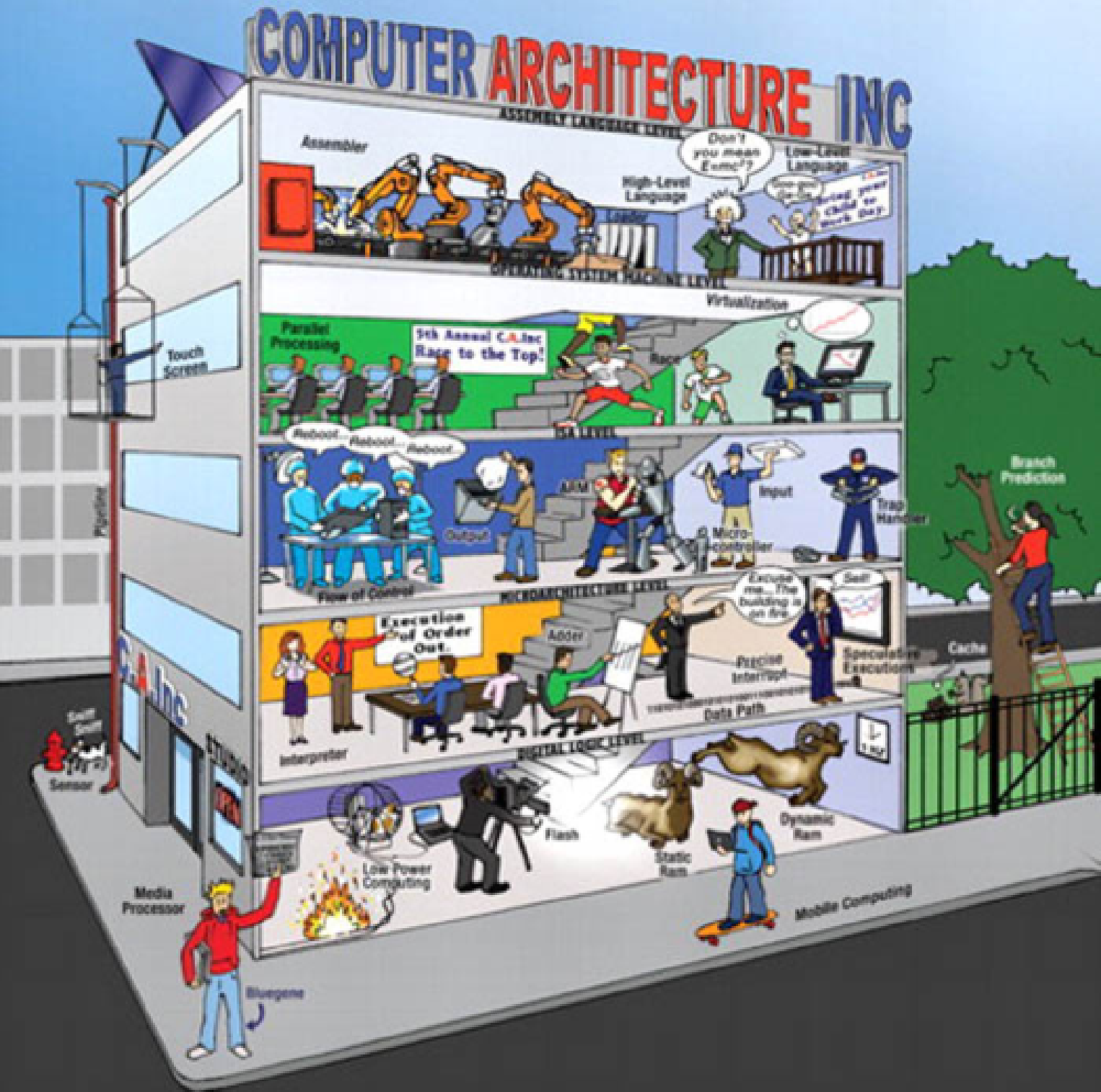
Structured Computer Organization

SIXTH EDITION

Andrew S. Tanenbaum • Todd Austin



STRUCTURED COMPUTER ORGANIZATION

SIXTH EDITION

This page intentionally left blank

STRUCTURED COMPUTER ORGANIZATION

SIXTH EDITION

ANDREW S. TANENBAUM
Vrije Universiteit
Amsterdam, The Netherlands

TODD AUSTIN
University of Michigan
Ann Arbor, Michigan, United States

PEARSON

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Editorial Director, ECS: Marcia Horton
Executive Editor: Tracy Johnson (Dunkelberger)
Associate Editor: Carole Snyder
Director of Marketing: Christy Lesko
Marketing Manager: Yez Alayan
Senior Marketing Coordinator: Kathryn Ferranti
Director of Production: Erin Gregg
Managing Editor: Jeff Holcomb
Associate Managing Editor: Robert Engelhardt
Manufacturing Buyer: Lisa McDowell
Art Director: Anthony Gemmellaro
Cover Illustrator: Jason Consalvo
Manager, Rights and Permissions: Michael Joyce
Media Editor: Daniel Sandin
Media Project Manager: Renata Butera
Printer/Binder: Courier/Westford
Cover Printer: Lehigh-Phoenix Color/Hagerstown

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear in the Credits section in the end matter of this text.

Library of Congress Cataloging-in-Publication Data
Tanenbaum, Andrew S.,
Structured computer organization / Andrew S. Tanenbaum, Todd Austin. -- 6th ed.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-13-291652-3
ISBN-10: 0-13-291652-5
1. Computer programming. 2. Computer organization. I. Austin, Todd. II. Title.

QA76.6.T38 2013
005.1--dc23

2012021627

10 9 8 7 6 5 4 3 2 1

PEARSON

ISBN 10: 0-13-291652-5
ISBN 13: 978-0-13-291652-3

AST: Suzanne, Barbara, Marvin, Aron and Nathan

TA: To Roberta, who made space (and time) for me to finish this project.

This page intentionally left blank

CONTENTS

This page intentionally left blank

PREFACE

The first five editions of this book were based on the idea that a computer can be regarded as a hierarchy of levels, each one performing some well-defined function. This fundamental concept is as valid today as it was when the first edition came out, so it has been retained as the basis for the sixth edition. As in the first five editions, the digital logic level, the microarchitecture level, the instruction set architecture level, the operating-system machine level, and the assembly language level are all discussed in detail.

Although the basic structure has been maintained, this sixth edition does contain many changes, both small and large, that bring it up to date in the rapidly changing computer industry. For example, the example machines used have been brought up to date. The current examples are the Intel Core i7, the Texas Instrument OMAP4430, and the Atmel ATmega168. The Core i7 is an example of a popular CPU used on laptops, desktops, and server machines. The OMAP4430 is an example of a popular ARM-based CPU, widely used in smartphones and tablets.

Although you have probably never heard of the ATmega168 microcontroller, you have probably interacted with one many times. The AVR-based ATmega168 microcontroller is found in many embedded systems, ranging from clock radios to microwave ovens. The interest in embedded systems is surging, and the ATmega168 is widely used due to its extremely low cost (pennies), the wealth of software and peripherals for it, and the large number of programmers available. The number of ATmega168s in the world certainly exceeds the number of Pentium and Core i3, i5, and i7 CPUs by orders of magnitude. The ATmega168s is also the processor found in the Arduino single-board embedded computer, a popular

hobbyist system designed at an Italian university to cost less than dinner at a pizza restaurant.

Over the years, many professors teaching from the course have repeatedly asked for material on assembly language programming. With the sixth edition, that material is now available on the book's Website (see below), where it can be easily expanded and kept evergreen. The assembly language chosen is the 8088 since it is a stripped-down version of the enormously popular iA32 instruction set used in the Core i7 processor. We could have used the ARM or AVR instruction set or some other ISA almost no one has ever heard of, but as a motivational tool, the 8088 is a better choice since large numbers of students have an 8088-compatible CPU at home. The full Core i7 is far too complex for students to understand in detail. The 8088 is similar but much simpler.

In addition, the Core i7, which is covered in great detail in this edition of the book, is capable of running 8088 programs. However, since debugging assembly code is very difficult, we have provided a set of tools for learning assembly language programming, including an 8088 assembler, a simulator, and a tracer. These tools are provided for Windows, UNIX, and Linux. The tools are on the book's Website.

The book has become longer over the years (the first edition was 443 pages; this one is 769 pages). Such an expansion is inevitable as a subject develops and there is more known about it. As a result, when the book is used for a course, it may not be possible to finish it in a single course (e.g., in a trimester system). A possible approach would be to do all of Chaps. 1, 2, and 3, the first part of Chap. 4 (up through and including Sec. 4.4), and Chap. 5 as a bare minimum. The remaining time could be filled with the rest of Chap. 4, and parts of Chaps. 6, 7, and 8, depending on the interests of the instructor and students.

A chapter-by-chapter rundown of the major changes since the fifth edition follows. Chapter 1 still contains an historical overview of computer architecture, pointing out how we got where we are now and what the milestones were along the way. Many students will be amazed to learn that the most powerful computers in the world in the 1960s, which cost millions of U.S. dollars, had far less than 1 percent of the computing power in their smartphones. Today's enlarged spectrum of computers that exist is discussed, including FPGAs, smartphones, tablets, and game consoles. Our three new example architectures (Core i7, OMAP4430, and ATmega168) are introduced.

In Chapter 2, the material on processing styles has expanded to include data-parallel processors including graphics processing units (GPUs). The storage landscape has been expanded to include the increasingly popular flash-based storage devices. New material has been added to the input/output section that details modern game controllers, including the Wiimote and the Kinect as well as the touch screens used on smartphones and tablets.

Chapter 3 has undergone revision in various places. It still starts at the beginning, with how transistors work, and builds up from there so that even students

with no hardware background at all will be able to understand in principle how a modern computer works. We provide new material on field-programmable gate arrays (FPGAs), programmable hardware fabrics that bring true large-scale gate-level design costs down to where they are widely used in the classroom today. The three new example architectures are described here at a high level.

Chapter 4 has always been popular for explaining how a computer really works, so most of it is unchanged since the fifth edition. However, there are new sections discussing the microarchitecture level of the Core i7, the OMAP4430, and the ATmega168.

Chapters 5 and 6 have been updated using the new example architectures, in particular with new sections describing the ARM and AVR instruction sets. Chapter 6 uses Windows 7 rather than Windows XP as an example.

Chapter 7, on assembly language programming, is largely unchanged from the fifth edition.

Chapter 8 has undergone many revisions to reflect new developments in the parallel computing arena. New details on the Core i7 multiprocessor architecture are included, and the NVIDIA Fermi general-purpose GPU architecture is described in detail. Finally, the BlueGene and Red Storm supercomputer sections have been updated to reflect recent upgrades to these enormous machines.

Chapter 9 has changed. The suggested readings have been moved to the Website, so the new Chap. 9 contains only the references cited in the book, many of which are new. Computer organization is a dynamic field.

Appendices A and B are unchanged since last time. Binary numbers and floating-point numbers haven't changed much in the past few years. Appendix C, about assembly language programming, was written by Dr. Evert Wattel of the Vrije Universiteit, Amsterdam. Dr. Wattel has had many years of experience teaching students using these tools. Our thanks to him for writing this appendix. It is largely unchanged since the fifth edition, but the tools are now on the Website rather than on a CD-ROM included with the book.

In addition to the assembly language tools, the Website also contains a graphical simulator to be used in conjunction with Chap. 4. This simulator was written by Prof. Richard Salter of Oberlin College. Students can use it to help grasp the principles discussed in this chapter. Our thanks to him for providing this software.

The Website, with the tools and so on, is located at

http://www.pearsonhighered.com/tanenbaum

From there, click on the Companion Website for this book and select the page you are looking. The student resources include:

* the assembler/tracer software
* the graphical simulator
* the suggested readings

The instructor resources include:

* PowerPoint sheets for the course
* solutions to the end-of-chapter exercises

The instructor resources require a password. Instructors should contact their Pearson Education representative to obtain one.

A number of people have read (parts of) the manuscript and provided useful suggestions or have been helpful in other ways. In particular, we would like to thank Anna Austin, Mark Austin, Livio Bertacco, Valeria Bertacco, Debapriya Chatterjee, Jason Clemons, Andrew DeOrio, Joseph Greathouse, and Andrea Pellegrini.

The following people reviewed the manuscript and suggested changes: Jason D. Bakos (University of South Carolina), Bob Brown (Southern Polytechnic State University), Andrew Chen (Minnesota State University, Moorhead), J. Archer Harris (James Madison University), Susan Krucke (James Madison University), A. Yavuz Oruc (University of Maryland), Frances Marsh (Jamestown Community College), and Kris Schindler (University at Buffalo). Our thanks to them.

Several people helped create new exercises. They are: Byron A. Jeff (Clayton University), Laura W. McFall (DePaul University), Taghi M. Mostafavi (University of North Carolina at Charlotte), and James Nystrom (Ferris State University). Again, we greatly appreciate the help.

Our editor, Tracy Johnson, has been ever helpful in many ways, large and small, as well as being very patient with us. The assistance of Carole Snyder in coordinating the various people involved in the project was much appreciated. Bob Englehardt did a great job with production.

I (AST) would like to thank Suzanne once more for her love and patience. It never ends, not even after 21 books. Barbara and Marvin are always a joy and now know what professors do for a living. Aron belongs to the next generation: kids who are heavy computer users before they hit nursery school. Nathan hasn't gotten that far yet, but after he figures out how to walk, the iPad is next.

Finally, I (TA) want to take this opportunity to thank my mother-in-law Roberta, who helped me carve out some quality time to work on this book. Her dining room table in Bassano Del Grappa, Italy had just the right amount of solitude, shelter, and vino to get this important task done.

ANDREW S. TANENBAUM
TODD AUSTIN

STRUCTURED COMPUTER ORGANIZATION

This page intentionally left blank

1

INTRODUCTION

A digital computer is a machine that can do work for people by carrying out instructions given to it. A sequence of instructions describing how to perform a certain task is called a **program**. The electronic circuits of each computer can recognize and directly execute a limited set of simple instructions into which all its programs must be converted before they can be executed. These basic instructions are rarely much more complicated than

Add two numbers.

Check a number to see if it is zero.

Copy a piece of data from one part of the computer's memory to another.

Together, a computer's primitive instructions form a language in which people can communicate with the computer. Such a language is called a **machine language**. The people designing a new computer must decide what instructions to include in its machine language. Usually, they try to make the primitive instructions as simple as possible consistent with the computer's intended use and performance requirements, in order to reduce the complexity and cost of the electronics needed. Because most machine languages are so simple, it is difficult and tedious for people to use them.

This simple observation has, over the course of time, led to a way of structuring computers as a sequence of abstractions, each abstraction building on the one

below it. In this way, the complexity can be mastered and computer systems can be designed in a systematic, organized way. We call this approach **structured computer organization** and have named the book after it. In the next section we will describe what we mean by this term. After that we will look at some historical developments, the state of the art, and some important examples.

1.1 STRUCTURED COMPUTER ORGANIZATION

As mentioned above, there is a large gap between what is convenient for people and what is convenient for computers. People want to do *X*, but computers can only do *Y*. This leads to a problem. The goal of this book is to explain how this problem can be solved.

1.1.1 Languages, Levels, and Virtual Machines

The problem can be attacked in two ways: both involve designing a new set of instructions that is more convenient for people to use than the set of built-in machine instructions. Taken together, these new instructions also form a language, which we will call L1, just as the built-in machine instructions form a language, which we will call L0. The two approaches differ in the way programs written in L1 are executed by the computer, which, after all, can only execute programs written in its machine language, L0.

One method of executing a program written in L1 is first to replace each instruction in it by an equivalent sequence of instructions in L0. The resulting program consists entirely of L0 instructions. The computer then executes the new L0 program instead of the old L1 program. This technique is called **translation**.

The other technique is to write a program in L0 that takes programs in L1 as input data and carries them out by examining each instruction in turn and executing the equivalent sequence of L0 instructions directly. This technique does not require first generating a new program in L0. It is called **interpretation** and the program that carries it out is called an **interpreter**.

Translation and interpretation are similar. In both methods, the computer carries out instructions in L1 by executing equivalent sequences of instructions in L0. The difference is that, in translation, the entire L1 program is first converted to an L0 program, the L1 program is thrown away, and then the new L0 program is loaded into the computer's memory and executed. During execution, the newly generated L0 program is running and in control of the computer.

In interpretation, after each L1 instruction is examined and decoded, it is carried out immediately. No translated program is generated. Here, the interpreter is in control of the computer. To it, the L1 program is just data. Both methods, and increasingly, a combination of the two, are widely used.

Rather than thinking in terms of translation or interpretation, it is often simpler to imagine the existence of a hypothetical computer or **virtual machine** whose machine language is L1. Let us call this virtual machine M1 (and let us call the machine corresponding to L0, M0). If such a machine could be constructed cheaply enough, there would be no need for having language L0 or a machine that executed programs in L0 at all. People could simply write their programs in L1 and have the computer execute them directly. Even if the virtual machine whose language is L1 is too expensive or complicated to construct out of electronic circuits, people can still write programs for it. These programs can be either interpreted or translated by a program written in L0 that itself can be directly executed by the existing computer. In other words, people can write programs for virtual machines, just as though they really existed.

To make translation or interpretation practical, the languages L0 and L1 must not be "too" different. This constraint often means that L1, although better than L0, will still be far from ideal for most applications. This result is perhaps discouraging in light of the original purpose for creating L1—relieving the programmer of the burden of having to express algorithms in a language more suited to machines than people. However, the situation is not hopeless.

The obvious approach is to invent still another set of instructions that is more people-oriented and less machine-oriented than L1. This third set also forms a language, which we will call L2 (and with virtual machine M2). People can write programs in L2 just as though a virtual machine with L2 as its machine language really existed. Such programs can be either translated to L1 or executed by an interpreter written in L1.

The invention of a whole series of languages, each one more convenient than its predecessors, can go on indefinitely until a suitable one is finally achieved. Each language uses its predecessor as a basis, so we may view a computer using this technique as a series of **layers** or **levels**, one on top of another, as shown in Fig. 1-1. The bottommost language or level is the simplest and the topmost language or level is the most sophisticated.

There is an important relation between a language and a virtual machine. Each machine has a machine language, consisting of all the instructions that the machine can execute. In effect, a machine defines a language. Similarly, a language defines a machine—namely, the machine that can execute all programs written in the language. Of course, the machine defined by a certain language may be enormously complicated and expensive to construct directly out of electronic circuits but we can imagine it nevertheless. A machine with C or C++ or Java as its machine language would be complex indeed but could be built using today's technology. There is a good reason, however, for not building such a computer: it would not be cost effective compared to other techniques. Merely being doable is not good enough: a practical design must be cost effective as well.

In a certain sense, a computer with n levels can be regarded as n different virtual machines, each one with a different machine language. We will use the terms

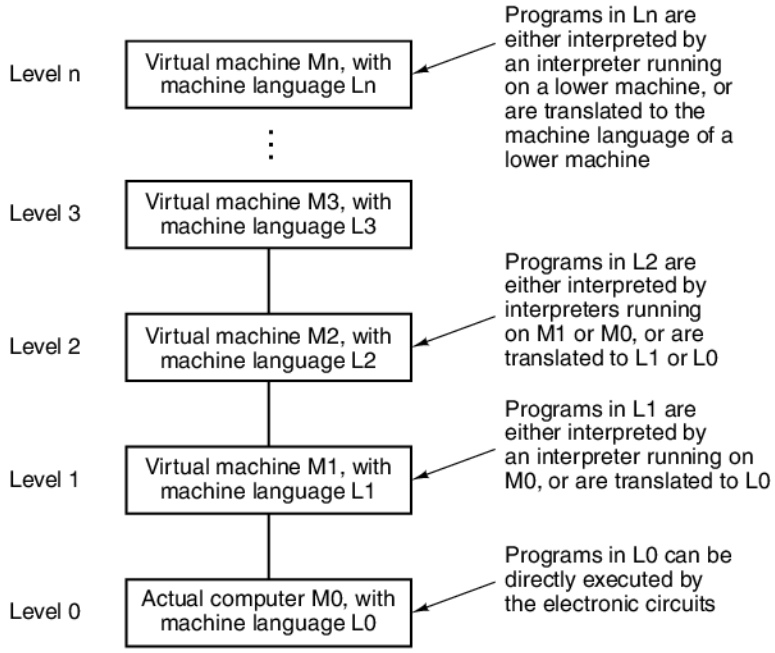


Figure 1-1. A multilevel machine.

"level" and "virtual machine" interchangeably. However, please note that like many terms in computer science, "virtual machine" has other meanings as well. We will look at another one of these later on in this book. Only programs written in language L0 can be directly carried out by the electronic circuits, without the need for intervening translation or interpretation. Programs written in L1, L2, ... L*n* must be either interpreted by an interpreter running on a lower level or translated to another language corresponding to a lower level.

A person who writes programs for the level *n* virtual machine need not be aware of the underlying interpreters and translators. The machine structure ensures that these programs will somehow be executed. It is of no real interest whether they are carried out step by step by an interpreter which, in turn, is also carried out by another interpreter, or whether they are carried out by the electronic circuits directly. The same result appears in both cases: the programs are executed.

Most programmers using an *n*-level machine are interested only in the top level, the one least resembling the machine language at the very bottom. However, people interested in understanding how a computer really works must study all the levels. People who design new computers or new levels must also be familiar with levels other than the top one. The concepts and techniques of constructing machines as a series of levels and the details of the levels themselves form the main subject of this book.

1.1.2 Contemporary Multilevel Machines

Most modern computers consist of two or more levels. Machines with as many as six levels exist, as shown in Fig. 1-2. Level 0, at the bottom, is the machine's true hardware. Its circuits carry out the machine-language programs of level 1. For the sake of completeness, we should mention the existence of yet another level below our level 0. This level, not shown in Fig. 1-2 because it falls within the realm of electrical engineering (and is thus outside the scope of this book), is called the **device level**. At this level, the designer sees individual transistors, which are the lowest-level primitives for computer designers. If one asks how transistors work inside, that gets us into solid-state physics.

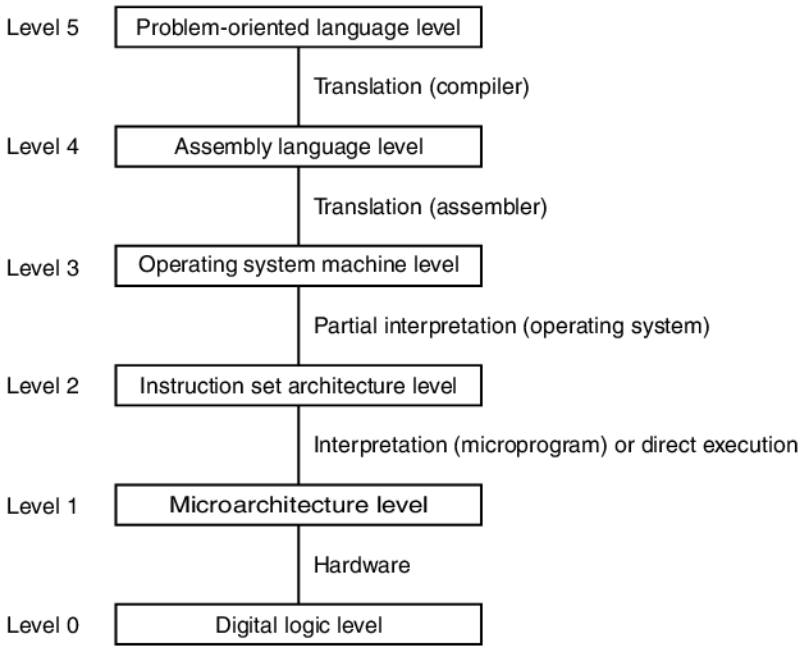


Figure 1-2. A six-level computer. The support method for each level is indicated below it (along with the name of the supporting program).

At the lowest level that we will study, the **digital logic level**, the interesting objects are called **gates**. Although built from analog components, such as transistors, gates can be accurately modeled as digital devices. Each gate has one or more digital inputs (signals representing 0 or 1) and computes as output some simple function of these inputs, such as AND or OR. Each gate is built up of at most a handful of transistors. A small number of gates can be combined to form a 1-bit memory, which can store a 0 or a 1. The 1-bit memories can be combined in groups of (for example) 16, 32, or 64 to form registers. Each **register** can hold a single binary

number up to some maximum. Gates can also be combined to form the main computing engine itself. We will examine gates and the digital logic level in detail in Chap. 3.

The next level up is the **microarchitecture level**. At this level we see a collection of (typically) 8 to 32 registers that form a local memory and a circuit called an **ALU** (**Arithmetic Logic Unit**), which is capable of performing simple arithmetic operations. The registers are connected to the ALU to form a **data path**, over which the data flow. The basic operation of the data path consists of selecting one or two registers, having the ALU operate on them (for example, adding them together), and storing the result back in some register.

On some machines the operation of the data path is controlled by a program called a **microprogram**. On other machines the data path is controlled directly by hardware. In early editions of this book, we called this level the "microprogramming level," because in the past it was nearly always a software interpreter. Since the data path is now often (partially) controlled directly by hardware, we changed the name to " microarchitecture level" to reflect this.

On machines with software control of the data path, the microprogram is an interpreter for the instructions at level 2. It fetches, examines, and executes instructions one by one, using the data path to do so. For example, for an ADD instruction, the instruction would be fetched, its operands located and brought into registers, the sum computed by the ALU, and finally the result routed back to the place it belongs. On a machine with hardwired control of the data path, similar steps would take place, but without an explicit stored program to control the interpretation of the level 2 instructions.

We will call level 2 the **Instruction Set Architecture level** (**ISA level**). Every computer manufacturer publishes a manual for each of the computers it sells, entitled "Machine Language Reference Manual," or "Principles of Operation of the Western Wombat Model 100X Computer," or something similar. These manuals are really about the ISA level, not the underlying levels. When they describe the machine's instruction set, they are in fact describing the instructions carried out interpretively by the microprogram or hardware execution circuits. If a computer manufacturer provides two interpreters for one of its machines, interpreting two different ISA levels, it will need to provide two "machine language" reference manuals, one for each interpreter.

The next level is usually a hybrid level. Most of the instructions in its language are also in the ISA level. (There is no reason why an instruction appearing at one level cannot be present at other levels as well.) In addition, there is a set of new instructions, a different memory organization, the ability to run two or more programs concurrently, and various other features. More variation exists between level 3 designs than between those at either level 1 or level 2.

