

1.6	Performance	28
1.7	The Power Wall	40
1.8	The Sea Change: The Switch from Uniprocessors to Multiprocessors	43
1.9	Real Stuff: Benchmarking the Intel Core i7	46
1.10	Fallacies and Pitfalls	49
1.11	Concluding Remarks	52
 1.12	Historical Perspective and Further Reading	54
1.13	Exercises	54

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 breath-taking rate. In the last 30 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 a second for a penny. 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 blind spot warnings, lane departure warnings, moving object detection, and air bag inflation to protect occupants in a crash.
- *Cell phones:* Who would have dreamed that advances in computer systems would lead to more than half of the planet having mobile phones, allowing person-to-person communication to almost anyone anywhere in the world?
- *Human genome project:* The cost of computer equipment to map and analyze human DNA sequences was 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 15 to 25 years earlier. Moreover, costs continue to drop; you will soon 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 web has transformed our society. For many, the web has replaced libraries and newspapers.
- *Search engines:* As the content of the web 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 glasses that augment reality, the cashless society, and cars that can drive themselves.

Classes of Computing Applications and Their Characteristics

Although a common set of hardware technologies (see Sections 1.4 and 1.5) 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.

Personal computers (PCs) are possibly the best known form of computing, which readers of this book have likely used extensively. Personal computers emphasize delivery of good performance to single users at low cost and usually execute third-party software. This class of computing drove the evolution of many computing technologies, which is only about 35 years old!

Servers are the modern form of what were once much larger computers, 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 computing, storage, 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 PC.

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 tens of thousands of processors and many **terabytes** of memory, and cost tens 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.

Embedded computers are the largest class of computers and span the widest range of applications and performance. Embedded computers include the microprocessors found in your car, the computers in a television set, 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!

personal computer (PC)

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 tens to hundreds of millions of dollars.

terabyte (TB)

Originally 1,099,511,627,776 (2^{40}) bytes, although communications and secondary storage systems developers started using the term to mean 1,000,000,000,000 (10^{12}) bytes. To reduce confusion, we now use the term **tebibyte (TiB)** for 2^{40} bytes, defining *terabyte* (TB) to mean 10^{12} bytes.

Figure 1.1 shows the full range of decimal and binary values and names.

embedded computer

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

Decimal term	Abbreviation	Value	Binary term	Abbreviation	Value	% Larger
kilobyte	KB	10^3	kibibyte	KiB	2^{10}	2%
megabyte	MB	10^6	mebibyte	MiB	2^{20}	5%
gigabyte	GB	10^9	gibibyte	GiB	2^{30}	7%
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}	10%
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}	13%
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}	15%
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}	18%
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}	21%

FIGURE 1.1 The 2^x vs. 10^y bytes ambiguity was resolved by adding a binary notation for all the common size terms. In the last column we note how much larger the binary term is than its corresponding decimal term, which is compounded as we head down the chart. These prefixes work for bits as well as bytes, so *gigabit* (Gb) is 10^9 bits while *gibibits* (GiB) is 2^{30} bits.

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. 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.

Welcome to the PostPC Era

The continuing march of technology brings about generational changes in computer hardware that shake up the entire information technology industry. Since the last edition of the book we have undergone such a change, as significant in the past as the switch starting 30 years ago to personal computers. Replacing the

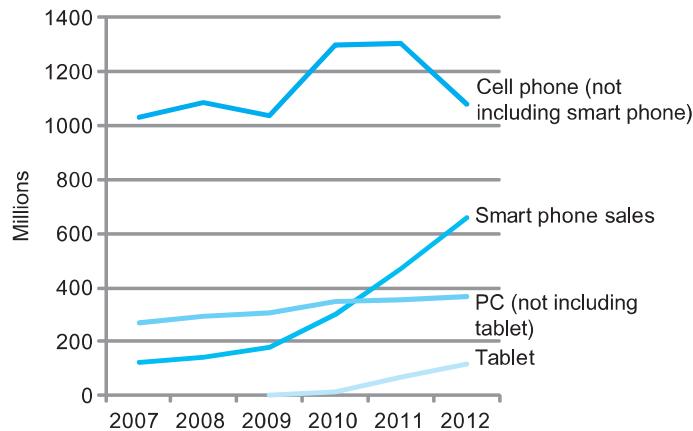


FIGURE 1.2 The number manufactured per year of tablets and smart phones, which reflect the PostPC era, versus personal computers and traditional cell phones. Smart phones represent the recent growth in the cell phone industry, and they passed PCs in 2011. Tablets are the fastest growing category, nearly doubling between 2011 and 2012. Recent PCs and traditional cell phone categories are relatively flat or declining.

