# 1.1 Introduction

> "Civilization advances by extending the number of important operations which we can perform without thinking about them.
> *Alfred North Whitehead, An Introduction to Mathematics, 1911*



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.

**PARTICIPATION ACTIVITY** 1.1.1: The information revolution.

1) The computing industry has not improved quite as rapidly as the transportation industry.
   - True
   - False

2) The agricultural and industrial revolutions each transformed society. Computers have led to a relatively recent information revolution.
   - True
   - False

3) Computer improvements have led to previously undreamt applications like cell phones, but most signs suggest the improvements are now coming to an end.
   - True
   - False



## Traditional classes of computing applications and their characteristics

Although a common set of hardware technologies (see COD Sections 1.4 (Under the covers) and 1.5 (Technologies for building processors and memory)) 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 dissimilar ways. Broadly speaking, computers are used in three different classes of applications: personal computers, servers, and embedded computers.



**PARTICIPATION ACTIVITY** | 1.1.2: Classes of computing: Personal computers, servers, and embedded computers.

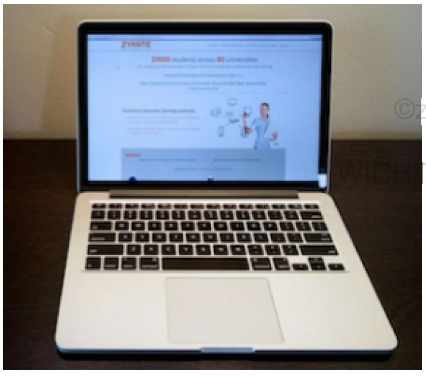
### Animation captions:

1. A personal computer (PC) is a computer designed for use by an individual, usually incorporating a graphics display, a keyboard, and a mouse.
2. An embedded computer is a computer inside another device used for running one predetermined application or collection of software.
3. A server is a computer used for running larger programs for multiple users, often simultaneously, and typically accessed only via a network.

*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!

***Personal computer*** (***PC***): A computer designed for use by an individual, usually incorporating a graphics display, a keyboard, and a mouse.

Figure 1.1.1: Personal computers: Desktop and laptop computers.





Source: zyBooks

*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 sizable 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.



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

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 COD Section 6.10 (Multiprocessor benchmarks and performance models)). 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 (see terabytes discussion near this section's end).

## Figure 1.1.2: Servers: Each rack contains multiple servers.

Source: Florian Hirzinger / GFDL or CC-BY-SA-3.0 via Wikimedia Commons



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 thus a proportionally tiny fraction of the overall computer market in terms of total revenue.

***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.

Figure 1.1.3: Supercomputers: Cray Y 190A supercomputer.



Source: NASA / Public domain via Wikimedia Commons

*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!



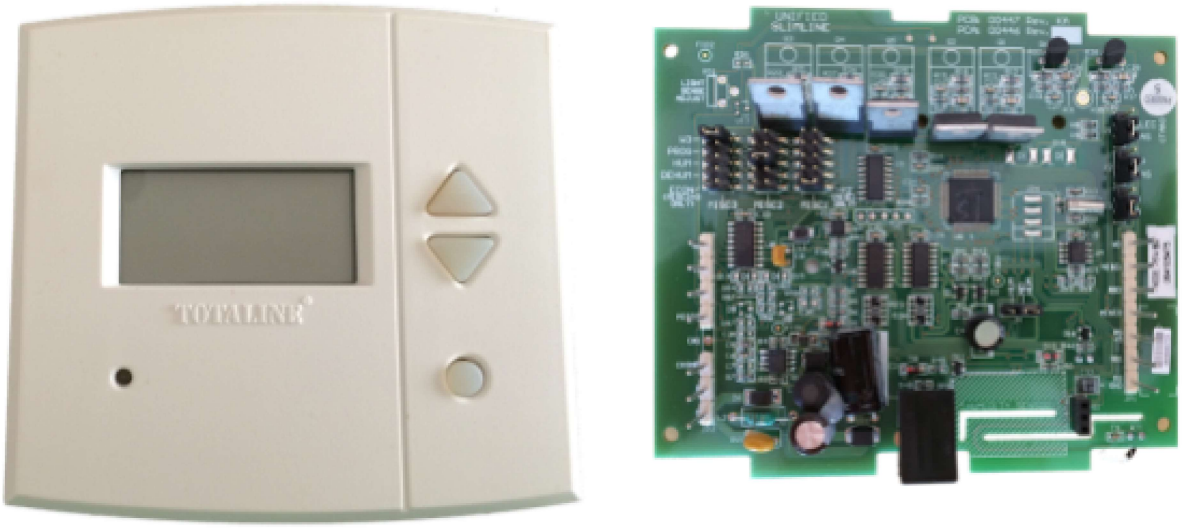
***Embedded computer***: A computer inside another device used for running one predetermined application or collection of software.

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 is the most important objective. 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.



Figure 1.1.4: Embedded computer: Thermostat.




Source: zyBooks

**PARTICIPATION ACTIVITY** 1.1.3: The three classes of computing applications.

Indicate to which class each computing application belongs.

1) A home computer kept on a desktop and used by family members for emails, web browsing, social networking, and movie watching.
   - ○ Embedded
   - ○ PC
   - ○ Server



2) A computer in an Amazon building accessed by thousands of people for online shopping.
   - ○

- Embedded
- PC
- Server

3) A computer in a cardiac pacemaker, which delivers electric shocks to keep a human's heart beating properly.

- Embedded
- PC
- Server



4) A computer at a federal laboratory that continually executes sophisticated algorithms on massive amounts of data from various weather stations to develop accurate weather forecasts.

- Embedded
- PC
- Server

## Elaboration

*Elaborations are short features 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 COD Chapter 4 (The Processor)). The core allows a designer to integrate other application-specific hardware with the processor core for fabrication on a single chip.*



## Welcome to the Post-PC 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 PC is the *personal mobile device (PMD)*. PMDs are battery operated with

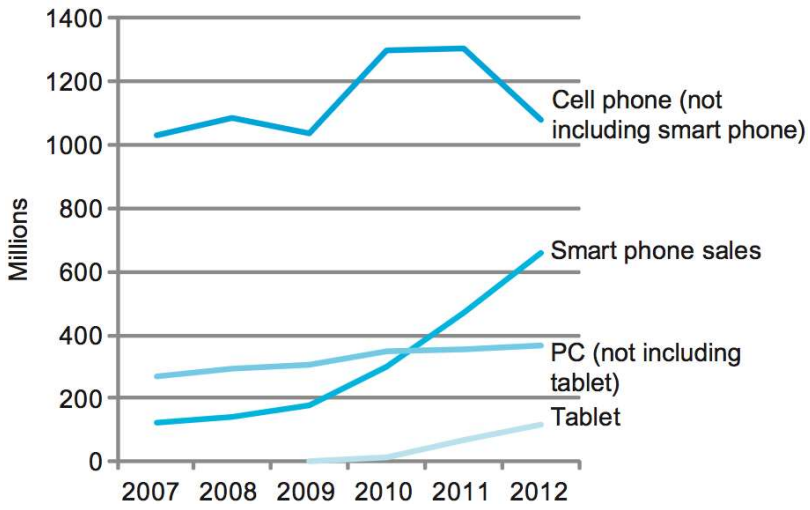
wireless connectivity to the Internet and typically cost hundreds of dollars, and, like PCs, users can download software ("apps") to run on them. Unlike PCs, they no longer have a keyboard and mouse, and are more likely to rely on a touch-sensitive screen or even speech input. Today's PMD is a smart phone or a tablet computer, but tomorrow it may include electronic glasses. The following figure shows the rapid growth over time of tablets and smart phones versus that of PCs and traditional cell phones.

***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.



Figure 1.1.5: The number manufactured per year of tablets and smart phones, which reflect the post-PC era, versus personal computers and traditional cell phones (COD Figure 1.2).

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.






Taking over from the conventional 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.

***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.



**PARTICIPATION ACTIVITY** 1.1.4: Post-PC era.

1) "Post-PC" era refers to today's situation of today's most widely-used computers being things other than PCs.
   - True
   - False

2) The number of PCs sold annually continues to increase at a dramatic rate.
   - True
   - False

3) A smart phone isn't actually a computer.
   - True
   - False

4) Cloud computing often involves thousands of servers in giant warehouses.
   - True
   - False



5) Software as a Service refers to installing large software applications on a PC, such as Microsoft Office.
   - True
   - False

6) In 2012, about 600 million smart

phones were sold.

- True
- False

## 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 popular software. In the 1960s and 1970s, a primary constraint on computer performance was the size of the computer's memory. Thus, programmers often followed a simple credo: minimize memory space to make programs fast. In the last decade, advances in computer design and memory technology have greatly reduced the importance of small memory size in most applications other than those in embedded computing systems.

Programmers interested in performance now need to understand the issues that have replaced the simple memory model of the 1960s: the parallel nature of processors and the hierarchical nature of memories. We demonstrate the importance of this understanding in COD Chapters 3 (Arithmetic for Computers) to 6 (Parallel Processors from Client to Cloud) by showing how to improve performance of a C program by a factor of 200. Moreover, as we explain in COD Section 1.7 (The power wall), 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 COD Chapter 6 (Parallel Processors from Client to Cloud)).
- 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 include a definition of each key term in a colored box near the first place a key term appears in the text. After a short time of working with the terminology, you will be fluent, and your friends will be impressed as you correctly use acronyms such as BIOS, CPU, DIMM, DRAM, PCIe, SATA, and many others.

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

**PARTICIPATION ACTIVITY** 1.1.5: What can be learned from this book.



1) Today's programmers of PCs and servers emphasize improving program performance by using a minimal amount of memory.
   - True

False

2) Today's programmers also emphasize energy efficiency of programs.
   - True
   - False

3) The performance of a program depends on many things, like the original program, how the program is translated to a computer's language, and the hardware.
   - True
   - False



4) A multicore microprocessor is a single processor capable of switching between multiple programs.
   - True
   - False

5) Acronyms are rarely used in the field of computers.
   - True
   - False

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

## Understanding program performance

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

| Hardware or software component | How this component affects performance | Where is this topic covered? |
|---|---|---|

| | | |
|---|---|---|
| Algorithm | Determines both the number of source-level statements and the number of I/O operations executed | Other books! |
| Programming language, compiler, and architecture | Determines the number of computer instructions for each source-level statement | COD Chapters 2 (Instructions: Language of the Computer) and 3 (Arithmetic for Computers) |
| Processor and memory system | Determines how fast instructions can be executed | COD Chapters 4 (The Processor), 5 (Large and Fast: Exploiting Memory Hierarchy), and 6 (Parallel Processor from Client to Cloud) |
| I/O system (hardware and operating system) | Determines how fast I/O operations may be executed | COD Chapters 4 (The Processor), 5 (Large and Fast: Exploiting Memory Hierarchy), and 6 (Parallel Processor from Client to Cloud) |



To demonstrate the impact of the ideas in this book, as mentioned above, 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 COD Chapter 3 (Arithmetic for Computers) we use *subword parallelism via C intrinsics* to increase performance by a factor of 3.8.
- In the category of *instruction level parallelism*, in COD Chapter 4 (The Processor) 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 COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy) we use *cache blocking* to increase performance on large matrices by another factor of 2.0 to 2.5.
- In the category of *thread level parallelism*, in COD Chapter 6 (Parallel Processor from Client to Cloud) we use *parallel for loops in OpenMP to exploit multicore hardware* to increase performance by another factor of 4 to 14.



## A "terabyte" and other sizes

The above discussion on supercomputers used the term "terabyte".

***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. The figure below shows the full range of decimal and binary values and names.



## Table 1.1.1: Size ambiguity resolved with binary notation for common size terms (COD 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.

| Decimal | 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% |



**PARTICIPATION ACTIVITY** 1.1.6: Terms for common sizes.

1) A terabyte is one _____bytes.

Check Show answer

2) $10^{15}$ bytes is a _____byte.

Check Show answer

3) A gibibyte is close, but not equal, to a _____byte.



