instructions	Execution Time
a.	1.00E+09	1.8 s	1.20E+09	1.8 s
b.	1.00E+09	1.1 s	1.20E+09	1.5 s

1.6.1 [5] <1.4> For the same program, two different compilers are used. The table above shows the execution time of the two different compiled programs. Find the average CPI for each program given that the processor has a clock cycle time of 1 ns.

1.6.2 [5] <1.4> Assume the average CPIs found in 1.6.1, but that the compiled programs run on two different processors. If the execution times on the two processors are the same, how much faster is the clock of the processor running compiler A's code versus the clock of the processor running compiler B's code?

1.6.3 [5] <1.4> A new compiler is developed that uses only 600 million instructions and has an average CPI of 1.1. What is the speedup of using this new compiler versus using Compiler A or B on the original processor of 1.6.1?

Consider two different implementations, P1 and P2, of the same instruction set. There are five classes of instructions (A, B, C, D, and E) in the instruction set. P1 has a clock rate of 4 GHz, and P2 has a clock rate of 6 GHz. The average number of cycles for each instruction class for P1 and P2 are listed in the following table.

		CPI Class A	CPI Class B	CPI Class C	CPI Class D	CPI Class E
a.	P1	1	2	3	4	5
	P2	3	3	3	5	5
b.	P1	1	2	3	4	5
	P2	2	2	2	2	6

1.6.4 [5] <1.4> Assume that peak performance is defined as the fastest rate that a computer can execute any instruction sequence. What are the peak performances of P1 and P2 expressed in instructions per second?

1.6.5 [5] <1.4> If the number of instructions executed in a certain program is divided equally among the five classes of instructions except for class A, which occurs twice as often as each of the others, how much faster is P2 than P1?

1.6.6 [5] <1.4> At what frequency does P1 have the same performance of P2 for the instruction mix given in 1.6.5?

Exercise 1.7

The following table shows the increase in clock rate and power of eight generations of Intel processors over 28 years.

Processor	Clock Rate	Power
80286 (1982)	12.5 MHz	3.3 W
80386 (1985)	16 MHz	4.1 W
80486 (1989)	25 MHz	4.9 W
Pentium (1993)	66 MHz	10.1 W
Pentium Pro (1997)	200 MHz	29.1 W
Pentium 4 Willamette (2001)	2 GHz	75.3 W
Pentium 4 Prescott (2004)	3.6 GHz	103 W
Core 2 Ketsfield (2007)	2.667 GHz	95 W

1.7.1 [5] <1.5> What is the geometric mean of the ratios between consecutive generations for both clock rate and power? (The geometric mean is described in Section 1.7.)

1.7.2 [5] <1.5> What is the largest relative change in clock rate and power between generations?

1.7.3 [5] <1.5> How much larger is the clock rate and power of the last generation with respect to the first generation?

Consider the following values for voltage in each generation.

Processor	Voltage
80286 (1982)	5
80386 (1985)	5
80486 (1989)	5
Pentium (1993)	5
Pentium Pro (1997)	3.3
Pentium 4 Willamette (2001)	1.75
Pentium 4 Prescott (2004)	1.25
Core 2 Ketsfield (2007)	1.1

1.7.4 [5] <1.5> Find the average capacitive loads, assuming a negligible static power consumption.

1.7.5 [5] <1.5> Find the largest relative change in voltage between generations.

1.7.6 [5] <1.5> Find the geometric mean of the voltage ratios in the generations since the Pentium.

Exercise 1.8

Suppose we have developed new versions of a processor with the following characteristics.

	Version	Voltage	Clock Rate
a.	Version 1	1.75 V	1.5 GHz
	Version 2	1.2 V	2 GHz
b.	Version 1	1.1 V	3 GHz
	Version 2	0.8 V	4 GHz

1.8.1 [5] <1.5> How much has the capacitive load varied between versions if the dynamic power has been reduced by 10%?

1.8.2 [5] <1.5> How much has the dynamic power been reduced if the capacitive load does not change?

1.8.3 [10] <1.5> Assuming that the capacitive load of version 2 is 80% the capacitive load of version 1, find the voltage for version 2 if the dynamic power of version 2 is reduced by 40% from version 1.

Suppose that the industry trends show that a new process generation varies as follows.

	Capacitance	Voltage	Clock Rate	Area
a.	1	$1/2^{1/2}$	1.15	$1/2^{1/2}$
b.	1	$1/2^{1/4}$	1.2	$1/2^{1/4}$

1.8.4 [5] <1.5> Find the scaling factor for the dynamic power.

1.8.5 [5] <1.5> Find the scaling of the capacitance per unit area unit.

1.8.6 [5] <1.5> Assuming a Core 2 processor with a clock rate of 2.667 GHz, a power consumption of 95 W, and a voltage of 1.1 V, find the voltage and clock rate of this processor for the next process generation.