PC is the **personal mobile device (PMD)**. PMDs are battery operated with wireless connectivity to the Internet and typically cost hundreds of dollars, and, like PCs, users can download software (“apps”) to run on them. Unlike PCs, they no longer have a keyboard and mouse, and are more likely to rely on a touch-sensitive screen or even speech input. Today’s PMD is a smart phone or a tablet computer, but tomorrow it may include electronic glasses. Figure 1.2 shows the rapid growth time of tablets and smart phones versus that of PCs and traditional cell phones.

Taking over from the traditional server is **Cloud Computing**, which relies upon giant datacenters that are now known as *Warehouse Scale Computers* (WSCs). Companies like Amazon and Google build these WSCs containing 100,000 servers and then let companies rent portions of them so that they can provide software services to PMDs without having to build WSCs of their own. Indeed, **Software as a Service (SaaS)** deployed via the cloud is revolutionizing the software industry just as PMDs and WSCs are revolutionizing the hardware industry. Today’s software developers will often have a portion of their application that runs on the PMD and a portion that runs in the Cloud.

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

Personal mobile devices (PMDs) are small wireless devices to connect to the Internet; they rely on batteries for power, and software is installed by downloading apps. Conventional examples are smart phones and tablets.

Cloud Computing refers to large collections of servers that provide services over the Internet; some providers rent dynamically varying numbers of servers as a utility.

Software as a Service (SaaS) delivers software and data as a service over the Internet, usually via a thin program such as a browser that runs on local client devices, instead of binary code that must be installed, and runs wholly on that device. Examples include web search and social networking.

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. Moreover, as we explain in Section 1.7, today's programmers need to worry about energy efficiency of their programs running either on the PMD or in the Cloud, which also requires understanding what is below your code. Programmers who seek to build competitive versions of software 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 techniques can be used by hardware designers to improve energy efficiency? What can the programmer do to help or hinder energy efficiency?
- 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 6).
- Since the first commercial computer in 1951, what great ideas did computer architects come up with that lay the foundation of modern computing?

**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 energy, 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.

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.

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.

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.

Understanding Program 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 6
I/O system (hardware and operating system)	Determines how fast I/O operations may be executed	Chapters 4, 5, and 6

To demonstrate the impact of the ideas in this book, we improve the performance of a C program that multiplies a matrix times a vector in a sequence of chapters. Each step leverages understanding how the underlying hardware really works in a modern microprocessor to improve performance by a factor of 200!

- In the category of *data level parallelism*, in Chapter 3 we use *subword parallelism via C intrinsics* to increase performance by a factor of 3.8.
- In the category of *instruction level parallelism*, in Chapter 4 we use *loop unrolling to exploit multiple instruction issue and out-of-order execution hardware* to increase performance by another factor of 2.3.
- In the category of *memory hierarchy optimization*, in Chapter 5 we use *cache blocking* to increase performance on large matrices by another factor of 2.5.
- In the category of *thread level parallelism*, in Chapter 6 we use *parallel for loops in OpenMP to exploit multicore hardware* to increase performance by another factor of 14.

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. The number of embedded processors sold every year greatly outnumbers the number of PC and even PostPC 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 conventional 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

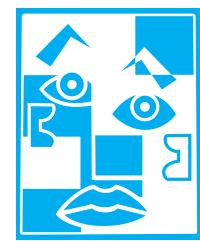
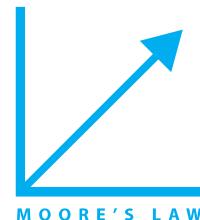
1.2

Eight Great Ideas in Computer Architecture

We now introduce eight great ideas that computer architects have been invented in the last 60 years of computer design. These ideas are so powerful they have lasted long after the first computer that used them, with newer architects demonstrating their admiration by imitating their predecessors. These great ideas are themes that we will weave through this and subsequent chapters as examples arise. To point out their influence, in this section we introduce icons and highlighted terms that represent the great ideas and we use them to identify the nearly 100 sections of the book that feature use of the great ideas.

Design for Moore's Law

The one constant for computer designers is rapid change, which is driven largely by **Moore's Law**. It states that integrated circuit resources double every 18–24 months. Moore's Law resulted from a 1965 prediction of such growth in IC capacity made by Gordon Moore, one of the founders of Intel. As computer designs can take years, the resources available per chip can easily double or quadruple between the start and finish of the project. Like a skeet shooter, computer architects must anticipate where the technology will be when the design finishes rather than design for where it starts. We use an “up and to the right” Moore's Law graph to represent designing for rapid change.



Use Abstraction to Simplify Design

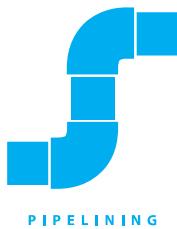
Both computer architects and programmers had to invent techniques to make themselves more productive, for otherwise design time would lengthen as dramatically as resources grew by Moore's Law. A major productivity technique for hardware and software is to use **abstractions** to represent the design at different levels of representation; lower-level details are hidden to offer a simpler model at higher levels. We'll use the abstract painting icon to represent this second great idea.