Check Show answer

**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.

1. The number of embedded processors sold every year greatly outnumbers the number of PC and even Post-PC 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

**Answer:** Discussion questions: many answers are acceptable.

# 1.2 Eight great ideas in computer architecture



We now introduce eight great ideas that computer architects have 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

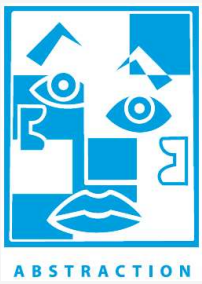
great ideas and we use them to identify the nearly 100 sections of the book that feature use of the great ideas.

## Design for Moore's Law


The one constant for computer designers is rapid change, which is driven largely by Moore's Law. ***Moore's Law*** 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 characterize 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 usually 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 COD Section 1.6 (Performance)). 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!

## Performance via parallelism


Since the dawn of computing, computer architects have offered designs that get more performance by computing 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***, which moves multiple operations through hardware units that each do a piece of an operation, akin to water flowing through a pipeline. 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 next great idea is prediction. The idea of ***prediction*** is that, 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 the 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 conflicting demands of fast, large, and cheap memory 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 COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy), caches give the programmer the illusion that main memory is almost 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 axles 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!)



**PARTICIPATION ACTIVITY** 1.2.1: Eight great ideas in computer architecture.

Match the situation with the closest analog of a great idea in computer architecture.

**Hierarchy of Memories** | **Use Abstraction to Simplify Design** | **Design for Moore's Law** | **Make the Common Case Fast**



| | |
|---|---|
| | A soccer player runs not to where the ball is, but to where the ball will be. |
| | A customer talks to a phone agent. If there's a problem, he talks to the agent's supervisor. |
| | A house architect first designs a house with 5 rooms, then designs room details like closets, windows, and flooring. |
| | A college student rents an apartment closer to campus than to her favorite weekend beach spot. |

Reset

**PARTICIPATION ACTIVITY** 1.2.2: Eight great ideas in computer architecture (continued).

Match the situation with the closest analog of a great idea in computer architecture.

**Dependability via Redundancy** | **Performance via Prediction** | **Performance via Parallelism** | **Performance via Pipelining**



| | |
|---|---|
| | A sister is hanging clothes to dry. Her brother helps by hanging clothes simultaneously. |

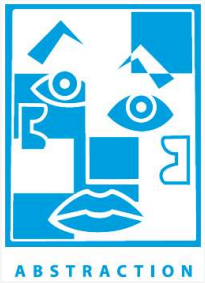
| | |
|---|---|
| | A brother is washing and drying dishes. His sister helps by drying each dish immediately after the brother washes each. |
| | A mom expects her son will be hungry after a long airplane flight, so she cooks dinner just in case. If he's not hungry, she'll whip up a dessert instead. |
| | A drummer's stick breaks, but he quickly grabs another one and continues playing the song. |

Reset



# 1.3 Below your program

> In Paris they simply stared when I spoke to them in French; I never did succeed in making those idiots understand their own language.
> *Mark Twain, The Innocents Abroad, 1869*




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 primitive 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**.



The animation below 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 application software.

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

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

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



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

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

**PARTICIPATION ACTIVITY** 1.3.1: A simplified view of hardware and software as hierarchical layers, shown as concentric circles with hardware in the center and applications software outermost (COD Figure 1.3).

### Animation captions:

1. Hardware is the physical machine that runs software.
2. Systems software, like an operating system, runs on the hardware and provides useful services.
3. Application software runs on top of systems software.

*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 COD Chapter 2 (Instructions: Language of the Computer).

***Compiler***: A program that translates high-level language statements into assembly language statements.



**PARTICIPATION ACTIVITY** 1.3.2: Below your program.

1) Behind a car's simple items like steering wheel, gas pedal, and brake pedal are complex

mechanical/computerized details. Those simple items represent _____.

- abstraction
- an operating system
- Linux

2) The physical machinery on which software runs.



- Instructions
- Hardware
- Compiler

3) Software that manages hardware resources on behalf of other programs.

- Hardware
- Compiler
- Operating system

4) A program to play tic-tac-toe.

- Systems software
- Application software

5) The collection of software on a computer that provides services to application software.

- Systems software
- Compiler

## From a high-level language to the language of hardware

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

```
1000110010100000
```

tell one computer to add two numbers. COD Chapter 2 (Instructions: Language of the Computer) 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.

***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.

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 software 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*.

***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.



**PARTICIPATION ACTIVITY** 1.3.3: Bits, assembly language, and machine language.

1) "Bit" is short for "binary digit."

- True

- False

2) Binary refers to $10^{-1}$.

- True
- False

3) Computers use binary because binary is more powerful than decimal numbers.

- True
- False



4) Although binary's alphabet contains only two "letters", 0 and 1, the binary alphabet can represent as much information as the English alphabet's 26 letters.

- True
- False

5) The number 12 can be represented in binary as 1100. If a computer's memory location contains 00001100, then that location contains the number 12.

- True
- False

6) The following could be a machine-language instruction: 1000110010100000.

- True
- False



7) The following could be an assembly language instruction: 1000110010100000.

- True
- False

8) An assembler translates assembly language instructions like

`ADD A,B`
to machine-language instructions like
1000110010100000.

- True
- False




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. The animation below shows the relationships among these programs and languages, which are more examples of the power of **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.

**PARTICIPATION ACTIVITY** 1.3.4: C program compiled into assembly language and then assembled into binary machine language (COD Figure 1.4).

## Animation captions:

1. A programmer writes a high-level language program (in the C language here).
2. A compiler converts the high-level language program to an assembly language program (in this case, for LEGv8).
3. An assembler converts the assembly language program into a binary machine language program (for LEGv8).



Note: Above, 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 COD Chapter 2 (Instructions: Language of the Computer).

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 the animation above). 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.

**PARTICIPATION ACTIVITY** 1.3.5: High-level programming language.

1) Which is a high-level language instruction?

- `00000000101000100000000100011000`
- `ADD  X2, X4, X2`
- temp = v[k];

2) What kind of language is C?



- Machine
- Assembly
- High-level

3) An advantage of a high-level language is allowing a programmer to _____.

- think more naturally

- think like a machine

4) An advantage of a high-level language is enabling a programmer to _____.
   - change a program
   - implement a program in less time

5) An advantage of a high-level language is that a program _____.
   - is specific to a particular machine
   - is independent of a particular machine



# 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 significant 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 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.

***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.

COD Chapters 5 (Large and Fast: Exploiting Memory Hierarchy) and 6 (Parallel Processor from Client to Cloud) 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.

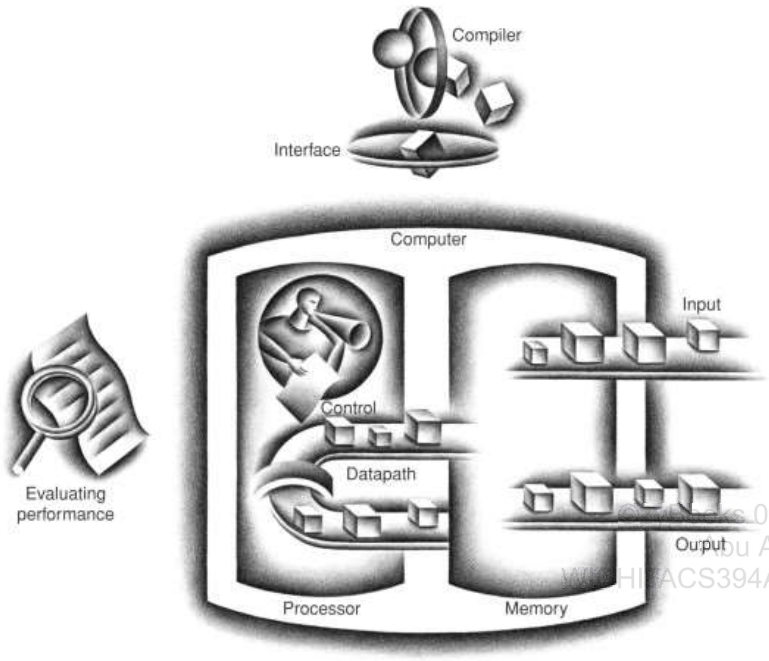
## The Big Picture

The five classic components of a computer are input, output, memory, datapath, and control, with the last two sometimes combined and called the processor. The figure below 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.



## Figure 1.4.1: The organization of a computer, showing the five classic components (COD Figure 1.5).

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.






**PARTICIPATION ACTIVITY** 1.4.1: The organization of a computer, showing the five classic components (COD Figure 1.5).

## Animation captions:

1. Input data from a keyboard, file, network, etc., is written in memory.
2. A datapath operates on data.
3. Output reads data from memory.
4. Control sends the signals that determine the operations of the datapath, memory, input, and output.



**PARTICIPATION ACTIVITY** 1.4.2: The five components of a computer.

**Output** **Memory** **Control** **Input** **Datapath**

| | |
|---|---|
| | Writes data to memory. Ex: Keyboard. |
| | Reads data from memory. Ex: Display. |
| | Stores instructions and data. |
| | Sends signals that determine the operation of the other components. |
| | Performs computations. |

**Reset**

## Through the looking glass

Source: zyBooks

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 control current

precisely and make sharper images. A red-green-blue mask associated with each dot on the display determines the intensity of the three-color components in the final image; in a color active matrix LCD, there are three transistor switches at each point.

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



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

The image is composed of a matrix of picture elements, or *pixels*, which can be represented as a matrix of bits, called a ***bit map***. Depending on the size of the screen and the resolution, the display matrix 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.

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

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

The goal of the bit map is to represent faithfully 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.

**PARTICIPATION ACTIVITY** 1.4.3: Each coordinate in the frame buffer on the right determines the color of the corresponding coordinate for the raster scan CRT display on the left (COD Figure 1.6).

### Animation captions:

1. A display conveys output to a user.
2. Each pixel has a corresponding (X, Y) coordinate.
3. A frame buffer determines the color/shade of each pixel. Pixel (X0, Y0)'s pattern of 0011 is different than pixel (X1, Y1)'s 1011.



> 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*

## Touchscreen

While PCs also use LCD displays, the tablets and smartphones of the post-PC era have replaced the keyboard and mouse with touch-sensitive displays, which has the wonderful user interface advantage of users pointing directly at 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 recognizes gestures that can lead to attractive user interfaces.

**PARTICIPATION ACTIVITY** 1.4.4: Displays.

1) A liquid crystal display works by having the liquid crystal generate different colors of light.
   - True
   - False

2) A typical computer display is made up of hundreds of pixels.
   - True
   - False

3) An active matrix display is an LCD that uses a transistor to control whether light passes for each pixel.
   - True
   - False

4) Each pixel of a display typically involves 24 bits, with 8 bits for each of red, blue, and green.
   - True
   - False

5) A frame buffer stores the pixel values for a display.
   - True

- False

6) A touchscreen combines a display for output with a touch-sensitive screen for input.

- True
- False



## Opening the box

The figure below 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.

### Figure 1.4.2: Components of the Apple iPad 2 A1395 (COD Figure 1.7).

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 3.8V, 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. The next figure 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 six-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 cable at the bottom is the connector between the logic board and the camera/volume control board. The board between the cable and the speaker assembly is the controller for the capacitive touchscreen. (Courtesy iFixit, www.ifixit.com)



The small rectangles in the figure below contain the devices that drive our advancing technology, called *integrated circuits* and nicknamed *chips*. The A5 package seen in the middle of the figure below contains two ARM processors that operate at 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*.

***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.

Figure 1.4.3: Logic board of Apple iPad 2 (COD Figure 1.8).

The logic board of Apple iPad 2 in the previous figure. The photo highlights five integrated circuits. The large integrated circuit in the middle is the Apple A5 chip, which contains dual ARM processor cores that run at 1 GHz as well as 512 MB of main memory inside the package. The next figure shows a photograph of the processor chip inside the A5 package. The similar-sized chip to the right is the 32GB 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 left of the A5 include power controller and I/O controller chips. (Courtesy iFixit, www.ifixit.com)

Descending even lower into the hardware, the figure below 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. COD Chapter 4 (The Processor) explains the datapath and control for a higher-performance design.