The new facilities added at level 3 are carried out by an interpreter running at level 2, which, historically, has been called an operating system. Those level 3 instructions that are identical to level 2's are executed directly by the microprogram

(or hardwired control), not by the operating system. In other words, some of the level 3 instructions are interpreted by the operating system and some are interpreted directly by the microprogram (or hardwired control). This is what we mean by "hybrid" level. Throughout this book we will call this level the **operating system machine level**.

There is a fundamental break between levels 3 and 4. The lowest three levels are not designed for use by the average garden-variety programmer. Instead, they are intended primarily for running the interpreters and translators needed to support the higher levels. These interpreters and translators are written by people called **systems programmers** who specialize in designing and implementing new virtual machines. Levels 4 and above are intended for the applications programmer with a problem to solve.

Another change occurring at level 4 is the method by which the higher levels are supported. Levels 2 and 3 are always interpreted. Levels 4, 5, and above are usually, although not always, supported by translation.

Yet another difference between levels 1, 2, and 3, on the one hand, and levels 4, 5, and higher, on the other, is the nature of the language provided. The machine languages of levels 1, 2, and 3 are numeric. Programs in them consist of long series of numbers, which are fine for machines but bad for people. Starting at level 4, the languages contain words and abbreviations meaningful to people.

Level 4, the assembly language level, is really a symbolic form for one of the underlying languages. This level provides a method for people to write programs for levels 1, 2, and 3 in a form that is not as unpleasant as the virtual machine languages themselves. Programs in assembly language are first translated to level 1, 2, or 3 language and then interpreted by the appropriate virtual or actual machine. The program that performs the translation is called an **assembler**.

Level 5 usually consists of languages designed to be used by applications programmers with problems to solve. Such languages are often called **high-level languages**. Literally hundreds exist. A few of the better-known ones are C, C++, Java, Perl, Python, and PHP. Programs written in these languages are generally translated to level 3 or level 4 by translators known as **compilers**, although occasionally they are interpreted instead. Programs in Java, for example, are usually first translated to a an ISA-like language called Java byte code, which is then interpreted.

In some cases, level 5 consists of an interpreter for a specific application domain, such as symbolic mathematics. It provides data and operations for solving problems in this domain in terms that people knowledgeable in the domain can understand easily.

In summary, the key thing to remember is that computers are designed as a series of levels, each one built on its predecessors. Each level represents a distinct abstraction, with different objects and operations present. By designing and analyzing computers in this fashion, we are temporarily able to suppress irrelevant detail and thus reduce a complex subject to something easier to understand.

The set of data types, operations, and features of each level is called its **architecture**. The architecture deals with those aspects that are visible to the user of that level. Features that the programmer sees, such as how much memory is available, are part of the architecture. Implementation aspects, such as what kind of technology is used to implement the memory, are not part of the architecture. The study of how to design those parts of a computer system that are visible to the programmers is called **computer architecture**. In common practice, however, computer architecture and computer organization mean essentially the same thing.

1.1.3 Evolution of Multilevel Machines

To provide some perspective on multilevel machines, we will briefly examine their historical development, showing how the number and nature of the levels has evolved over the years. Programs written in a computer's true machine language (level 1) can be directly executed by the computer's electronic circuits (level 0), without any intervening interpreters or translators. These electronic circuits, along with the memory and input/output devices, form the computer's **hardware**. Hardware consists of tangible objects—integrated circuits, printed circuit boards, cables, power supplies, memories, and printers—rather than abstract ideas, algorithms, or instructions.

Software, in contrast, consists of **algorithms** (detailed instructions telling how to do something) and their computer representations—namely, programs. Programs can be stored on hard disk, CD-ROM, or other media, but the essence of software is the set of instructions that makes up the programs, not the physical media on which they are recorded.

In the very first computers, the boundary between hardware and software was crystal clear. Over time, however, it has blurred considerably, primarily due to the addition, removal, and merging of levels as computers have evolved. Nowadays, it is often hard to tell them apart (Vahid, 2003). In fact, a central theme of this book is

Hardware and software are logically equivalent.

Any operation performed by software can also be built directly into the hardware, preferably after it is sufficiently well understood. As Karen Panetta put it: "Hardware is just petrified software." Of course, the reverse is also true: any instruction executed by the hardware can also be simulated in software. The decision to put certain functions in hardware and others in software is based on such factors as cost, speed, reliability, and frequency of expected changes. There are few hard-and-fast rules to the effect that X must go into the hardware and Y must be programmed explicitly. These decisions change with trends in technology economics, demand, and computer usage.

The Invention of Microprogramming

The first digital computers, back in the 1940s, had only two levels: the ISA level, in which all the programming was done, and the digital logic level, which executed these programs. The digital logic level's circuits were complicated, difficult to understand and build, and unreliable.

In 1951, Maurice Wilkes, a researcher at the University of Cambridge, suggested designing a three-level computer in order to drastically simplify the hardware and thus reduce the number of (unreliable) vacuum tubes needed (Wilkes, 1951). This machine was to have a built-in, unchangeable interpreter (the microprogram), whose function was to execute ISA-level programs by interpretation. Because the hardware would now only have to execute microprograms, which have a limited instruction set, instead of ISA-level programs, which have a much larger instruction set, fewer electronic circuits would be needed. Because electronic circuits were then made from vacuum tubes, such a simplification promised to reduce tube count and hence enhance reliability (i.e., the number of crashes per day).

A few of these three-level machines were constructed during the 1950s. More were constructed during the 1960s. By 1970 the idea of having the ISA level be interpreted by a microprogram, instead of directly by the electronics, was dominant. All the major machines of the day used it.

The Invention of the Operating System

In these early years, most computers were "open shop," which meant that the programmer had to operate the machine personally. Next to each machine was a sign-up sheet. A programmer wanting to run a program signed up for a block of time, say Wednesday morning 3 to 5 A.M. (many programmers liked to work when it was quiet in the machine room). When the time arrived, the programmer headed for the machine room with a deck of 80-column punched cards (an early input medium) in one hand and a sharpened pencil in the other. Upon arriving in the computer room, he or she gently nudged the previous programmer toward the door and took over the computer.

If the programmer wanted to run a FORTRAN program, the following steps were necessary:

1. He† went over to the cabinet where the program library was kept, took out the big green deck labeled FORTRAN compiler, put it in the card reader, and pushed the START button.
2. He put his FORTRAN program in the card reader and pushed the CONTINUE button. The program was read in.

† "He" should be read as "he or she" throughout this book.

3. When the computer stopped, he read his FORTRAN program in a second time. Although some compilers required only one pass over the input, many required two or more. For each pass, a large card deck had to be read in.
4. Finally, the translation neared completion. The programmer often became nervous near the end because if the compiler found an error in the program, he had to correct it and start the entire process all over again. If there were no errors, the compiler punched out the translated machine language program on cards.
5. The programmer then put the machine language program in the card reader along with the subroutine library deck and read them both in.
6. The program began executing. More often than not it did not work and unexpectedly stopped in the middle. Generally, the programmer fiddled with the console switches and looked at the console lights for a while. If lucky, he figured out the problem, corrected the error, and went back to the cabinet containing the big green FORTRAN compiler to start over again. If less fortunate, he made a printout of the contents of memory, called a **core dump**, and took it home to study.

This procedure, with minor variations, was normal at many computer centers for years. It forced the programmers to learn how to operate the machine and to know what to do when it broke down, which was often. The machine was frequently idle while people were carrying cards around the room or scratching their heads trying to find out why their programs were not working properly.

Around 1960 people tried to reduce the amount of wasted time by automating the operator's job. A program called an **operating system** was kept in the computer at all times. The programmer provided certain control cards along with the program that were read and carried out by the operating system. Figure 1-3 shows a sample job for one of the first widespread operating systems, FMS (FORTRAN Monitor System), on the IBM 709.

The operating system read the *JOB card and used the information on it for accounting purposes. (The asterisk was used to identify control cards, so they would not be confused with program and data cards.) Later, it read the *FORTRAN card, which was an instruction to load the FORTRAN compiler from a magnetic tape. The compiler then read in and compiled the FORTRAN program. When the compiler finished, it returned control back to the operating system, which then read the *DATA card. This was an instruction to execute the translated program, using the cards following the *DATA card as the data.

Although the operating system was designed to automate the operator's job (hence the name), it was also the first step in the development of a new virtual machine. The *FORTRAN card could be viewed as a virtual "compile program" instruction. Similarly, the *DATA card could be regarded as a virtual "execute

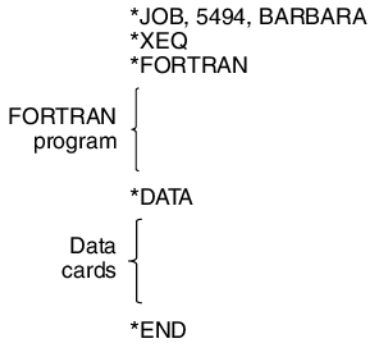


Figure 1-3. A sample job for the FMS operating system.

program" instruction. A level with only two instructions was not much of a level but it was a start in that direction.

In subsequent years, operating systems became more and more sophisticated. New instructions, facilities, and features were added to the ISA level until it began to take on the appearance of a new level. Some of this new level's instructions were identical to the ISA-level instructions, but others, particularly input/output instructions, were completely different. The new instructions were often known as **operating system macros** or **supervisor calls**. The usual term now is **system call**.

Operating systems developed in other ways as well. The early ones read card decks and printed output on the line printer. This organization was known as a **batch system**. Usually, there was a wait of several hours between the time a program was submitted and the time the results were ready. Developing software was difficult under those circumstances.

In the early 1960s researchers at Dartmouth College, M.I.T., and elsewhere developed operating systems that allowed (multiple) programmers to communicate directly with the computer. In these systems, remote terminals were connected to the central computer via telephone lines. The computer was shared among many users. A programmer could type in a program and get the results typed back almost immediately, in the office, in a garage at home, or wherever the terminal was located. These systems were called **timesharing systems**.

Our interest in operating systems is in those parts that interpret the instructions and features present in level 3 and not present in the ISA level rather than in the timesharing aspects. Although we will not emphasize it, you should keep in mind that operating systems do more than just interpret features added to the ISA level.

The Migration of Functionality to Microcode

Once microprogramming had become common (by 1970), designers realized that they could add new instructions by just extending the microprogram. In other words, they could add "hardware" (new machine instructions) by programming.

This revelation led to a virtual explosion in machine instruction sets, as designers competed with one another to produce bigger and better instruction sets. Many of these instructions were not essential in the sense that their effect could be easily achieved by existing instructions, but often they were slightly faster than a sequence of existing instructions. For example, many machines had an instruction INC (INCrement) that added 1 to a number. Since these machines also had a general ADD instruction, having a special instruction to add 1 (or to add 720, for that matter) was not necessary. However, the INC was usually a little faster than the ADD, so it got thrown in.

For the same reason, many other instructions were added to the microprogram. These often included

1. Instructions for integer multiplication and division.
2. Floating-point arithmetic instructions.
3. Instructions for calling and returning from procedures.
4. Instructions for speeding up looping.
5. Instructions for handling character strings.

Furthermore, once machine designers saw how easy it was to add new instructions, they began looking around for other features to add to their microprograms. A few examples of these additions include

1. Features to speed up computations involving arrays (indexing and indirect addressing).
2. Features to permit programs to be moved in memory after they have started running (relocation facilities).
3. Interrupt systems that signal the computer as soon as an input or output operation is completed.
4. The ability to suspend one program and start another in a small number of instructions (process switching).
5. Special instructions for processing audio, image, and multimedia files.

Numerous other features and facilities have been added over the years as well, usually for speeding up some particular activity.

The Elimination of Microprogramming

Microprograms grew fat during the golden years of microprogramming (1960s and 1970s). They also tended to get slower and slower as they acquired more bulk. Finally, some researchers realized that by eliminating the microprogram, vastly

reducing the instruction set, and having the remaining instructions be directly executed (i.e., hardware control of the data path), machines could be speeded up. In a certain sense, computer design had come full circle, back to the way it was before Wilkes invented microprogramming in the first place.

But the wheel is still turning. Modern processors still rely on microprogramming to translate complex instructions to internal microcode that can be executed directly on streamlined hardware.

The point of this discussion is to show that the boundary between hardware and software is arbitrary and constantly changing. Today's software may be tomorrow's hardware, and vice versa. Furthermore, the boundaries between the various levels are also fluid. From the programmer's point of view, how an instruction is actually implemented is unimportant (except perhaps for its speed). A person programming at the ISA level can use its multiply instruction as though it were a hardware instruction without having to worry about it, or even be aware of whether it really is a hardware instruction. One person's hardware is another person's software. We will come back to all these topics later in this book.

1.2 MILESTONES IN COMPUTER ARCHITECTURE

Hundreds of different kinds of computers have been designed and built during the evolution of the modern digital computer. Most have been long forgotten, but a few have had a significant impact on modern ideas. In this section we will give a brief sketch of some of the key historical developments in order to get a better understanding of how we got where we are now. Needless to say, this section only touches on the highlights and leaves many stones unturned. Figure 1-4 lists some of the milestone machines to be discussed in this section. Slater (1987) is a good place to look for additional historical material on the people who founded the computer age. For short biographies and beautiful color photographs by Louis Fabian Bachrach of some of the key people who founded the computer age, see Morgan's coffee-table book (1997).

1.2.1 The Zeroth Generation—Mechanical Computers (1642–1945)

The first person to build a working calculating machine was the French scientist Blaise Pascal (1623–1662), in whose honor the programming language Pascal is named. This device, built in 1642, when Pascal was only 19, was designed to help his father, a tax collector for the French government. It was entirely mechanical, using gears, and powered by a hand-operated crank.

Pascal's machine could do only addition and subtraction operations, but thirty years later the great German mathematician Baron Gottfried Wilhelm von Leibniz (1646–1716) built another mechanical machine that could multiply and divide as

Year	Name	Made by	Comments
1834	Analytical Engine	Babbage	First attempt to build a digital computer
1936	Z1	Zuse	First working relay calculating machine
1943	COLOSSUS	British gov't	First electronic computer
1944	Mark I	Aiken	First American general-purpose computer
1946	ENIAC	Eckert/Mauchley	Modern computer history starts here
1949	EDSAC	Wilkes	First stored-program computer
1951	Whirlwind I	M.I.T.	First real-time computer
1952	IAS	Von Neumann	Most current machines use this design
1960	PDP-1	DEC	First minicomputer (50 sold)
1961	1401	IBM	Enormously popular small business machine
1962	7094	IBM	Dominated scientific computing in the early 1960s
1963	B5000	Burroughs	First machine designed for a high-level language
1964	360	IBM	First product line designed as a family
1964	6600	CDC	First scientific supercomputer
1965	PDP-8	DEC	First mass-market minicomputer (50,000 sold)
1970	PDP-11	DEC	Dominated minicomputers in the 1970s
1974	8080	Intel	First general-purpose 8-bit computer on a chip
1974	CRAY-1	Cray	First vector supercomputer
1978	VAX	DEC	First 32-bit superminicomputer
1981	IBM PC	IBM	Started the modern personal computer era
1981	Osborne-1	Osborne	First portable computer
1983	Lisa	Apple	First personal computer with a GUI
1985	386	Intel	First 32-bit ancestor of the Pentium line
1985	MIPS	MIPS	First commercial RISC machine
1985	XC2064	Xilinx	First field-programmable gate array (FPGA)
1987	SPARC	Sun	First SPARC-based RISC workstation
1989	GridPad	Grid Systems	First commercial tablet computer
1990	RS6000	IBM	First superscalar machine
1992	Alpha	DEC	First 64-bit personal computer
1992	Simon	IBM	First smartphone
1993	Newton	Apple	First palmtop computer (PDA)
2001	POWER4	IBM	First dual-core chip multiprocessor

Figure 1-4. Some milestones in the development of the modern digital computer.

well. In effect, Leibniz had built the equivalent of a four-function pocket calculator three centuries ago.

Nothing much happened for the next 150 years until a professor of mathematics at the University of Cambridge, Charles Babbage (1792–1871), the inventor of

the speedometer, designed and built his **difference engine**. This mechanical device, which like Pascal's could only add and subtract, was designed to compute tables of numbers useful for naval navigation. The entire construction of the machine was designed to run a single algorithm, the method of finite differences using polynomials. The most interesting feature of the difference engine was its output method: it punched its results into a copper engraver's plate with a steel die, thus foreshadowing later write-once media such as punched cards and CD-ROMs.

Although the difference engine worked reasonably well, Babbage quickly got bored with a machine that could run only one algorithm. He began to spend increasingly large amounts of his time and family fortune (not to mention 17,000 pounds of the government's money) on the design and construction of a successor called the **analytical engine**. The analytical engine had four components: the store (memory), the mill (computation unit), the input section (punched-card reader), and the output section (punched and printed output). The store consisted of 1000 words of 50 decimal digits, each used to hold variables and results. The mill could accept operands from the store, then add, subtract, multiply, or divide them, and finally return the result to the store. Like the difference engine, it was entirely mechanical.

The great advance of the analytical engine was that it was general purpose. It read instructions from punched cards and carried them out. Some instructions commanded the machine to fetch two numbers from the store, bring them to the mill, be operated on (e.g., added), and have the result sent back to the store. Other instructions could test a number and conditionally branch depending on whether it was positive or negative. By punching a different program on the input cards, it was possible to have the analytical engine perform different computations, something not true of the difference engine.

Since the analytical engine was programmable in a simple assembly language, it needed software. To produce this software, Babbage hired a young woman named Augusta Ada Lovelace, who was the daughter of famed British poet Lord Byron. Ada Lovelace was thus the world's first computer programmer. The programming language Ada is named in her honor.

Unfortunately, like many modern designers, Babbage never quite got the hardware debugged. The problem was that he needed thousands upon thousands of cogs and wheels and gears produced to a degree of precision that nineteenth-century technology was unable to provide. Nevertheless, his ideas were far ahead of his time, and even today most modern computers have a structure very similar to the analytical engine, so it is certainly fair to say that Babbage was the (grand)father of the modern digital computer.

The next major development occurred in the late 1930s, when a German engineering student named Konrad Zuse built a series of automatic calculating machines using electromagnetic relays. He was unable to get government funding after WWII began because government bureaucrats expected to win the war so quickly that the new machine would not be ready until after it was over. Zuse was unaware

of Babbage's work, and his machines were destroyed by the Allied bombing of Berlin in 1944, so his work did not have any influence on subsequent machines. Still, he was one of the pioneers of the field.

Slightly later, in the United States, two people also designed calculators, John Atanasoff at Iowa State College and George Stibbitz at Bell Labs. Atanasoff's machine was amazingly advanced for its time. It used binary arithmetic and had capacitors for memory, which were periodically refreshed to keep the charge from leaking out, a process he called "jogging the memory." Modern dynamic memory (DRAM) chips work the same way. Unfortunately the machine never really became operational. In a way, Atanasoff was like Babbage: a visionary who was ultimately defeated by the inadequate hardware technology of his time.

Stibbitz' computer, although more primitive than Atanasoff's, actually worked. Stibbitz gave a public demonstration of it at a conference at Dartmouth College in 1940. Among those in the audience was John Mauchley, an unknown professor of physics at the University of Pennsylvania. The computing world would hear more about Prof. Mauchley later.

While Zuse, Stibbitz, and Atanasoff were designing automatic calculators, a young man named Howard Aiken was grinding out tedious numerical calculations by hand as part of his Ph.D. research at Harvard. After graduating, Aiken recognized the importance of being able to do calculations by machine. He went to the library, discovered Babbage's work, and decided to build out of relays the general-purpose computer that Babbage had failed to build out of toothed wheels.

Aiken's first machine, the Mark I, was completed at Harvard in 1944. It had 72 words of 23 decimal digits each and had an instruction time of 6 sec. Input and output used punched paper tape. By the time Aiken had completed its successor, the Mark II, relay computers were obsolete. The electronic era had begun.

1.2.2 The First Generation—Vacuum Tubes (1945–1955)

The stimulus for the electronic computer was World War II. During the early part of the war, German submarines were wreaking havoc on British ships. Commands were sent from the German admirals in Berlin to the submarines by radio, which the British could, and did, intercept. The problem was that these messages were encoded using a device called the **ENIGMA**, whose forerunner was designed by amateur inventor and former U.S. president, Thomas Jefferson.

Early in the war, British intelligence managed to acquire an ENIGMA machine from Polish Intelligence, which had stolen it from the Germans. However, to break a coded message, a huge amount of computation was needed, and it was needed very soon after the message was intercepted to be of any use. To decode these messages, the British government set up a top secret laboratory that built an electronic computer called the COLOSSUS. The famous British mathematician Alan Turing helped design this machine. The COLOSSUS was operational in 1943, but since the British government kept virtually every aspect of the project classified as

a military secret for 30 years, the COLOSSUS line was basically a dead end. It is worth noting only because it was the world's first electronic digital computer.

In addition to destroying Zuse's machines and stimulating the construction of the COLOSSUS, the war also affected computing in the United States. The army needed range tables for aiming its heavy artillery. It produced these tables by hiring hundreds of women to crank them out using hand calculators (women were thought to be more accurate than men). Nevertheless, the process was time consuming and errors often crept in.

John Mauchley, who knew of Atanasoff's work as well as Stibbitz', was aware that the army was interested in mechanical calculators. Like many computer scientists after him, he put together a grant proposal asking the army for funding to build an electronic computer. The proposal was accepted in 1943, and Mauchley and his graduate student, J. Presper Eckert, proceeded to build an electronic computer, which they called the **ENIAC** (**Electronic Numerical Integrator And Computer**). It consisted of 18,000 vacuum tubes and 1500 relays. The ENIAC weighed 30 tons and consumed 140 kilowatts of power. Architecturally, the machine had 20 registers, each capable of holding a 10-digit decimal number. (A decimal register is very small memory that can hold one number up to some maximum number of decimal digits, somewhat like the odometer that keeps track of how far a car has traveled in its lifetime.) The ENIAC was programmed by setting up 6000 multiposition switches and connecting a multitude of sockets with a veritable forest of jumper cables.

The machine was not finished until 1946, too late to be of any use for its original purpose. However, since the war was over, Mauchley and Eckert were allowed to organize a summer school to describe their work to their scientific colleagues. That summer school was the beginning of an explosion of interest in building large digital computers.

After that historic summer school, many other researchers set out to build electronic computers. The first one operational was the EDSAC (1949), built at the University of Cambridge by Maurice Wilkes. Others included the JOHNNIAC at the Rand Corporation, the ILLIAC at the University of Illinois, the MANIAC at Los Alamos Laboratory, and the WEIZAC at the Weizmann Institute in Israel.

Eckert and Mauchley soon began working on a successor, the **EDVAC** (**Electronic Discrete Variable Automatic Computer**). However, that project was fatally wounded when they left the University of Pennsylvania to form a startup company, the Eckert-Mauchley Computer Corporation, in Philadelphia (Silicon Valley had not yet been invented). After a series of mergers, this company became the modern Unisys Corporation.

As a legal aside, Eckert and Mauchley filed for a patent claiming they invented the digital computer. In retrospect, this would not be a bad patent to own. After years of litigation, the courts decided that the Eckert-Mauchley patent was invalid and that John Atanasoff invented the digital computer, even though he never patented it, effectively putting the invention in the public domain.

While Eckert and Mauchley were working on the EDVAC, one of the people involved in the ENIAC project, John von Neumann, went to Princeton's Institute of Advanced Studies to build his own version of the EDVAC, the **IAS machine**. Von Neumann was a genius in the same league as Leonardo Da Vinci. He spoke many languages, was an expert in the physical sciences and mathematics, and had total recall of everything he ever heard, saw, or read. He was able to quote verbatim from memory the text of books he had read years earlier. At the time he became interested in computers, he was already the most eminent mathematician in the world.

It was soon apparent to him that programming computers with huge numbers of switches and cables was slow, tedious, and inflexible. He came to realize that the program could be represented in digital form in the computer's memory, along with the data. He also saw that the clumsy serial decimal arithmetic used by the ENIAC, with each digit represented by 10 vacuum tubes (1 on and 9 off) could be replaced by using parallel binary arithmetic, something Atanasoff had realized years earlier.

The basic design, which he first described, is now known as a **von Neumann machine**. It was used in the EDSAC, the first stored-program computer, and even now, more than half a century later, is still the basis for nearly all digital computers. This design, and the IAS machine, built in collaboration with Herman Goldstine, has had such an enormous influence that it is worth describing briefly. Although Von Neumann's name is always attached to this design, Goldstine and others made major contributions to it as well. A sketch of the architecture is given in Fig. 1-5.

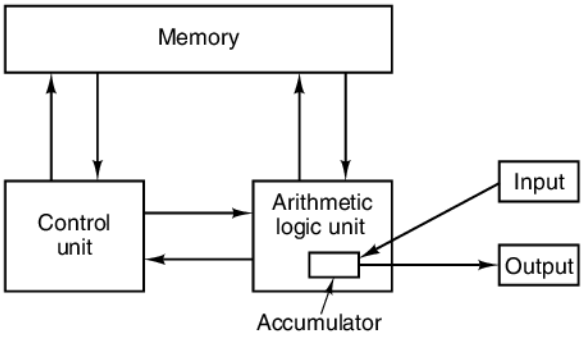


Figure 1-5. The original von Neumann machine.

The von Neumann machine had five basic parts: the memory, the arithmetic logic unit, the control unit, and the input and output equipment. The memory consisted of 4096 words, a word holding 40 bits, each a 0 or a 1. Each word held either two 20-bit instructions or a 40-bit signed integer. The instructions had 8 bits devoted to telling the instruction type and 12 bits for specifying one of the 4096

memory words. Together, the arithmetic logic unit and the control unit formed the "brain" of the computer. In modern computers they are combined onto a single chip called the **CPU** (**Central Processing Unit**).

Inside the arithmetic logic unit was a special internal 40-bit register called the **accumulator**. A typical instruction added a word of memory to the accumulator or stored the contents of the accumulator in memory. The machine did not have floating-point arithmetic because von Neumann felt that any competent mathematician ought to be able to keep track of the decimal point (actually the binary point) in his or her head.

At about the same time von Neumann was building the IAS machine, researchers at M.I.T. were also building a computer. Unlike IAS, ENIAC and other machines of its type, which had long word lengths and were intended for heavy number crunching, the M.I.T. machine, the Whirlwind I, had a 16-bit word and was designed for real-time control. This project led to the invention of the magnetic core memory by Jay Forrester, and then eventually to the first commercial minicomputer.