Make the Common Case Fast

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. This common sense advice implies that you know what the common case is, which is only possible with careful experimentation and measurement (see Section 1.6). We use a sports car as the icon for making the common case fast, as the most common trip has one or two passengers, and it's surely easier to make a fast sports car than a fast minivan!



PARALLELISM



PIPELINING



PREDICTION



HIERARCHY



DEPENDABILITY

Performance via Parallelism

Since the dawn of computing, computer architects have offered designs that get more performance by performing operations in parallel. We'll see many examples of parallelism in this book. We use multiple jet engines of a plane as our icon for parallel performance.

Performance via Pipelining

A particular pattern of parallelism is so prevalent in computer architecture that it merits its own name: **pipelining**. For example, before fire engines, a “bucket brigade” would respond to a fire, which many cowboy movies show in response to a dastardly act by the villain. The townsfolk form a human chain to carry a water source to fire, as they could much more quickly move buckets up the chain instead of individuals running back and forth. Our pipeline icon is a sequence of pipes, with each section representing one stage of the pipeline.

Performance via Prediction

Following the saying that it can be better to ask for forgiveness than to ask for permission, the final great idea is **prediction**. In some cases it can be faster on average to guess and start working rather than wait until you know for sure, assuming that the mechanism to recover from a misprediction is not too expensive and your prediction is relatively accurate. We use the fortune-teller's crystal ball as our prediction icon.

Hierarchy of Memories

Programmers want memory to be fast, large, and cheap, as memory speed often shapes performance, capacity limits the size of problems that can be solved, and the cost of memory today is often the majority of computer cost. Architects have found that they can address these conflicting demands with a **hierarchy of memories**, with the fastest, smallest, and most expensive memory per bit at the top of the hierarchy and the slowest, largest, and cheapest per bit at the bottom. As we shall see in Chapter 5, caches give the programmer the illusion that main memory is nearly as fast as the top of the hierarchy and nearly as big and cheap as the bottom of the hierarchy. We use a layered triangle icon to represent the memory hierarchy. The shape indicates speed, cost, and size: the closer to the top, the faster and more expensive per bit the memory; the wider the base of the layer, the bigger the memory.

Dependability via Redundancy

Computers not only need to be fast; they need to be dependable. Since any physical device can fail, we make systems **dependable** by including redundant components that can take over when a failure occurs *and* to help detect failures. We use the tractor-trailer as our icon, since the dual tires on each side of its rear axels allow the truck to continue driving even when one tire fails. (Presumably, the truck driver heads immediately to a repair facility so the flat tire can be fixed, thereby restoring redundancy!)

1.3

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, an example of the great idea of **abstraction**.

Figure 1.3 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.

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:

- 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, iOS, and Windows.

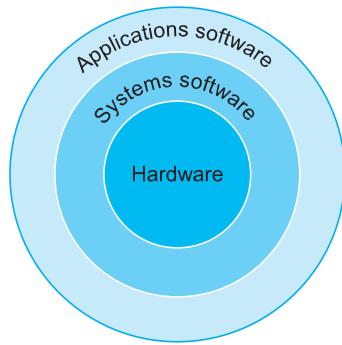
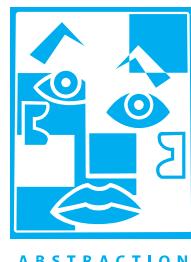


FIGURE 1.3 A simplified view of hardware and software as hierarchical layers, shown as concentric circles with hardware in the center and applications software outermost. In complex applications, there are often multiple layers of application software as well. For example, a database system may run on top of the systems software hosting an application, which in turn runs on top of the database.

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



systems software

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

operating system

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

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

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 in Appendix A.

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.

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

assembly language A symbolic representation of machine instructions.

machine language A binary representation of machine instructions.

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

add A, B

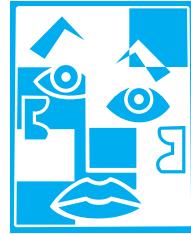
and the assembler would translate this notation into

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**.

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 1.4 shows the relationships among these programs and languages, which are more examples of the power of **abstraction**.



ABSTRACTION

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.