***Datapath***: The component of the processor that performs arithmetic operations.

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

The A5 package in the figure above also includes two memory chips, each with 2 gibibits of capacity, thereby supplying 512MiB. 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.

***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 1.4.4: Processor integrated circuit (COD 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 COD Section 1.5 (Technologies for building processors and memory)). 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).

**PARTICIPATION ACTIVITY** 1.4.5: Processors, chips, and the iPad 2.

1) An integrated circuit is often called a _____.
   - chip
   - CPU



2) The CPU chip physically occupies _____ of the size of the iPad 2.
   - most
   - a small fraction

3) The iPad 2 consists of how many chips?
   - 1
   - 2
   - 5

4) The A5 package has a chip containing ____ ARM processors.
   - 1
   - 2
   - 5

5) A CPU is also known as _____.
   - a datapath
   - control
   - a processor

Descending into the depths of any component of the hardware reveals insights into the computer. Inside the processor is another type of memory—cache memory.


*Cache memory* consists of a small, fast memory that acts as a buffer for the DRAM memory. (The nontechnical definition of *cache* is a safe place for hiding things.) Cache is built using a different memory technology, *static random access memory (SRAM)*. SRAM is faster but less dense, and hence more expensive, than DRAM (see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)). 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.



**PARTICIPATION ACTIVITY** 1.4.6: RAM and cache.

**SRAM** **DRAM** **Cache**

| | |
|---|---|
| | Large memory where most data is stored. |
| | A faster memory technology than DRAM, but using more area to store a bit. |
| | A small memory that keeps a copy of data from larger memory. |

**Reset**


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)*.



***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.

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) separately 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.

***Implementation***: Hardware that obeys the architecture abstraction.



## The 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.

**PARTICIPATION ACTIVITY** 1.4.7: Instruction set architecture.

1) An instruction set architecture enables a machine language program to run on different hardware implementations.
   - ○ True
   - ○ False



2) While different hardware implementations may run the same program, designers strive to keep the performance of new hardware implementations the same as older implementations.

- True
- False

## 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 therefore a *nonvolatile memory* technology.



***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.


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. A previous figure (COD 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 (personal mobile devices). 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. COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy) describes disks and flash memory in more detail.

***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.



**PARTICIPATION ACTIVITY** 1.4.8: Memories.

**Main memory** **Volatile memory** **Flash memory** **Magnetic disk**

**Secondary memory** **Nonvolatile memory**

| | |
|---|---|
| | Memory layer used to hold programs and data while programs are running. |
| | Memory layer used to store programs and data between runs. |
| | A form of memory that retains data only if the memory is receiving power. |
| | A form of memory that retains data in the absence of a power. |
| | A nonvolatile semiconductor memory often used as secondary memory for personal mobile devices. |
| | A form of nonvolatile secondary memory composed of rotating platters coated with a magnetic recording material |



Reset

## 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 a previous figure (COD 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; 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.

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

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

Networks have changed the face of computing in the last 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 considerably cheaper and had a significantly 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 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 post-PC 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.



**PARTICIPATION ACTIVITY** 1.4.9: Networks.

1) Network connectivity is a key part of modern computing devices like personal computers, tablets, and smartphones.
   - True
   - False

2) Ethernet is an example of a wide area network.
   - True
   - False

3) Wireless networks continue to become more common.
   - True
   - False

**PARTICIPATION ACTIVITY** 1.4.10: Check yourself: DRAM memory, flash memory, and disk storage characteristics.



1) _____ memory is volatile.
   - DRAM
   - Flash
   - Disk

2) _____ memory is the most expensive per GB.

- DRAM
- Flash
- Disk

3) _____ memory has the longest access time.

- DRAM
- Flash
- Disk



# 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. The table below shows the technologies that have 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.

Table 1.5.1: Relative performance per unit cost of technologies used in computers over time (COD Figure 1.10).

| 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 |

Source: Computer Museum, Boston, with 2013 extrapolated by the authors.

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 forecasting 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*.



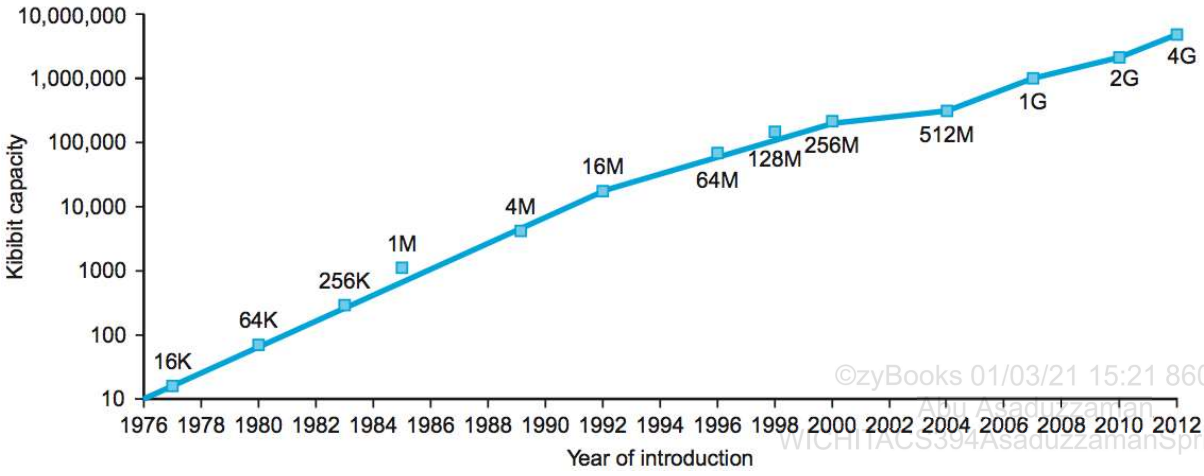
***Transistor***: An on/off switch controlled by an electric signal.

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

This rate of increasing integration has been remarkably stable. The figure below shows the growth in DRAM capacity since 1977. For 35 years, the industry has consistently quadrupled capacity every 3 years, resulting in an increase in excess of 16,000 times!

## Figure 1.5.1: Growth of capacity per DRAM chip over time (COD Figure 1.11).

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.






**PARTICIPATION ACTIVITY** 1.5.1: Technologies in processors and memory.

1) Very large-scale integrated circuits

combine dozens to hundreds of vacuum tubes into a single chip.

- True
- False

2) From the 1970s to 1990s, DRAM capacity has quadrupled every three years.

- True
- False



3) DRAM capacity today continues to quadruple every three years.

- True
- False

To understand how to 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)
- Excellent insulators from electricity (like plastic sheathing or glass)
- Areas that can conduct or insulate under special conditions (as a switch)

***Silicon***: A natural element that is a semiconductor.

***Semiconductor***: A substance that does not conduct electricity well.

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

The manufacturing process for integrated circuits is critical to the cost of the chips and hence important to computer designers. The figure below 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.

***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.



**PARTICIPATION ACTIVITY** 1.5.2: The chip manufacturing process (COD Figure 1.12).

### Animation captions:

1. After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers.
2. These patterned wafers are then tested with a wafer tester, and a map of the good parts is made.
3. Then, the wafers are diced into dies.
4. 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.

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

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

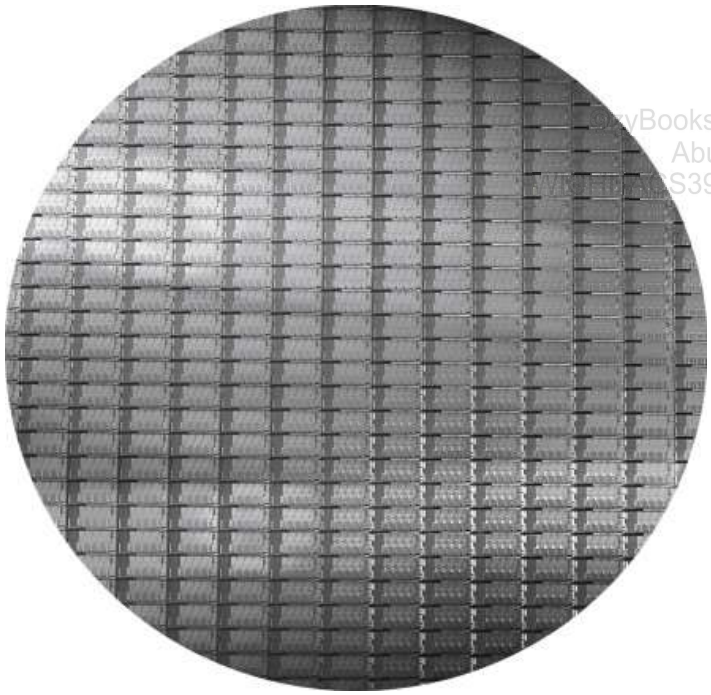
***Die***: The individual rectangular sections that are cut from a wafer, more informally known as ***chips***.



Figure 1.5.2: A 12-inch (300mm) wafer of Intel Core i7 (Courtesy Intel) (COD Figure 1.13).

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.





**PARTICIPATION ACTIVITY** 1.5.3: Chip manufacturing process.

1) Silicon is _____.
   - a rare element
   - a common element found in sand

2) The manufacturing process begins with a silicon _____.
   - ingot
   - wafer
   - die



3) Blank wafers undergo 20 to 40 chemical processing steps to create _____.
   - transistors, conductors, and insulators
   - silicon
   -

dies

4) A wafer tester evaluates the patterned wafer for the presence of _____.
   - patterns
   - defects
   - components

5) After the patterned wafer is chopped up into dies, the dies are _____.
   - packaged
   - shipped
   - diced



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.

***Yield***: 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 to the fewer 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.

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 the figure above). 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.*



**PARTICIPATION ACTIVITY** 1.5.4: Check yourself: Yield and chips costs.

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.
   - ○ True
   - ○ False

2) It is less work to design a high-volume part than a low-volume part.
   - ○ True
   - ○ False

3) The masks used to make the chip are expensive, so the cost per chip is lower for higher volumes.
   - ○ True
   - ○ False



4) Engineering development costs are high and largely independent of volume; thus, the development cost per die is lower with high-volume parts.
   - ○ True
   - ○ False

5) High-volume parts always have smaller die sizes than low-volume parts and therefore have higher yield per wafer.

- True
- False



# 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 those 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. The table below 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.



Table 1.6.1: The capacity, range, and speed for a number of commercial airplanes (COD Figure 1.14).

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).

| Airplane | Passenger capacity | Cruising range (miles) | Cruising speed (m.p.h.) | Passenger throughput (passengers x 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 |



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 distinct ways.

If you were running a program on two different desktop computers, you'd say that the faster one is the desktop computer that gets the job done first. If you were running a datacenter that had several servers running jobs submitted by many users, you'd say that the faster computer was the one that completed the most jobs during a day. As an individual computer user, you are interested in reducing *response time*—the time between the start and completion of a task—also referred to as *execution time*. Datacenter managers often care about 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.

***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.

**PARTICIPATION ACTIVITY** 1.6.1: Execution time and throughput.

1) At Joe's Car Wash, a car enters the wash, and exits 5 minutes later (nice and clean). 5 minutes is the _____.
   - execution time
   - throughput

2) At Joe's Car Wash, a car enters the wash, and exits 5 minutes later. However, 1 minute after a car enters, the next car can enter. Thus, once full, the car wash outputs 1 clean car per minute. 1 car per minute is the _____.
   - response time
   - throughput



3) Response time is synonymous with _____.
   - execution time
   - throughput

4) Cars drive 60 km/h over a 1 km long bridge. A car thus requires 1 minute to cross the bridge. Cars stay separated by about 100 m, so 1 car enters and another exits the bridge every 6 seconds. The execution time is _____.
   - 1 minute
   - 6 seconds

5) Cars drive 60 km/h over a 1 km long bridge. A car thus requires 1 minute to cross the bridge. Cars stay separated by about 100 m, so 1 car enters and another exits the bridge every 6 seconds. The throughput is _____.
   - 6 cars / sec
   - 10 cars / minute



6) Increasing the speed limit for cars driving over a bridge _____ the execution time.