While all this was going on, IBM was a small company engaged in the business of producing card punches and mechanical card-sorting machines. Although IBM had provided some of Aiken's financing, it was not terribly interested in computers until it produced the 701 in 1953, long after Eckert and Mauchley's company was number one in the commercial market with its UNIVAC computer. The 701 had 2048 36-bit words, with two instructions per word. It was the first in a series of scientific machines that came to dominate the industry within a decade. Three years later came the 704, which initially had 4096 words of core memory, 36-bit instructions, and a new innovation, floating-point hardware. In 1958, IBM began production of its last vacuum-tube machine, the 709, which was basically a beefed-up 704.

1.2.3 The Second Generation—Transistors (1955–1965)

The transistor was invented at Bell Labs in 1948 by John Bardeen, Walter Brattain, and William Shockley, for which they were awarded the 1956 Nobel Prize in physics. Within 10 years the transistor revolutionized computers, and by the late 1950s, vacuum tube computers were obsolete. The first transistorized computer was built at M.I.T.'s Lincoln Laboratory, a 16-bit machine along the lines of the Whirlwind I. It was called the **TX-0** (**Transistorized eXperimental computer 0**) and was merely intended as a device to test the much fancier TX-2.

The TX-2 never amounted to much, but one of the engineers working at the Laboratory, Kenneth Olsen, formed a company, Digital Equipment Corporation (DEC), in 1957 to manufacture a commercial machine much like the TX-0. It was four years before this machine, the PDP-1, appeared, primarily because the venture capitalists who funded DEC firmly believed that there was no market for computers. After all, T.J. Watson, former president of IBM, once said that the world

market for computers was about four or five units. Instead, DEC mostly sold small circuit boards to companies to integrate into their products.

When the PDP-1 finally appeared in 1961, it had 4096 18-bit words of core memory and could execute 200,000 instructions/sec. This performance was half that of the IBM 7090, the transistorized successor to the 709, and the fastest computer in the world at the time. The PDP-1 cost $120,000; the 7090 cost millions. DEC sold dozens of PDP-1s, and the minicomputer industry was born.

One of the first PDP-1s was given to M.I.T., where it quickly attracted the attention of some of the budding young geniuses so common at M.I.T. One of the PDP-1's many innovations was a visual display and the ability to plot points anywhere on its 512-by-512 pixel screen. Before long, the students had programmed the PDP-1 to play Spacewar, and the world had its first video game.

A few years later DEC introduced the PDP-8, which was a 12-bit machine, but much cheaper than the PDP-1 ($16,000). The PDP-8 had a major innovation: a single bus, the omnibus, as shown in Fig. 1-6. A **bus** is a collection of parallel wires used to connect the components of a computer. This architecture was a major departure from the memory-centered IAS machine and has been adopted by nearly all small computers since. DEC eventually sold 50,000 PDP-8s, which established it as the leader in the minicomputer business.

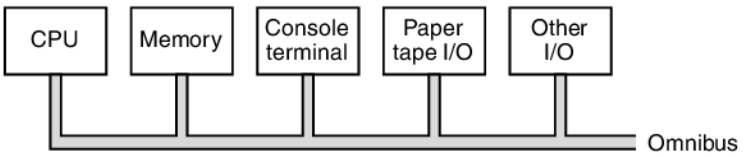


Figure 1-6. The PDP-8 omnibus.

Meanwhile, IBM's reaction to the transistor was to build a transistorized version of the 709, the 7090, as mentioned above, and later the 7094. The 7094 had a cycle time of 2 microsec and a core memory consisting of 32,768 words of 36 bits each. The 7090 and 7094 marked the end of the ENIAC-type machines, but they dominated scientific computing for years in the 1960s.

At the same time that IBM had become a major force in scientific computing with the 7094, it was making a huge amount of money selling a little business-oriented machine called the 1401. This machine could read and write magnetic tapes, read and punch cards, and print output almost as fast as the 7094, and at a fraction of the price. It was terrible at scientific computing, but for business record keeping it was perfect.

The 1401 was unusual in that it did not have any registers, or even a fixed word length. Its memory was 4000 8-bit bytes, although later models supported up to a then-astounding 16,000 bytes. Each byte contained a 6-bit character, an administrative bit, and a bit used to indicate end-of-word. A MOVE instruction, for example, had a source and a destination address and began moving bytes from the source to the destination until it hit one with the end-of-word bit set to 1.

In 1964 a tiny, unknown company, Control Data Corporation (CDC), introduced the 6600, a machine that was nearly an order of magnitude faster than the mighty 7094 and every other machine in existence at the time. It was love at first sight among the number crunchers, and CDC was launched on its way to success. The secret to its speed, and the reason it was so much faster than the 7094, was that inside the CPU was a highly parallel machine. It had several functional units for doing addition, others for doing multiplication, and still another for division, and all of them could run in parallel. Although getting the most out of it required careful programming, with some work it was possible to have 10 instructions being executed at once.

As if this was not enough, the 6600 had a number of little computers inside to help it, sort of like Snow White and the Seven Vertically Challenged People. This meant that the CPU could spend all its time crunching numbers, leaving all the details of job management and input/output to the smaller computers. In retrospect, the 6600 was decades ahead of its time. Many of the key ideas found in modern computers can be traced directly back to the 6600.

The designer of the 6600, Seymour Cray, was a legendary figure, in the same league as Von Neumann. He devoted his entire life to building faster and faster machines, now called **supercomputers**, including the 6600, 7600, and Cray-1. He also invented a now-famous algorithm for buying cars: you go to the dealer closest to your house, point to the car closest to the door, and say: "I'll take that one." This algorithm wastes the least time on unimportant things (like buying cars) to leave you the maximum time for doing important things (like designing supercomputers).

There were many other computers in this era, but one stands out for quite a different reason and is worth mentioning: the Burroughs B5000. The designers of machines like the PDP-1, 7094, and 6600 were all totally preoccupied with the hardware, making it either cheap (DEC) or fast (IBM and CDC). Software was almost completely irrelevant. The B5000 designers took a different tack. They built a machine specifically with the intention of having it programmed in Algol 60, a forerunner of C and Java, and included many features in the hardware to ease the compiler's task. The idea that software also counted was born. Unfortunately it was forgotten almost immediately.

1.2.4 The Third Generation—Integrated Circuits (1965–1980)

The invention of the silicon integrated circuit by Jack Kilby and Robert Noyce (working independently) in 1958 allowed dozens of transistors to be put on a single chip. This packaging made it possible to build computers that were smaller, faster, and cheaper than their transistorized predecessors. Some of the more significant computers from this generation are described below.

By 1964 IBM was the leading computer company and had a big problem with its two highly successful and profitable machines, the 7094 and the 1401: they

were as incompatible as two machines could be. One was a high-speed number cruncher using parallel binary arithmetic on 36-bit registers, and the other was a glorified input/output processor using serial decimal arithmetic on variable-length words in memory. Many of its corporate customers had both and did not like the idea of having two separate programming departments with nothing in common.

When the time came to replace these two series, IBM took a radical step. It introduced a single product line, the System/360, based on integrated circuits, that was designed for both scientific and commercial computing. The System/360 contained many innovations, the most important of which was that it was a family of about a half-dozen machines with the same assembly language, and increasing size and power. A company could replace its 1401 with a 360 Model 30 and its 7094 with a 360 Model 75. The Model 75 was bigger and faster (and more expensive), but software written for one of them could, in principle, run on the other. In practice, a program written for a small model would run on a large model without problems. However, the reverse was not true. When moving a program written for a large model to a smaller machine, the program might not fit in memory. Still, this was a major improvement over the situation with the 7094 and 1401. The idea of machine families caught on instantly, and within a few years most computer manufacturers had a family of common machines spanning a wide range of price and performance. Some characteristics of the initial 360 family are shown in Fig. 1-7. Other models were introduced later.

Property	Model 30	Model 40	Model 50	Model 65
Relative performance	1	3.5	10	21
Cycle time (in billionths of a sec)	1000	625	500	250
Maximum memory (bytes)	65,536	262,144	262,144	524,288
Bytes fetched per cycle	1	2	4	16
Maximum number of data channels	3	3	4	6

Figure 1-7. The initial offering of the IBM 360 product line.

Another major innovation in the 360 was **multiprogramming**, having several programs in memory at once, so that when one was waiting for input/output to complete, another could compute. This resulted in a higher CPU utilization.

The 360 also was the first machine that could emulate (simulate) other computers. The smaller models could emulate the 1401, and the larger ones could emulate the 7094, so that customers could continue to run their old unmodified binary programs while converting to the 360. Some models ran 1401 programs so much faster than the 1401 itself that many customers never converted their programs.

Emulation was easy on the 360 because all the initial models and most of the later models were microprogrammed. All IBM had to do was write three microprograms, for the native 360 instruction set, the 1401 instruction set, and the 7094

instruction set. This flexibility was one of the main reasons microprogramming was introduced in the 360. Wilkes' motivation of reducing tube count no longer mattered, of course, since the 360 did not have any tubes.

The 360 solved the dilemma of binary-parallel versus serial decimal with a compromise: the machine had 16 32-bit registers for binary arithmetic, but its memory was byte-oriented, like that of the 1401. It also had 1401 style serial instructions for moving variably sized records around memory.

Another major feature of the 360 was a (for that time) huge address space of 2^{24} (16,777,216) bytes. With memory costing several dollars per byte in those days, this much memory looked very much like infinity. Unfortunately, the 360 series was later followed by the 370, 4300, 3080, 3090, 390 and z series, all using essentially the same architecture. By the mid 1980s, the memory limit became a real problem, and IBM had to partially abandon compatibility when it went to 32-bit addresses needed to address the new 2^{32}-byte memory.

With hindsight, it can be argued that since they had 32-bit words and registers anyway, they probably should have had 32-bit addresses as well, but at the time no one could imagine a machine with 16 million bytes of memory. While the transition to 32-bit addresses was successful for IBM, it was again only a temporary solution to the memory-addressing problem, as computing systems would soon require the ability to address more than 2^{32} (4,294,967,296) bytes of memory. In a few more years computers with 64-bit addresses would appear on the scene.

The minicomputer world also took a big step forward in the third generation with DEC's introduction of the PDP-11 series, a 16-bit successor to the PDP-8. In many ways, the PDP-11 series was like a little brother to the 360 series just as the PDP-1 was like a little brother to the 7094. Both the 360 and PDP-11 had word-oriented registers and a byte-oriented memory and both came in a range spanning a considerable price/performance ratio. The PDP-11 was enormously successful, especially at universities, and continued DEC's lead over the other minicomputer manufacturers.

1.2.5 The Fourth Generation—Very Large Scale Integration (1980–?)

By the 1980s, **VLSI** (**Very Large Scale Integration**) had made it possible to put first tens of thousands, then hundreds of thousands, and finally millions of transistors on a single chip. This development soon led to smaller and faster computers. Before the PDP-1, computers were so big and expensive that companies and universities had to have special departments called **computer centers** to run them. With the advent of the minicomputer, a department could buy its own computer. By 1980, prices had dropped so low that it was feasible for a single individual to have his or her own computer. The personal computer era had begun.

Personal computers were used in a very different way than large computers. They were used for word processing, spreadsheets, and numerous highly interactive applications (such as games) that the larger computers could not handle well.

The first personal computers were usually sold as kits. Each kit contained a printed circuit board, a bunch of chips, typically including an Intel 8080, some cables, a power supply, and perhaps an 8-inch floppy disk. Putting the parts together to make a computer was up to the purchaser. Software was not supplied. If you wanted any, you wrote your own. Later, the CP/M operating system, written by Gary Kildall, became popular on 8080s. It was a true (floppy) disk operating system, with a file system, and user commands typed in from the keyboard to a command processor (shell).

Another early personal computer was the Apple and later the Apple II, designed by Steve Jobs and Steve Wozniak in the proverbial garage. This machine was enormously popular with home users and at schools and made Apple a serious player almost overnight.

After much deliberating and observing what other companies were doing, IBM, then the dominant force in the computer industry, finally decided it wanted to get into the personal computer business. Rather than design the entire machine from scratch, using only IBM parts, made from IBM transistors, made from IBM sand, which would have taken far too long, IBM did something quite uncharacteristic. It gave an IBM executive, Philip Estridge, a large bag of money and told him to go build a personal computer far from the meddling bureaucrats at corporate headquarters in Armonk, NY. Estridge, working 2000 km away in Boca Raton, Florida, chose the Intel 8088 as his CPU, and built the IBM Personal Computer from commercial components. It was introduced in 1981 and instantly became the best-selling computer in history. When the PC hit 30, a number of articles about its history were published, including those by Bradley (2011), Goth (2011), Bride (2011), and Singh (2011).

IBM also did something uncharacteristic that it would later come to regret. Rather than keeping the design of the machine totally secret (or at least, guarded by a gigantic and impenetrable wall of patents), as it normally did, it published the complete plans, including all the circuit diagrams, in a book that it sold for $49. The idea was to make it possible for other companies to make plug-in boards for the IBM PC, to increase its flexibility and popularity. Unfortunately for IBM, since the design was now completely public and all the parts were easily available from commercial vendors, numerous other companies began making **clones** of the PC, often for far less money than IBM was charging. Thus, an entire industry started.

Although other companies made personal computers using non-Intel CPUs, including Commodore, Apple, and Atari, the momentum of the IBM PC industry was so large that the others were steamrollered. Only a few survived, and these were in niche markets.

One that did survive, although barely, was the Apple Macintosh. The Macintosh was introduced in 1984 as the successor to the ill-fated Apple Lisa, which was the first computer to come with a **GUI** (**Graphical User Interface**), similar to the now-popular Windows interface. The Lisa failed because it was too expensive, but

the lower-priced Macintosh introduced a year later was a huge success and inspired love and passion among its many admirers.

The early personal computer market also led to the then-unheard of desire for portable computers. At that time, a portable computer made as much sense as a portable refrigerator does now. The first true portable personal computer was the Osborne-1, which at 11 kg was more of a luggable computer than a portable computer. Still, it proved that portables were possible. The Osborne-1 was a modest commercial success, but a year later Compaq brought out its first portable IBM PC clone and was quickly established as the leader in the market for portable computers.

The initial version of the IBM PC came equipped with the MS-DOS operating system supplied by the then-tiny Microsoft Corporation. As Intel was able to produce increasingly powerful CPUs, IBM and Microsoft were able to develop a successor to MS-DOS called OS/2, which featured a graphical user interface, similar to that of the Apple Macintosh. Meanwhile, Microsoft also developed its own operating system, Windows, which ran on top of MS-DOS, just in case OS/2 did not catch on. To make a long story short, OS/2 did not catch on, IBM and Microsoft had a big and extremely public falling out, and Microsoft went on to make Windows a huge success. How tiny Intel and even tinier Microsoft managed to dethrone IBM, one of the biggest, richest, and most powerful corporations in the history of the world, is a parable no doubt related in great detail in business schools around the globe.

With the success of the 8088 in hand, Intel went on to make bigger and better versions of it. Particularly noteworthy was the 80386, released in 1985, which was a 32-bit CPU. This was followed by a souped-up version, naturally called the 80486. Subsequent versions went by the names Pentium and Core. These chips are used in nearly all modern PCs. The generic name many people use to describe the architecture of these processors is **x86**. The compatible chips manufactured by AMD are also called x86s.

By the mid-1980s, a new development called RISC (discussed in Chap. 2) began to take over, replacing complicated (CISC) architectures with much simpler (but faster) ones. In the 1990s, superscalar CPUs began to appear. These machines could execute multiple instructions at the same time, often in a different order than they appeared in the program. We will introduce the concepts of CISC, RISC, and superscalar in Chap. 2 and discuss them at length throughout this book.

Also in the mid-1980s, Ross Freeman with his colleagues at Xilinx developed a clever approach to building integrated circuits that did not require wheelbarrows full of money or access to a silicon fabrication facility. This new kind of computer chip, called a **field-programmable gate array** (**FPGA**), contained a large supply of generic logic gates that could be "programmed" into any circuit that fit into the device. This remarkable new approach to hardware design made FPGA hardware as malleable as software. Using FPGAs that cost tens to hundreds of U.S. dollars, it became possible to build computing systems specialized for unique applications

that served only a few users. Fortunately, silicon fabrication companies could still produce faster, lower-power and less expensive chips for applications that needed millions of chips. But, for applications with only a few users, such as prototyping, low-volume design applications, and education, FPGAs remain a popular tool for building hardware.

Up until 1992, personal computers were either 8-bit, 16-bit, or 32-bit. Then DEC came out with the revolutionary 64-bit Alpha, a true 64-bit RISC machine that outperformed all other personal computers by a wide margin. It had a modest success, but almost a decade elapsed before 64-bit machines began to catch on in a big way, and then mostly as high-end servers.

Throughout the 1990s computing systems were getting faster and faster using a variety of microarchitectural optimizations, many of which we will examine in this book. Users of these systems were pampered by computer vendors, because each new system they bought would run their programs much faster than their old system. However, by the end of the 1990s this trend was beginning to wane because of two important obstacles in computer design: architects were running out of tricks to make programs faster, and the processors were getting too expensive to cool. Desperate to continue building faster processors, most computer companies began turning toward parallel architectures as a way to squeeze out more performance from their silicon. In 2001 IBM introduced the POWER4 dual-core architecture. This was the first time that a mainstream CPU incorporated two processors onto the same die. Today, most desktop and server class processors, and even some embedded processors, incorporate multiple processors on chip. The performance of these multiprocessors has unfortunately been less than stellar for the typical user, because (as we will see in later chapters) parallel machines require programmers to explicitly parallelize programs, which is a difficult and error-prone task.

1.2.6 The Fifth Generation—Low-Power and Invisible Computers

In 1981, the Japanese government announced that they were planning to spend $500 million to help Japanese companies develop fifth-generation computers, which would be based on artificial intelligence and represent a quantum leap over "dumb" fourth-generation computers. Having seen Japanese companies take over the market in many industries, from cameras to stereos to televisions, American and European computer makers went from 0 to full panic in a millisecond, demanding government subsidies and more. Despite lots of fanfare, the Japanese fifth-generation project basically failed and was quietly abandoned. In a sense, it was like Babbage's analytical engine—a visionary idea but so far ahead of its time that the technology for actually building it was nowhere in sight.

Nevertheless, what might be called the fifth generation did happen, but in an unexpected way: computers shrank. In 1989, Grid Systems released the first tablet computer, called the GridPad. It consisted of a small screen on which the users could write with a special pen to control the system. Systems such as the GridPad

showed that computers did not need to sit on a desk or in a server room, but instead, could be put into an easy-to-carry package with touchscreens and handwriting recognition to make them even more valuable.

The Apple Newton, released in 1993, showed that a computer could be built in a package no bigger than a portable audiocassette player. Like the GridPad, the Newton used handwriting for user input, which in this case proved to be a big stumbling block to its success. However, later machines of this class, now called **PDAs** (**Personal Digital Assistants**), have improved user interfaces and are very popular. They have now evolved into **smartphones**.

Eventually, the writing interface of the PDA was perfected by Jeff Hawkins, who had created a company called Palm to develop a low-cost PDA for the mass consumer market. Hawkins was an electrical engineer by training, but he had a keen interest in neuroscience, which is the study of the human brain. He realized that handwriting recognition could be made more reliable by training users to write in a manner that was more easily readable by computers, an input technique he called "Graffiti." It required a small amount of training for the user, but in the end it led to faster and more reliable writing, and the first Palm PDA, called the Palm Pilot, was a huge success. Graffiti is one of the great successes in computing, demonstrating the power of the human mind to take advantage of the power of the human mind.

Users of PDAs swore by the devices, religiously using them to manage their schedules and contacts. When cell phones started gaining popularity in the early 1990s, IBM jumped at the opportunity to integrate the cell phone with the PDA, creating the "smartphone." The first smartphone, called **Simon**, used a touchscreen for input, and it gave the user all of the capabilities of a PDA plus telephone, games, and email. Shrinking component sizes and cost eventually led to the wide use of smartphones, embodied in the popular Apple iPhone and Google Android platforms.

But even the PDAs and smartphones are not really revolutionary. Even more important are the "invisible" computers, which are embedded into appliances, watches, bank cards, and numerous other devices (Bechini et al., 2004). These processors allow increased functionality and lower cost in a wide variety of applications. Whether these chips form a true generation is debatable (they have been around since the 1970s), but they are revolutionizing how thousands of appliances and other devices work. They are already starting to have a major impact on the world and their influence will increase rapidly in the coming years. One unusual aspects of these embedded computers is that the hardware and software are often **codesigned** (Henkel et al., 2003). We will come back to them later in this book.

If we see the first generation as vacuum-tube machines (e.g. ENIAC), the second generation as transistor machines (e.g., the IBM 7094), the third generation as early integrated-circuit machines (e.g., the IBM 360), and the fourth generation as personal computers (e.g., the Intel CPUs), the real fifth generation is more a paradigm shift than a specific new architecture. In the future, computers will be

everywhere and embedded in everything—indeed, invisible. They will be part of the framework of daily life, opening doors, turning on lights, dispensing money, and doing thousands of other things. This model, devised by Mark Weiser, was originally called **ubiquitous computing**, but the term **pervasive computing** is also used frequently now (Weiser, 2002). It will change the world as profoundly as the industrial revolution did. We will not discuss it further in this book, but for more information about it, see Lyytinen and Yoo (2002), Saha and Mukherjee (2003), and Sakamura (2002).

1.3 THE COMPUTER ZOO

In the previous section, we gave a very brief history of computer systems. In this one we will look at the present and gaze toward the future. Although personal computers are the best known computers, there are other kinds of machines around these days, so it is worth taking a brief look at what else is out there.

1.3.1 Technological and Economic Forces

The computer industry is moving ahead like no other. The primary driving force is the ability of chip manufacturers to pack more and more transistors per chip every year. More transistors, which are tiny electronic switches, means larger memories and more powerful processors. Gordon Moore, co-founder and former chairman of Intel, once joked that if aviation technology had moved ahead as fast as computer technology, an airplane would cost $500 and circle the earth in 20 minutes on 5 gallons of fuel. However, it would be the size of a shoebox.

Specifically, while preparing a speech for an industry group, Moore noticed that each new generation of memory chips was being introduced 3 years after the previous one. Since each new generation had four times as much memory as its predecessor, he realized that the number of transistors on a chip was increasing at a constant rate and predicted this growth would continue for decades to come. This observation has become known as **Moore's law**. Today, Moore's law is often expressed as the doubling of the number of transistors every 18 months. Note that this is equivalent to about a 60 percent increase in transistor count per year. The sizes of the memory chips and their dates of introduction shown in Fig. 1-8 confirm that Moore's law has held for over four decades.

Of course, Moore's law is not a law at all, but simply an empirical observation about how fast solid-state physicists and process engineers are advancing the state of the art, and a prediction that they will continue at the same rate in the future. Some industry observers expect Moore's law to continue to hold for at least another decade, maybe longer. Other observers expect energy dissipation, current leakage, and other effects to kick in earlier and cause serious problems that need to be

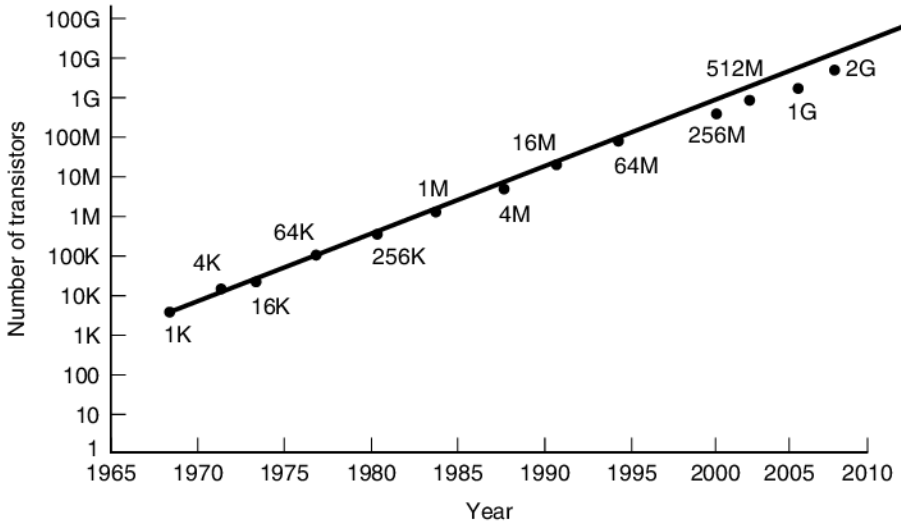


Figure 1-8. Moore's law predicts a 60 percent annual increase in the number of transistors that can be put on a chip. The data points given above and below the line are memory sizes, in bits.

solved (Bose, 2004, Kim et al., 2003). However, the reality of shrinking transistors is that the thickness of these devices is soon to be only a few atoms. At that point transistors will consist of too few atoms to be reliable, or we will simply reach a point where further size decreases will require subatomic building blocks. (As a matter of good advice, it is recommended that anyone working in a silicon fabrication plant take the day off on the day they decide to split the one-atom transistor!) Despite the many challenges in extending Moore's law trends, there are hopeful technologies on the horizon, including advances in quantum computing (Oskin et al., 2002) and carbon nanotubes (Heinze et al., 2002) that may create opportunities to scale electronics beyond the limits of silicon.

Moore's law has created what economists call a **virtuous circle**. Advances in technology (transistors/chip) lead to better products and lower prices. Lower prices lead to new applications (nobody was making video games for computers when computers cost $10 million each although when the price dropped to $120,000 M.I.T. students took up the challenge). New applications lead to new markets and new companies springing up to take advantage of them. The existence of all these companies leads to competition, which in turn creates economic demand for better technologies with which to beat the others. The circle is then round.

Another factor driving technological improvement is Nathan's first law of software (due to Nathan Myhrvold, a former top Microsoft executive). It states: "Software is a gas. It expands to fill the container holding it." Back in the 1980s, word

processing was done with programs like troff (still used for this book). Troff occupies kilobytes of memory. Modern word processors occupy many megabytes of memory. Future ones will no doubt require gigabytes of memory. (To a first approximation, the prefixes kilo, mega, giga, and tera mean thousand, million, billion, and trillion, respectively, but see Sec. 1.5 for details.) Software that continues to acquire features (not unlike boats that continue to acquire barnacles) creates a constant demand for faster processors, bigger memories, and more I/O capacity.