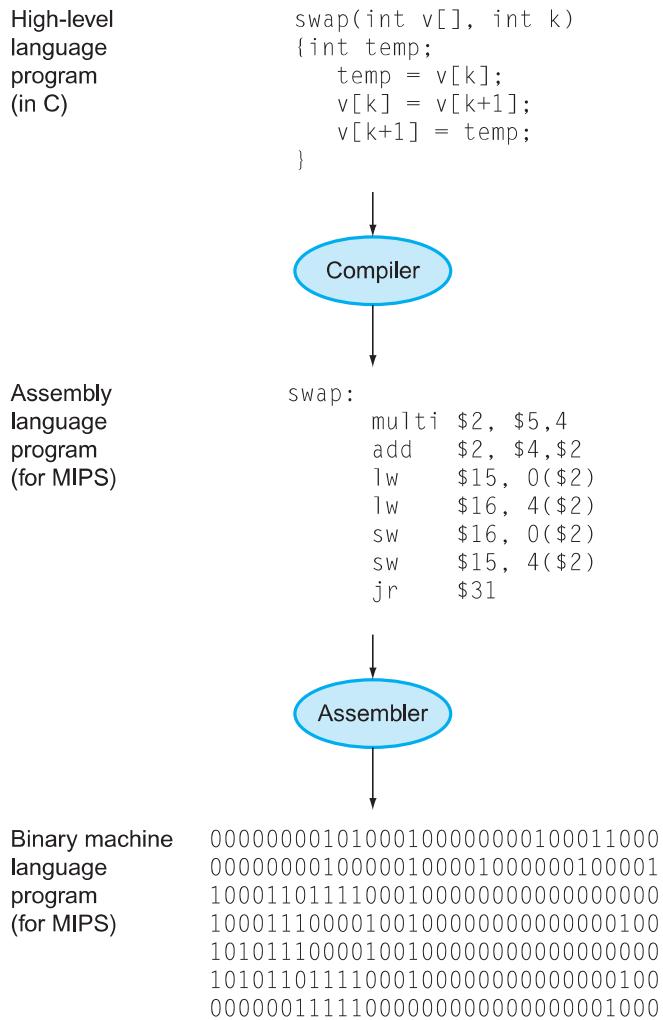


FIGURE 1.4 C program compiled into assembly language and then assembled into binary machine language. Although the translation from high-level language to binary machine language is shown in two steps, some compilers cut out the middleman and produce binary machine language directly. These languages 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 1.4](#)). 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.

1.4 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.

Two key components of computers are **input devices**, such as the microphone, and **output devices**, such as the speaker. As the names suggest, input feeds the

input device

A mechanism through which the computer is fed information, such as a keyboard.

output device

A mechanism that conveys the result of a computation to a user, such as a display, or to another computer.

computer, and output is the result of computation sent to the user. Some devices, such as wireless networks, provide both input and output to the computer.

Chapters 5 and 6 describe 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.

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 1.5](#) 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.

The BIG Picture

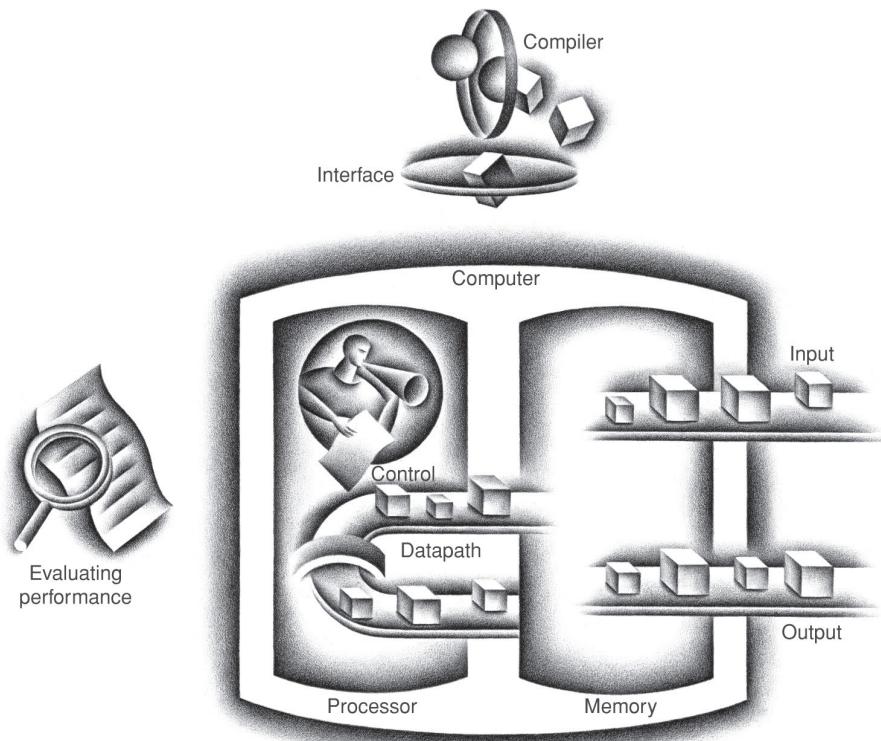


FIGURE 1.5 The organization of a computer, showing the five classic components. The processor gets instructions and data from memory. Input writes data to memory, and output reads data from memory. Control sends the signals that determine the operations of the datapath, memory, input, and output.

Through the Looking Glass

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.

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

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

The most fascinating I/O device is probably the graphics display. Most personal mobile devices 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.

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 in a typical tablet ranges in size from 1024×768 to 2048×1536 . 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.

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.

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.