Exercise 1.9

Although the dynamic power is the primary source of power dissipation in CMOS, leakage current produces a static power dissipation $V \times I_{leak}$. The smaller the on-chip dimensions, the more significant is the static power. Assume the figures shown in the following table for static and dynamic power dissipation for several generations of processors.

	Technology	Dynamic Power (W)	Static Power (W)	Voltage (V)
a.	180 nm	50	10	1.2
b.	70 nm	90	60	0.9

1.9.1 [5] <1.5> Find the percentage of the total dissipated power comprised by static power.

1.9.2 [5] <1.5> If the total dissipated power is reduced by 10% while maintaining the static to total power rate of problem 1.9.1, how much should the voltage be reduced to maintain the same leakage current?

1.9.3 [5] <1.5> Determine the ratio of static power to dynamic power for each technology.

Consider now the dynamic power dissipation of different versions of a given processor for three different voltages given in the following table.

	1.2 V	1.0 V	0.8 V
a.	75 W	60 W	35 W
b.	62 W	50 W	30 W

1.9.4 [5] <1.5> Determine the static power at 0.8 V, assuming a static to dynamic power ratio of 0.6.

1.9.5 [5] <1.5> Determine the static and dynamic power dissipation assuming the rates obtained in problem 1.9.1.

1.9.6 [10] <1.5> Determine the geometric mean of the power variations between versions.

Exercise 1.10

The table below shows the instruction type breakdown of a given application executed on 1, 2, 4, or 8 processors. Using this data, you will be exploring the speed-up of applications on parallel processors.

	Processors	No. Instructions per Processor			CPI		
		Arithmetic	Load/Store	Branch	Arithmetic	Load/Store	Branch
a.	1	2560	1280	256	1	4	2
	2	1280	640	128	1	5	2
	4	640	320	64	1	7	2
	8	320	160	32	1	12	2

	Processors	No. Instructions per Processor			CPI		
		Arithmetic	Load/Store	Branch	Arithmetic	Load/Store	Branch
b.	1	2560	1280	256	1	4	2
	2	1280	640	128	1	6	2
	4	640	320	64	1	8	2
	8	320	160	32	1	10	2

1.10.1 [5] <1.4, 1.6> The table above shows the number of instructions required per processor to complete a program on a multiprocessor with 1, 2, 4, or 8 processors. What is the total number of instructions executed per processor? What is the aggregate number of instructions executed across all processors?

1.10.2 [5] <1.4, 1.6> Given the CPI values on the right of the table above, find the total execution time for this program on 1, 2, 4, and 8 processors. Assume that each processor has a 2 GHz clock frequency.

1.10.3 [10] <1.4, 1.6> If the CPI of the arithmetic instructions was doubled, what would the impact be on the execution time of the program on 1, 2, 4, or 8 processors?

The table below shows the number of instructions per processor core on a multicore processor as well as the average CPI for executing the program on 1, 2, 4, or 8 cores. Using this data, you will be exploring the speedup of applications on multicore processors.

	Cores per Processor	Instructions per Core	Average CPI
a.	1	1.00E+10	1.2
	2	5.00E+09	1.4
	4	2.50E+09	1.8
	8	1.25E+09	2.6

	Cores per Processor	Instructions per Core	Average CPI
b.	1	1.00E+10	1.0
	2	5.00E+09	1.2
	4	2.50E+09	1.4
	8	1.25E+09	1.7

1.10.4 [10] <1.4, 1.6> Assuming a 3 GHz clock frequency, what is the execution time of the program using 1, 2, 4, or 8 cores?

1.10.5 [10] <1.5, 1.6> Assume that the power consumption of a processor core can be described by the following equation:

$$\text{Power} = \frac{5.0\text{mA}}{\text{MHz}}\text{Voltage}^2$$

where the operation voltage of the processor is described by the following equation:

$$\text{Voltage} = \frac{1}{5}\text{Frequency} + 0.4$$

with the frequency measured in GHz. So, at 5 GHz, the voltage would be 1.4 V. Find the power consumption of the program executing on 1, 2, 4, and 8 cores assuming that each core is operating at a 3 GHz clock frequency. Likewise, find the power consumption of the program executing on 1, 2, 4, or 8 cores assuming that each core is operating at 500 MHz.

1.10.6 [10] <1.5, 1.6> If using a single core, find the required CPI for this core to get an execution time equal to the time obtained by using the number of cores in the table above (execution times in problem 1.10.4). Note that the number of instructions should be the aggregate number of instructions executed across all the cores.

Exercise 1.11

The following table shows manufacturing data for various processors.