While the gains in transistors per chip have been dramatic over the years, the gains in other computer technologies have been hardly less so. For example, the IBM PC/XT was introduced in 1982 with a 10-megabyte hard disk. Thirty years later, 1-TB hard disks are common on the PC/XT's successors. This improvement of five orders of magnitude in 30 years represents an annual capacity increase of nearly 50 percent. However, measuring disk improvement is trickier, since there are other parameters besides capacity, such as data rate, seek time, and price. Nevertheless, almost any metric will show that the price/performance ratio has increased since 1982 by about 50 percent per year. These enormous gains in disk performance, coupled with the fact that the dollar volume of disks shipped from Silicon Valley has exceeded that of CPU chips, led Al Hoagland to suggest that the place was named wrong: it should have been called Iron Oxide Valley (since this is the recording medium used on disks). Slowly this trend is shifting back in favor of silicon as silicon-based flash memories have begun to replace traditional spinning disks in many systems.

Another area that has seen spectacular gains has been telecommunication and networking. In less than two decades, we have gone from 300 bit/sec modems to analog modems at 56,000 bits/sec to fiber-optic networks at 10^{12} bits/sec. Fiber-optic transatlantic telephone cables, such as TAT-12/13, cost about $700 million, last for 10 years, and can carry 300,000 simultaneous calls, which comes to under 1 cent for a 10-minute intercontinental call. Optical communication systems running at 10^{12} bits/sec over distances exceeding 100 km without amplifiers have been proven feasible. The exponential growth of the Internet hardly needs comment here.

1.3.2 The Computer Spectrum

Richard Hamming, a former researcher at Bell Labs, once observed that a change of an order of magnitude in quantity causes a change in quality. Thus, a racing car that can go 1000 km/hour in the Nevada desert is a fundamentally different kind of machine than a normal car that goes 100 km/hour on a highway. Similarly, a 100-story skyscraper is not just a scaled up 10-story apartment building. And with computers, we are not talking about factors of 10, but over the course of four decades, factors of a million.

The gains afforded by Moore's law can be used by chip vendors in several different ways. One way is to build increasingly powerful computers at constant

price. Another approach is to build the same computer for less and less money every year. The computer industry has done both of these and more, so that a wide variety of computers are available now. A very rough categorization of current computers is given in Fig. 1-9.

Type	Price ($)	Example application
Disposable computer	0.5	Greeting cards
Microcontroller	5	Watches, cars, appliances
Mobile and game computers	50	Home video games and smartphones
Personal computer	500	Desktop or notebook computer
Server	5K	Network server
Mainframe	5M	Batch data processing in a bank

Figure 1-9. The current spectrum of computers available. The prices should be taken with a grain (or better yet, a metric ton) of salt.

In the following sections we will examine each of these categories and discuss their properties briefly.

1.3.3 Disposable Computers

At the bottom end, we find single chips glued to the inside of greeting cards for playing "Happy Birthday," "Here Comes the Bride," or some equally appalling ditty. The authors have not yet spotted a condolence card that plays a funeral dirge, but having now released this idea into the public domain, we expect it shortly. To anyone who grew up with multimillion-dollar mainframes, the idea of disposable computers makes about as much sense as disposable aircraft.

However, disposable computers are here to stay. Probably the most important development in the area of throwaway computers is the **RFID** (**Radio Frequency IDentification**) chip. It is now possible to manufacture, for a few cents, battery-less RFID chips smaller than 0.5 mm on edge that contain a tiny radio transponder and a built-in unique 128-bit number. When pulsed from an external antenna, they are powered by the incoming radio signal long enough to transmit their number back to the antenna. While the chips are tiny, their implications are certainly not.

Let us start with a mundane application: removing bar codes from products. Experimental trials have already been held in which products in stores have RFID chips (instead of bar codes) attached by the manufacturer. Customers select their products, put them in a shopping cart, and just wheel them out of the store, bypassing the checkout counter. At the store's exit, a reader with an antenna sends out a signal asking each product to identify itself, which it does by a short wireless transmission. Customers are also identified by chips on their debit or credit card. At the end of the month, the store sends each customer an itemized bill for this month's purchases. If the customer does not have a valid RFID bank or credit

card, an alarm is sounded. Not only does this system eliminate the need for cashiers and the corresponding wait in line, but it also serves as an antitheft system because hiding a product in a pocket or bag has no effect.

An interesting property of this system is that while bar codes identify the product type, they do not identify the specific item. With 128 bits available, RFID chips do. As a consequence, every package of, say, aspirins, on a supermarket shelf will have a different RFID code. This means that if a drug manufacturer discovers a manufacturing defect in a batch of aspirins after they have been shipped, supermarkets all over the world can be told to sound the alarm when a customer buys any package whose RFID number lies in the affected range, even if the purchase happens in a distant country months later. Aspirins not in the defective batch will not sound the alarm.

But labeling packages of aspirins, cookies, and dog biscuits is only the start. Why stop at labeling the dog biscuits when you can label the dog? Pet owners are already asking veterinarians to implant RFID chips in their animals, allowing them to be traced if they are stolen or lost. Farmers want their livestock tagged as well. The obvious next step is for nervous parents to ask their pediatrician to implant RFID chips in their children in case they get stolen or lost. While we are at it, why not have hospitals put them in all newborns to avoid mixups at the hospital? Governments and the police can no doubt think of many good reasons for tracking all citizens all the time. By now, the "implications" of RFID chips alluded to earlier may be getting a bit clearer.

Another (slightly less controversial) application of RFID chips is vehicle tracking. When a string of railroad cars with embedded RFID chips passes by a reader, the computer attached to the reader then has a list of which cars passed by. This system makes it easy to keep track of the location of all railroad cars, which helps suppliers, their customers, and the railroads. A similar scheme can be applied to trucks. For cars, the idea is already being used to collect tolls electronically (e.g., the E-Z Pass system).

Airline baggage systems and many other package transport systems can also use RFID chips. An experimental system tested at Heathrow airport in London allowed arriving passengers to remove the lugging from their luggage. Bags carried by passengers purchasing this service were tagged with RFID chips, routed separately within the airport, and delivered directly to the passengers' hotels. Other uses of RFID chips include having cars arriving at the painting station of the assembly line specify what color they are supposed to be, studying animal migrations, having clothes tell the washing machine what temperature to use, and many more. Some chips may be integrated with sensors so that the low-order bits may contain the current temperature, pressure, humidity or other environmental variable.

Advanced RFID chips also contain permanent storage. This capability led the European Central Bank to make a decision to put RFID chips in euro banknotes in the coming years. The chips would record where they have been. Not only would

this make counterfeiting euro notes virtually impossible, but it would make tracing kidnapping ransoms, the loot taken from robberies, and laundered money much easier to track and possibly remotely invalidate. When cash is no longer anonymous, standard police procedure in the future may be to check out where the suspect's money has been recently. Who needs to implant chips in people when their wallets are full of them? Again, when the public learns about what RFID chips can do, there is likely to be some public discussion about the matter.

The technology used in RFID chips is developing rapidly. The smallest ones are passive (not internally powered) and capable only of transmitting their unique numbers when queried. However, larger ones are active, can contain a small battery and a primitive computer, and are capable of doing some calculations. Smart cards used in financial transactions fall into this category.

RFID chips differ not only in being active or passive, but also in the range of radio frequencies they respond to. Those operating at low frequencies have a limited data rate but can be sensed at great distances from the antenna. Those operating at high frequencies have a higher data rate and a shorter range. The chips also differ in other ways and are being improved all the time. The Internet is full of information about RFID chips, with *www.rfid.org* being one good starting point.

1.3.4 Microcontrollers

Next up the ladder we have computers that are embedded inside devices that are not sold as computers. The embedded computers, sometimes called **microcontrollers**, manage the devices and handle the user interface. Microcontrollers are found in a large variety of different devices, including the following. Some examples of each category are given in parentheses.

1. Appliances (clock radio, washer, dryer, microwave, burglar alarm).
2. Communications gear (cordless phone, cell phone, fax, pager).
3. Computer peripherals (printer, scanner, modem, CD ROM-drive).
4. Entertainment devices (VCR, DVD, stereo, MP3 player, set-top box).
5. Imaging devices (TV, digital camera, camcorder, lens, photocopier).
6. Medical devices (X-ray, MRI, heart monitor, digital thermometer).
7. Military weapon systems (cruise missile, ICBM, torpedo).
8. Shopping devices (vending machine, ATM, cash register).
9. Toys (talking doll, game console, radio-controlled car or boat).

A car can easily contain 50 microcontrollers, running subsystems including the antilock brakes, fuel injection, radio, lights, and GPS. A jet plane can easily have 200 or more. A family might easily own several hundred computers without even

knowing it. Within a few years, practically everything that runs on electricity or batteries will contain a microcontroller. The number of microcontrollers sold every year dwarfs that of all other kinds of computers except disposable computers by orders of magnitude.

While RFID chips are minimal systems, microcontrollers are small, but complete, computers. Each microcontroller has a processor, memory, and I/O capability. The I/O capability usually includes sensing the device's buttons and switches and controlling the device's lights, display, sound, and motors. In most cases, the software is built into the chip in the form of a read-only memory created when the microcontroller is manufactured. Microcontrollers come in two general types: general purpose and special purpose. The former are just small, but ordinary computers; the latter have an architecture and instruction set tuned to some specific application, such as multimedia. Microcontrollers come in 4-bit, 8-bit, 16-bit, and 32-bit versions.

However, even the general-purpose microcontrollers differ from standard PCs in important ways. First, they are extremely cost sensitive. A company buying millions of units may make the choice based on a 1-cent price difference per unit. This constraint leads manufacturers to make architectural choices based much more on manufacturing costs, a criteria less dominant on chips costing hundreds of dollars. Microcontroller prices vary greatly depending on how many bits wide they are, how much and what kind of memory they have, and other factors. To get an idea, an 8-bit microcontroller purchased in large enough volume can probably be had for as little as 10 cents per unit. This price is what makes it possible to put a computer inside a $9.95 clock radio.

Second, virtually all microcontrollers operate in real time. They get a stimulus and are expected to give an instantaneous response. For example, when the user presses a button, often a light goes on, and there should not be any delay between the button being pressed and the light going on. The need to operate in real time often has impact on the architecture.

Third, embedded systems often have physical constraints in terms of size, weight, battery consumption, and other electrical and mechanical limits. The microcontrollers used in them have to be designed with these restrictions in mind.

One particularly fun application of microcontrollers is in the Arduino embedded control platform. Arduino was designed by Massimo Banzi and David Cuartielles in Ivrea, Italy. Their goal for the project was to produce a complete embedded computing platform that costs less than a large pizza with extra toppings, making it easily accessible to students and hobbyists. (This was a difficult task, because there is a glut of pizzas in Italy, so they are really cheap.) They achieved their goal well: a complete Arduino system costs less than 20 US dollars!

The Arduino system is an open-source hardware design, which means that all its details are published and free so that anyone can build (and even sell) an Arduino system. It is based on the Atmel AVR 8-bit RISC microcontroller, and most board designs also include basic I/O support. The board is programmed using

an embedded programming language called Wiring which has built-in all the bells and whistles required to control real-time devices. What makes the Arduino platform fun to use is its large and active development community. There are thousands of published projects using the Arduino, ranging from an electronic pollutant sniffer, to a biking jacket with turn signals, a moisture detector that sends email when a plant needs to be watered, and an unmanned autonomous airplane. To learn more about the Arduino and get your hands dirty on your own Arduino projects, go to *www.arduino.cc*.

1.3.5 Mobile and Game Computers

A step up are the mobile platforms and video game machines. They are normal computers, often with special graphics and sound capability but with limited software and little extensibility. They started out as low-end CPUs for simple phones and action games like ping pong on TV sets. Over the years they have evolved into far more powerful systems, rivaling or even outperforming personal computers in certain dimensions.

To get an idea of what is inside these systems, consider the specifications of three popular products. First, the Sony PlayStation 3. It contains a 3.2-GHz multicore proprietary CPU (called the Cell microprocessor), which is based on the IBM PowerPC RISC CPU, and seven 128-bit Synergistic Processing Elements (SPEs). The PlayStation 3 also contains 512 MB of RAM, a 550-MHz custom Nvidia graphics chip, and a Blu-ray player. Second, the Microsoft Xbox 360. It contains a 3.2-GHz IBM triple-core PowerPC CPU with 512 MB of RAM, a 500-MHz custom ATI graphics chip, a DVD player, and a hard disk. Third, the Samsung Galaxy Tablet (on which this book was proofread). It contains two 1-GHz ARM cores plus a graphics processing unit (integrated into the Nvidia Tegra 2 system-on-a-chip), 1 GB of RAM, dual cameras, a 3-axis gyroscope, and flash memory storage.

While these machines are not quite as powerful as high-end personal computers produced in the same time period, they are not that far behind, and in some ways they are ahead (e.g., the 128-bit SPE in the PlayStation 3 is wider than the CPU in any PC). The main difference between these machines and a PC is not so much the CPU as it is their being closed systems. Users may not expand them with plug-in cards, although USB or FireWire interfaces are sometimes provided. Also, and perhaps most important, these platforms are carefully optimized for a few application domains: highly interactive applications with 3D graphics and multimedia output. Everything else is secondary. These hardware and software restrictions, lack of extensibility, small memories, absence of a high-resolution monitor, and small (or sometime absent) hard disk make it possible to build and sell these machines more cheaply than personal computers. Despite these restrictions, millions of these devices have been sold and their numbers are growing all the time.

Mobile computers have the added requirement that they use as little energy as possible to perform their tasks. The less energy they use the longer their battery will last. This is a challenging design task because mobile platforms such as tablets and smartphones must be frugal in their energy use, but at the same time, users of these devices expect high-performance capabilities, such as 3D graphics, high-definition multimedia processing, and gaming.

1.3.6 Personal Computers

Next, we come to the personal computers that most people think of when they hear the term "computer." These include desktop and notebook models. They usually come with a few gigabytes of memory, a hard disk holding up to terabytes of data, a CD-ROM/DVD/Blu-ray drive, sound card, network interface, high-resolution monitor, and other peripherals. They have elaborate operating systems, many expansion options, and a huge range of available software.

The heart of every personal computer is a printed circuit board at the bottom or side of the case. It usually contains the CPU, memory, various I/O devices (such as a sound chip and possibly a modem), as well as interfaces to the keyboard, mouse, disk, network, etc., and some expansion slots. A photo of one of these circuit boards is given in Fig. 1-10.

Notebook computers are basically PCs in a smaller package. They use the same hardware components, but manufactured in smaller sizes. They also run the same software as desktop PCs. Since most readers are probably quite familiar with notebook and personal computers, additional introductory material is hardly needed.

Yet another variant on this theme is the tablet computer, such as the popular iPad. These devices are just normal PCs in a smaller package, with a solid-state disk instead of a rotating hard disk, a touch screen, and a different CPU than the x86. But from an architectural perspective, tablets are just notebooks with a different form factor.

1.3.7 Servers

Beefed-up personal computers or workstations are often used as network servers, both for local area networks (typically within a single company), and for the Internet. These come in single-processor and multiple-processor configurations, and have gigabytes of memory, terabytes of hard disk space, and high-speed networking capability. Some of them can handle thousands of transactions per second.

Architecturally, however, a single-processor server is not really very different from a single-processor personal computer. It is just faster, bigger, and has more disk space and possibly a faster network connection. Servers run the same operating systems as personal computers, typically some flavor of UNIX or Windows.

Figure 1-10. A printed circuit board is at the heart of every personal computer. This one is the Intel DQ67SW board. © 2011 Intel Corporation. Used by permission.

Clusters

Owing to almost continuous improvements in the price/performance ratio of servers, in recent years system designers have begun connecting large numbers of them together to form **clusters**. They consist of standard server-class systems connected by gigabit/sec networks and running special software that allow all the machines to work together on a single problem, often in business, science or engineering. Normally, they are what are called **COTS** (**Commodity Off The Shelf**) computers that anyone can buy from a PC vendor. The main addition is high-speed networking, but sometimes that is also a standard commercial network card, too.

Large clusters are typically housed in special-purpose rooms or buildings called **data centers**. Data centers can scale quite large, from a handful of machines to 100,000 or more of them. Usually, the amount of money available is the limiting factor. Owing to their low component price, individual companies can now own such machines for internal use. Many people use the terms "cluster" and "data center" interchangeably although technically the former is the collection of servers and the latter is the room or building.

A common use for a cluster is as an Internet Web server. When a Website expects thousands of requests per second for its pages, the most economical solution

is often to build a data center with hundreds, or even thousands, of servers. The incoming requests are then sprayed among the servers to allow them to be processed in parallel. For example, Google has data centers all over the world to service search requests, the largest one, in The Dalles, Oregon, is a facility that is as large as two (U.S.) football fields. The location was chosen because data centers require vast amounts of electric power and The Dalles is the site of a 2 GW hydroelectric dam on the Columbia River that can provide it. Altogether, Google is thought to have more than 1,000,000 servers in its data centers.

The computer business is a dynamic one, with things changing all the time. In the 1960s, computing was dominated by giant mainframe computers (see below) costing tens of millions of dollars to which users connected using small remote terminals. This was a very centralized model. Then in the 1980s personal computers arrived on the scene, millions of people bought one, and computing was decentralized.

With the advent of data centers, we are starting to relive the past in the form of **cloud computing**, which is mainframe computing V2.0. The idea here is that everyone will have one or more simple devices, including PCs, notebooks, tablets, and smartphones that are essentially user interfaces to the cloud (i.e, the data center) where all the user's photos, videos, music, and other data are stored. In this model, the data are accessible from different devices anywhere and at any time without the user having to keep track of where they are. Here, the data center full of servers has replaced the single large centralized computer, but the paradigm has reverted back to the old one: the users have simple terminals and data and computing power is centralized somewhere else.

Who knows how long this model will be popular? It could easily happen in 10 years that so many people have stored so many songs, photos, and videos in the cloud that the (wireless) infrastructure for communicating with it has become completely bogged down. This could lead to a new revolution: personal computers, where people store their own data on their own machines locally, thus bypassing the traffic jam over the air.

The take-home message here is that the model of computing popular in a given era depends a lot on the technology, economics, and applications available at the time and can change when these factors change.

1.3.8 Mainframes

Now we come to the mainframes: room-sized computers that hark back to the 1960s. These machines are the direct descendants of IBM 360 mainframes acquired decades ago. For the most part, they are not much faster than powerful servers, but they always have more I/O capacity and are often equipped with vast disk farms, often holding thousands of gigabytes of data. While expensive, they are often kept running due to the immense investment in software, data, operating procedures, and personnel that they represent. Many companies find it cheaper to just

pay a few million dollars once in a while for a new one, than to even contemplate the effort required to reprogram all their applications for smaller machines.

It is this class of computer that led to the now-infamous Year 2000 problem, which was caused by (mostly COBOL) programmers in the 1960s and 1970s representing the year as two decimal digits (in order to save memory). They never envisioned their software lasting three or four decades. While the predicted disaster never occurred due to a huge amount of work put into fixing the problem, many companies have repeated the same mistake by simply adding two more digits to the year. The authors hereby predict the end of civilization at midnight on Dec. 31, 9999, when 8000 years worth of old COBOL programs crash simultaneously.

In addition to their use for running 40-year-old legacy software, the Internet has breathed new life into mainframes. They have found a new niche as powerful Internet servers, for example, by handling massive numbers of e-commerce transactions per second, particularly in businesses with huge databases.

Up until recently, there was another category of computers even more powerful than mainframes: **supercomputers**. They had enormously fast CPUs, many gigabytes of main memory, and very fast disks and networks. They were used for massive scientific and engineering calculations such as simulating colliding galaxies, synthesizing new medicines, or modeling the flow of air around an airplane wing. However, in recent years, data centers constructed from commodity components have come to offer as much computing power at much lower prices, and the true supercomputers are now a dying breed.

1.4 EXAMPLE COMPUTER FAMILIES

In this book we will focus on three popular instruction set architectures (ISAs): x86, ARM and AVR. The x86 architecture is found in nearly all personal computers (including Windows and Linux PCs and Macs) and server systems. Personal computers are of interest because every reader has undoubtedly used one. Servers are of interest because they run all the services on the Internet. The ARM architecture dominates the mobile market. For example, most smartphones and tablet computers are based on ARM processors. Finally, the AVR architecture is found in very low-cost microcontrollers found in many embedded computing applications. Embedded computers are invisible to their users but control cars, televisions, microwave ovens, washing machines, and practically every other electrical device costing more than $50. In this section, we will briefly introduce the three instruction set architectures that will be used as examples in the rest of the book.

1.4.1 Introduction to the x86 Architecture

In 1968, Robert Noyce, inventor of the silicon integrated circuit; Gordon Moore, of Moore's law fame; and Arthur Rock, a San Francisco venture capitalist, formed the Intel Corporation to make memory chips. In its first year of operation,

Intel sold only $3000 worth of chips, but business has picked up since then (Intel is now the world's largest CPU chip manufacturer).

In the late 1960s, calculators were large electromechanical machines the size of a modern laser printer and weighing 20 kg. In Sept. 1969, a Japanese company, Busicom, approached Intel with a request that it manufacture 12 custom chips for a proposed electronic calculator. The Intel engineer assigned to this project, Ted Hoff, looked at the plan and realized that he could put a 4-bit general-purpose CPU on a single chip that would do the same thing and be simpler and cheaper as well. Thus, in 1970, the first single-chip CPU, the 2300-transistor 4004, was born (Faggin et al., 1996).

It is worth noting that neither Intel nor Busicom had any idea what they had just done. When Intel decided that it might be worth a try to use the 4004 in other projects, it offered to buy back all the rights to the new chip from Busicom by returning the $60,000 Busicom had paid Intel to develop it. Intel's offer was quickly accepted, at which point it began working on an 8-bit version of the chip, the 8008, introduced in 1972. The Intel family, starting with the 4004 and 8008, is shown in Fig. 1-11, giving the introduction date, clock rate, transistor count, and memory.

Chip	Date	MHz	Trans.	Memory	Notes
4004	4/1971	0.108	2300	640	First microprocessor on a chip
8008	4/1972	0.108	3500	16 KB	First 8-bit microprocessor
8080	4/1974	2	6000	64 KB	First general-purpose CPU on a chip
8086	6/1978	5–10	29,000	1 MB	First 16-bit CPU on a chip
8088	6/1979	5–8	29,000	1 MB	Used in IBM PC
80286	2/1982	8–12	134,000	16 MB	Memory protection present
80386	10/1985	16–33	275,000	4 GB	First 32-bit CPU
80486	4/1989	25–100	1.2M	4 GB	Built-in 8-KB cache memory
Pentium	3/1993	60–233	3.1M	4 GB	Two pipelines; later models had MMX
Pentium Pro	3/1995	150–200	5.5M	4 GB	Two levels of cache built in
Pentium II	5/1997	233–450	7.5M	4 GB	Pentium Pro plus MMX instructions
Pentium III	2/1999	650–1400	9.5M	4 GB	SSE Instructions for 3D graphics
Pentium 4	11/2000	1300–3800	42M	4 GB	Hyperthreading; more SSE instructions
Core Duo	1/2006	1600–3200	152M	2 GB	Dual cores on a single die
Core	7/2006	1200–3200	410M	64 GB	64-bit quad core architecture
Core i7	1/2011	1100–3300	1160M	24 GB	Integrated graphics processor

Figure 1-11. Key members of the Intel CPU family. Clock speeds are measured in MHz (megahertz), where 1 MHz is 1 million cycles/sec.

Intel did not expect much demand for the 8008, so it set up a low-volume production line. Much to everyone's amazement, there was an enormous amount of interest, so Intel set about designing a new CPU chip that got around the 8008's limit

of 16 kilobytes of memory (imposed by the number of pins on the chip). This design resulted in the 8080, a small, general-purpose CPU, introduced in 1974. Much like the PDP-8, this product took the industry by storm and instantly became a mass-market item. Only instead of selling thousands, as DEC had, Intel sold millions.

In 1978 came the 8086, a genuine 16-bit CPU on a single chip. The 8086 was designed to be similar to the 8080, but it was not completely compatible with the 8080. The 8086 was followed by the 8088, which had the same architecture as the 8086 and ran the same programs but had an 8-bit bus instead of a 16-bit bus, making it both slower and cheaper than the 8086. When IBM chose the 8088 as the CPU for the original IBM PC, this chip quickly became the personal computer industry standard.

Neither the 8088 nor the 8086 could address more than 1 megabyte of memory. By the early 1980s this had become a serious problem, so Intel designed the 80286, an upward compatible version of the 8086. The basic instruction set was essentially the same as that of the 8086 and 8088, but the memory organization was quite different, and rather awkward, due to the requirement of compatibility with the older chips. The 80286 was used in the IBM PC/AT and in the midrange PS/2 models. Like the 8088, it was a huge success, mostly because people viewed it as a faster 8088.

The next logical step was a true 32-bit CPU on a chip, the 80386, brought out in 1985. Like the 80286, this one was more or less compatible with everything back to the 8080. Being backward compatible was a boon to people for whom running old software was important, but a nuisance to people who would have preferred a simple, clean, modern architecture unencumbered by the mistakes and technology of the past.

Four years later the 80486 came out. It was essentially a faster version of the 80386 that also had a floating-point unit and 8 kilobytes of cache memory on chip. **Cache memory** is used to hold the most commonly used memory words inside or close to the CPU, in order to avoid (slow) accesses to main memory. The 80486 also had built-in multiprocessor support, allowing manufacturers to build systems containing multiple CPUs sharing a common memory.

At this point, Intel found out the hard way (by losing a trademark infringement lawsuit) that numbers (like 80486) cannot be trademarked, so the next generation got a name: **Pentium** (from the Greek word for five, *πεντε*). Unlike the 80486, which had one internal pipeline, the Pentium had two of them, which helped make it twice as fast (we will discuss pipelines in detail in Chap. 2).

Later in the production run, Intel added special **MMX** (**MultiMedia eXtension**) instructions. These instructions were intended to speed up computations required to process audio and video, making the addition of special multimedia coprocessors unnecessary.

When the next generation appeared, people who were hoping for the Sexium (*sex* is Latin for six) were sorely disappointed. The name Pentium was now so

well known that the marketing people wanted to keep it, and the new chip was called the Pentium Pro. Despite the small name change from its predecessor, this processor represented a major break with the past. Instead, of having two or more pipelines, the Pentium Pro had a very different internal organization and could execute up to five instructions at a time.

Another innovation found in the Pentium Pro was a two-level cache memory. The processor chip itself had 8 kilobytes of fast memory to hold commonly used instructions and another 8 kilobytes of fast memory to hold commonly used data. In the same cavity within the Pentium Pro package (but not on the chip itself) was a second cache memory of 256 kilobytes.