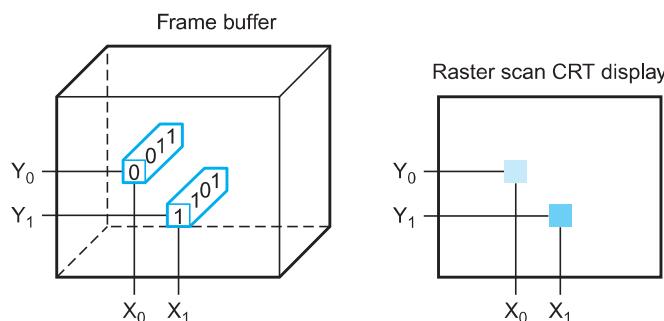


FIGURE 1.6 Each coordinate in the frame buffer on the left determines the shade of the corresponding coordinate for the raster scan CRT display on the right. Pixel (X_0, Y_0) contains the bit pattern 0011, which is a lighter shade on the screen than the bit pattern 1101 in pixel (X_1, Y_1) .

Touchscreen

While PCs also use LCD displays, the tablets and smartphones of the PostPC era have replaced the keyboard and mouse with touch sensitive displays, which has the wonderful user interface advantage of users pointing directly what they are interested in rather than indirectly with a mouse.

While there are a variety of ways to implement a touch screen, many tablets today use capacitive sensing. Since people are electrical conductors, if an insulator like glass is covered with a transparent conductor, touching distorts the electrostatic field of the screen, which results in a change in capacitance. This technology can allow multiple touches simultaneously, which allows gestures that can lead to attractive user interfaces.

Opening the Box

Figure 1.7 shows the contents of the Apple iPad 2 tablet computer. Unsurprisingly, of the five classic components of the computer, I/O dominates this reading device. The list of I/O devices includes a capacitive multitouch LCD display, front facing camera, rear facing camera, microphone, headphone jack, speakers, accelerometer, gyroscope, Wi-Fi network, and Bluetooth network. The datapath, control, and memory are a tiny portion of the components.

The small rectangles in Figure 1.8 contain the devices that drive our advancing technology, called **integrated circuits** and nicknamed **chips**. The A5 package seen in the middle of in Figure 1.8 contains two ARM processors that operate with a clock rate of 1 GHz. The *processor* is the active part of the computer, following the instructions of a program to the letter. It adds numbers, tests numbers, signals I/O devices to activate, and so on. 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.

The A5 package in Figure 1.8 also includes two memory chips, each with 2 gibibits of capacity, thereby supplying 512 MiB. The **memory** is where the programs are kept when they are running; it also contains the data needed by the running programs. The memory is built from DRAM chips. DRAM stands for **dynamic random access memory**. Multiple 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.

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.

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

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.

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.

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

dynamic random access memory (DRAM)
Memory built as an integrated circuit; it provides random access to any location. Access times are 50 nanoseconds and cost per gigabyte in 2012 was \$5 to \$10.



FIGURE 1.7 Components of the Apple iPad 2 A1395. The metal back of the iPad (with the reversed Apple logo in the middle) is in the center. At the top is the capacitive multitouch screen and LCD display. To the far right is the 3.8 V, 25 watt-hour, polymer battery, which consists of three Li-ion cell cases and offers 10 hours of battery life. To the far left is the metal frame that attaches the LCD to the back of the iPad. The small components surrounding the metal back in the center are what we think of as the computer; they are often L-shaped to fit compactly inside the case next to the battery. Figure 1.8 shows a close-up of the L-shaped board to the lower left of the metal case, which is the logic printed circuit board that contains the processor and the memory. The tiny rectangle below the logic board contains a chip that provides wireless communication: Wi-Fi, Bluetooth, and FM tuner. It fits into a small slot in the lower left corner of the logic board. Near the upper left corner of the case is another L-shaped component, which is a front-facing camera assembly that includes the camera, headphone jack, and microphone. Near the right upper corner of the case is the board containing the volume control and silent/screen rotation lock button along with a gyroscope and accelerometer. These last two chips combine to allow the iPad to recognize 6-axis motion. The tiny rectangle next to it is the rear-facing camera. Near the bottom right of the case is the L-shaped speaker assembly. The board between the cable and the speaker assembly is the controller for the capacitive touchscreen. (Courtesy iFixit, www.ifixit.com)

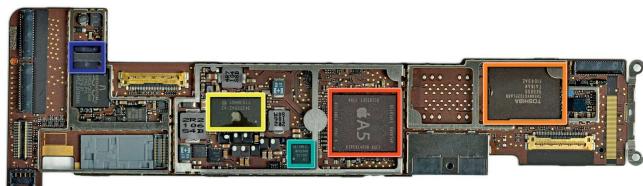


FIGURE 1.8 The logic board of Apple iPad 2 in Figure 1.7. The photo highlights five integrated circuits. The large integrated circuit in the middle is the Apple A5 chip, which contains a dual ARM processor cores that run at 1 GHz as well as 512 MB of main memory inside the package. Figure 1.9 shows a photograph of the processor chip inside the A5 package. The similar sized chip to the left is the 32 GB flash memory chip for non-volatile storage. There is an empty space between the two chips where a second flash chip can be installed to double storage capacity of the iPad. The chips to the right of the A5 include power controller and I/O controller chips. (Courtesy iFixit, www.ifixit.com)