- improves
- worsens

7) Increasing the speed limit for cars driving over a bridge while keeping the same minimum separation between cars _____ the throughput.

- improves
- worsens



**PARTICIPATION ACTIVITY** 1.6.2: Example of throughput and response time.

1) Replacing a processor in a computer with a faster processor has what effect?

- Decreases response time
- Increases throughput
- Both (decreases response time and increases throughput)

2) Adding additional processors to a system that uses multiple processors for separate tasks -- for example, searching the web -- has what effect? Assume that before adding processors, tasks do not wait to execute (tasks do not "queue up").

- Decreases response time
- Increases throughput
- Both (decreases response time and increases throughput)

3) Adding additional processors to a system that uses multiple processors for separate tasks -- for example, searching the web -- has what effect? Assume that before adding processors, tasks often must wait to execute due to another task executing (tasks "queue up").

Decreases response time

- Increases throughput
- Both (decreases response time and increases throughput)

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.



**PARTICIPATION ACTIVITY** 1.6.3: Performance.

### Animation captions:

1. For a given task, Computer X has less execution time (is faster) than Computer Y.
2. Desired is a measure of "performance" where bigger is better ("more performance" means faster).
3. One such performance measure is the inverse of execution time.
4. The performance of two computers can be related quantitatively. Computer X is n times as fast as computer Y.
5. The execution time of two computers can be related quantitatively. The execution time on computer Y is n times as long as computer X.

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:

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



## Example 1.6.1: 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?

**Answer**

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

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

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."

**PARTICIPATION ACTIVITY** 1.6.4: Performance.

Computer A requires 10 seconds to compress a file. Computer B requires 5 seconds.

1) Which computer has higher execution time?
   - A
   - B



2) Which computer has higher performance?
   - A
   - B

3) If performance is 1 / execution time,

what is A's performance?

- 0.1
- 10

4) If performance is 1 / execution time, what is B's performance?

- 0.2
- 5



5) How many times faster is B than A at compressing a file?

- 2
- 1/2

6) To determine how many times faster Computer C is than Computer D, which is the correct calculation?

- PerfC / PerfD
- PerfD / PerfC

## 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 between operating systems.

***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.



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.

**PARTICIPATION ACTIVITY** 1.6.5: Measuring performance.

A task runs alone on a CPU. The task starts by running for 5 ms. The task then waits for 4 ms while the operating system runs some instructions to access disk. The CPU is then idle for 2 ms while waiting for data from disk. Finally, the task runs another 10 ms and completes.

1) The elapsed time is ____ ms.

Check Show answer

2) The user CPU time is _____ ms.

Check Show answer

3) The CPU time is _____ ms.



Check Show answer

4) The system performance is _____ ms.

Check Show answer

5) The CPU performance is _____ ms.

Check Show answer



## 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 find 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). ***Clock rate*** is the inverse of the clock period. In the next subsection, we will formalize the relationship between the clock cycles of the hardware designer and the seconds of the computer user.

***Clock cycle***: Also called ***tick***, ***clock tick***, ***clock period***, ***clock***, 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.

PARTICIPATION

**ACTIVITY** 1.6.6: Clock cycle and clock period.

1) A particular processor has a clock rate of 1 GHz. The clock thus ticks one billion times per second.
   - ○ True
   - ○ False



2) A clock rate of 1 GHz corresponds to a period of 1 nanosecond, which is $1x10^{-9}$ seconds.
   - ○ True
   - ○ False

**PARTICIPATION ACTIVITY** 1.6.7: Check yourself: Throughput and response time.

Suppose we know that an application that uses both personal mobile devices and the Cloud is limited by network performance. For the following changes, determine what improvements to the application's performance is/are made, if any.

1) 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).
   - ○ Only the throughput improves.
   - ○ Both response time and throughput improve.
   - ○ Neither response time or throughput improves.

2) The networking software is improved, thereby reducing the network communication delay, but not increasing throughput.
   - ○ Only the throughput improves.
   - ○ Only the response time improves.
   - ○ Neither response time or

throughput improves.

3) More memory is added to the computer.

- Both response time and throughput improve.
- Neither response time or throughput improves.



**PARTICIPATION ACTIVITY** 1.6.8: Check yourself: Program performance.

1) 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?

sec

Check Show answer

## CPU performance and its factors

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

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

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 1.6.2: 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 designe has determined that a substantial increase in the clock rate is possible, but this increase will affect th 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:

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

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

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

CPU time for B can be found using this equation:

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

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

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

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

**PARTICIPATION ACTIVITY** 1.6.9: Improving performance.



Refer to the above example.

1) What is the CPU clock cycles for computer A?

- $2 \times 10^9 \frac{\text{cycles}}{\text{sec}}$
-

- $20 \times 10^9$ cycles
- 10 sec

2) Computer B's performance is improved by reducing the _____.

- number of clock cycles required to execute the program
- length of a clock cycle



## 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

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

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 identical 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.

### Example 1.6.3: 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?



**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.

**PARTICIPATION ACTIVITY** 1.6.10: Using the performance equation.

Refer to the above example.

1) How does one know that each computer executes the same number of instructions for the program?
   - All computers use the same number of instructions for a given program.
   - Both computers use the same instruction set architecture.
   - Both computers use the same implementation.

2) Which computer has a faster clock?
   - Computer A
   - Computer B



3) Which computer requires fewer clock cycles to execute a single instruction?
   - Computer A
   - Computer B

4) If Computer A executes 1000 instructions for the program, what is the program's CPU time on Computer A?

- 1000 instr * 2.0 cycle/instr * 250 ps/cycle = 500,000 ps.
- 1000 instr * 1.2 cycle/instr * 500 ps/cycle = 600,000 ps.



5) If Computer A executes 1000 instructions for the program, how many instructions does Computer B execute for the program?

- 1000
- 1000 * 1.2 = 1200
- 1000 * 2.0 = 2000

6) For a particular program, Computers A and B execute 2000 instructions. A's CPU time is 2000 * 2.0 * 250 = 1,000,000 ps. B's is 2000 * 1.2 * 500 = 1,200,000 ps. How much faster is Computer A than B?

- 1.2
- 200,000

7) Computer A is better than Computer B.

- Yes
- Unclear

## The classic CPU performance equation

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



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

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.

***Instruction count***: The number of instructions executed by the program.



## Example 1.6.4: 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?

**Answer**

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



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

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

This yields

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

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

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



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

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

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

**PARTICIPATION ACTIVITY** 1.6.11: Comparing code segments.

Refer to the above example.

1) Instruction class ____ requires the largest number of cycles per instruction.

Check Show answer

2) Code sequence 2 executes ____ instructions.

Check Show answer

3) Code sequence 2 requires ____ CPU clock cycles.



Check Show answer

4) Assume a new code sequence 3 contains the following instruction

counts for each instruction class. What is code sequence 3's CPU clock cycles?

| Code sequence | Instruction counts for each instruction class | | |
|---|---|---|---|
| | A | B | C |
| 3 | 10 | 4 | 6 |



Check Show answer

5) Assume a new code sequence 3 contains the following instruction counts for each instruction class. What is code sequence 3's CPI?

| Code sequence | Instruction counts for each instruction class | | |
|---|---|---|---|
| | A | B | C |
| 3 | 10 | 4 | 6 |

Check Show answer

## The Big Picture

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

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

Always bear in mind that the only complete and reliable measure of computer

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



Table 1.6.2: The basic components of performance and how each is measured (COD Figure 1.15).

| 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 |

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 COD Chapter 4 (The Processor) and COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)), 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 equal. 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 COD Section 1.10 (Fallacies and pitfalls), 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.



**PARTICIPATION ACTIVITY** 1.6.12: Performance components.

1) Assume CPI and clock cycle time remain constant. Reducing the instruction count will reduce the program's execution time.
   - True
   - False

2) For a given number of instructions, assume CPI is increased by 20%, and clock cycle time is decreased by 10%. The program execution time decreases.
   - True
   - False

## Understanding program performance

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



| Hardware or software component | Affects what? | How? |
|---|---|---|
| Algorithm | Instruction count, possibly | The algorithm determines the number of source program instructions executed and |

| | | |
|---|---|---|
| | CPI | 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 varied 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 COD Chapter 4 (The Processor), 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 two 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.*



**PARTICIPATION ACTIVITY** 1.6.13: Check yourself: Program execution time.

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.

1) How fast can we expect the application to run using this new compiler?

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

# 1.7 The power wall

The figure below 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.



**PARTICIPATION ACTIVITY** 1.7.1: Clock rate and power for Intel x86 microprocessors over eight generations and 30 years (COD Figure 1.16).

## Animation captions:

1. The Pentium 4 made a dramatic jump in clock rate and power but less so in performance.
2. The Prescott thermal problems led to the abandonment of the Pentium 4 line.
3. The Core 2 line reverts to a simpler pipeline with lower clock rates and multiple processors per chip.
4. The Core i5 pipelines follow in its footsteps.



**PARTICIPATION ACTIVITY** 1.7.2: Clock rate and power trends.

1) As clock rates increased in early Intel processors, power _____.
   - increased
   - decreased

2) The _____ Intel processor consumed the most power.
   - 2004
   - 2010

3) Clock rates stopped increasing around 2004 because of power.
   - True
   - False

Although power provides a limit to what we can cool, in the post-PC era the really valuable resource is energy. Battery life can trump performance in the personal mobile device, and the architects of warehouse scale computers try to reduce the costs of powering and cooling 100,000 servers as the costs are high at this scale. Just as measuring time in seconds is a safer evaluation of program performance than a rate like MIPS (see COD Section 1.10 (Fallacies and pitfalls)), the energy metric joules is a better measure than a power rate like watts, which is just joules/second.

The dominant technology for integrated circuits is called CMOS (complementary metal oxide semiconductor). For CMOS, the primary source of energy consumption is so-called dynamic energy—that is, energy that is consumed when transistors switch states from 0 to 1 and vice versa. The dynamic energy depends on the capacitive loading of each transistor and the voltage applied:

$$Energy \propto Capacitive\ load \times Voltage^2$$

This equation is the energy of a pulse during the logic transition of $0 \rightarrow 1 \rightarrow 0$ or $1 \rightarrow 0 \rightarrow 1$. The energy of a single transition is then

$$Energy \propto 1/2 \times Capacitive\ load \times Voltage^2$$

The power required per transistor is just the product of energy of a transition and the frequency of transitions:

$$Power \propto 1/2 \times Capacitive\ load \times Voltage^2 \times Frequency\ switched$$



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

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

**PARTICIPATION ACTIVITY** 1.7.3: Energy and power.

1) Voltage = 4 V, frequency = 1 GHz, and power = 3 W. Frequency is increased to 6 GHz. What is the new power?