	Wafer Diameter	Dies per Wafer	Defects per Unit Area	Cost per Wafer
a.	15 cm	84	0.020 defects/cm^2	12
b.	20 cm	100	0.031 defects/cm^2	15

1.11.1 [10] <1.7> Find the yield.

1.11.2 [5] <1.7> Find the cost per die.

1.11.3 [10] <1.7> If the number of dies per wafer is increased by 10% and the defects per area unit increases by 15%, find the die area and yield.

Suppose that, with the evolution of the electronic devices manufacturing technology, the yield varies as shown in the following table.

	T1	T2	T3	T4
Yield	0.85	0.89	0.92	0.95

1.11.4 [10] <1.7> Find the defects per area unit for each technology given a die area of 200 mm^2.

1.11.5 [5] <1.7> Represent graphically the variation of the yield together with the variation of defects per unit area.

Exercise 1.12

The following table shows results for SPEC CPU2006 benchmark programs running on an AMD Barcelona.

	Name	Intr. Count $\times 10^9$	Execution Time (seconds)	Reference Time (seconds)
a.	bzip2	2389	750	9650
b.	go	1658	700	10,490

1.12.1 [5] <1.7> Find the CPI if the clock cycle time is 0.333 ns.

1.12.2 [5] <1.7> Find the SPECratio.

1.12.3 [5] <1.7> For these two benchmarks, find the geometric mean of the SPECratio.

The following table shows data for further benchmarks.

	Name	CPI	Clock Rate	SPECratio
a.	libquantum	1.61	4 GHz	19.8
b.	astar	1.79	4 GHz	9.1

1.12.4 [5] <1.7> Find the increase in CPU time if the number of instructions of the benchmark is increased by 10% without affecting the CPI.

1.12.5 [5] <1.7> Find the increase in CPU time if the number of instructions of the benchmark is increased by 10% and the CPI is increased by 5%.

1.12.6 [5] <1.7> Find the change in the SPECratio for the change described in 1.12.5.

Exercise 1.13

Suppose that we are developing a new version of the AMD Barcelona processor with a 4 GHz clock rate. We have added some additional instructions to the instruction set in such a way that the number of instructions has been reduced by 15% from the values shown for each benchmark in Exercise 1.12. The execution times obtained are shown in the following table.

	Name	Execution Time (seconds)	Reference Time (seconds)	SPECratio
a.	bzip2	700	9650	13.7
b.	go	620	10490	16.9

1.13.1 [10] <1.8> Find the new CPI.

1.13.2 [10] <1.8> In general, these CPI values are larger than those obtained in previous exercises for the same benchmarks. This is due mainly to the clock rate used in both cases, 3 GHz and 4 GHz. Determine whether the increase in the CPI is similar to that of the clock rate. If they are dissimilar, why?

1.13.3 [5] <1.8> How much has the CPU time been reduced?

The following table shows data for further benchmarks.

	Name	Execution Time (seconds)	CPI	Clock Rate
a.	libquantum	960	1.61	3 GHz
b.	astar	690	1.79	3 GHz

1.13.4 [10] <1.8> If the execution time is reduced by an additional 10% without affecting to the CPI and with a clock rate of 4 GHz, determine the number of instructions.

1.13.5 [10] <1.8> Determine the clock rate required to give a further 10% reduction in CPU time while maintaining the number of instructions and with the CPI unchanged.

1.13.6 [10] <1.8> Determine the clock rate if the CPI is reduced by 15% and the CPU time by 20% while the number of instructions is unchanged.

Exercise 1.14

Section 1.8 cites as a pitfall the utilization of a subset of the performance equation as a performance metric. To illustrate this, consider the following data for the execution of a program in different processors.

	Processor	Clock Rate	CPI	No. Instr.
a.	P1	4 GHz	0.9	5.00E+06
	P2	3 GHz	0.75	1.00E+06
b.	P1	3 GHz	1.1	3.00E+06
	P2	2.5 GHz	1.0	0.50E+06

1.14.1 [5] <1.8> One usual fallacy is to consider the computer with the largest clock rate as having the largest performance. Check if this is true for P1 and P2.

1.14.2 [10] <1.8> Another fallacy is to consider that the processor executing the largest number of instructions will need a larger CPU time. Considering that processor P1 is executing a sequence of 10^6 instructions and that the CPI of processors P1 and P2 do not change, determine the number of instructions that P2 can execute in the same time that P1 needs to execute 10^6 instructions.

1.14.3 [10] <1.8> A common fallacy is to use MIPS (millions of instructions per second) to compare the performance of two different processors, and consider that the processor with the largest MIPS has the largest performance. Check if this is true for P1 and P2.

Another common performance figure is MFLOPS (million of floating-point operations per second), defined as

$$\text{MFLOPS} = \text{No. FP operations} / (\text{execution time} \times 10^6)$$

but this figure has the same problems as MIPS. Consider the program in the following table, running on the two processors below.