Although the Pentium Pro had a big cache, it lacked the MMX instructions (because Intel was unable to manufacture such a large chip with acceptable yields). When the technology improved enough to get both the MMX instructions and the cache on one chip, the combined product was released as the Pentium II. Next, yet more multimedia instructions, called **SSE** (**Streaming SIMD Extensions**), were added for enhanced 3D graphics (Raman et al., 2000). The new chip was dubbed the Pentium III, but internally it was essentially a Pentium II.

The next Pentium, released in Nov. 2000, was based on a different internal architecture but had the same instruction set as the earlier Pentiums. To celebrate this event, Intel switched from Roman numerals to Arabic numbers and called it the Pentium 4. As usual, the Pentium 4 was faster than all its predecessors. The 3.06-GHz version also introduced an intriguing new feature—hyperthreading. This feature allowed programs to split their work into two threads of control which the Pentium 4 could run in parallel, speeding up execution. In addition, another batch of SSE instructions was added to speed up audio and video processing even more.

In 2006, Intel changed the brand name from Pentium to Core and released a dual core chip, the **Core 2 duo**. When Intel decided it wanted a cheaper single-core version of the chip, it just sold Core 2 duos with one core disabled because wasting a little silicon on each chip manufacturered was ultimately cheaper than incurring the enormous expense of designing and testing a new chip from scratch. The Core series has continued to evolve, with the i3, i5, and i7 being popular variants for low-, medium-, and high-performance computers. No doubt more variants will follow. A photo of the i7 is presented in Fig. 1-12. There are actually eight cores on it, but except in the Xeon version, only six are enabled. This approach means that a chip with one or two defective cores can still be sold by disabling the defective one(s). Each core has its own level 1 and level 2 caches, but there is also a shared level 3 (L3) cache used by all the cores. We will discuss caches in detail later in this book.

In addition to the mainline desktop CPUs discussed so far, Intel has manufactured variants of some of the Pentium chips for special markets. In early 1998, Intel introduced a new product line called the **Celeron**, which was basically a low-price, low-performance version of the Pentium 2 intended for low-end PCs. Since

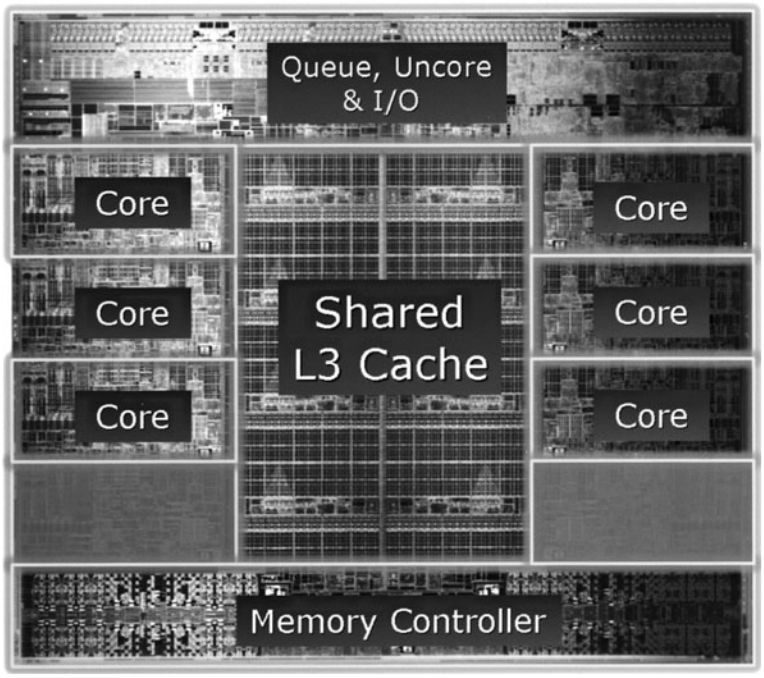


Figure 1-12. The Intel Core i7-3960X die. The die is 21 by 21 mm and has 2.27 billion transistors. © 2011 Intel Corporation. Used by permission.

the Celeron has the same architecture as the Pentium 2, we will not discuss it further in this book. In June 1998, Intel introduced a special version of the Pentium 2 for the upper end of the market. This processor, called the **Xeon**, had a larger cache, a faster bus, and better multiprocessor support but was otherwise a normal Pentium 2, so we will not discuss it separately either. The Pentium III also had a Xeon version as do more recent chips. On more recent chips, one feature of the Xeon is more cores.

1n 2003, Intel introduced the Pentium M (as in Mobile), a chip designed for notebook computers. This chip was part of the Centrino architecture, whose goals were lower power consumption for longer battery lifetime; smaller, lighter, computers; and built-in wireless networking capability using the IEEE 802.11 (WiFi) standard. The Pentium M was very low power and much smaller than the Pentium 4, two characteristics that would soon allow it (and its successors) to subsume the Pentium 4 microarchitecture in future Intel products.

All the Intel chips are backward compatible with their predecessors as far back as the 8086. In other words, a Pentium 4 or Core can run old 8086 programs without modification. This compatibility has always been a design requirement for Intel, to allow users to maintain their existing investment in software. Of course,

the Core is four orders of magnitude more complex than the 8086, so it can do quite a few things that the 8086 could not do. These piecemeal extensions have resulted in an architecture that is not as elegant as it might have been had someone given the Pentium 4 architects 42 million transistors and instructions to start all over again.

It is interesting to note that although Moore's law has long been associated with the number of bits in a memory, it applies equally well to CPU chips. By plotting the transistor counts given in Fig. 1-8 against the date of introduction of each chip on a semilog scale, we see that Moore's law holds here too. This graph is given in Fig. 1-13.

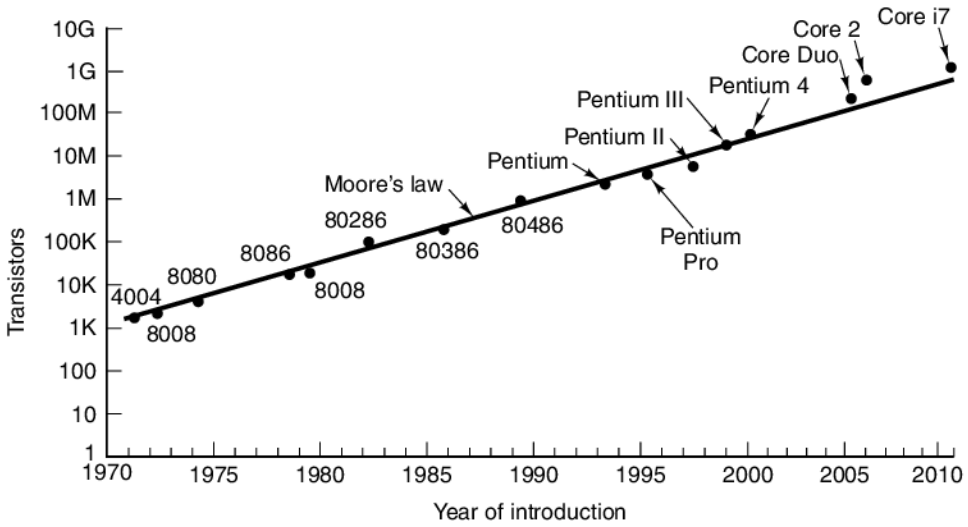


Figure 1-13. Moore's law for (Intel) CPU chips.

While Moore's law will probably continue to hold for some years to come, another problem is starting to overshadow it: heat dissipation. Smaller transistors make it possible to run at higher clock frequencies, which requires using a higher voltage. Power consumed and heat dissipated is proportional to the square of the voltage, so going faster means having more heat to get rid of. At 3.6 GHz, the Pentium 4 consumes 115 watts of power. That means it gets about as hot as a 100-watt light bulb. Speeding up the clock makes the problem worse.

In November 2004, Intel canceled the 4-GHz Pentium 4 due to problems dissipating the heat. Large fans can help but the noise they make is not popular with users, and water cooling, while used on large mainframes, is not an option for desktop machines (and even less so for notebook computers). As a consequence, the once-relentless march of the clock may have ended, at least until Intel's engineers figure out an efficient way to get rid of all the heat generated. Instead, Intel CPU designs now put two or more CPUs on a single chip, along with large shared

cache. Because of the way power consumption is related to voltage and clock speed, two CPUs on a chip consume far less power than one CPU at twice the speed. As a consequence, the gain offered by Moore's law may be increasingly exploited in the future to include more cores and larger on-chip caches, rather than higher and higher clock speeds. Taking advantage of these multiprocessors poses great challenges to programmers, because unlike the sophisticated uniprocessor microarchitectures of the past that could extract more performance from existing programs, multiprocessors require the programmer to explicitly orchestrate parallel execution, using threads, semaphores, shared memory and other headache- and bug-inducing technologies.

1.4.2 Introduction to the ARM Architecture

In the early 80s, the U.K.-based company Acorn Computer, flush with the success of their 8-bit BBC Micro personal computer, began working on a second machine with the hope of competing with the recently released IBM PC. The BBC Micro was based on the 8-bit 6502 processor, and Steve Furber and his colleagues at Acorn felt that the 6502 did not have the muscle to compete with the IBM PC's 16-bit 8086 processor. They began looking at the options in the marketplace, and decided that they were too limited.

Inspired by the Berkeley RISC project, in which a small team designed a remarkably fast processor (which eventually led to the SPARC architecture), they decided to build their own CPU for the project. They called their design the Acorn RISC Machine (or ARM, which would later be rechristened the Advanced RISC machine when ARM eventually split from Acorn). The design was completed in 1985. It included 32-bit instructions and data, and a 26-bit address space, and it was manufactured by VLSI Technology.

The first ARM architecture (called the ARM2) appeared in the Acorn Archimedes personal computer. The Archimedes was a very fast and inexpensive machine for its day, running up to 2 MIPS (millions of instructions per second) and costing only 899 British pounds at launch. The machine became very popular in the UK, Ireland, Australia and New Zealand, especially in schools.

Based on the success of the Archimedes, Apple approached Acorn to develop an ARM processor for their upcoming Apple Newton project, the first palmtop computer. To better focus on the project, the ARM architecture team left Acorn to create a new company called Advanced RISC Machines (ARM). Their new processor was called the ARM 610, which powered the Apple Newton when it was release in 1993. Unlike the original ARM design, this new ARM processor incorporated a 4-KB cache that significantly improved the design's performance. Although the Apple Newton was not a great success, the ARM 610 did see other successful applications including Acorn's RISC PC computer.

In the mid 1990s, ARM collaborated with Digital Equipment Corporation to develop a high-speed, low-power version of the ARM, intended for energy-frugal

mobile applications such as PDAs. They produced the StrongARM design, which from its first appearance sent waves through the industry due to its high speed (233 MHz) and ultralow power demands (1 watt). It gained efficiency through a simple, clean design that included two 16-KB caches for instructions and data. The StrongARM and its successors at DEC were moderately successful in the marketplace, finding their way into a number of PDAs, set-top boxes, media devices, and routers.

Perhaps the most venerable of the ARM architectures is the ARM7 design, first released by ARM in 1994 and still in wide use today. The design included separate instruction and data caches, and it also incorporated the 16-bit Thumb instruction set. The Thumb instruction set is a shorthand version of the full 32-bit ARM instruction set, allowing programmers to encode many of the most common operations into smaller 16-bit instructions, significantly reducing the amount of program memory needed. The processor has worked well for a wide range of low- to middle-end embedded applications such as toasters, engine control, and even the Nintendo Gameboy Advance hand-held gaming console.

Unlike many computer companies, ARM does not manufacture any microprocessors. Instead, it creates designs and ARM-based developer tools and libraries, and licenses them to system designers and chip manufacturers. For example, the CPU used in the Samsung Galaxy Tab Android-based tablet computer is an ARM-based processor. The Galaxy Tab contains the Tegra 2 system-on-chip processor, which includes two ARM Cortex-A9 processors and an Nvidia GeForce graphics processing unit. The Tegra 2 cores were designed by ARM, integrated into a system-on-a-chip design by Nvidia, and manufactured by Taiwan Semiconductor Manufacturing Company (TSMC). It's an impressive collaboration by companies in different countries in which all of the companies contributed value to the final design.

Figure 1-14 shows a die photo of the Nvidia's Tegra 2 system-on-a-chip. The design contains three ARM processors: two 1.2-GHz ARM Cortex-A9 cores plus an ARM7 core. The Cortex-A9 cores are dual-issue out-of-order cores with a 1-MB L2 cache and support for shared-memory multiprocessing. (That's a lot of buzzwords that we will get into in later chapters. For now, just know that these features make the design very fast!) The ARM7 core is an older and smaller ARM core used for system configuration and power management. The graphics core is a 333-MHz GeForce graphics processing unit (GPU) design optimized for low-power operation. Also included on the Tegra 2 are a video encoder/decoder, an audio processor and an HDMI video output interface.

The ARM architecture has found great success in the low-power, mobile and embedded markets. In January 2011, ARM announced that it had sold 15 billion ARM processors since its inception, and indicated that sales were continuing to grow. While tailored for lower-end markets, the ARM architecture does have the computational capability to perform in any market, and there are hints that it may be expanding its horizons. For example, in October 2011, a 64-bit ARM was

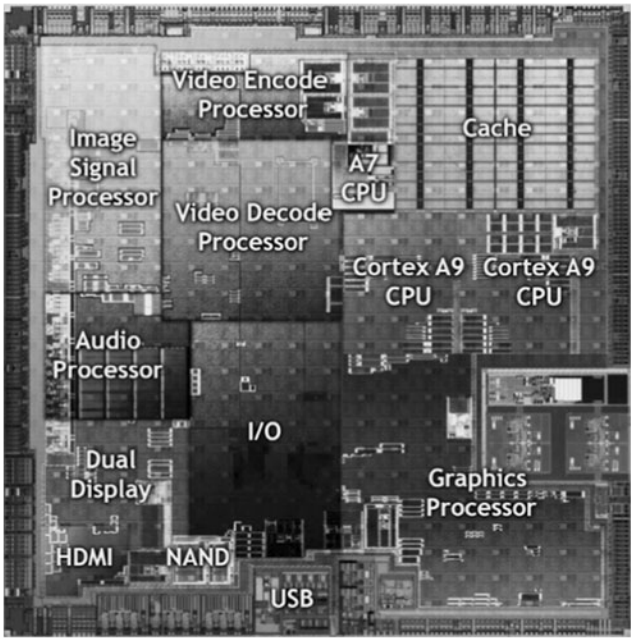


Figure 1-14. The Nvidia Tegra 2 system on a chip. © 2011 Nvidia Corporation. Used by permission.

announced. Also in January 2011, Nvidia announced "Project Denver," an ARM-based system-on-a-chip being developed for the server and other markets. The design will incorporate multiple 64-bit ARM processors plus a general-purpose GPU (GPGPU). The low-power aspects of the design will help to reduce the cooling requirements of server farms and data centers.

1.4.3 Introduction to the AVR Architecture

Our third example is very different from our first (the x86 architecture, used in personal computers and servers) and second (the ARM architecture, used in PDAs and smartphones). It is the AVR architecture, which is used in very low-end embedded systems. The AVR story starts in 1996 at the Norwegian Institute of Technology, where students Alf-Egil Bogen and Vegard Wollan designed an 8-bit RISC CPU called the AVR. It was reportedly given this name because it was "(A)lf and (V)egard's (R)ISC processor." Shortly after the design was completed, Atmel bought the design and started Atmel Norway, where the two architects continued to refine the AVR processor design. Atmel released their first AVR microcontroller, the AT90S1200, in 1997. To ease its adoption by system designers, they implemented the pinout to be exactly the same as that of the Intel 8051, which was one

of the most popular microcontrollers at the time. Today there is much interest in the AVR architecture because it is at the heart of the very popular open-source Arduino embedded controller platform.

The AVR architecture is implemented in three classes of microcontrollers, listed in Fig. 1-15. The lowest class, the tinyAVR is designed for the most area-, power- and cost-constrained applications. It includes an 8-bit CPU, basic digital I/O support, and analog input support (for example, reading temperature values off a thermistor). The tinyAVR is so small that its pins work double duty, such that they can be reprogrammed at run time to any of the digital or analog functions supported by the microcontroller. The megaAVR, which is found in the popular Arduino open-source embedded system, also adds serial I/O support, internal clocks, and programmable analog outputs. The top end of the bottom end is the AVR XMEGA microcontroller, which also incorporates an accelerator for cryptographic operations plus built-in support for USB interfaces.

Chip	Flash	EEPROM	RAM	Pins	Features
tinyAVR	0.5–16 KB	0–512 B	32–512 B	6–32	Tiny, digital I/O, analog input
megaAVR	8–256 KB	0.5–4 KB	0.25–8 KB	28-100	Many peripherals, analog out
AVR XMEGA	16–256 KB	1–4 KB	2–16 KB	44–100	Crypto acceleration, USB I/O

Figure 1-15. Microcontroller classes in the AVR family.

Along with various additional peripherals, each AVR processor class includes some additional memory resources. Microcontrollers typically have three types of memory on board: flash, EEPROM, and RAM. Flash memory is programmable using an external interface and high voltages, and this is where program code and data are stored. Flash RAM is nonvolatile, so even if the system is powered down, the flash memory will remember what was written to it. Like flash, EEPROM is also nonvolatile, but unlike flash RAM, it can be changed by the program while it is running. This is the storage in which an embedded system would keep user configuration information, such as whether your alarm clock displays time in 12- or 24-hour format. Finally, the RAM is where program variables will be stored as the program runs. This memory is volatile, so any value stored here will be lost once the system loses power. We study volatile and nonvolatile RAM types in detail in Chap. 2.

The recipe for success in the microcontroller business is to cram into the chip everything it may possibly need (and the kitchen sink, too, if it can be reduced to a square millimeter) and then put it into an inexpensive and small package with very few pins. By integrating lots of features into the microcontroller, it can work for many applications, and by making it cheap and small, it can serve many form factors. To get a sense of how many things get packed onto a modern microcontroller, let's take a look at the peripherals included in the Atmel megaAVR-168:

1. Three timers (two 8-bit timers and one 16-bit timer).
2. Real-time clock with oscillator.
3. Six pulse-width modulation channels used, for example, to control light intensity or motor speed.
4. Eight analog-to-digital conversion channels used to read voltage levels.
5. Universal serial receiver/transmitter.
6. I2C serial interface, a common standard for interfacing to sensors.
7. Programmable watchdog timer that detects when the system has locked up.
8. On-chip analog comparator that compares two input voltages.
9. Power brown-out detector that interrupts the system when power is failing.
10. Internal programmable clock oscillator to drive the CPU clock.

1.5 METRIC UNITS

To avoid any confusion, it is worth stating explicitly that in this book, as in computer science in general, metric units are used instead of traditional English units (the furlong-stone-fortnight system). The principal metric prefixes are listed in Fig. 1-16. The prefixes are typically abbreviated by their first letters, with the units greater than 1 capitalized (KB, MB, etc.). One exception (for historical reasons) is kbps for kilobits/sec. Thus, a 1-Mbps communication line transmits 10^6 bits/sec and a 100-psec (or 100-ps) clock ticks every 10^{-10} seconds. Since milli and micro both begin with the letter "m," a choice had to be made. Normally, "m" is for milli and "μ" (the Greek letter mu) is for micro.

It is also worth pointing out that in common industry practice for measuring memory, disk, file, and database sizes, the units have slightly different meanings. There, kilo means 2^{10} (1024) rather than 10^3 (1000) because memories are always a power of two. Thus, a 1-KB memory contains 1024 bytes, not 1000 bytes. Similarly, a 1-MB memory contains 2^{20} (1,048,576) bytes, a 1-GB memory contains 2^{30} (1,073,741,824) bytes, and a 1-TB database contains 2^{40} (1,099,511,627,776) bytes.

However, a 1-kbps communication line can transmit 1000 bits per second and a 10-Mbps LAN runs at 10,000,000 bits/sec because these speeds are not powers of two. Unfortunately, many people tend to mix up these two systems, especially for disk sizes.

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	yotta

Figure 1-16. The principal metric prefixes.

To avoid ambiguity, the standards organizations have introduced the new terms kibibyte for 2^{10} bytes, mebibyte for 2^{20} bytes, gibibyte for 2^{30} bytes, and tebibyte for 2^{40} bytes, but the industry has been slow to adopt them. We feel that until these new terms are in wider use, it is better to stick with the symbols KB, MB, GB, and TB for 2^{10}, 2^{20}, 2^{30}, and 2^{40} bytes, respectively, and the symbols kbps, Mbps, Gbps, and Tbps for 10^{3}, 10^{6}, 10^{9}, and 10^{12} bits/sec, respectively.

1.6 OUTLINE OF THIS BOOK

This book is about multilevel computers (which includes nearly all modern computers) and how they are organized. We will examine four levels in considerable detail—namely, the digital logic level, the microarchitecture level, the ISA level, and the operating system machine level. Some of the basic issues to be examined include the overall design of the level (and why it was designed that way), the kinds of instructions and data available, the memory organization and addressing, and the method by which the level is implemented. The study of these topics, and similar ones, is called computer organization or computer architecture.

We are primarily concerned with concepts rather than details or formal mathematics. For that reason, some of the examples given will be highly simplified, in order to emphasize the central ideas and not the details.

To provide some insight into how the principles presented in this book can be, and are, applied in practice, we will use the x86, ARM, and AVR architectures as running examples throughout the book. These three have been chosen for several reasons. First, all are widely used and the reader is likely to have access to at least one of them. Second, each one has its own unique architecture, which provides a basis for comparison and encourages a "what are the alternatives?" attitude. Books dealing with only one machine often leave the reader with a "true machine design revealed" feeling, which is absurd in light of the many compromises and arbitrary decisions that designers are forced to make. The reader is encouraged to

study these and all other computers with a critical eye and to try to understand why things are the way they are, as well as how they could have been done differently, rather than simply accepting them as given.

It should be made clear from the beginning that this is not a book about how to program the x86, ARM, or AVR architectures. These machines will be used for illustrative purposes where appropriate, but we make no pretense of being complete. Readers wishing a thorough introduction to one of them should consult the manufacturer's publications.

Chapter 2 is an introduction to the basic components of a computer—processors, memories, and input/output equipment. It is intended to provide an overview of the system architecture and an introduction to subsequent chapters.

Chapters 3, 4, 5, and 6 each deal with one specific level shown in Fig. 1-2. Our treatment is bottom-up, because machines have traditionally been designed that way. The design of level k is largely determined by the properties of level $k-1$, so it is hard to understand any level unless you already have a good grasp of the underlying level that motivated it. Also, it is educationally sound to proceed from the simpler lower levels to the more complex higher levels rather than vice versa.

Chapter 3 is about the digital logic level, the machine's true hardware. It discusses what gates are and how they can be combined into useful circuits. Boolean algebra, a tool for analyzing digital circuits, is also introduced. Computer buses are explained, especially the popular PCI bus. Numerous examples from industry are discussed in this chapter, including the three running examples mentioned above.

Chapter 4 introduces the architecture of the microarchitecture level and its control. Since the function of this level is to interpret the level 2 instructions in the layer above it, we will concentrate on this topic and illustrate it by means of examples. The chapter also contains discussions of the microarchitecture level of some real machines.

Chapter 5 discusses the ISA level, the one most computer vendors advertise as the machine language. We will look at our example machines here in detail.

Chapter 6 covers some of the instructions, memory organization, and control mechanisms present at the operating system machine level. The examples used here are the Windows operating system (popular on x86 based desktop systems) and UNIX, used on many x86 and ARM based systems.

Chapter 7 is about the assembly language level. It covers both assembly language and the assembly process. The subject of linking also comes up here.

Chapter 8 discusses parallel computers, an increasingly important topic nowadays. Some of these parallel computers have multiple CPUs that share a common memory. Others have multiple CPUs without common memory. Some are supercomputers; some are systems on a chip; others are clusters of computers.

Chapter 9 contains an alphabetical list of literature citations. Suggested readings are on the book's Website. See: *www.prenhall.com/tanenbaum*.

PROBLEMS

1. Explain each of the following terms in your own words:

a. Translator.
b. Interpreter.
c. Virtual machine.

2. Is it conceivable for a compiler to generate output for the microarchitecture level instead of for the ISA level? Discuss the pros and cons of this proposal.

3. Can you imagine any multilevel computer in which the device level and digital logic levels were not the lowest levels? Explain.

4. Consider a multilevel computer in which all the levels are different. Each level has instructions that are m times as powerful as those of the level below it; that is, one level r instruction can do the work of m level $r - 1$ instructions. If a level-1 program requires k seconds to run, how long would equivalent programs take at levels 2, 3, and 4, assuming n level r instructions are required to interpret a single $r + 1$ instruction?

5. Some instructions at the operating system machine level are identical to ISA language instructions. These instructions are carried out directly by the microprogram or hardware rather than by the operating system. In light of your answer to the preceding problem, why do you think this is the case?

6. Consider a computer with identical interpreters at levels 1, 2, and 3. It takes an interpreter n instructions to fetch, examine, and execute one instruction. A level-1 instruction takes k nanoseconds to execute. How long does it take for an instruction at levels 2, 3, and 4?

7. In what sense are hardware and software equivalent? In what sense are they not equivalent?

8. Babbage's difference engine had a fixed program that could not be changed. Is this essentially the same thing as a modern CD-ROM that cannot be changed? Explain your answer.

9. One of the consequences of von Neumann's idea to store the program in memory is that programs can be modified, just like data. Can you think of an example where this facility might have been useful? (*Hint*: Think about doing arithmetic on arrays.)

10. The performance ratio of the 360 model 75 was 50 times that of the 360 model 30, yet the cycle time was only five times as fast. How do you account for this discrepancy?

11. Two system designs are shown in Fig. 1-5 and Fig. 1-6. Describe how input/output might occur in each system. Which one has the potential for better overall system performance?

12. Suppose that each of the 300 million people in the United States fully consumes two packages of goods a day bearing RFID tags. How many RFID tags have to be pro-

duced annually to meet that demand? At a penny a tag, what is the total cost of the tags? Given the size of GDP, is this amount of money going to be an obstacle to their use on every package offered for sale?

13. Name three appliances that are candidates for being run by an embedded CPU.

14. At a certain point in time, a transistor on a chip was 0.1 micron in diameter. According to Moore's law, how big would a transistor be on next year's model?

15. It has been shown that Moore's law not only applies to semiconductor density, but it also predicts the increase in (reasonable) simulation sizes, and the reduction in computational simulation run-times. First show for a fluid mechanics simulation that takes 4 hours to run on a machine today, that it should only take 1 hour to run on machines built 3 years from now, and only 15 minutes on machines built 6 years from now. Then show that for a large simulation that has an estimated run-time of 5 years that it would complete sooner if we waited 3 years to start the simulation.