W

Check Show answer

2) Voltage = 4 V, frequency = 1 GHz, and power = 3 W. Voltage is decreased to 2 V. What is the new power? Type as: 0.##

W

Check Show answer



## Example 1.7.1: Relative power.

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

**Answer**

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

Thus the power ratio is

$$0.85^4 = 0.52$$

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



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

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

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

## Elaboration

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



## Elaboration

*Power is a challenge for integrated circuits for two reasons. First, power must be brought in and distributed around the chip; modern microprocessors use hundreds of pins just for power and ground! Similarly, multiple levels of chip interconnect are used solely for power and ground distribution to portions of the chip. Second, power is dissipated as heat and must be removed. Server chips can burn more than 100 watts,*

*and cooling the chip and the surrounding system is a major expense in warehouse scale computers (see COD Chapter 6 (Parallel Processors from Client to Cloud)).*

**PARTICIPATION ACTIVITY** 1.7.4: Power wall.



1) $\text{Processor}_A$ has 75% of the capacitive load of $\text{processor}_B$. $\text{Processor}_A$ also has a 20% voltage reduction and 10% shrink in frequency. What is the relative impact on dynamic power?
   - $0.75 \times 0.80^2 \times 0.90 = 0.432$
   - $0.75 \times 0.20^2 \times 0.10 = 0.003$
   - $0.75 \times 0.80^2 \times 0.90 = 0.432\ \text{V}$

2) Which improvement has a bigger impact on power?
   - 25% reduction in voltage
   - 25% reduction in frequency switching

3) In the past 20 years, voltages have decreased from 5 V to 1 V. Why don't manufacturers continue to lower voltages to reduce power consumption?
   - Further lowering of voltage results in transistor leakage.
   - Voltage has no impact on power

4) Over the past 30 years, processor frequencies have continued to increase.
   - True
   - False

# 1.8 The sea change: The switch from uniprocessors to multiprocessors

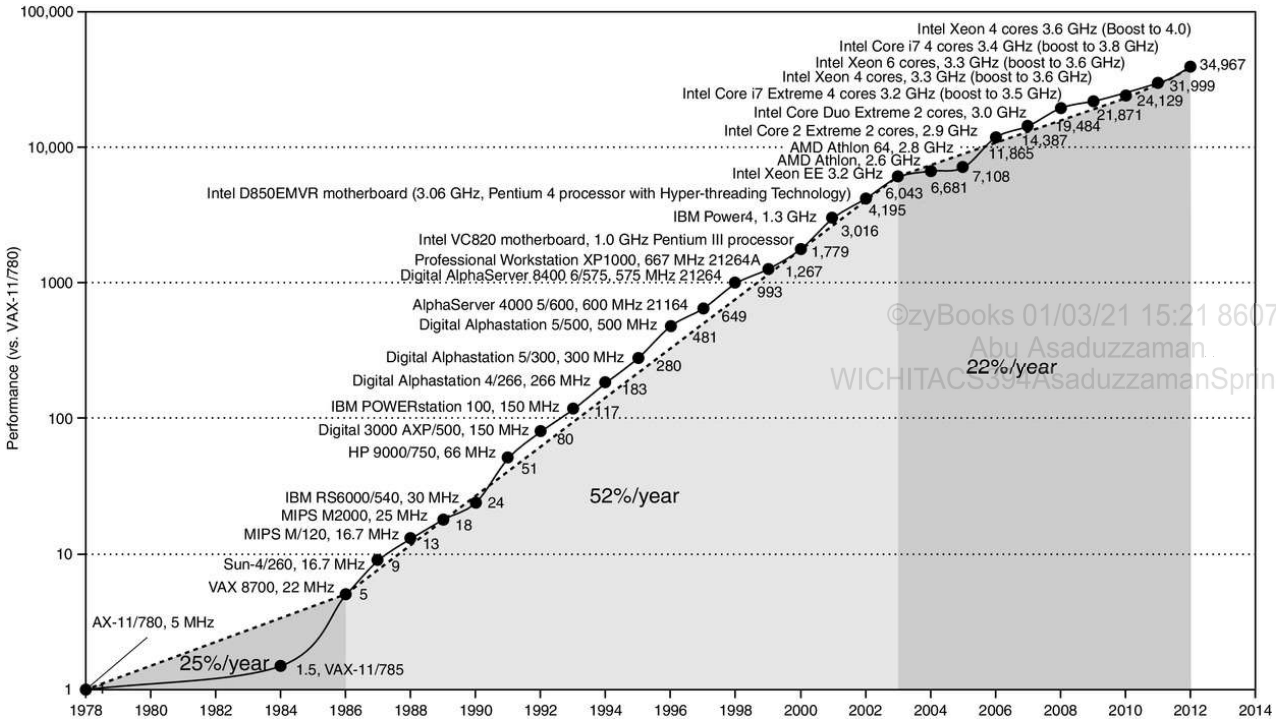
> Up to now, most software has been like music written for a solo performer; with the current generation of chips we're getting a little experience with duets and quartets and other small ensembles; but scoring a work for large orchestra and chorus is a different kind of challenge. *Brian Hayes, Computing in a Parallel Universe, 2007.*



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

## Figure 1.8.1: Growth in processor performance since the mid-1980s (COD Figure 1.17).

This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see COD Section 1.10 (Fallacies and pitfalls)). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. The higher annual performance improvement of 52% since the mid-1980s meant performance was about a factor of seven larger in 2002 than it would have been had it stayed at 25%. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 22% per year.






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

**PARTICIPATION ACTIVITY** 1.8.1: Slowing in uniprocessor performance has led to a switch to multiprocessor systems.

## Animation captions:

1. From the mid-1980s to 2000s, processor performance improved 52% annually. Performance growth was largely attributable to more advanced architectural and organizational ideas.
2. In 2005, the limits of power, available instruction-level parallelism, and long memory latency slowed uniprocessor performance. Multiprocessor systems began to appear.
3. Future systems may contain numerous processors.



**PARTICIPATION ACTIVITY** 1.8.2: Changes in processor design.

1) From the mid-1980s to early-2000s, processor performance improved each year at an average of 52%.

True

False

2) Growth in processor performance slowed in 2002.

True

False



3) Power was a factor in the slowing of processor performance growth.

True

False

4) Manufacturers continue to design single processor systems and increase processor performance through new technology-driven improvements.

True

False

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

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

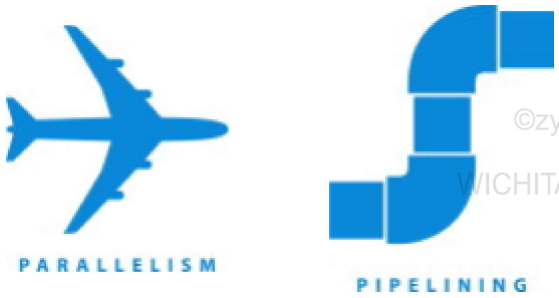
## Hardware/Software Interface



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

Forcing programmers to be aware of the parallel hardware and to rewrite their programs to be parallel had been the "third rail" of computer architecture, for

companies in the past that depended on such a change in behavior failed (see COD Section 6.15 (Historical perspective and further reading)). From this historical perspective, it's startling that the whole IT industry has bet its future that programmers will finally successfully switch to explicitly parallel programming.






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

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

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



**PARTICIPATION ACTIVITY** 1.8.3: Programs for multicore systems.

1) As computing systems move to multicore microprocessors, programmers _____ to obtain performance benefits.

- don't need to change any code
- need to rewrite their programs

2) Parallel programming seeks to improve program _____.

- pipelining
- performance
- correctness



3) How should programmers write code to maximize the benefits of parallel programming?

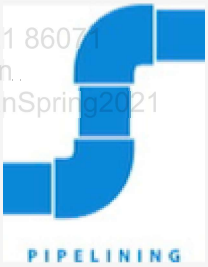
- Run all program tasks on a single processor
- Run programs in a round-robin fashion to ensure even wear of processors
- Divide a program into sub-tasks so all processors run about the same amount of time

4) Parallel programming becomes more difficult as the number of processor cores increases.

- True
- False

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


- *COD Chapter 2 (Instructions: Language of the Computer) Section 2.11: Parallelism and Instructions: Synchronization.* Usually independent parallel tasks need to coordinate at times, such as to say when they have completed their work. This chapter explains the instructions used by multicore processors to synchronize tasks.
- *COD Chapter 3 (Arithmetic for Computers), Section 3.6: Parallelism and Computer Arithmetic: Subword Parallelism.* Perhaps the simplest form of parallelism to build involves computing on elements in parallel, such as when multiplying two vectors. Subword parallelism takes advantage of the resources supplied by *Moore's Law* to provide wider arithmetic units that can operate on many operands simultaneously.









- *COD Chapter 4 (The Processor), Section 4.10: Parallelism via Instructions.* Given the difficulty of explicitly parallel programming, tremendous effort was invested in the 1990s in having the hardware and the compiler uncover implicit parallelism, initially via *pipelining*. This chapter describes some of these aggressive techniques, including fetching and executing multiple instructions concurrently and guessing on the outcomes of decisions, and executing instructions speculatively using *prediction*.
- *COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy), Section 5.10: Parallelism and Memory Hierarchies: Cache Coherence.* One way to lower the cost of communication is to have all processors use the same address space, so that any processor can read or write any data. Given that all processors today use caches to keep a temporary copy of the data in faster memory near the processor, it's easy to imagine that parallel programming would be even more difficult if the caches associated with each processor had inconsistent values of the shared data. This chapter describes the mechanisms that keep the data in all caches consistent.
- *COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy), Section 5.11: Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks.* This section describes how using many disks in conjunction can offer much higher throughput, which was the original inspiration of ***Redundant Arrays of Inexpensive Disks*** (***RAID***). The real popularity of RAID proved to be the much greater dependability offered by including a modest number of redundant disks. The section explains the differences in performance, cost, and dependability between the various RAID levels.



In addition to these sections, there is a full chapter on parallel processing. COD Chapter 6 (Parallel Processor from Client to Cloud) goes into more detail on the challenges of parallel programming; presents the two contrasting approaches to communication of shared addressing and explicit message passing; describes a restricted model of parallelism that is easier to program; discusses the difficulty of benchmarking parallel processors; introduces a new simple performance model for multicore microprocessors; and, finally, describes and evaluates four examples of multicore microprocessors using this model.

As mentioned above, COD Chapters 3 (Arithmetic for Computers) to 6 (Parallel Processor from Client to Cloud) use matrix vector multiply as a running example to show how each type of parallelism can significantly increase performance.

COD Appendix B (Graphics and Computing GPUs) describes an increasingly popular hardware component that is included with desktop computers. The ***graphics processing unit*** (***GPU***) is a hardware component that accelerates graphics. GPUs are becoming programming platforms in their own right. As you might expect, given these times, GPUs rely on *parallelism*.

COD Appendix B (Graphics and Computing GPUs) describes the NVIDIA GPU and highlights parts of its parallel programming environment.

# 1.9 Real stuff: Benchmarking the Intel Core i7

> "I thought [computers] would be a universally applicable idea, like a book is. But I didn't think it would develop as fast as it did, because I didn't envision we'd be able to get as many parts on a chip as we finally got. The transistor came along unexpectedly. It all happened much faster than we expected.
> *J. Presper Eckert, coinventor of ENIAC, speaking in 1991*



Each chapter has a section entitled "Real Stuff " that ties the concepts in the book with a computer you may use every day. These sections cover the technology underlying modern computers. For this first "Real Stuff" section, we look at how integrated circuits are manufactured and how performance and power are measured, with the Intel Core i7 as the example.

## SPEC CPU benchmark


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

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

***Benchmark***: A program selected for use in comparing computer performance.



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

The floating-point benchmarks include structured grid codes for finite element modeling, particle method codes for molecular dynamics, and sparse linear algebra codes for fluid dynamics.

The figure below describes the SPEC integer benchmarks and their execution time on the Intel Core i7 and shows the factors that explain execution time: instruction count, CPI, and clock cycle time. Note that CPI varies by more than a factor of 5.



## Table 1.9.1: SPECINTC2006 benchmarks running on a 2.66GHz Intel Core i7 920 (COD Figure 1.18).

Execution time is the product of the three factors in this table: instruction count in billions, clocks per instruction (CPI), and clock cycle time in nanoseconds. SPECratio is simply the reference time, which is supplied by SPEC, divided by the measured execution time. The single number quoted as SPECINTC2006 is the geometric mean of the SPECratios.

| Description | Name | Instruction Count x $10^9$ | CPI | Clock cycle time (seconds x $10^{-9}$) | Execution Time (seconds) | Reference Time (seconds) | SPECratio |
|---|---|---|---|---|---|---|---|
| Interpreted string processing | perl | 2252 | 0.60 | 0.376 | 508 | 9770 | 19.2 |
| Block-sorting compression | bzip2 | 2390 | 0.70 | 0.376 | 629 | 9650 | 15.4 |
| GNU C compiler | gcc | 794 | 1.20 | 0.376 | 358 | 8050 | 22.5 |
| Combinatorial optimization | mcf | 221 | 2.66 | 0.376 | 221 | 9120 | 41.2 |
| Go game (AI) | go | 1274 | 1.10 | 0.376 | 527 | 10490 | 19.9 |
| Search gene sequence | hmmer | 2616 | 0.60 | 0.376 | 590 | 9330 | 15.8 |
| Chess game (AI) | sjeng | 1948 | 0.80 | 0.376 | 586 | 12100 | 20.7 |
| Quantum computer | libquantum | 659 | 0.44 | 0.376 | 109 | 20720 | 190.0 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| simulation | | | | | | | |
| Video compression | h264avc | 3793 | 0.50 | 0.376 | 713 | 22130 | 31.0 |
| Discrete event simulation library | omnetpp | 367 | 2.10 | 0.376 | 290 | 6250 | 21.5 |
| Games/path finding | astar | 1250 | 1.00 | 0.376 | 470 | 7020 | 14.9 |
| XML parsing | xalancbmk | 1045 | 0.70 | 0.376 | 275 | 6900 | 25.1 |
| Geometric mean | - | - | - | - | - | - | 25.7 |



To simplify the marketing of computers, SPEC decided to report a single number summarizing all 12 integer benchmarks. Dividing the execution time of a reference processor by the execution time of the evaluated computer normalizes the execution time measurements; this normalization yields a measure, called the ***SPECratio***, which has the advantage that bigger numeric results indicate faster performance. That is, the SPECratio is the inverse of execution time. A CINT2006 or CFP2006 summary measurement is obtained by taking the geometric mean of the SPECratios.