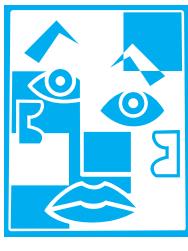
FIGURE 1.9 The processor integrated circuit inside the A5 package. The size of chip is 12.1 by 10.1 mm, and it was manufactured originally in a 45-nm process (see Section 1.5). It has two identical ARM processors or cores in the middle left of the chip and a PowerVR graphical processor unit (GPU) with four datapaths in the upper left quadrant. To the left and bottom side of the ARM cores are interfaces to main memory (DRAM). (Courtesy Chipworks, www.chipworks.com)

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). SRAM and DRAM are two layers of the **memory hierarchy**.

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

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, and so on.

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

The BIG Picture

implementation

Hardware that obeys the architecture abstraction.

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. A DVD disk is nonvolatile.

As mentioned above, one of the great ideas to improve design is abstraction. 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.

Both hardware and software consist of hierarchical layers using abstraction, with each lower layer hiding details from the level above. 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 disk doesn't forget the movie 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 former, and **secondary memory** for the latter. Secondary memory forms the next lower layer of the **memory hierarchy**. DRAMs have dominated main memory since 1975, but **magnetic disks** dominated secondary memory starting even earlier. Because of their size and form factor, personal Mobile Devices use **flash memory**, a nonvolatile semiconductor memory, instead of disks. Figure 1.8 shows the chip containing the flash memory of the iPad 2. While slower than DRAM, it is much cheaper than DRAM in addition to being nonvolatile. Although costing more per bit than disks, it is smaller, it comes in much smaller capacities, it is more rugged, and it is more power efficient than disks. Hence, flash memory is the standard secondary memory for PMDs. Alas, 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 5 describes disks and flash memory in more detail.

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 1.5 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 personal mobile device or server without a 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, computers on the network can share I/O devices.
- **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 up to 40 gigabits per second. Its length and speed make Ethernet useful to connect computers on the same floor of a building;



HIERARCHY

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

secondary memory Nonvolatile memory used to store programs and data between runs; typically consists of flash memory in PMDs and magnetic disks in servers.

magnetic disk Also called **hard disk**. A form of nonvolatile secondary memory composed of rotating platters coated with a magnetic recording material. Because they are rotating mechanical devices, access times are about 5 to 20 milliseconds and cost per gigabyte in 2012 was \$0.05 to \$0.10.

flash memory A nonvolatile semiconductor memory. It is cheaper and slower than DRAM but more expensive per bit and faster than magnetic disks. Access times are about 5 to 50 microseconds and cost per gigabyte in 2012 was \$0.75 to \$1.00.

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.

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 web. They are typically based on optical fibers and are leased from telecommunication companies.

Networks have changed the face of computing in the last 30 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 30 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 1 to 40 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 30 years.

For the last decade another innovation in networking is reshaping the way computers communicate. Wireless technology is widespread, which enabled the PostPC Era. 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 memory, flash memory, and disk storage differ significantly. For each technology, list its volatility, approximate relative access time, and approximate relative cost compared to DRAM.

1.5**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 1.10](#) shows the technologies that have

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
2013	Ultra large-scale integrated circuit	250,000,000,000

FIGURE 1.10 Relative performance per unit cost of technologies used in computers over time. Source: Computer Museum, Boston, with 2013 extrapolated by the authors. See [Section 1.12](#).

been used over time, with an estimate of the relative performance per unit cost for each technology. 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.

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. When Gordon Moore predicted the continuous doubling of resources, he was predicting the growth rate of the number of transistors per 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 1.11 shows the growth in DRAM capacity since 1977. For decades, the industry has consistently quadrupled capacity every 3 years, resulting in an increase in excess of 16,000 times!

To understand how manufacture integrated circuits, we 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)

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.

silicon A natural element that is a semiconductor.

semiconductor A substance that does not conduct electricity well.

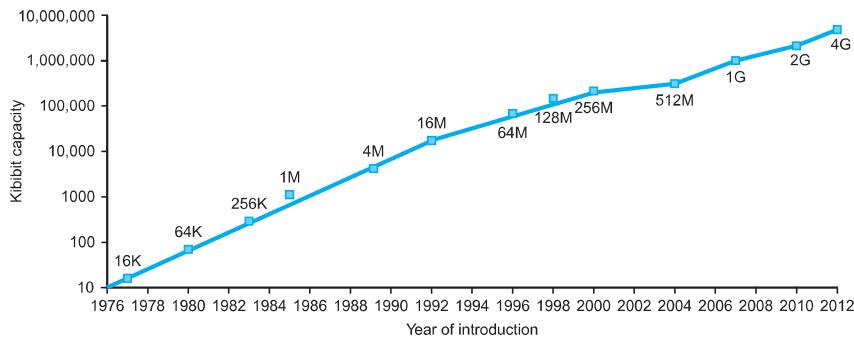


FIGURE 1.11 Growth of capacity per DRAM chip over time. The y-axis is measured in kibibits (2^{10} bits). The DRAM industry quadrupled capacity almost every three years, a 60% increase per year, for 20 years. In recent years, the rate has slowed down and is somewhat closer to doubling every two years to three years.