16. In 1959, an IBM 7090 could execute about 500,000 instructions/sec, had a memory of 32,768 36-bit words, and cost $3 million. Compare this to a current computer and determine how much better the current one is by multiplying the ratio of memory sizes and speeds and then dividing this by the ratio of the prices. Now see what the same gains would have done to aviation in the same time period. The Boeing 707 was delivered to the airlines in substantial quantities in 1959. Its speed was 950 km/hr and its capacity was initially 180 passengers. It cost $4 million. What would the speed, capacity, and cost of an aircraft now be if it had the same gains as a computer? Clearly, state your assumptions about speed, memory size, and price.

17. Developments in the computer industry are often cyclic. Originally, instruction sets were hardwired, then they were microprogrammed, then RISC machines came along and they were hardwired again. Originally, computing was centralized on large mainframe computers. List two developments that demonstrate the cyclic behavior here as well.

18. The legal issue of who invented the computer was settled in April 1973 by Judge Earl Larson, who handled a patent infringement lawsuit filed by the Sperry Rand Corporation, which had acquired the ENIAC patents. Sperry Rand's position was that everybody making a computer owed them royalties because it owned the key patents. The case went to trial in June 1971 and over 30,000 exhibits were entered. The court transcript ran to over 20,000 pages. Study this case more carefully using the extensive information available on the Internet and write a report discussing the technical aspects of the case. What exactly did Eckert and Mauchley patent and why did the judge feel their system was based on Atanasoff's earlier work?

19. Pick the three people you think were most influential in creating modern computer hardware and write a short report describing their contributions and why you picked them.

20. Pick the three people you think were most influential in creating modern computer systems software and write a short report describing their contributions and why you picked them.

21. Pick the three people you think were most influential in creating modern Websites that draw a lot of traffic and write a short report describing their contributions and why you picked them.

2

COMPUTER SYSTEMS ORGANIZATION

A digital computer consists of an interconnected system of processors, memories, and input/output devices. This chapter is an introduction to these three components and to their interconnection, as background for a more detailed examination of the specific levels in the five subsequent chapters. Processors, memories, and input/output are key concepts and will be present at every level, so we will start our study of computer architecture by looking at all three in turn.

2.1 PROCESSORS

The organization of a simple bus-oriented computer is shown in Fig. 2-1. The **CPU** (**Central Processing Unit**) is the "brain" of the computer. Its function is to execute programs stored in the main memory by fetching their instructions, examining them, and then executing them one after another. The components are connected by a **bus**, which is a collection of parallel wires for transmitting address, data, and control signals. Buses can be external to the CPU, connecting it to memory and I/O devices, but also internal to the CPU, as we will see shortly. Modern computers have multiple buses.

The CPU is composed of several distinct parts. The control unit is responsible for fetching instructions from main memory and determining their type. The arithmetic logic unit performs operations such as addition and Boolean AND needed to carry out the instructions.

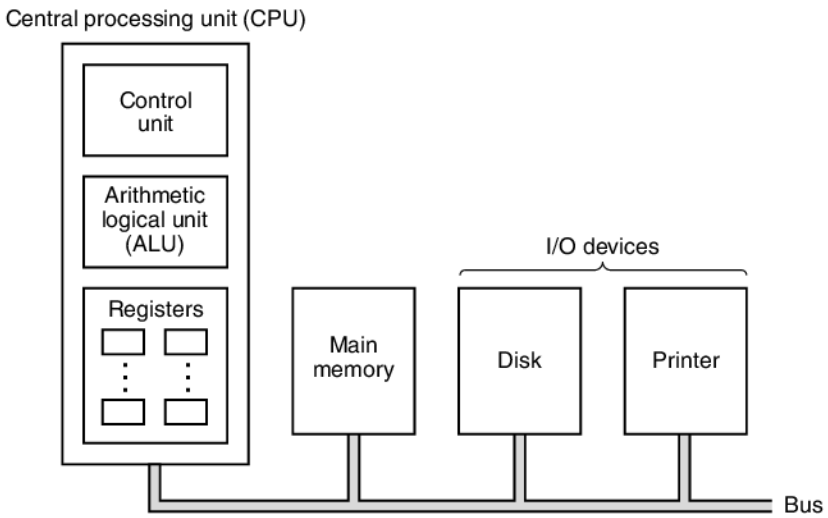


Figure 2-1. The organization of a simple computer with one CPU and two I/O devices.

The CPU also contains a small, high-speed memory used to store temporary results and certain control information. This memory is made up of a number of registers, each having has a certain size and function. Usually, all the registers have the same size. Each register can hold one number, up to some maximum determined by its size. Registers can be read and written at high speed since they are internal to the CPU.

The most important register is the **Program Counter** (**PC**), which points to the next instruction to be fetched for execution. (The name "program counter" is somewhat misleading because it has nothing to do with *counting* anything, but the term is universally used.) Also important is the **Instruction Register** (**IR**), which holds the instruction currently being executed. Most computers have numerous other registers as well, some of them general purpose as well as some for specific purposes. Yet other registers are used by the operating system to control the computer.

2.1.1 CPU Organization

The internal organization of part of a simple von Neumann CPU is shown in Fig. 2-2 in more detail. This part is called the **data path** and consists of the registers (typically 1 to 32), the **ALU** (**Arithmetic Logic Unit**), and several buses connecting the pieces. The registers feed into two ALU input registers, labeled *A* and *B* in the figure. These registers hold the ALU input while the ALU is performing

some computation. The data path is important in all machines and we will discuss it at great length throughout this book.

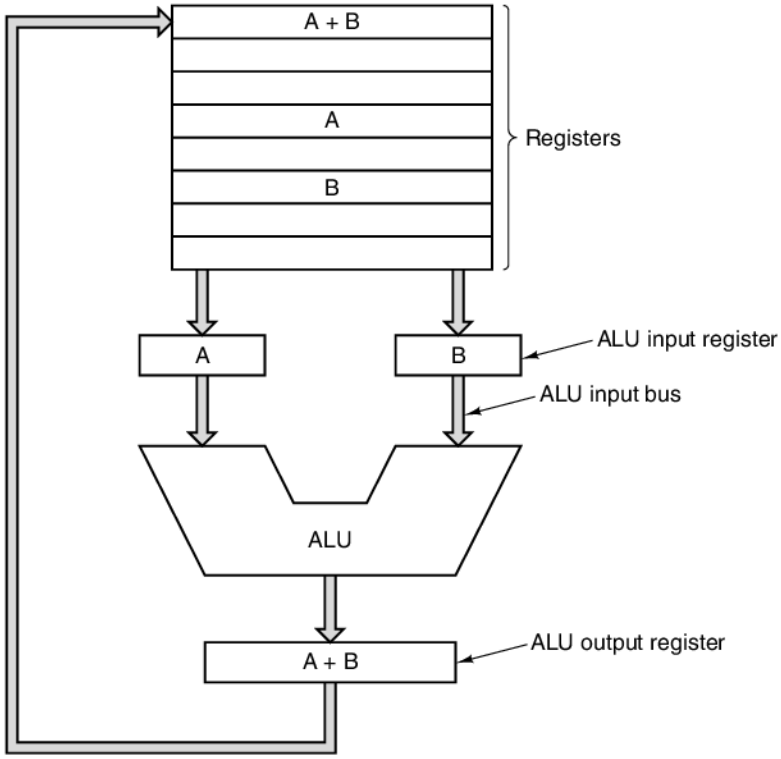


Figure 2-2. The data path of a typical von Neumann machine.

The ALU itself performs addition, subtraction, and other simple operations on its inputs, thus yielding a result in the output register. This output register can be stored back into a register. Later on, the register can be written (i.e., stored) into memory, if desired. Not all designs have the *A*, *B*, and output registers. In the example, addition is illustrated, but ALUs can also perform other operations.

Most instructions can be divided into one of two categories: register-memory or register-register. Register-memory instructions allow memory words to be fetched into registers, where, for example, they can be used as ALU inputs in subsequent instructions. ("Words" are the units of data moved between memory and registers. A word might be an integer. We will discuss memory organization later in this chapter.) Other register-memory instructions allow registers to be stored back into memory.

The other kind of instruction is register-register. A typical register-register instruction fetches two operands from the registers, brings them to the ALU input registers, performs some operation on them (such as addition or Boolean AND),

and stores the result back in one of the registers. The process of running two operands through the ALU and storing the result is called the **data path cycle** and is the heart of most CPUs. To a considerable extent, it defines what the machine can do. Modern computers have multiple ALUs operating in parallel and specialized for different functions. The faster the data path cycle is, the faster the machine runs.

2.1.2 Instruction Execution

The CPU executes each instruction in a series of small steps. Roughly speaking, the steps are as follows:

1. Fetch the next instruction from memory into the instruction register.
2. Change the program counter to point to the following instruction.
3. Determine the type of instruction just fetched.
4. If the instruction uses a word in memory, determine where it is.
5. Fetch the word, if needed, into a CPU register.
6. Execute the instruction.
7. Go to step 1 to begin executing the following instruction.

This sequence of steps is frequently referred to as the **fetch-decode-execute** cycle. It is central to the operation of all computers.

This description of how a CPU works closely resembles a program written in English. Figure 2-3 shows this informal program rewritten as a Java method (i.e., procedure) called *interpret*. The machine being interpreted has two registers visible to user programs: the program counter (PC), for keeping track of the address of the next instruction to be fetched, and the accumulator (AC), for accumulating arithmetic results. It also has internal registers for holding the current instruction during its execution (instr), the type of the current instruction (instr_type), the address of the instruction's operand (data_loc), and the current operand itself (data). Instructions are assumed to contain a single memory address. The memory location addressed contains the operand, for example, the data item to add to the accumulator.

The very fact that it is possible to write a program that can imitate the function of a CPU shows that a program need not be executed by a "hardware" CPU consisting of a box full of electronics. Instead, a program can be carried out by having another program fetch, examine, and execute its instructions. A program (such as the one in Fig. 2-3) that fetches, examines, and executes the instructions of another program is called an **interpreter**, as mentioned in Chap. 1.

```
public class Interp {
   static int PC;                          // program counter holds address of next instr
   static int AC;                          // the accumulator, a register for doing arithmetic
   static int instr;                       // a holding register for the current instruction
   static int instr_type;                  // the instruction type (opcode)
   static int data_loc;                    // the address of the data, or –1 if none
   static int data;                        // holds the current operand
   static boolean run_bit = true;          // a bit that can be turned off to halt the machine

   public static void interpret(int memory[], int starting_address) {
      // This procedure interprets programs for a simple machine with instructions having
      // one memory operand. The machine has a register AC (accumulator), used for
      // arithmetic. The ADD instruction adds an integer in memory to the AC, for example.
      // The interpreter keeps running until the run bit is turned off by the HALT instruction.
      // The state of a process running on this machine consists of the memory, the
      // program counter, the run bit, and the AC. The input parameters consist of
      // the memory image and the starting address.

      PC = starting_address;
      while (run_bit) {
         instr = memory[PC];               // fetch next instruction into instr
         PC = PC + 1;                      // increment program counter
         instr_type = get_instr_type(instr);   // determine instruction type
         data_loc = find_data(instr, instr_type);   // locate data (–1 if none)
         if (data_loc >= 0)                // if data_loc is –1, there is no operand
            data = memory[data_loc];       // fetch the data
         execute(instr_type, data);        // execute instruction
      }

   }

   private static int get_instr_type(int addr) { ... }
   private static int find_data(int instr, int type) { ... }
   private static void execute(int type, int data) { ... }
}
```

Figure 2-3. An interpreter for a simple computer (written in Java).

This equivalence between hardware processors and interpreters has important implications for computer organization and the design of computer systems. After having specified the machine language, *L*, for a new computer, the design team can decide whether they want to build a hardware processor to execute programs in *L* directly or whether they want to write an interpreter to interpret programs in *L* instead. If they choose to write an interpreter, they must also provide some hardware machine to run the interpreter. Certain hybrid constructions are also possible, with some hardware execution as well as some software interpretation.

An interpreter breaks the instructions of its target machine into small steps. As a consequence, the machine on which the interpreter runs can be much simpler and less expensive than a hardware processor for the target machine would be. This

saving is especially significant if the target machine has a large number of instructions and they are fairly complicated, with many options. The saving comes essentially from the fact that hardware is being replaced by software (the interpreter) and it costs more to replicate hardware than software.

Early computers had small, simple sets of instructions. But the quest for more powerful computers led, among other things, to more powerful individual instructions. Very early on, it was discovered that more complex instructions often led to faster program execution even though individual instructions might take longer to execute. A floating-point instruction is an example of a more complex instruction. Direct support for accessing array elements is another. Sometimes it was as simple as observing that the same two instructions often occurred consecutively, so a single instruction could accomplish the work of both.

The more complex instructions were better because the execution of individual operations could sometimes be overlapped or otherwise executed in parallel using different hardware. For expensive, high-performance computers, the cost of this extra hardware could be readily justified. Thus expensive, high-performance computers came to have many more instructions than lower-cost ones. However, instruction compatibility requirements and the rising cost of software development created the need to implement complex instructions even on low-end computers where cost was more important than speed.

By the late 1950s, IBM (then the dominant computer company) had recognized that supporting a single family of machines, all of which executed the same instructions, had many advantages, both for IBM and for its customers. IBM introduced the term **architecture** to describe this level of compatibility. A new family of computers would have one architecture but many different implementations that could all execute the same program, differing only in price and speed. But how to build a low-cost computer that could execute all the complicated instructions of high-performance, expensive machines?

The answer lay in interpretation. This technique, first suggested by Maurice Wilkes (1951), permitted the design of simple, lower-cost computers that could nevertheless execute a large number of instructions. The result was the IBM System/360 architecture, a compatible family of computers, spanning nearly two orders of magnitude, in both price and capability. A direct hardware (i.e., not interpreted) implementation was used only on the most expensive models.

Simple computers with interpreted instructions also some had other benefits. Among the most important were

1. The ability to fix incorrectly implemented instructions in the field, or even make up for design deficiencies in the basic hardware.
2. The opportunity to add new instructions at minimal cost, even after delivery of the machine.
3. Structured design that permitted efficient development, testing, and documenting of complex instructions.

As the market for computers exploded dramatically in the 1970s and computing capabilities grew rapidly, the demand for low-cost computers favored designs of computers using interpreters. The ability to tailor the hardware and the interpreter for a particular set of instructions emerged as a highly cost-effective design for processors. As the underlying semiconductor technology advanced rapidly, the advantages of the cost outweighed the opportunities for higher performance, and interpreter-based architectures became the conventional way to design computers. Nearly all new computers designed in the 1970s, from minicomputers to mainframes, were based on interpretation.

By the late 70s, the use of simple processors running interpreters had become very widespread except among the most expensive, highest-performance models, such as the Cray-1 and the Control Data Cyber series. The use of an interpreter eliminated the inherent cost limitations of complex instructions so designers began to explore much more complex instructions, particularly the ways to specify the operands to be used.

This trend reached its zenith with Digital Equipment Corporation's VAX computer, which had several hundred instructions and more than 200 different ways of specifying the operands to be used in each instruction. Unfortunately, the VAX architecture was conceived from the beginning to be implemented with an interpreter, with little thought given to the implementation of a high-performance model. This mind set resulted in the inclusion of a very large number of instructions which were of marginal value and difficult to execute directly. This omission proved to be fatal to the VAX, and ultimately to DEC as well (Compaq bought DEC in 1998 and Hewlett-Packard bought Compaq in 2001).

Though the earliest 8-bit microprocessors were very simple machines with very simple instruction sets, by the late 70s, even microprocessors had switched to interpreter-based designs. During this period, one of the biggest challenges facing microprocessor designers was dealing with the growing complexity made possible by integrated circuits. A major advantage of the interpreter-based approach was the ability to design a simple processor, with the complexity largely confined to the memory holding the interpreter. Thus a complex hardware design could be turned into a complex software design.

The success of the Motorola 68000, which had a large interpreted instruction set, and the concurrent failure of the Zilog Z8000 (which had an equally large instruction set, but without an interpreter) demonstrated the advantages of an interpreter for bringing a new microprocessor to market quickly. This success was all the more surprising given Zilog's head start (the Z8000's predecessor, the Z80, was far more popular than the 68000's predecessor, the 6800). Of course, other factors were instrumental here, too, not the least of which was Motorola's long history as a chip manufacturer and Exxon's (Zilog's owner) long history of being an oil company, not a chip manufacturer.

Another factor working in favor of interpretation during that era was the existence of fast read-only memories, called **control stores**, to hold the interpreters.

Suppose that a typical interpreted instruction took the interpreter 10 instructions, called **microinstructions**, at 100 nsec each, and two references to main memory, at 500 nsec each. Total execution time was then 2000 nsec, only a factor-of-two worse than the best that direct execution could achieve. Had the control store not been available, the instruction would have taken 6000 nsec. A factor-of-six penalty is a lot harder to swallow than a factor-of-two penalty.

2.1.3 RISC versus CISC

During the late 70s there was experimentation with very complex instructions, made possible by the interpreter. Designers tried to close the "semantic gap" between what machines could do and what high-level programming languages required. Hardly anyone thought about designing simpler machines, just as now not a lot of research goes into designing less powerful spreadsheets, networks, Web servers, etc. (perhaps unfortunately).

One group that bucked the trend and tried to incorporate some of Seymour Cray's ideas in a high-performance minicomputer was led by John Cocke at IBM. This work led to an experimental minicomputer, named the **801**. Although IBM never marketed this machine and the results were not published until years later (Radin, 1982), word got out and other people began investigating similar architectures.

In 1980, a group at Berkeley led by David Patterson and Carlo Séquin began designing VLSI CPU chips that did not use interpretation (Patterson, 1985, Patterson and Séquin, 1982). They coined the term **RISC** for this concept and named their CPU chip the RISC I CPU, followed shortly by the RISC II. Slightly later, in 1981, across the San Francisco Bay at Stanford, John Hennessy designed and fabricated a somewhat different chip he called the **MIPS** (Hennessy, 1984). These chips evolved into commercially important products, the SPARC and the MIPS, respectively.

These new processors were significantly different than commercial processors of the day. Since they did not have to be backward compatible with existing products, their designers were free to choose new instruction sets that would maximize total system performance. While the initial emphasis was on simple instructions that could be executed quickly, it was soon realized that designing instructions that could be **issued** (started) quickly was the key to good performance. How long an instruction actually took mattered less than how many could be started per second.

At the time these simple processors were being first designed, the characteristic that caught everyone's attention was the relatively small number of instructions available, typically around 50. This number was far smaller than the 200 to 300 on established computers such as the DEC VAX and the large IBM mainframes. In fact, the acronym RISC stands for **Reduced Instruction Set Computer**, which was contrasted with CISC, which stands for **Complex Instruction Set Computer** (a thinly veiled reference to the VAX, which dominated university

Computer Science Departments at the time). Nowadays, few people think that the size of the instruction set is a major issue, but the name stuck.

To make a long story short, a great religious war ensued, with the RISC supporters attacking the established order (VAX, Intel, large IBM mainframes). They claimed that the best way to design a computer was to have a small number of simple instructions that execute in one cycle of the data path of Fig. 2-2 by fetching two registers, combining them somehow (e.g., adding or ANDing them), and storing the result back in a register. Their argument was that even if a RISC machine takes four or five instructions to do what a CISC machine does in one instruction, if the RISC instructions are 10 times as fast (because they are not interpreted), RISC wins. It is also worth pointing out that by this time the speed of main memories had caught up to the speed of read-only control stores, so the interpretation penalty had greatly increased, strongly favoring RISC machines.

One might think that given the performance advantages of RISC technology, RISC machines (such as the Sun UltraSPARC) would have mowed down CISC machines (such as the Intel Pentium) in the marketplace. Nothing like this has happened. Why not?

First of all, there is the issue of backward compatibility and the billions of dollars companies have invested in software for the Intel line. Second, surprisingly, Intel has been able to employ the same ideas even in a CISC architecture. Starting with the 486, the Intel CPUs contain a RISC core that executes the simplest (and typically most common) instructions in a single data path cycle, while interpreting the more complicated instructions in the usual CISC way. The net result is that common instructions are fast and less common instructions are slow. While this hybrid approach is not as fast as a pure RISC design, it gives competitive overall performance while still allowing old software to run unmodified.

2.1.4 Design Principles for Modern Computers

Now that more than two decades have passed since the first RISC machines were introduced, certain design principles have come to be accepted as a good way to design computers given the current state of the hardware technology. If a major change in technology occurs (e.g., a new manufacturing process suddenly makes memory cycle time 10 times faster than CPU cycle time), all bets are off. Thus machine designers should always keep an eye out for technological changes that may affect the balance among the components.

That said, there is a set of design principles, sometimes called the **RISC design principles**, that architects of new general-purpose CPUs do their best to follow. External constraints, such as the requirement of being backward compatible with some existing architecture, often require compromises from time to time, but these principles are goals that most designers strive to meet. Next we will discuss the major ones.

All Instructions Are Directly Executed by Hardware

All common instructions are directly executed by the hardware. They are not interpreted by microinstructions. Eliminating a level of interpretation provides high speed for most instructions. For computers that implement CISC instruction sets, the more complex instructions may be broken into separate parts, which can then be executed as a sequence of microinstructions. This extra step slows the machine down, but for less frequently occurring instructions it may be acceptable.

Maximize the Rate at Which Instructions Are Issued

Modern computers resort to many tricks to maximize their performance, chief among which is trying to start as many instructions per second as possible. After all, if you can issue 500 million instructions/sec, you have built a 500-MIPS processor, no matter how long the instructions actually take to complete. (**MIPS** stands for Millions of Instructions Per Second. The MIPS processor was so named as to be a pun on this acronym. Officially it stands for Microprocessor without Interlocked Pipeline Stages.) This principle suggests that parallelism can play a major role in improving performance, since issuing large numbers of slow instructions in a short time interval is possible only if multiple instructions can execute at once.

Although instructions are always encountered in program order, they are not always issued in program order (because some needed resource might be busy) and they need not finish in program order. Of course, if instruction 1 sets a register and instruction 2 uses that register, great care must be taken to make sure that instruction 2 does not read the register until it contains the correct value. Getting this right requires a lot of bookkeeping but has the potential for performance gains by executing multiple instructions at once.

Instructions Should Be Easy to Decode

A critical limit on the rate of issue of instructions is decoding individual instructions to determine what resources they need. Anything that can aid this process is useful. That includes making instructions regular, of fixed length, and with a small number of fields. The fewer different formats for instructions, the better.

Only Loads and Stores Should Reference Memory

One of the simplest ways to break operations into separate steps is to require that operands for most instructions come from—and return to—CPU registers. The operation of moving operands from memory into registers can be performed in separate instructions. Since access to memory can take a long time, and the delay is unpredictable, these instructions can best be overlapped with other instructions assuming they do nothing except move operands between registers and memory.

This observation means that only LOAD and STORE instructions should reference memory. All other instructions should operate only on registers.

Provide Plenty of Registers

Since accessing memory is relatively slow, many registers (at least 32) need to be provided, so that once a word is fetched, it can be kept in a register until it is no longer needed. Running out of registers and having to flush them back to memory only to later reload them is undesirable and should be avoided as much as possible. The best way to accomplish this is to have enough registers.

2.1.5 Instruction-Level Parallelism

Computer architects are constantly striving to improve performance of the machines they design. Making the chips run faster by increasing their clock speed is one way, but for every new design, there is a limit to what is possible by brute force at that moment in history. Consequently, most computer architects look to parallelism (doing two or more things at once) as a way to get even more performance for a given clock speed.

Parallelism comes in two general forms, namely, instruction-level parallelism and processor-level parallelism. In the former, parallelism is exploited within individual instructions to get more instructions/sec out of the machine. In the latter, multiple CPUs work together on the same problem. Each approach has its own merits. In this section we will look at instruction-level parallelism; in the next one, we will look at processor-level parallelism.

Pipelining

It has been known for years that the actual fetching of instructions from memory is a major bottleneck in instruction execution speed. To alleviate this problem, computers going back at least as far as the IBM Stretch (1959) have had the ability to fetch instructions from memory in advance, so they would be there when they were needed. These instructions were stored in a special set of registers called the **prefetch buffer**. This way, when an instruction was needed, it could usually be taken from the prefetch buffer rather than waiting for a memory read to complete.

In effect, prefetching divides instruction execution into two parts: fetching and actual execution. The concept of a **pipeline** carries this strategy much further. Instead of being divided into only two parts, instruction execution is often divided into many (often a dozen or more) parts, each one handled by a dedicated piece of hardware, all of which can run in parallel.

Figure 2-4(a) illustrates a pipeline with five units, also called **stages**. Stage 1 fetches the instruction from memory and places it in a buffer until it is needed. Stage 2 decodes the instruction, determining its type and what operands it needs.

Stage 3 locates and fetches the operands, either from registers or from memory. Stage 4 actually does the work of carrying out the instruction, typically by running the operands through the data path of Fig. 2-2. Finally, stage 5 writes the result back to the proper register.

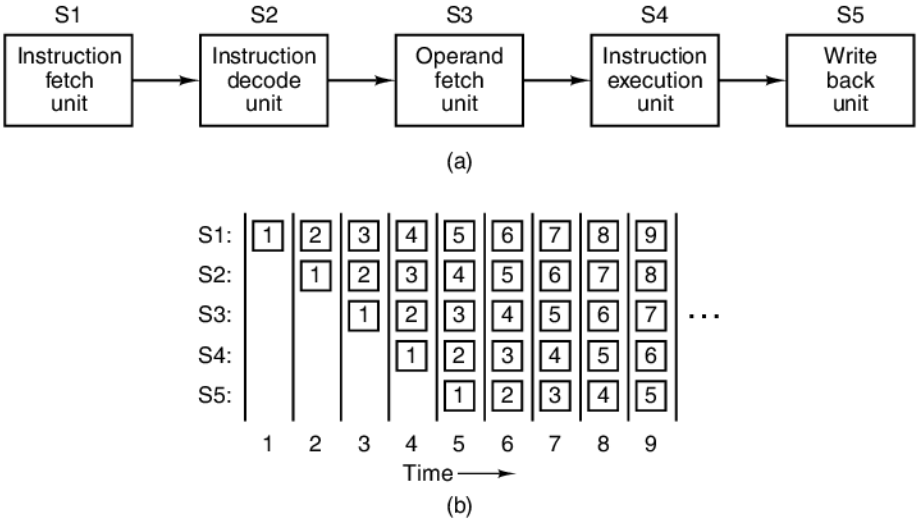


Figure 2-4. (a) A five-stage pipeline. (b) The state of each stage as a function of time. Nine clock cycles are illustrated.