## Elaboration

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

*The formula for the geometric mean is*

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



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

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

## SPEC power benchmark

Given the increasing importance of energy and power, SPEC added a benchmark to measure power. The SPEC power benchmark reports power consumption of servers at different workload levels, divided into 10% increments, over a period of time. The figure below shows the results for a server using Intel Nehalem processors similar to the above.



Table 1.9.2: SPECpower_ssj2008 running on a dual socket 2.66GHz Intel Xeon X5650 with 16GB of DRAM and one 100GB SSD disk (COD Figure 1.19).

| Target Load % | Performance (ssj_ops) | Average Power (watts) |
|---|---|---|
| 100% | 865,618 | 258 |
| 90% | 786,688 | 242 |
| 80% | 698,051 | 224 |
| 70% | 607,826 | 204 |
| 60% | 521,391 | 185 |
| 50% | 436,757 | 170 |
| 40% | 345,919 | 157 |
| 30% | 262,071 | 146 |
| 20% | 176,061 | 135 |
| 10% | 86,784 | 121 |
| 0% | 0 | 80 |
| Overall Sum | 4,787,166 | 1922 |
| Σssj_ops / Σpower = | | 2490 |

SPECpower started with another SPEC benchmark for Java business applications (SPECJBB2005), which exercises the processors, caches, and main memory as well as the Java virtual machine, compiler, garbage collector, and pieces of the operating system. Performance is measured in throughput, and the units are business operations per second. Once again, to simplify the marketing of computers, SPEC boils these numbers down to one number, called "overall ssj_ops per watt." The formula for this single summarizing metric is

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



where $\text{ssj\_ops}_i$ is performance at each 10% increment and $\text{power}_i$ is power consumed at each performance level.

**PARTICIPATION ACTIVITY** 1.9.1: SPEC benchmark.

1) "Go game (AI)" is a benchmark.
   - True
   - False

2) A workload is typically a set of benchmarks.
   - True
   - False

3) All SPEC benchmarks have similar instruction count, CPI, and clock cycle time characteristics.
   - True
   - False

4) A smaller SPECratio indicates better performance.
   - True
   - False



5) Power is not a useful evaluation metric in the SPEC benchmarks.
   - True
   - False

# 1.10 Fallacies and pitfalls

> Science must begin with myths, and the criticism of myths.
> *Sir Karl Popper, The Philosophy of Science, 1957*



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

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

The great idea of making the *common case fast* has a demoralizing corollary that has plagued designers of both hardware and software. It reminds us that the opportunity for improvement is affected by how much time the event consumes.


**PARTICIPATION ACTIVITY** 1.10.1: Improvement is affected by how much time a feature is used.

#### Animation captions:

1. A program runs in 100 seconds on a computer. Add operations require 60 seconds and all other instructions require 40 seconds.
2. Improving add operations by 2X results in a program execution time of 70 seconds, or a speedup of 1.4.
3. Improving add operations by 4X results in a program execution time of 55 seconds, or a speedup of 1.8.
4. Even if add operations could be reduced to run in 0 seconds, the program still requires 40 seconds to execute, which is a speedup of 2.5.
5. The performance enhancement possible with a given improvement is limited by the amount that the improved feature is used.



A simple design problem illustrates it well. Suppose a program runs in 100 seconds on a computer, with multiply operations responsible for 80 seconds of this time. How much do I have to improve the

speed of multiplication if I want my program to run five times faster?

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

**PARTICIPATION ACTIVITY** 1.10.2: Amdahl's Law.



**Animation captions:**

1. A program runs in 100 seconds on a computer, with multiply operations responsible for 80 seconds of time.
2. Use Amdahl's Law to calculate the improvement needed for multiplication operations to run the program five times faster.
3. Solve for n.
4. Multiply operations must run in 0 seconds to run the program 5 times faster.

That is, there is no amount by which we can enhance multiply to achieve a fivefold increase in performance, if multiply accounts for only 80% of the workload. The performance enhancement possible with a given improvement is limited by the amount that the improved feature is used. In everyday life this concept also yields what we call the law of diminishing returns.

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

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

Amdahl's Law is also used to argue for practical limits to the number of parallel processors. We examine this argument in the Fallacies and Pitfalls section of COD Chapter 6 (Parallel Processor from Client to Cloud).

**PARTICIPATION ACTIVITY** 1.10.3: Amdahl's Law.



1) A program runs in 100 seconds. Multiply operations are responsible for 30 of those seconds. If extensive designer effort is applied such that multiply operations are made to run 2 times faster, what is the program's new execution time?

sec

Check Show answer

2) If some aspect of a computer accounts for 50% of program execution time, what is the limit on how many times faster programs can run if engineers focus on improving that aspect?



times

Check Show answer

## Fallacy: Computers at low utilization use little power.

Power efficiency matters at low utilizations because server workloads vary. Utilization of servers in Google's warehouse scale computer, for example, is between 10% and 50% most of the time and at 100% less than 1% of the time. Even given 5 years to learn how to run the SPECpower benchmark well, the specially configured computer with the best results in 2012 still uses 33% of the peak power at 10% of the load. Systems in the field that are not configured for the SPECpower benchmark are surely worse.

Since servers' workloads vary but use a large fraction of peak power, Luiz Barroso and Urs Hölzle [2007] argue that we should redesign hardware to achieve "energy-proportional computing." If future servers used, say, 10% of peak power at 10% workload, we could reduce the electricity bill of datacenters and become good corporate citizens in an era of increasing concern about CO2 emissions.

## Fallacy: Designing for performance and designing for energy efficiency are unrelated goals.

Since energy is power over time, it is often the case that hardware or software optimizations that take less time save energy overall even if the optimization takes a bit more energy when it is used. One reason is that all the rest of the computer is consuming energy while the program is running, so even if the optimized portion uses a little more energy, the reduced time can save the energy of the whole system.



**PARTICIPATION ACTIVITY** 1.10.4: Fallacies.

1) Google's warehouse scale computer uses 5% of the peak power when

running at 10% utilization.

- ○ True
- ○ False

2) Designers must choose between energy and performance. If a computer is designed for improved performance, then the computer's energy consumption will increase.

- ○ True
- ○ False



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

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

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

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

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

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

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

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

The CPI varied by a factor of 5 for SPEC CPU2006 on an Intel Core i7 computer in COD Figure 1.18 (SPECINTC2006 benchmarks running on a 2.66GHz Intel Core i7 920), so MIPS does as well. Finally, and most importantly, if a new program executes more instructions but each instruction is faster, MIPS can vary independently from performance!



**PARTICIPATION ACTIVITY** 1.10.5: Check yourself: MIPS.

Consider the following performance measurements for a program:

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

1) Which computer has the higher MIPS rating?
   - Computer A
   - Computer B

2) Which computer is faster for that program?
   - Computer A
   - Computer B

# 1.11 Concluding remarks



> Where ... the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have 1,000 vacuum tubes and perhaps weigh just 1½ tons.
> *Popular Mechanics, March 1949*

Although it is difficult to predict exactly what level of cost/performance computers will have in the future, it's a safe bet that they will be much better than they are today. To participate in these

advances, computer designers and programmers must understand a wider variety of issues.


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



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

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

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

## The Big Picture

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




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

Energy efficiency has replaced die area as the most critical resource of microprocessor design. Conserving power while trying to increase performance has forced the hardware industry to switch to multicore microprocessors, thereby requiring the software industry to switch to programming parallel hardware. *Parallelism* is now required for performance.


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




**PARTICIPATION ACTIVITY** 1.11.1: Computer technologies.

1) The most reliable method to evaluate performance is execution time.
   - True
   - False

2) Software developers do not need an understanding of hardware to write efficient programs.
   - True
   - False

3) Die area is the most critical resource of microprocessor design.
   - True
   - False

4) Power limitations have forced computer designers to exploit parallelism to improve system performance.
   - True
   - False



## Road map for this book

At the bottom of these abstractions is the five classic components of a computer: datapath, control, memory, input, and output. These five components also serve as the framework for the rest of the chapters in this book:

- Datapath: COD Chapter 3 (Arithmetic for Computers), COD Chapter 4 (The Processor), COD Chapter 6 (Parallel Processor from Client to Cloud), and COD Appendix B (Graphics and Computing GPUs)
- Control: COD Chapter 4 (The Processor), COD Chapter 6 (Parallel Processor from Client to Cloud), and COD Appendix B (Graphics and Computing GPUs)
- Memory: COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)
- Input: COD Chapters 5 (Large and Fast: Exploiting Memory Hierarchy) and 6 (Parallel Processor from Client to Cloud)
- Output: COD Chapters 5 (Large and Fast: Exploiting Memory Hierarchy) and 6 (Parallel Processor from Client to Cloud)



As mentioned above, COD Chapter 4 (The Processor) describes how processors exploit implicit parallelism, COD Chapter 6 (Parallel Processor from Client to Cloud) describes the explicitly parallel multicore microprocessors that are at the heart of the parallel revolution, and COD Appendix B (Graphics and Computing GPUs) describes the highly parallel graphics processor chip. COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy) describes how a memory hierarchy exploits locality. COD Chapter 2 (Instructions: Language of the Computer) describes instruction sets—the interface between compilers and the computer—and emphasizes the role of compilers and programming languages in using the features of the instruction set. COD Chapter 3 (Arithmetic for Computers) describes how computers handle arithmetic data. COD Appendix A (The Basics of Logic Design) introduces logic design.

# 1.12 Historical perspective and reading

> An active field of science is like an immense anthill; the individual almost vanishes into the mass of minds tumbling over each other, carrying information from place to place, passing it around at the speed of light.
> *Lewis Thomas, "Natural Science," in The Lives of a Cell, 1974*

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



The historical perspective for this chapter provides a background for some of the key ideas presented therein. Its purpose is to give you the human story behind the technological advances and to place achievements in their historical context. By learning the past, you may be better able to understand the forces that will shape computing in the future. Each historical perspective section ends with suggestions for additional reading, which are also collected separately in the online section "Further Reading."