- 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.

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 inches thick, used to create chips.

The manufacturing process for integrated circuits is critical to the cost of the chips and hence important to computer designers. [Figure 1.12](#) 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 inches thick. These wafers then go through a series of processing steps, during which patterns of chemicals are placed on 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.

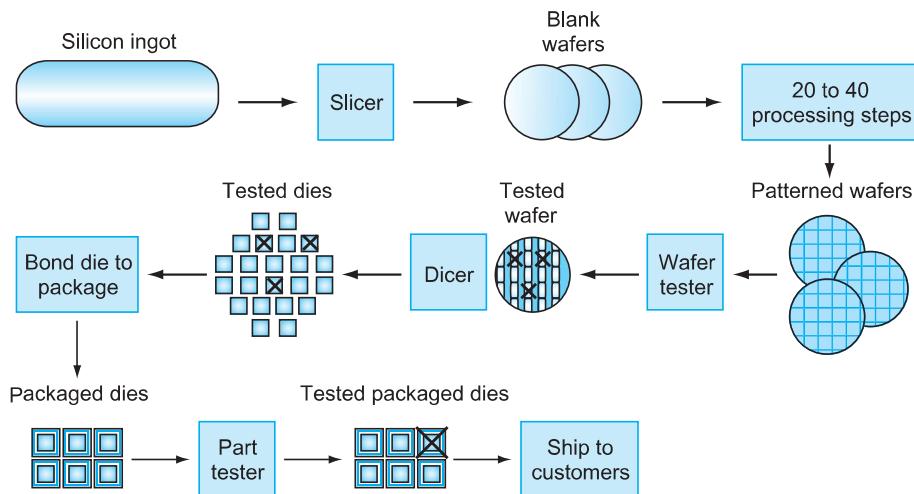


FIGURE 1.12 The chip manufacturing process. After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers (see [Figure 1.13](#)). These patterned wafers are then tested with a wafer tester, and a map of the good parts is made. Then, the wafers are diced into dies (see [Figure 1.19](#)). 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.

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

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. The simplest way to cope with imperfection is to place many independent components on a single wafer. The patterned wafer is then chopped up, or *diced*, into these components,

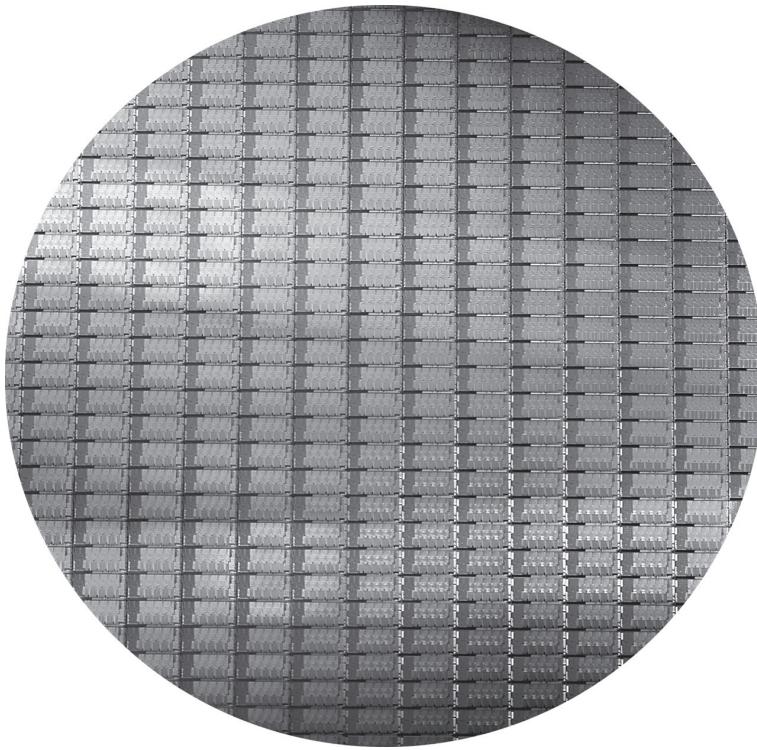


FIGURE 1.13 A 12-inch (300 mm) wafer of Intel Core i7 (Courtesy Intel). The number of dies on this 300 mm (12 inch) wafer at 100% yield is 280, each 20.7 by 10.5 mm. The several dozen partially rounded chips at the boundaries of the wafer are useless; they are included because it's easier to create the masks used to pattern the silicon. This die uses a 32-nanometer technology, which means that the smallest features are approximately 32 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.

called **dies** and more informally known as **chips**. Figure 1.13 shows a photograph of a wafer containing microprocessors before they have been diced; earlier, Figure 1.9 shows an individual microprocessor die.