In Fig. 2-4(b) we see how the pipeline operates as a function of time. During clock cycle 1, stage S1 is working on instruction 1, fetching it from memory. During cycle 2, stage S2 decodes instruction 1, while stage S1 fetches instruction 2. During cycle 3, stage S3 fetches the operands for instruction 1, stage S2 decodes instruction 2, and stage S1 fetches the third instruction. During cycle 4, stage S4 executes instruction 1, S3 fetches the operands for instruction 2, S2 decodes instruction 3, and S1 fetches instruction 4. Finally, in cycle 5, S5 writes the result of instruction 1 back, while the other stages work on the following instructions.

Let us consider an analogy to clarify the concept of pipelining. Imagine a cake factory in which the baking of the cakes and the packaging of the cakes for shipment are separated. Suppose that the shipping department has a long conveyor belt with five workers (processing units) lined up along it. Every 10 sec (the clock cycle), worker 1 places an empty cake box on the belt. The box is carried down to worker 2, who places a cake in it. A little later, the box arrives at worker 3's station, where it is closed and sealed. Then it continues to worker 4, who puts a label on the box. Finally, worker 5 removes the box from the belt and puts it in a large container for later shipment to a supermarket. Basically, this is the way computer pipelining works, too: each instruction (cake) goes through several processing steps before emerging completed at the far end.

Getting back to our pipeline of Fig. 2-4, suppose that the cycle time of this machine is 2 nsec. Then it takes 10 nsec for an instruction to progress all the way through the five-stage pipeline. At first glance, with an instruction taking 10 nsec, it might appear that the machine can run at 100 MIPS, but in fact it does much better than this. At every clock cycle (2 nsec), one new instruction is completed, so the actual rate of processing is 500 MIPS, not 100 MIPS.

Pipelining allows a trade-off between **latency** (how long it takes to execute an instruction), and **processor bandwidth** (how many MIPS the CPU has). With a cycle time of T nsec, and n stages in the pipeline, the latency is nT nsec because each instruction passes through n stages, each of which takes T nsec.

Since one instruction completes every clock cycle and there are $10^9/T$ clock cycles/second, the number of instructions executed per second is $10^9/T$. For example, if $T = 2$ nsec, 500 million instructions are executed each second. To get the number of MIPS, we have to divide the instruction execution rate by 1 million to get $(10^9/T)/10^6 = 1000/T$ MIPS. Theoretically, we could measure instruction execution rate in BIPS instead of MIPS, but nobody does that, so we will not either.

Superscalar Architectures

If one pipeline is good, then surely two pipelines are better. One possible design for a dual pipeline CPU, based on Fig. 2-4, is shown in Fig. 2-5. Here a single instruction fetch unit fetches pairs of instructions together and puts each one into its own pipeline, complete with its own ALU for parallel operation. To be able to run in parallel, the two instructions must not conflict over resource usage (e.g., registers), and neither must depend on the result of the other. As with a single pipeline, either the compiler must guarantee this situation to hold (i.e., the hardware does not check and gives incorrect results if the instructions are not compatible), or conflicts must be detected and eliminated during execution using extra hardware.

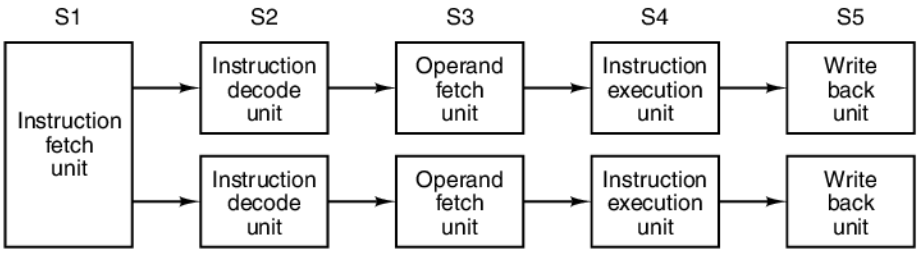


Figure 2-5. Dual five-stage pipelines with a common instruction fetch unit.

Although pipelines, single or double, were originally used on RISC machines (the 386 and its predecessors did not have any), starting with the 486 Intel began

introducing data pipelines into its CPUs. The 486 had one pipeline and the original Pentium had two five-stage pipelines roughly as in Fig. 2-5, although the exact division of work between stages 2 and 3 (called decode-1 and decode-2) was slightly different than in our example. The main pipeline, called the **u pipeline**, could execute an arbitrary Pentium instruction. The second pipeline, called the **v pipeline**, could execute only simple integer instructions (and also one simple floating-point instruction—FXCH).

Fixed rules determined whether a pair of instructions were compatible so they could be executed in parallel. If the instructions in a pair were not simple enough or incompatible, only the first one was executed (in the u pipeline). The second one was then held and paired with the instruction following it. Instructions were always executed in order. Thus Pentium-specific compilers that produced compatible pairs could produce faster-running programs than older compilers. Measurements showed that a Pentium running code optimized for it was exactly twice as fast on integer programs as a 486 running at the same clock rate (Pountain, 1993). This gain could be attributed entirely to the second pipeline.

Going to four pipelines is conceivable, but doing so duplicates too much hardware (computer scientists, unlike folklore specialists, do not believe in the number three). Instead, a different approach is used on high-end CPUs. The basic idea is to have just a single pipeline but give it multiple functional units, as shown in Fig. 2-6. For example, the Intel Core architecture has a structure similar to this figure. It will be discussed in Chap. 4. The term **superscalar architecture** was coined for this approach in 1987 (Agerwala and Cocke, 1987). Its roots, however, go back more than 40 years to the CDC 6600 computer. The 6600 fetched an instruction every 100 nsec and passed it off to one of 10 functional units for parallel execution while the CPU went off to get the next instruction.

The definition of "superscalar" has evolved somewhat over time. It is now used to describe processors that issue multiple instructions—often four or six—in a single clock cycle. Of course, a superscalar CPU must have multiple functional units to hand all these instructions to. Since superscalar processors generally have one pipeline, they tend to look like Fig. 2-6.

Using this definition, the 6600 was technically not superscalar because it issued only one instruction per cycle. However, the effect was almost the same: instructions were issued at a much higher rate than they could be executed. The conceptual difference between a CPU with a 100-nsec clock that issues one instruction every cycle to a group of functional units and a CPU with a 400-nsec clock that issues four instructions per cycle to the same group of functional units is very small. In both cases, the key idea is that the issue rate is much higher than the execution rate, with the workload being spread across a collection of functional units.

Implicit in the idea of a superscalar processor is that the S3 stage can issue instructions considerably faster than the S4 stage is able to execute them. If the S3 stage issued an instruction every 10 nsec and all the functional units could do their work in 10 nsec, no more than one would ever be busy at once, negating the whole

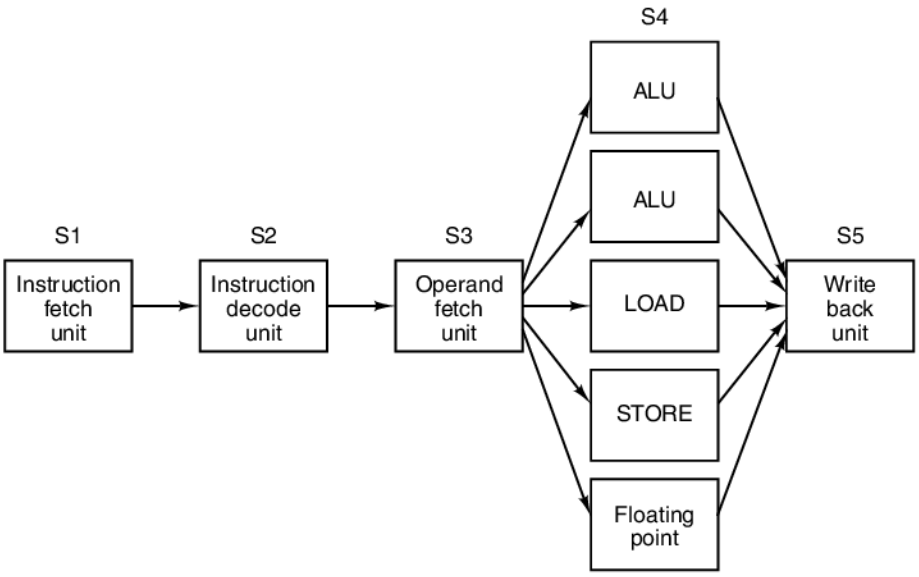


Figure 2-6. A superscalar processor with five functional units.

idea. In reality, most of the functional units in stage 4 take appreciably longer than one clock cycle to execute, certainly the ones that access memory or do floating-point arithmetic. As can be seen from the figure, it is possible to have multiple ALUs in stage S4.

2.1.6 Processor-Level Parallelism

The demand for ever faster computers seems to be insatiable. Astronomers want to simulate what happened in the first microsecond after the big bang, economists want to model the world economy, and teenagers want to play 3D interactive multimedia games over the Internet with their virtual friends. While CPUs keep getting faster, eventually they are going to run into the problems with the speed of light, which is likely to stay at 20 cm/nanosecond in copper wire or optical fiber, no matter how clever Intel's engineers are. Faster chips also produce more heat, whose dissipation is a huge problem. In fact, the difficulty of getting rid of the heat produced is the main reason CPU clock speeds have stagnated in the past decade.

Instruction-level parallelism helps a little, but pipelining and superscalar operation rarely win more than a factor of five or ten. To get gains of 50, 100, or more, the only way is to design computers with multiple CPUs, so we will now take a look at how some of these are organized.

Data Parallel Computers

A substantial number of problems in computational domains such as the physical sciences, engineering, and computer graphics involve loops and arrays, or otherwise have a highly regular structure. Often the same calculations are performed repeatedly on many different sets of data. The regularity and structure of these programs makes them especially easy targets for speed-up through parallel execution. Two primary methods have been used to execute these highly regular programs quickly and efficiently: SIMD processors and vector processors. While these two schemes are remarkably similar in most ways, ironically, the first is generally thought of as a parallel computer while the second is considered an ex extension to a single processor.

Data parallel computers have found many successful applications as a consequence of their remarkable efficiency. They are able to produce significant computational power with fewer transistors than alternative approaches. Gordon Moore (of Moore's law) famously noted that silicon costs about $1 billion per acre (4047 square meters). Thus, the more computational muscle that can be squeezed out of that acre of silicon, the more money a computer company can make selling silicon. Data parallel processors are one of the most efficient means to squeeze performance out of silicon. Because all of the processors are running the same instruction, the system needs only one "brain" controlling the computer. Consequently, the processor needs only one fetch stage, one decode stage, and one set of control logic. This is a huge saving in silicon that gives data parallel computers a big edge over other processors, as long as the software they are running is highly regular with lots of parallelism.

A **Single Instruction-stream Multiple Data-stream** or **SIMD processor** consists of a large number of identical processors that perform the same sequence of instructions on different sets of data. The world's first SIMD processor was the University of Illinois ILLIAC IV computer (Bouknight et al., 1972). The original ILLIAC IV design consisted of four quadrants, each quadrant having an 8×8 square grid of processor/memory elements. A single control unit per quadrant broadcast a single instruction to all processors, which was executed by all processors in lockstep each using its own data from its own memory. Owing to funding constraints only one 50 megaflops (million floating-point operations per second) quadrant was ever built; had the entire 1-gigaflop machine been completed, it would have doubled the computing power of the entire world.

Modern graphics processing units (GPUs) heavily rely on SIMD processing to provide massive computational power with few transistors. Graphics processing lends itself to SIMD processors because most of the algorithms are highly regular, with repeated operations on pixels, vertices, textures, and edges. Fig. 2-7 shows the SIMD processor at the core of the Nvidia Fermi GPU. A Fermi GPU contains up to 16 SIMD stream multiprocessors (SM), with each SM containing 32 SIMD processors. Each cycle, the scheduler selects two threads to execute on the SIMD

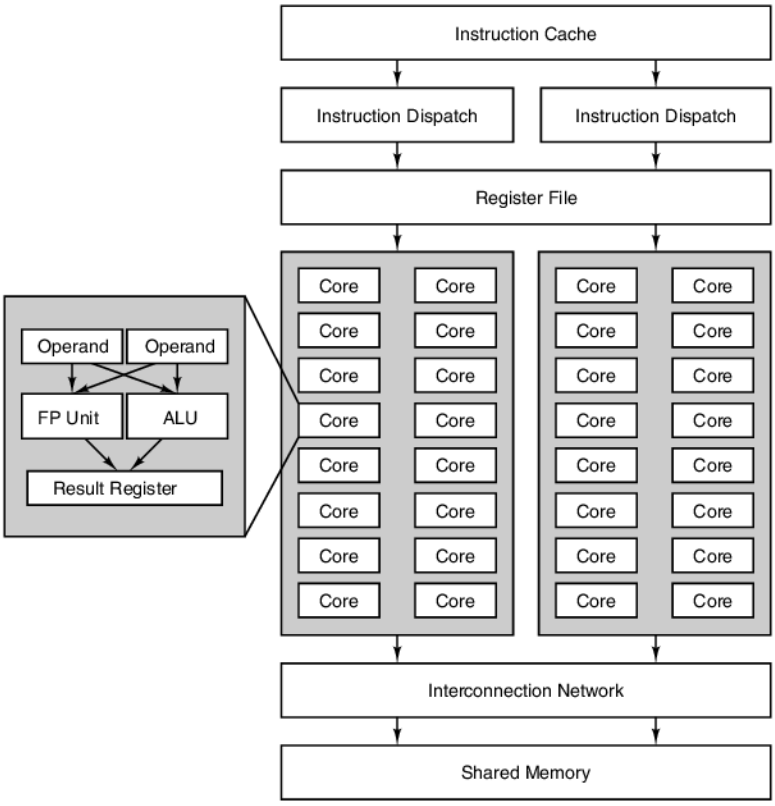


Figure 2-7. The SIMD core of the Fermi graphics processing unit.

processor. The next instruction from each thread then executes on up to 16 SIMD processors, although possibly fewer if there is not enough data parallelism. If each thread is able to perform 16 operations per cycle, a fully loaded Fermi GPU core with 32 SMs will perform a whopping 512 operations per cycle. This is an impressive feat considering that a similar-sized general purpose quad-core CPU would struggle to achieve 1/32 as much processing.

A **vector processor** appears to the programmer very much like a SIMD processor. Like a SIMD processor, it is very efficient at executing a sequence of operations on pairs of data elements. But unlike a SIMD processor, all of the operations are performed in a single, heavily pipelined functional unit. The company Seymour Cray founded, Cray Research, produced many vector processors, starting with the Cray-1 back in 1974 and continuing through current models.

Both SIMD processors and vector processors work on arrays of data. Both execute single instructions that, for example, add the elements together pairwise for two vectors. But while the SIMD processor does it by having as many adders as elements in the vector, the vector processor has the concept of a **vector register**,

which consists of a set of conventional registers that can be loaded from memory in a single instruction, which actually loads them from memory serially. Then a vector addition instruction performs the pairwise addition of the elements of two such vectors by feeding them to a pipelined adder from the two vector registers. The result from the adder is another vector, which can either be stored into a vector register or used directly as an operand for another vector operation. The SSE (Streaming SIMD Extension) instructions available on the Intel Core architecture use this execution model to speed up highly regular computation, such as multimedia and scientific software. In this respect, the Intel Core architecture has the ILLIAC IV as one of its ancestors.

Multiprocessors

The processing elements in a data parallel processor are not independent CPUs, since there is only one control unit shared among all of them. Our first parallel system with multiple full-blown CPUs is the **multiprocessor**, a system with more than one CPU sharing a common memory, like a group of people in a room sharing a common blackboard. Since each CPU can read or write any part of memory, they must coordinate (in software) to avoid getting in each other's way. When two or more CPUs have the ability to interact closely, as is the case with multiprocessors, they are said to be tightly coupled.

Various implementation schemes are possible. The simplest one is to have a single bus with multiple CPUs and one memory all plugged into it. A diagram of such a bus-based multiprocessor is shown in Fig. 2-8(a).

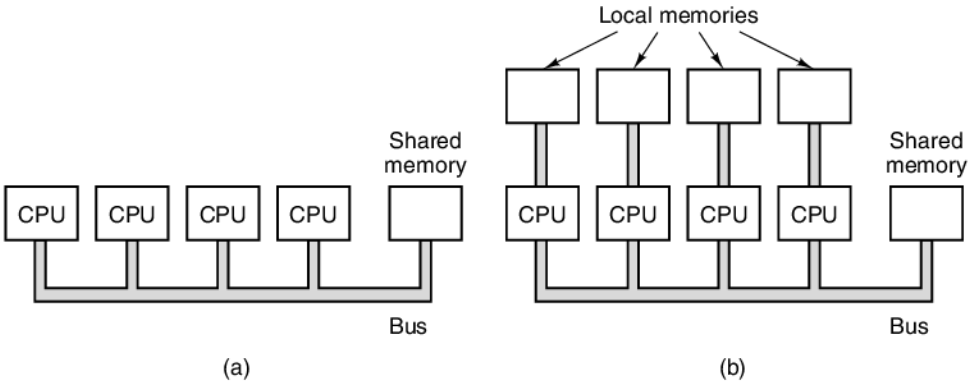


Figure 2-8. (a) A single-bus multiprocessor. (b) A multicomputer with local memories.

It does not take much imagination to realize that with a large number of fast processors constantly trying to access memory over the same bus, conflicts will result. Multiprocessor designers have come up with various schemes to reduce this

contention and improve performance. One design, shown in Fig. 2-8(b), gives each processor some local memory of its own, not accessible to the others. This memory can be used for program code and those data items that need not be shared. Access to this private memory does not use the main bus, greatly reducing bus traffic. Other schemes (e.g., caching—see below) are also possible.

Multiprocessors have the advantage over other kinds of parallel computers that the programming model of a single shared memory is easy to work with. For example, imagine a program looking for cancer cells in a photograph of some tissue taken through a microscope. The digitized photograph could be kept in the common memory, with each processor assigned some region of the photograph to hunt in. Since each processor has access to the entire memory, studying a cell that starts in its assigned region but straddles the boundary into the next region is no problem.

Multicomputers

Although multiprocessors with a modest number of processors (≤ 256) are relatively easy to build, large ones are surprisingly difficult to construct. The difficulty is in connecting so many the processors to the memory. To get around these problems, many designers have simply abandoned the idea of having a shared memory and just build systems consisting of large numbers of interconnected computers, each having its own private memory, but no common memory. These systems are called **multicomputers**. The CPUs in a multicomputer are said to be **loosely coupled**, to contrast them with the **tightly coupled** multiprocessor CPUs.

The CPUs in a multicomputer communicate by sending each other messages, something like email, but much faster. For large systems, having every computer connected to every other computer is impractical, so topologies such as 2D and 3D grids, trees, and rings are used. As a result, messages from one computer to another often must pass through one or more intermediate computers or switches to get from the source to the destination. Nevertheless, message-passing times on the order of a few microseconds can be achieved without much difficulty. Multicomputers with over 250,000 CPUs, such as IBM's Blue Gene/P, have been built.

Since multiprocessors are easier to program and multicomputers are easier to build, there is much research on designing hybrid systems that combine the good properties of each. Such computers try to present the illusion of shared memory without going to the expense of actually constructing it. We will go into multiprocessors and multicomputers in detail in Chap. 8.

2.2 PRIMARY MEMORY

The **memory** is that part of the computer where programs and data are stored. Some computer scientists (especially British ones) use the term **store** or **storage** rather than memory, although more and more, the term "storage" is used to refer

to disk storage. Without a memory from which the processors can read and write information, there would be no stored-program digital computers.

2.2.1 Bits

The basic unit of memory is the binary digit, called a **bit**. A bit may contain a 0 or a 1. It is the simplest possible unit. (A device capable of storing only zeros could hardly form the basis of a memory system; at least two values are needed.)

People often say that computers use binary arithmetic because it is "efficient." What they mean (although they rarely realize it) is that digital information can be stored by distinguishing between different values of some continuous physical quantity, such as voltage or current. The more values that must be distinguished, the less separation between adjacent values, and the less reliable the memory. The binary number system requires only two values to be distinguished. Consequently, it is the most reliable method for encoding digital information. If you are not familiar with binary numbers, see Appendix A.

Some computers, such as the large IBM mainframes, are advertised as having decimal as well as binary arithmetic. This trick is accomplished by using 4 bits to store one decimal digit using a code called **BCD** (**Binary Coded Decimal**). Four bits provide 16 combinations, used for the 10 digits 0 through 9, with six combinations not used. The number 1944 is shown below encoded in decimal and in pure binary, using 16 bits in each example:

decimal: 0001 1001 0100 0100 binary: 0000011110011000

Sixteen bits in the decimal format can store the numbers from 0 to 9999, giving only 10,000 combinations, whereas a 16-bit pure binary number can store 65,536 different combinations. For this reason, people say that binary is more efficient.

Consider, however, what would happen if some brilliant young electrical engineer invented a highly reliable electronic device that could directly store the digits 0 to 9 by dividing the region from 0 to 10 volts into 10 intervals. Four of these devices could store any decimal number from 0 to 9999. Four such devices would provide 10,000 combinations. They could also be used to store binary numbers, by only using 0 and 1, in which case, four of them could store only 16 combinations. With such devices, the decimal system would obviously be more efficient.

2.2.2 Memory Addresses

Memories consist of a number of **cells** (or **locations**), each of which can store a piece of information. Each cell has a number, called its **address**, by which programs can refer to it. If a memory has n cells, they will have addresses 0 to $n - 1$. All cells in a memory contain the same number of bits. If a cell consists of k bits,

it can hold any one of 2^k different bit combinations. Figure 2-9 shows three different organizations for a 96-bit memory. Note that adjacent cells have consecutive addresses (by definition).

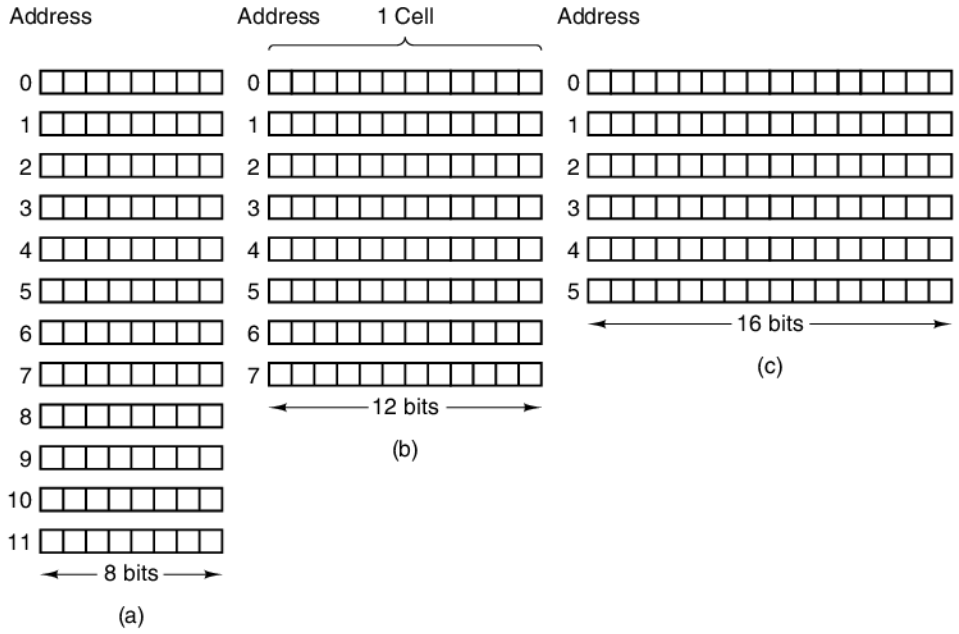


Figure 2-9. Three ways of organizing a 96-bit memory.

Computers that use the binary number system (including octal and hexadecimal notation for binary numbers) express memory addresses as binary numbers. If an address has m bits, the maximum number of cells addressable is 2^m. For example, an address used to reference the memory of Fig. 2-9(a) needs at least 4 bits in order to express all the numbers from 0 to 11. A 3-bit address is sufficient for Fig. 2-9(b) and (c), however. The number of bits in the address determines the maximum number of directly addressable cells in the memory and is independent of the number of bits per cell. A memory with 2^{12} cells of 8 bits each and a memory with 2^{12} cells of 64 bits each need 12-bit addresses.

The number of bits per cell for some computers that have been sold commercially is listed in Fig. 2-10.

The significance of the cell is that it is the smallest addressable unit. In recent years, nearly all computer manufacturers have standardized on an 8-bit cell, which is called a **byte**. The term **octet** is also used. Bytes are grouped into **words**. A computer with a 32-bit word has 4 bytes/word, whereas a computer with a 64-bit word has 8 bytes/word. The significance of a word is that most instructions operate on entire words, for example, adding two words together. Thus a 32-bit machine will have 32-bit registers and instructions for manipulating 32-bit words,

Computer	Bits/cell
Burroughs B1700	1
IBM PC	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Cyber	60

Figure 2-10. Number of bits per cell for some historically interesting commercial computers.

whereas a 64-bit machine will have 64-bit registers and instructions for moving, adding, subtracting, and otherwise manipulating 64-bit words.

2.2.3 Byte Ordering

The bytes in a word can be numbered from left to right or right to left. At first it might seem that this choice is unimportant, but as we shall see shortly, it has major implications. Figure 2-11(a) depicts part of the memory of a 32-bit computer whose bytes are numbered from left to right, such as the SPARC or the big IBM mainframes. Figure 2-11(b) gives the analogous representation of a 32-bit computer using right-to-left numbering, such as the Intel family. The former system, where the numbering begins at the "big" (i.e., high-order) end is called a **big endian** computer, in contrast to the **little endian** of Fig. 2-11(b). These terms are due to Jonathan Swift, whose *Gulliver's Travels* satirized politicians who made war over their dispute about whether eggs should be broken at the big end or the little end. The term was first used in computer architecture in a delightful article by Cohen (1981).

It is important to understand that in both the big endian and little endian systems, a 32-bit integer with the numerical value of, say, 6, is represented by the bits 110 in the rightmost (low-order) 3 bits of a word and zeros in the leftmost 29 bits. In the big endian scheme, the 110 bits are in byte 3 (or 7, or 11, etc.), whereas in the little endian scheme they are in byte 0 (or 4, or 8, etc.). In both cases, the word containing this integer has address 0.