	Processor	Instr. Count	No. Instructions			CPI			Clock Rate
			L/S	FP	Branch	L/S	FP	Branch	
a.	P1	1.00E+06	50%	40%	10%	0.75	1.0	1.5	4 GHz
	P2	5.00E+06	40%	40%	20%	1.25	0.8	1.25	3 GHz
b.	P1	5.00E+06	30%	30%	40%	1.5	1.0	2.0	4 GHz
	P2	2.00E+06	40%	30%	30%	1.25	1.0	2.5	3 GHz

1.14.4 [10] <1.8> Find the MFLOPS figures for the programs.

1.14.5 [10] <1.8> Find the MIPS figures for the programs.

1.14.6 [10] <1.8> Find the performance for the programs and compare it with MIPS and MFLOPS.

Exercise 1.15

Another pitfall cited in Section 1.8 is expecting to improve the overall performance of a computer by improving only one aspect of the computer. This might be true, but not always. Consider a computer running programs with CPU times shown in the following table.

	FP Instr.	INT Instr.	L/S Instr.	Branch Instr.	Total Time
a.	70 s	85 s	55 s	40 s	250 s
b.	40 s	90 s	60 s	20 s	210 s

1.15.1 [5] <1.8> How much is the total time reduced if the time for FP operations is reduced by 20%?

1.15.2 [5] <1.8> How much is the time for INT operations reduced if the total time is reduced by 20%?

1.15.3 [5] <1.8> Can the total time can be reduced by 20% by reducing only the time for branch instructions?

The following table shows the instruction type breakdown per processor of given applications executed in different numbers of processors.

	Processors	FP Instr.	INT Instr.	L/S Instr.	Branch Instr.	CPI (FP)	CPI (INT)	CPI (L/S)	CPI (Branch)
a.	2	280×10^6	1000×16^6	640×10^6	128×10^6	1	1	4	2
b.	16	50×10^6	110×10^6	80×10^6	16×10^6	1	1	4	2

Assume that each processor has a 2 GHz clock rate.

1.15.4 [10] <1.8> How much must we improve the CPI of FP instructions if we want the program to run two times faster?

1.15.5 [10] <1.8> How much must we improve the CPI of L/S instructions if we want the program to run two times faster?

1.15.6 [5] <1.8> How much is the execution time of the program improved if the CPI of INT and FP instructions is reduced by 40% and the CPI of L/S and Branch is reduced by 30%?

Exercise 1.16

Another pitfall, related to the execution of programs in multiprocessor systems, is expecting improvement in performance by improving only the execution time of part of the routines. The following table shows the execution time of five routines of a program running on different numbers of processors.

	No. Processors	Routine A (ms)	Routine B (ms)	Routine C (ms)	Routine D (ms)	Routine E (ms)
a.	4	12	45	6	36	3
b.	32	2	7	1	6	2

1.16.1 [10] <1.8> Find the total execution time and by how much it is reduced if the time of routines A, C, and E is improved by 15%.

1.16.2 [10] <1.8> How much is the total time reduced if routine B is improved by 10%?

1.16.3 [10] <1.8> How much is the total time reduced if routine D is improved by 10%?

Execution time in a multiprocessor system can be split into computing time for the routines plus routing time spent sending data from one processor to another. Consider the execution time and routing time given in the following table. In this case, the routing time is an important component of the total time.

No. Processors	Routine A (ms)	Routine B (ms)	Routine C (ms)	Routine D (ms)	Routine E (ms)	Routing Time (ms)
2	40	78	9	70	4	11
4	29	60	4	36	2	13
8	15	45	3	19	3	17
16	7	35	1	11	2	22
32	4	23	1	6	1	23
64	2	12	0.5	3	1	26

1.16.4 [10] <1.8> For each doubling of the number of processors, determine the ratio of new to old computing time and the ratio of new to old routing time.

1.16.5 [5] <1.8> Using the geometric means of the ratios, extrapolate to find the computing time and routing time in a 128-processor system.

1.16.6 [10] <1.8> Find the computing time and routing time for a system with one processor.

Answers to Check Yourself

§1.1, page 9: Discussion questions: many answers are acceptable.
§1.3, page 25: Disk memory: nonvolatile, long access time (milliseconds), and cost $0.20–$2.00/GB. Semiconductor memory: volatile, short access time (nanoseconds), and cost $20–$75/GB.
§1.4, page 31: 1. a: both, b: latency, c: neither. 2. 7 seconds.
§1.4, page 38: b.
§1.7, page 50: 1, 3, and 4 are valid reasons. Answer 5 can be generally true because high volume can make the extra investment to reduce die size by, say, 10% a good economic decision, but it doesn't have to be true.
§1.8, page 53: a. Computer A has the higher MIPS rating. b. Computer B is faster.