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, using the next generation process shrinks a large die as it uses smaller sizes for both transistors and wires. This improves the yield and the die count per wafer. A 32-nanometer (nm) process was typical in 2012, which means essentially that the smallest feature size on the die is 32 nm.

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.

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.

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

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

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

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

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 1.13](#)). 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 the die area.

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.

1.6

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

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 1.14 lists 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 was the Concorde (retired from service in 2003), the plane with the longest range is the DC-8, and the plane with the largest capacity is the 747.

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 1.14 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).

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 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

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.

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 personal mobile devices, which are more focused on response time, versus servers, which are more focused on throughput.

EXAMPLE

ANSWER

Throughput and Response Time

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 web

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

$$\begin{aligned} \text{Performance}_X &> \text{Performance}_Y \\ \frac{1}{\text{Execution time}_X} &> \frac{1}{\text{Execution time}_Y} \\ \text{Execution time}_Y &> \text{Execution time}_X \end{aligned}$$

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 as fast as Y, then the execution time on Y is n times as long as it is on X:

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

Relative Performance

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?

EXAMPLE

We know that A is n times as fast as 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 as fast as 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

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

ANSWER

For simplicity, we will normally use the terminology *as fast as* 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 over which 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 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.

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.

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.

1. Suppose we know that an application that uses both personal mobile devices and the Cloud 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 PMD and the Cloud, 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 as fast as 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

clock cycle Also called **tick**, **clock tick**, **clock period**, **clock**, or **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

execution time. A simple formula relates the most basic metrics (clock cycles and clock cycle time) to CPU time:

$$\text{CPU execution time for a program} = \frac{\text{CPU clock cycles for a program}}{\text{Clock cycle time}}$$

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

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

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:

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

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

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

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. 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

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \frac{\text{Average clock cycles}}{\text{per instruction}}$$

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 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.

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

Using the Performance Equation

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?

EXAMPLE

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:

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

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

Now we can compute the CPU time for each computer:

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

Likewise, for B:

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

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

$$\frac{\text{CPU performance}_A}{\text{CPU performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{600 \times I \text{ ps}}{500 \times I \text{ ps}} = 1.2$$

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

The Classic CPU Performance Equation

instruction count The number of instructions executed by the program.

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:

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

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

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:

Code sequence	Instruction counts for each instruction class		
	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?

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:

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

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

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

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

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

EXAMPLE

ANSWER

The BIG Picture

Figure 1.15 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:

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

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 1.15 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 Chapter 4 and Chapter 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 1.10](#), 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.

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.

Understanding Program Performance

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 divides, 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.

Elaboration: Although clock cycle time has traditionally been fixed, to save energy or temporarily boost performance, today's processors can vary their clock rates, so we would need to use the average clock rate for a program. For example, the Intel Core i7 will temporarily increase clock rate by about 10% until the chip gets too warm. Intel calls this *Turbo mode*.

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:

- $\frac{15 \times 0.6}{1.1} = 8.2 \text{ sec}$
- $15 \times 0.6 \times 1.1 = 9.9 \text{ sec}$
- $\frac{15 \times 1.1}{0.6} = 27.5 \text{ sec}$

1.7 The Power Wall

Figure 1.16 shows the increase in clock rate and power of eight generations of Intel microprocessors over 30 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.

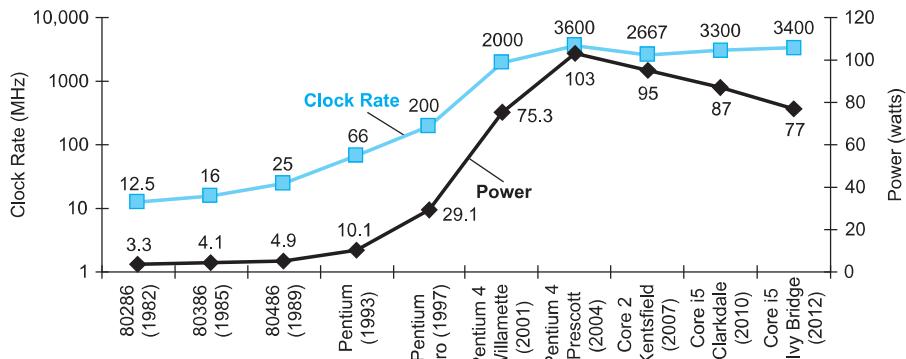


FIGURE 1.16 Clock rate and Power for Intel x86 microprocessors over eight generations and 25 years.

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 Core i5 pipelines follow in its footsteps.