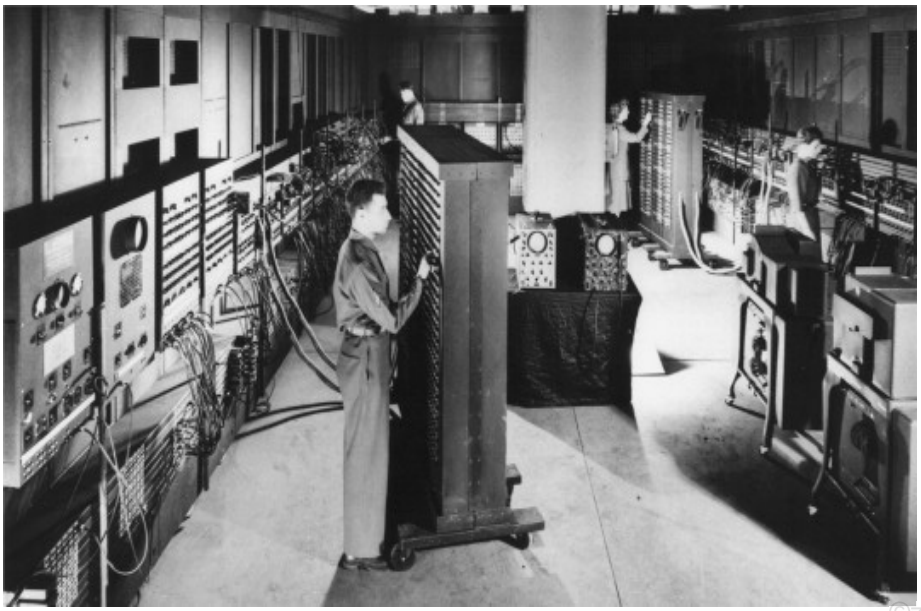
## The first electronic computers

***J. Presper Eckert*** and ***John Mauchly*** at the Moore School of the University of Pennsylvania built what is widely accepted to be the world's first operational electronic, general-purpose computer called the ***ENIAC*** (Electronic Numerical Integrator and Calculator). This machine was funded by the United States Army and started working during World War II but was not publicly disclosed until 1946. ENIAC was a general-purpose machine used for computing artillery-firing tables. The figure below shows the U-shaped computer, which was 80 feet long by 8.5 feet high and several feet wide. Each of the 20 10-digit registers was 2 feet long. In total, ENIAC used 18,000 vacuum tubes.



In size, ENIAC was two orders of magnitude bigger than machines built today, yet it was more than eight orders of magnitude slower, performing 1900 additions per second. ENIAC provided conditional jumps and was programmable, clearly distinguishing it from earlier calculators. Programming was done manually by plugging cables and setting switches, and data were entered on punched cards. Programming for typical calculations required from half an hour to a whole day. ENIAC was a general-purpose machine, limited primarily by a small amount of storage and tedious programming.

Figure 1.12.1: ENIAC, the world's first general-purpose electronic computer (COD Figure e1.12.1).





In 1944, John von Neumann was attracted to the ENIAC project. The group wanted to improve the way programs were entered and discussed storing programs as numbers; ***von Neumann*** helped crystallize the ideas and wrote a memo proposing a stored-program computer called ***EDVAC*** (Electronic Discrete Variable Automatic Computer). Herman Goldstine distributed the memo and put von Neumann's name on it, much to the dismay of Eckert and Mauchly, whose names were omitted. This memo has served as the basis for the commonly used term von Neumann computer. Several

early pioneers in the computer field believe that this term gives too much credit to von Neumann, who wrote up the ideas, and too little to the engineers, Eckert and Mauchly, who worked on the machines. For this reason, the term does not appear elsewhere in this book or in the online sections.

In 1946, Maurice Wilkes of Cambridge University visited the Moore School to attend the latter part of a series of lectures on developments in electronic computers. When he returned to Cambridge, Wilkes decided to embark on a project to build a stored-program computer named ***EDSAC*** (Electronic Delay Storage Automatic Calculator). EDSAC started working in 1949 and was the world's first full-scale, operational, stored-program computer [Wilkes, 1985]. (A small prototype called the ***Mark-I***, built at the University of Manchester in 1948, might be called the first operational stored-program machine. ) COD Section 2.5 (Representing instructions in the Computer) in COD Chapter 2 (Instructions: Language of the Computer) explains the stored-program concept.



In 1947, Eckert and Mauchly applied for a patent on electronic computers. The dean of the Moore School demanded that the patent be turned over to the university, which may have helped Eckert and Mauchly conclude that they should leave. Their departure crippled the EDVAC project, delaying completion until 1952.

Goldstine left to join von Neumann at the Institute for Advanced Study (IAS) at Princeton in 1946. Together with Arthur Burks, they issued a report based on the memo written earlier [Burks et al., 1946]. The paper was incredible for the period; reading it today, you would never guess this landmark paper was written more than 50 years ago, because it discusses most of the architectural concepts seen in modern computers. This paper led to the IAS machine built by Julian Bigelow. It had a total of 1024 40-bit words and was roughly 10 times faster than ENIAC. The group thought about uses for the machine, published a set of reports, and encouraged visitors. These reports and visitors inspired the development of a number of new computers.

Recently, there has been some controversy about the work of ***John Atanasoff***, who built a small-scale electronic computer in the early 1940s. His machine, designed at Iowa State University, was a special-purpose computer that was never completely operational. Mauchly briefly visited Atanasoff before he built ENIAC. The presence of the Atanasoff machine, together with delays in filing the ENIAC patents (the work was classified and patents could not be filed until after the war) and the distribution of von Neumann's EDVAC paper, was used to break the Eckert-Mauchly patent. Though controversy still rages over Atanasoff's role, Eckert and Mauchly are usually given credit for building the first working, general-purpose, electronic computer [Stern, 1980].

Another pioneering computer that deserves credit was a special-purpose machine built by ***Konrad Zuse*** in Germany in the late 1930s and early 1940s. Although Zuse had the design for a programmable computer ready, the German government decided not to fund scientific investigations taking more than two years because the bureaucrats expected the war would be won by that deadline.



Across the English Channel, during World War II special-purpose electronic computers were built to decrypt intercepted German messages. A team at Bletchley Park, including ***Alan Turing***, built the ***Colossus*** in 1943. The machines were kept secret until 1970; after the war, the group had little impact on commercial British computers.

While work on ENIAC went forward, ***Howard Aiken*** was building an electro-mechanical computer called the Mark-I at Harvard (a name that Manchester later adopted for its machine). He followed the

Mark-I with a relay machine, the Mark-II, and a pair of vacuum tube machines, the Mark-III and Mark-IV. In contrast to earlier machines like EDSAC, which used a single memory for instructions and data, the Mark-III and Mark-IV had separate memories for instructions and data. The machines were regarded as reactionary by the advocates of stored-program computers; the term ***Harvard architecture*** was coined to describe machines with distinct memories. Paying respect to history, this term is used today in a different sense to describe machines with a single main memory but with separate caches for instructions and data.



The ***Whirlwind project*** was begun at MIT in 1947 and was aimed at applications in real-time radar signal processing. Although it led to several inventions, its most important innovation was magnetic core memory. Whirlwind had 2048 16-bit words of magnetic core. Magnetic cores served as the main memory technology for nearly 30 years.

**PARTICIPATION ACTIVITY** 1.12.1: First electronic computers.

1) The _____ is the abbreviation for what is considered the world's first operational electronic, general-purpose computer.

Check Show answer

2) The ENIAC was developed in the 1940s at the University of _____.

Check Show answer

3) A team including Alan Turing built Colossus to decrypt _____ messages during World War II.

Check Show answer



4) The Mark-III and Mark-IV computers developed at Harvard used separate memories for instructions and data. A computer using similar separate memories is known as a _____ architecture.

Check Show answer

## Commercial developments



In December 1947, Eckert and Mauchly formed Eckert-Mauchly Computer Corporation. Their first machine, the ***BINAC***, was built for Northrop and was shown in August 1949. After some financial difficulties, their firm was acquired by Remington-Rand, where they built the UNIVAC I (Universal Automatic Computer), designed to be sold as a general-purpose computer (shown below). Originally delivered in June 1951, ***UNIVAC I*** sold for about $1 million and was the first successful commercial computer—48 systems were built! This early machine, along with many other fascinating pieces of computer lore, may be seen at the Computer History Museum in Mountain View, California.

Figure 1.12.2: UNIVAC I, the first commercial computer in the United States (COD Figure e1.12.2).

UNIVAC correctly predicted the outcome of the 1952 presidential election, but its initial forecast was withheld from broadcast because experts doubted the use of such early results.




IBM had been in the punched card and office automation business but didn't start building computers until 1950. The first IBM computer, the ***IBM 701***, shipped in 1952, and eventually 19 units were sold. In

the early 1950s, many people were pessimistic about the future of computers, believing that the market and opportunities for these "highly specialized" machines were quite limited.

In 1964, after investing $5 billion, IBM made a bold move with the announcement of the System/360. An IBM spokesman said the following at the time:

*We are not at all humble in this announcement. This is the most important product announcement that this corporation has ever made in its history. It's not a computer in any previous sense. It's not a product, but a line of products … that spans in performance from the very low part of the computer line to the very high.*



Figure 1.12.3: IBM System/360 computers: models 40, 50, 65, and 75 were all introduced in 1964 (COD Figure e1.12.3).

The ***IBM System/360*** is a family of computers developed to address a wide range of computing needs. These four models varied in cost and performance by a factor of almost 10; it grows to 25 if we include models 20 and 30 (not shown). The clock rate, range of memory sizes, and approximate price for only the processor and memory of average size: (a) model 40, 1.6 MHz, 32 KB-256 KB, $225,000; (b) model 50, 2.0MHz, 128KB-256KB, $550,000; (c) model 65, 5.0MHz, 256KB-1MB, $1,200,000; and (d) model 75, 5.1MHz, 256KB-1MB, $1,900,000. Adding I/O devices typically increased the price by factors of 1.8 to 3.5, with higher factors for cheaper models.

a. c.

b. d.




Moving the idea of the architecture *abstraction* into commercial reality, IBM announced six implementations of the System/360 architecture that varied in price and performance by a factor of 25. The figure above shows four of these models. IBM bet its company on the success of a *computer family*, and IBM won. The System/360 and its successors dominated the large computer market.

About a year later, ***Digital Equipment Corporation*** (***DEC***) unveiled the ***PDP-8***, the first commercial minicomputer. The ***minicomputer*** was a small machine that was a breakthrough in low-cost design, allowing DEC to offer a computer for under $20,000. Minicomputers were the forerunners of microprocessors, with Intel inventing the first microprocessor in 1971—the ***Intel 4004***.

In 1963 came the announcement of the first ***supercomputer***, an extremely fast computer targeted to perform a large number of computations typically needed by scientific applications. This announcement came neither from the large companies nor even from the high-tech centers. Seymour Cray led the design of the Control Data Corporation CDC 6600 in Minnesota. ***Seymour Cray*** is often credited as the "father of supercomputing" and regarded as a pioneer of supercomputing. This machine included many ideas that are beginning to be found in the latest microprocessors. Cray later left CDC to form Cray Research, Inc., in Wisconsin. In 1976, he announced the Cray-1 (figure below). The ***Cray-1*** was simultaneously the fastest in the world, the most expensive, and the computer with the best cost/performance for scientific programs.

Figure 1.12.4: Cray-1, the first commercial vector supercomputer, announced in 1976 (COD Figure e1.12.4).

This machine had the unusual distinction of being both the fastest computer for scientific applications and the computer with the best price/performance for those applications. Viewed from the top, the computer looks like the letter C. Seymour Cray passed away in 1996 because of injuries sustained in an automobile accident. At the time of his death, this 70-year-old computer pioneer was working on his vision of the next generation of supercomputers. (See www.cray.com for more details.)



**PARTICIPATION ACTIVITY** 1.12.2: Commercial development of computers.

PDP-8 | IBM System/360 | Cray-1 | Intel 4004

| | |
|---|---|
| | Product line of computers with varying cost and performance. |
| | Minicomputer |
| | Supercomputer |
| | Microprocessor |

Reset



While Seymour Cray was creating the world's most expensive computer, other designers around the world were looking at using the microprocessor to create a computer so cheap that you could have it at home. There is no single fountainhead for the *personal computer*, but in 1977, the ***Apple IIe*** (figure below) from ***Steve Jobs*** and ***Steve Wozniak*** set standards for low cost, high volume, and high reliability that defined the personal computer industry.

## Figure 1.12.5: The Apple IIe Plus (COD Figure e1.12.5).

Designed by Steve Wozniak, the Apple IIe set standards of cost and reliability for the industry.


However, even with a 4-year head start, Apple's personal computers finished second in popularity. The IBM Personal Computer, announced in 1981, became the best-selling computer of any kind; its success gave Intel the most popular microprocessor and Microsoft the most popular operating system. Of course, over the more than 30 years that the IBM-compatible personal computer has existed, it has evolved greatly. In fact, the first personal computers had 16-bit processors and 64 kilobytes of *memory*, and a low-density, slow floppy disk was the only nonvolatile storage! Floppy disks were originally developed by IBM for loading diagnostic programs in mainframes, but were a major I/O device in personal computers for almost 20 years before the advent of CDs and networking made them obsolete as a method for exchanging data.



Of course, Intel microprocessors have also evolved since the first PC, which used a 16-bit processor with an 8-bit external interface! In COD Chapter 2 (Instructions: Language of the Computer), we write about the evolution of the Intel architecture.

## Figure 1.12.6: The Xerox Alto was the primary inspiration for the modern desktop computer (COD Figure e1.12.6).

It included a mouse, a bit-mapped scheme, a Windows-based user interface, and a local network connection.

The first personal computers were quite simple, with little or no graphics capability, no pointing devices, and primitive operating systems compared to those of today. The computer that inspired many of the architectural and software concepts that characterize the modern desktop machines was the ***Xerox Alto***, shown in the figure above. The Alto was created as an experimental prototype of a future computer; there were several hundred Altos built, including a significant number that were donated to universities. Among the technologies incorporated in the Alto were:



- A bit-mapped graphics display integrated with a computer (earlier graphics displays acted as terminals, usually connected to larger computers)
- A mouse, which was invented earlier, but included on every Alto and used extensively in the user interface
- A local area network (LAN), which became the precursor to the Ethernet
- A user interface based on Windows and featuring a WYSIWYG (what you see is what you get) editor and interactive drawing programs

In addition, both file servers and print servers were developed and interfaced via the local area network, and connections between the local area network and the wide area ***ARPAnet*** produced the first versions of Internet-style networking. The Xerox Alto was incredibly influential and clearly affected the design of a wide variety of computers and software systems, including the Apple Macintosh, the IBM-compatible PC, MacOS and Windows, and Sun and other early workstations.

**PARTICIPATION ACTIVITY** 1.12.3: Personal computers.

1) In the 1980's, the _____ was the most popular personal computer.
   - Apple IIe
   - Microsoft
   - IBM Personal Computer

2) The first personal computers did not come with a pointing device, such as a mouse.
   - True
   - False



3) _____ were the only nonvolatile storage in the first personal computers.
   - Floppy drives

- Flash drives
- CD-ROMs

## Measuring performance

From the earliest days of computing, designers have specified performance goals—ENIAC was to be 1000 times faster than the Harvard Mark-I, and the IBM Stretch (7030) was to be 100 times faster than the fastest computer then in existence. What wasn't clear, though, was how this performance was to be measured.



The original measure of performance was the time required to perform an individual operation, such as addition. Since most instructions took the same execution time, the timing of one was the same as the others. As the execution times of instructions in a computer became more diverse, however, the time required for one operation was no longer useful for comparisons.

To consider these differences, an instruction mix was calculated by measuring the relative frequency of instructions in a computer across many programs. Multiplying the time for each instruction by its weight in the mix gave the user the *average instruction execution time*. (If measured in clock cycles, average instruction execution time is the same as average CPI.) Since instruction sets were similar, this was a more precise comparison than add times. From average instruction execution time, then, it was only a small step to MIPS. MIPS had the virtue of being easy to understand; hence, it grew in popularity.

## The quest for an average program

As processors were becoming more sophisticated and relied on memory hierarchies (the topic of COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)) and pipelining (the topic of COD Chapter 4 (The Processor)), a single execution time for each instruction no longer existed; neither execution time nor MIPS, therefore, could be calculated from the instruction mix and the manual.

Although it might seem obvious today that the right thing to do would have been to develop a set of real applications that could be used as standard benchmarks, this was a difficult task until relatively recent times. Variations in operating systems and language standards made it hard to create large programs that could be moved from computer to computer simply by recompiling.

Instead, the next step was benchmarking using synthetic programs. The ***Whetstone*** synthetic program was created by measuring scientific programs written in Algol-60 (see Curnow and Wichmann's [1976] description). This program was converted to Fortran and was widely used to characterize scientific program performance. Whetstone performance is typically quoted in Whetstones per second—the number of executions of a single iteration of the Whetstone benchmark! ***Dhrystone*** is another synthetic benchmark that is still used in some embedded computing circles (see Weicker's [1984] description and methodology).



About the same time Whetstone was developed, the concept of *kernel benchmarks* gained popularity. ***Kernels*** are small, time-intensive pieces from real programs that are extracted and then used as benchmarks. This approach was developed primarily for benchmarking high-end computers,

especially supercomputers. ***Livermore Loops*** and ***Linpack*** are the best-known examples of kernel benchmarks. The Livermore Loops consist of a series of 21 small loop fragments. Linpack consists of a portion of a linear algebra subroutine package. Kernels are best used to isolate the performance of individual features of a computer and to explain the reasons for differences in the performance of real programs. Because scientific applications often use small pieces of code that execute for a long time, characterizing performance with kernels is most popular in this application class. Although kernels help illuminate performance, they frequently overstate the performance on real applications.



## SPECulating about performance

An important advance in performance evaluation was the formation of the System Performance Evaluation Cooperative (SPEC) group in 1988. SPEC comprises representatives of many computer companies—the founders being Apollo/Hewlett-Packard, DEC, MIPS, and Sun—who have agreed on a set of real programs and inputs that all will run. It is worth noting that SPEC couldn't have come into being before portable operating systems and the popularity of high-level languages. Now compilers, too, are accepted as a proper part of the performance of computer systems and must be measured in any evaluation.

History teaches us that while the SPEC effort may be useful with current computers, it will not meet the needs of the next generation without changing. In 1991, a throughput measure was added, based on running multiple versions of the benchmark. It is most useful for evaluating timeshared usage of a uniprocessor or a multiprocessor. Other system benchmarks that include OS-intensive and I/O-intensive activities have also been added. Another change was the decision to drop some benchmarks and add others. One result of the difficulty in finding benchmarks was that the initial version of the SPEC benchmarks (called SPEC89) contained six floating-point benchmarks but only four integer benchmarks. Calculating a single summary measurement using the geometric mean of execution times normalized to a VAX-11/780 meant that this measure favored computers with strong floating-point performance.

In 1992, a new benchmark set (called SPEC92) was introduced. It incorporated additional benchmarks, dropped matrix300, and provided separate means (SPEC INT and SPECFP) for integer and floating-point programs. In addition, the SPECbase measure, which disallows program-specific optimization flags, was added to provide users with a performance measurement that would more closely match what they might experience on their own programs. The SPECFP numbers show the largest increase versus the base SPECFP measurement, typically ranging from 15% to 30% higher.

In 1995, the benchmark set was once again updated, adding some new integer and floating-point benchmarks, as well as removing some benchmarks that suffered from flaws or had running times that had become too small given the factor of 20 or more performance improvement since the first SPEC release. SPEC95 also changed the base computer for normalization to a Sun SPARC Station 10/40, since operating versions of the original base computer were becoming difficult to find!



The most recent version of SPEC is SPEC2006. What is perhaps most surprising is that all floating-point programs in SPEC2006 are new, and for integer programs just two are from SPEC2000, one from SPEC95, none from SPEC92, and one from SPEC89. The sole survivor from SPEC89 is the gcc compiler.

SPEC has also added benchmark suites beyond the original suites targeted at CPU performance. In 2008, ***SPEC*** provided benchmark sets for graphics, high-performance scientific computing, object-oriented computing, file systems, Web servers and clients, Java, engineering CAD applications, and power.

**PARTICIPATION ACTIVITY** 1.12.4: Evaluating performance.



1) Comparing the time required to perform an add instruction is a reliable method to evaluate performance between computers.
   - ○ True
   - ○ False

2) Computer A is faster than Computer B.

   Benchmark results:
   Computer A: 10 Whetstones per second
   Computer B: 12 Whetstones per second.
   - ○ True
   - ○ False

3) The SPEC benchmarks change as computers evolve.
   - ○ True
   - ○ False

## The growth of embedded computing

Embedded processors have been around for a very long time; in fact, the first minicomputers and the first microprocessors were originally developed for controlling functions in a laboratory or industrial application. For many years, the dominant use of embedded processors was for industrial control applications, and although this use continued to grow, the processors tended to be very cheap and the performance relatively low. For example, the best-selling processor in the world remains an 8-bit micro controller used in cars, some home appliances, and other simple applications.

The late 1980s and early 1990s saw the emergence of new opportunities for embedded processors, ranging from more advanced video games and set-top boxes to cell phones and personal digital assistants. The rapidly increasing number of information appliances and the growth of networking have driven dramatic surges in the number of embedded processors, as well as the performance

requirements. To evaluate performance, the embedded community was inspired by SPEC to create the ***Embedded Microprocessor Benchmark Consortium*** (***EEMBC***). Started in 1997, it consists of a collection of kernels organized into suites that address different portions of the embedded industry. They announced the second generation of these benchmarks in 2007.

## A half-century of progress



Since 1951, there have been thousands of new computers using a wide range of technologies and having widely varying capabilities. The table below summarizes the key characteristics of some machines mentioned in this section and shows the dramatic changes that have occurred in just over *50 years*. After adjusting for inflation, price/performance has improved by almost 100 billion in 55 years, or about 58% per year. Another way to say it is we've seen a factor of 10,000 improvement in cost and a factor of 10,000,000 improvement in performance.


Table 1.12.1: Characteristics of key commercial computers since 1950, in actual dollars and in 2007 dollars adjusted for inflation (COD Figure e1.12.7).

The last row assumes we can fully utilize the potential performance of the four cores in Barcelona. In above, here the price of the IBM S/360 model 50 includes I/O devices.

| Year | Name | Size (cu. ft.) | Power (watts) | Performance (adds/sec) | Memory (KB) | Price | Price/ performance vs. UNIVAC |
|---|---|---|---|---|---|---|---|
| 1951 | UNIVAC I | 1,000 | 125,000 | 2,000 | 48 | $1,000,000 | |
| 1964 | IBM S/360 model 50 | 60 | 10,000 | 500,000 | 64 | $1,000,000 | 26 |
| 1965 | PDP-8 | 8 | 500 | 330,000 | 4 | $16,000 | 10,85 |
| 1976 | Cray-1 | 58 | 60,000 | 166,000,000 | 32,000 | $4,000,000 | 21,84 |
| 1981 | IBM PC | 1 | 150 | 240,000 | 256 | $3,000 | 42,10 |
| 1991 | HP 9000/model 750 | 2 | 500 | 50,000,000 | 16,384 | $7,400 | 3,556,18 |
| 1996 | Intel PPro | 2 | 500 | 400,000,000 | 16,384 | $4,400 | 47,846,89 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | PC (200 MHz) | | | | | | |
| 2003 | Intel Pentium 4 PC (3.0 GHz) | 2 | 500 | 6,000,000,000 | 262,144 | $1,600 | 1,875,000,00 |
| 2007 | AMD Barcelona PC (2.5 GHz) | 2 | 250 | 20,000,000,000 | 2,097,152 | $800 | 12,500,000,00 |



(Source: The Computer History Museum and Producer Price Index for Industrial Commodities.)

**PARTICIPATION ACTIVITY** 1.12.5: Commercial computer trends.

Refer to the table above.

1) Computers first became affordable to the general public around 1970.
   - True
   - False

2) The 1996 Intel PPro PC was able to perform more add instructions per second than the first Cray supercomputer of 1976.
   - True
   - False

Readers interested in computer history should consult *Annals of the History of Computing*, a journal devoted to the history of computing. Several books describing the early days of computing have also appeared, many written by the pioneers including Goldstine [1972]; Metropolis et al., [1980]; and Wilkes [1985].



## Further reading

Barroso, L. and U. Hölzle [2007]. "The case for energy-proportional computing," IEEE Computer, December.
*A plea to change the nature of computer components so that they use much less power when lightly utilized.*

Bell, C. G. [1996]. Computer Pioneers and Pioneer Computers, ACM and the Computer Museum, videotapes.
*Two videotapes on the history of computing, produced by Gordon and Gwen Bell, including the following machines and*