If computers stored only integers, there would be no problem. However, many applications require a mixture of integers, character strings, and other data types. Consider, for example, a simple personnel record consisting of a string (employee

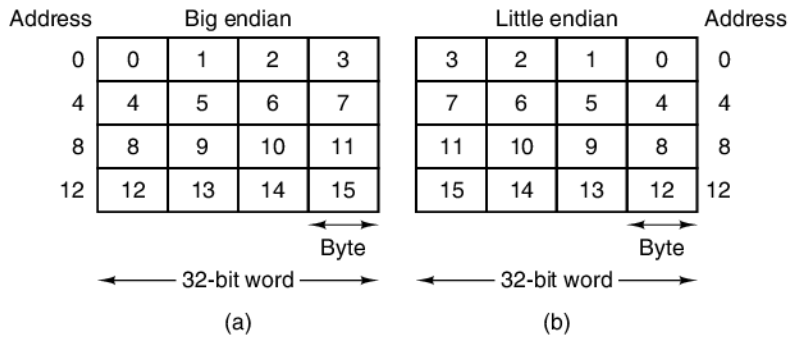


Figure 2-11. (a) Big endian memory. (b) Little endian memory.

name) and two integers (age and department number). The string is terminated with 1 or more 0 bytes to fill out a word. The big endian representation is shown in Fig. 2-12(a); the little endian representation is shown in Fig. 2-12(b) for Jim Smith, age 21, department 260 ($1 \times 256 + 4 = 260$).

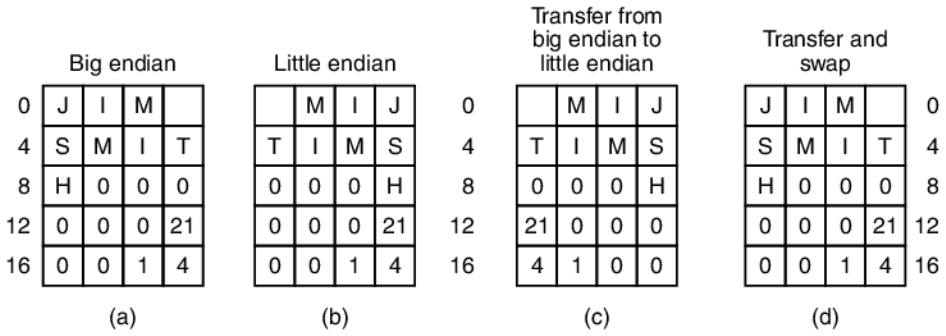


Figure 2-12. (a) A personnel record for a big endian machine. (b) The same record for a little endian machine. (c) The result of transferring the record from a big endian to a little endian. (d) The result of byte swapping (c).

Both of these representations are fine and internally consistent. The problems begin when one of the machines tries to send the record to the other one over a network. Let us assume that the big endian sends the record to the little endian one byte at a time, starting with byte 0 and ending with byte 19. (We will be optimistic and assume the bits of the bytes are not reversed by the transmission, as we have enough problems as is.) Thus the big endian's byte 0 goes into the little endian's memory at byte 0, and so on, as shown in Fig. 2-12(c).

When the little endian tries to print the name, it works fine, but the age comes out as 21×2^{24} and the department is just as garbled. This situation arises because

the transmission has reversed the order of the characters in a word, as it should, but it has also reversed the bytes in an integer, which it should not.

An obvious solution is to have the software reverse the bytes within a word after the copy has been made. Doing this leads to Fig. 2-12(d) which makes the two integers fine but turns the string into "MIJTIMS" with the "H" hanging in the middle of nowhere. This reversal of the string occurs because when reading it, the computer first reads byte 0 (a space), then byte 1 (M), and so on.

There is no simple solution. One way that works, but is inefficient, is to include a header in front of each data item telling what kind of data follows (string, integer, or other) and how long it is. This allows the receiver to perform only the necessary conversions. In any event, it should be clear that the lack of a standard for byte ordering is a big nuisance when moving data between different machines.

2.2.4 Error-Correcting Codes

Computer memories occasionally make errors due to voltage spikes on the power line, cosmic rays, or other causes. To guard against such errors, some memories use error-detecting or error-correcting codes. When these codes are used, extra bits are added to each memory word in a special way. When a word is read out of memory, the extra bits are checked to see if an error has occurred.

To understand how errors can be handled, it is necessary to look closely at what an error really is. Suppose that a memory word consists of m data bits to which we will add r redundant, or check, bits. Let the total length be n (i.e., $n = m + r$). An n-bit unit containing m data and r check bits is often referred to as an n-bit **codeword**.

Given any two codewords, say, 10001001 and 10110001, it is possible to determine how many corresponding bits differ. In this case, 3 bits differ. To determine how many bits differ, just compute the bitwise Boolean EXCLUSIVE OR of the two codewords and count the number of 1 bits in the result. The number of bit positions in which two codewords differ is called the **Hamming distance** (Hamming, 1950). Its main significance is that if two codewords are a Hamming distance d apart, it will require d single-bit errors to convert one into the other. For example, the codewords 11110001 and 00110000 are a Hamming distance 3 apart because it takes 3 single-bit errors to convert one into the other.

With an m-bit memory word, all 2^m bit patterns are legal, but due to the way the check bits are computed, only 2^m of the 2^n codewords are valid. If a memory read turns up an invalid codeword, the computer knows that a memory error has occurred. Given the algorithm for computing the check bits, it is possible to construct a complete list of the legal codewords, and from this list find the two codewords whose Hamming distance is minimum. This distance is the Hamming distance of the complete code.

The error-detecting and error-correcting properties of a code depend on its Hamming distance. To detect d single-bit errors, you need a distance $d + 1$ code

because with such a code there is no way that d single-bit errors can change a valid codeword into another valid codeword. Similarly, to correct d single-bit errors, you need a distance $2d + 1$ code because that way the legal codewords are so far apart that even with d changes, the original codeword is still closer than any other codeword, so it can be uniquely determined.

As a simple example of an error-detecting code, consider a code in which a single **parity bit** is appended to the data. The parity bit is chosen so that the number of 1 bits in the codeword is even (or odd). Such a code has a distance 2, since any single-bit error produces a codeword with the wrong parity. In other words, it takes two single-bit errors to go from a valid codeword to another valid codeword. It can be used to detect single errors. Whenever a word containing the wrong parity is read from memory, an error condition is signaled. The program cannot continue, but at least no incorrect results are computed.

As a simple example of an error-correcting code, consider a code with only four valid codewords:

0000000000, 0000011111, 1111100000, and 1111111111

This code has a distance 5, which means that it can correct double errors. If the codeword 0000000111 arrives, the receiver knows that the original must have been 0000011111 (if there was no more than a double error). If, however, a triple error changes 0000000000 into 0000000111, the error cannot be corrected.

Imagine that we want to design a code with m data bits and r check bits that will allow all single-bit errors to be corrected. Each of the 2^m legal memory words has n illegal codewords at a distance 1 from it. These are formed by systematically inverting each of the n bits in the n-bit codeword formed from it. Thus each of the 2^m legal memory words requires $n + 1$ bit patterns dedicated to it (for the n possible errors and correct pattern). Since the total number of bit patterns is 2^n, we must have $(n + 1)2^m \leq 2^n$. Using $n = m + r$, this requirement becomes $(m + r + 1) \leq 2^r$. Given m, this puts a lower limit on the number of check bits needed to correct single errors. Figure 2-13 shows the number of check bits required for various memory word sizes.

Word size	Check bits	Total size	Percent overhead
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2

Figure 2-13. Number of check bits for a code that can correct a single error.

This theoretical lower limit can be achieved using a method due to Richard Hamming (1950). Before taking a look at Hamming's algorithm, let us look at a simple graphical representation that clearly illustrates the idea of an error-correcting code for 4-bit words. The Venn diagram of Fig. 2-14(a) contains three circles, *A*, *B*, and *C*, which together form seven regions. As an example, let us encode the 4-bit memory word 1100 in the regions *AB*, *ABC*, *AC*, and *BC*, 1 bit per region (in alphabetical order). This encoding is shown in Fig. 2-14(a).

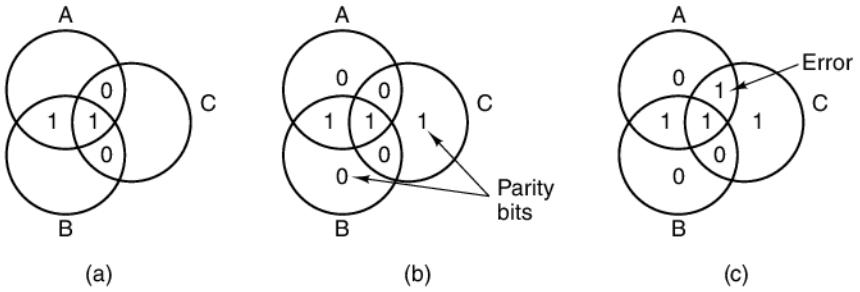


Figure 2-14. (a) Encoding of 1100. (b) Even parity added. (c) Error in *AC*.

Next we add a parity bit to each of the three empty regions to produce even parity, as illustrated in Fig. 2-14(b). By definition, the sum of the bits in each of the three circles, *A*, *B*, and *C*, is now an even number. In circle *A*, we have the four numbers 0, 0, 1, and 1, which add up to 2, an even number. In circle *B*, the numbers are 1, 1, 0, and 0, which also add up to 2, an even number. Finally, in circle *C*, we have the same thing. In this example all the circles happen to be the same, but sums of 0 and 4 are also possible in other examples. This figure corresponds to a codeword with 4 data bits and 3 parity bits.

Now suppose that the bit in the *AC* region goes bad, changing from a 0 to a 1, as shown in Fig. 2-14(c). The computer can now see that circles *A* and *C* have the wrong (odd) parity. The only single-bit change that corrects them is to restore *AC* back to 0, thus correcting the error. In this way, the computer can repair single-bit memory errors automatically.

Now let us see how Hamming's algorithm can be used to construct error-correcting codes for any size memory word. In a Hamming code, r parity bits are added to an m-bit word, forming a new word of length $m + r$ bits. The bits are numbered starting at 1, not 0, with bit 1 the leftmost (high-order) bit. All bits whose bit number is a power of 2 are parity bits; the rest are used for data. For example, with a 16-bit word, 5 parity bits are added. Bits 1, 2, 4, 8, and 16 are parity bits, and all the rest are data bits. In all, the memory word has 21 bits (16 data, 5 parity). We will (arbitrarily) use even parity in this example.

Each parity bit checks specific bit positions; the parity bit is set so that the total number of 1s in the checked positions is even. The bit positions checked by the parity bits are

Bit 1 checks bits 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21.

Bit 2 checks bits 2, 3, 6, 7, 10, 11, 14, 15, 18, 19.

Bit 4 checks bits 4, 5, 6, 7, 12, 13, 14, 15, 20, 21.

Bit 8 checks bits 8, 9, 10, 11, 12, 13, 14, 15.

Bit 16 checks bits 16, 17, 18, 19, 20, 21.

In general, bit b is checked by those bits $b_1, b_2, ..., b_j$ such that $b_1 + b_2 + ... + b_j = b$. For example, bit 5 is checked by bits 1 and 4 because $1 + 4 = 5$. Bit 6 is checked by bits 2 and 4 because $2 + 4 = 6$, and so on.

Figure 2-15 shows construction of a Hamming code for the 16-bit memory word 1111000010101110. The 21-bit codeword is 001011100000101101110. To see how error correction works, consider what would happen if bit 5 were inverted by an electrical surge on the power line. The new codeword would be 001001100000101101110 instead of 001011100000101101110. The 5 parity bits will be checked, with the following results:

Parity bit 1 incorrect (1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21 contain five 1s).

Parity bit 2 correct (2, 3, 6, 7, 10, 11, 14, 15, 18, 19 contain six 1s).

Parity bit 4 incorrect (4, 5, 6, 7, 12, 13, 14, 15, 20, 21 contain five 1s).

Parity bit 8 correct (8, 9, 10, 11, 12, 13, 14, 15 contain two 1s).

Parity bit 16 correct (16, 17, 18, 19, 20, 21 contain four 1s).

The total number of 1s in bits 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, and 21 should be an even number because even parity is being used. The incorrect bit must be one of the bits checked by parity bit 1—namely, bit 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, or 21. Parity bit 4 is incorrect, meaning that one of bits 4, 5, 6, 7, 12, 13, 14, 15, 20, or 21 is incorrect. The error must be one of the bits in both lists, namely, 5, 7, 13, 15, or 21. However, bit 2 is correct, eliminating 7 and 15. Similarly, bit 8 is correct, eliminating 13. Finally, bit 16 is correct, eliminating 21. The only bit left is bit 5, which is the one in error. Since it was read as a 1, it should be a 0. In this manner, errors can be corrected.

A simple method for finding the incorrect bit is first to compute all the parity bits. If all are correct, there was no error (or more than one). Then add up all the incorrect parity bits, counting 1 for bit 1, 2 for bit 2, 4 for bit 4, and so on. The resulting sum is the position of the incorrect bit. For example, if parity bits 1 and 4 are incorrect but 2, 8, and 16 are correct, bit 5 (1 + 4) has been inverted.

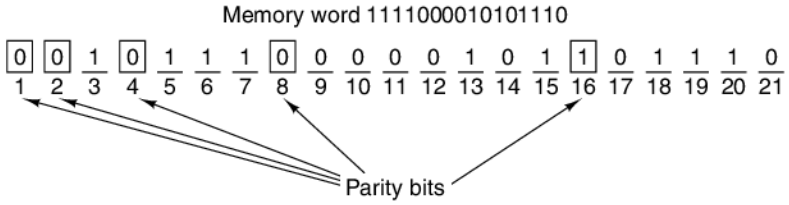


Figure 2-15. Construction of the Hamming code for the memory word 1111000010101110 by adding 5 check bits to the 16 data bits.

2.2.5 Cache Memory

Historically, CPUs have always been faster than memories. As memories have improved, so have CPUs, preserving the imbalance. In fact, as it becomes possible to put more and more circuits on a chip, CPU designers are using these new facilities for pipelining and superscalar operation, making CPUs go even faster. Memory designers have usually used new technology to increase the capacity of their chips, not the speed, so the problem appears to be getting worse over time. What this imbalance means in practice is that after the CPU issues a memory request, it will not get the word it needs for many CPU cycles. The slower the memory, the more cycles the CPU will have to wait.

As we pointed out above, there are two ways to deal with this problem. The simplest way is to just start memory READs when they are encountered but continue executing and stall the CPU if an instruction tries to use the memory word before it has arrived. The slower the memory, the greater the penalty when it does occur. For example, if one instruction in five touches memory and the memory access time is five cycles, execution time will be twice what it would have been with instantaneous memory. But if the memory access time is 50 cycles, then execution time will be up by a factor of 11 (5 cycles for executing instructions plus 50 cycles for waiting for memory).

The other solution is to have machines that do not stall but instead require the compilers not to generate code to use words before they have arrived. The trouble is that this approach is far easier said than done. Often after a LOAD there is nothing else to do, so the compiler is forced to insert NOP (no operation) instructions, which do nothing but occupy a slot and waste time. In effect, this approach is a software stall instead of a hardware stall, but the performance degradation is the same.

Actually, the problem is not technology, but economics. Engineers know how to build memories that are as fast as CPUs, but to run them at full speed, they have to be located on the CPU chip (because going over the bus to memory is very slow). Putting a large memory on the CPU chip makes it bigger, which makes it more expensive, and even if cost were not an issue, there are limits to how big a

CPU chip can be made. Thus the choice comes down to having a small amount of fast memory or a large amount of slow memory. What we would prefer is a large amount of fast memory at a low price.

Interestingly enough, techniques are known for combining a small amount of fast memory with a large amount of slow memory to get the speed of the fast memory (almost) and the capacity of the large memory at a moderate price. The small, fast memory is called a **cache** (from the French *cacher*, meaning to hide, and pronounced "cash"). Below we will briefly describe how caches are used and how they work. A more detailed description will be given in Chap. 4.

The basic idea behind a cache is simple: the most heavily used memory words are kept in the cache. When the CPU needs a word, it first looks in the cache. Only if the word is not there does it go to main memory. If a substantial fraction of the words are in the cache, the average access time can be greatly reduced.

Success or failure thus depends on what fraction of the words are in the cache. For years, people have known that programs do not access their memories completely at random. If a given memory reference is to address A, it is likely that the next memory reference will be in the general vicinity of A. A simple example is the program itself. Except for branches and procedure calls, instructions are fetched from consecutive locations in memory. Furthermore, most program execution time is spent in loops, in which a limited number of instructions are executed over and over. Similarly, a matrix manipulation program is likely to make many references to the same matrix before moving on to something else.

The observation that the memory references made in any short time interval tend to use only a small fraction of the total memory is called the **locality principle** and forms the basis for all caching systems. The general idea is that when a word is referenced, it and some of its neighbors are brought from the large slow memory into the cache, so that the next time it is used, it can be accessed quickly. A common arrangement of the CPU, cache, and main memory is illustrated in Fig. 2-16. If a word is read or written k times in a short interval, the computer will need 1 reference to slow memory and $k - 1$ references to fast memory. The larger k is, the better the overall performance.

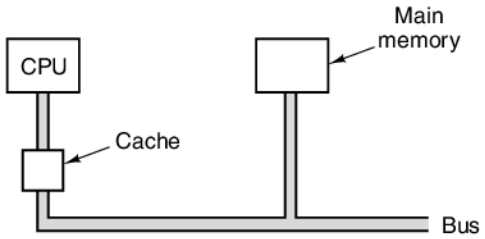


Figure 2-16. The cache is logically between the CPU and main memory. Physically, there are several possible places it could be located.

We can formalize this calculation by introducing c, the cache access time, m, the main memory access time, and h, the **hit ratio**, which is the fraction of all references that can be satisfied out of the cache. In our little example of the previous paragraph, $h = (k - 1)/k$. Some authors also define the **miss ratio**, which is $1 - h$.

With these definitions, we can calculate the mean access time as follows:

$$\text{mean access time} = c + (1 - h)m$$

As $h \rightarrow 1$, all references can be satisfied out of the cache, and the access time approaches c. On the other hand, as $h \rightarrow 0$, a memory reference is needed every time, so the access time approaches $c + m$, first a time c to check the cache (unsuccessfully), and then a time m to do the memory reference. On some systems, the memory reference can be started in parallel with the cache search, so that if a cache miss occurs, the memory cycle has already been started. However, this strategy requires that the memory can be stopped in its tracks on a cache hit, making the implementation more complicated.

Using the locality principle as a guide, main memories and caches are divided up into fixed-size blocks. When talking about these blocks inside the cache, we will often refer to them as **cache lines**. When a cache miss occurs, the entire cache line is loaded from the main memory into the cache, not just the word needed. For example, with a 64-byte line size, a reference to memory address 260 will pull the line consisting of bytes 256 to 319 into one cache line. With a little bit of luck, some of the other words in the cache line will be needed shortly. Operating this way is more efficient than fetching individual words because it is faster to fetch k words all at once than one word k times. Also, having cache entries be more than one word means there are fewer of them, hence a smaller overhead is required. Finally, many computers can transfer 64 or 128 bits in parallel on a single bus cycle, even on 32-bit machines.

Cache design is an increasingly important subject for high-performance CPUs. One issue is cache size. The bigger the cache, the better it performs, but also the slower it is to access and the more it costs. A second issue is the size of the cache line. A 16-KB cache can be divided up into 1024 lines of 16 bytes, 2048 lines of 8 bytes, and other combinations. A third issue is how the cache is organized, that is, how does the cache keep track of which memory words are currently being held? We will examine caches in detail in Chap. 4.

A fourth design issue is whether instructions and data are kept in the same cache or different ones. Having a **unified cache** (instructions and data use the same cache) is a simpler design and automatically balances instruction fetches against data fetches. Nevertheless, the trend these days is toward a **split cache**, with instructions in one cache and data in the other. This design is also called a **Harvard architecture**, the reference going all the way back to Howard Aiken's Mark III computer, which had different memories for instructions and data. The force driving designers in this direction is the widespread use of pipelined CPUs. The instruction fetch unit needs to access instructions at the same time the operand

fetch unit needs access to data. A split cache allows parallel accesses; a unified one does not. Also, since instructions are not modified during execution, the contents of the instruction cache never has to be written back into memory.

Finally, a fifth issue is the number of caches. It is common these days to have chips with a primary cache on chip, a secondary cache off chip but in the same package as the CPU chip, and a third cache still further away.

2.2.6 Memory Packaging and Types

From the early days of semiconductor memory until the early 1990s, memory was manufactured, bought, and installed as single chips. Chip densities went from 1K bits to 1M bits and beyond, but each chip was sold as a separate unit. Early PCs often had empty sockets into which additional memory chips could be plugged, if and when the purchaser needed them.

Since the early 1990s, a different arrangement has been used. A group of chips, typically 8 or 16, is mounted on a printed circuit board and sold as a unit. This unit is called a **SIMM** (**Single Inline Memory Module**) or a **DIMM** (**Dual Inline Memory Module**), depending on whether it has a row of connectors on one side or both sides of the board. SIMMs have one edge connector with 72 contacts and transfer 32 bits per clock cycle. They are rarely used these days. DIMMs usually have edge connectors with 120 contacts on each side of the board, for a total of 240 contacts, and transfer 64 bits per clock cycle. The most common ones at present are DDR3 DIMMS, which is the third version of the double data-rate memories. A typical DIMM is illustrated in Fig. 2-17.

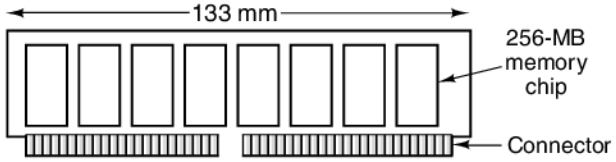


Figure 2-17. Top view of a DIMM holding 4 GB with eight chips of 256 MB on each side. The other side looks the same.

A typical DIMM configuration might have eight data chips with 256 MB each. The entire module would then hold 2 GB. Many computers have room for four modules, giving a total capacity of 8 GB when using 2-GB modules and more when using larger ones.

A physically smaller DIMM, called an **SO-DIMM** (**Small Outline DIMM**), is used in notebook computers. DIMMS can have a parity bit or error correction added, but since the average error rate of a module is one error every 10 years, for most garden-variety computers, error detection and correction are omitted.

2.3 SECONDARY MEMORY

No matter how big the main memory is, it is always way too small. People always want to store more information than it can hold, primarily because as technology improves, people begin thinking about storing things that were previously entirely in the realm of science fiction. For example, as the U.S. government's budget discipline forces government agencies to generate their own revenue, one can imagine the Library of Congress deciding to digitize and sell its full contents as a consumer article ("All of human knowledge for only $299.95"). Roughly 50 million books, each with 1 MB of text and 1 MB of compressed pictures, requires storing 10^{14} bytes or 100 terabytes. Storing all 50,000 movies ever made is also in this general ballpark. This amount of information is not going to fit in main memory, at least not for a few decades.

2.3.1 Memory Hierarchies

The traditional solution to storing a great deal of data is a memory hierarchy, as illustrated in Fig. 2-18. At the top are the CPU registers, which can be accessed at full CPU speed. Next comes the cache memory, which is currently on the order of 32 KB to a few megabytes. Main memory is next, with sizes currently ranging from 1 GB for entry-level systems to hundreds of gigabytes at the high end. After that come solid-state and magnetic disks, the current workhorses for permanent storage. Finally, we have magnetic tape and optical disks for archival storage.

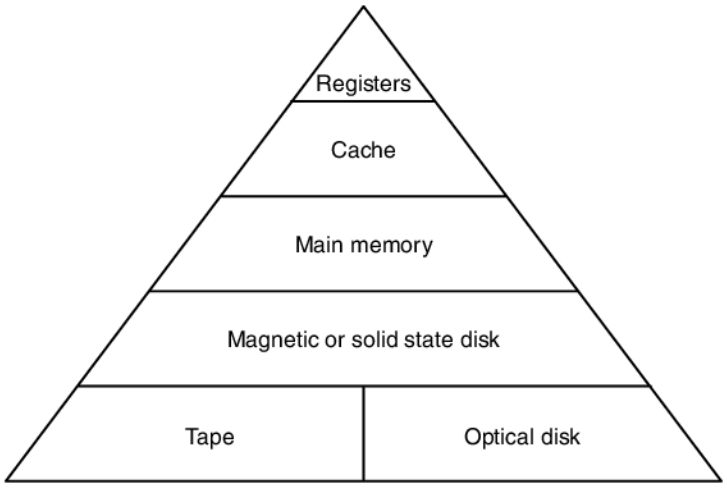


Figure 2-18. A five-level memory hierarchy.

As we move down the hierarchy, three key parameters increase. First, the access time gets bigger. CPU registers can be accessed in a nanosecond or less.

Cache memories take a small multiple of CPU registers. Main memory accesses are typically 10 nanoseconds. Now comes a big gap, as disk access times are at least 10 times slower for solid-state disks and hundreds of times slower for magnetic disks. Tape and optical disk access can be measured in seconds if the media have to be fetched and inserted into a drive.

Second, the storage capacity increases as we go downward. CPU registers are good for perhaps 128 bytes, caches for tens of megabytes, main memories for a few gigabytes, solid-state disks for hundreds of gigabytes, and magnetic disks for terabytes. Tapes and optical disks are usually kept off-line, so their capacity is limited only by the owner's budget.

Third, the number of bits you get per dollar spent increases down the hierarchy. Although the actual prices change rapidly, main memory is measured in dollars/megabyte, solid-state disk in dollars/gigabyte, and magnetic disk and tape storage in pennies/gigabyte.

We have already looked at registers, cache, and main memory. In the following sections we will look at magnetic and solid-state disks; after that, we will study optical ones. We will not study tapes because they are rarely used except for backup, and there is not a lot to say about them anyway.

2.3.2 Magnetic Disks

A magnetic disk consists of one or more aluminum platters with a magnetizable coating. Originally these platters were as much as 50 cm in diameter, but at present they are typically 3 to 9 cm, with disks for notebook computers already under 3 cm and still shrinking. A disk head containing an induction coil floats just over the surface, resting on a cushion of air. When a positive or negative current passes through the head, it magnetizes the surface just beneath the head, aligning the magnetic particles facing left or facing right, depending on the polarity of the drive current. When the head passes over a magnetized area, a positive or negative current is induced in the head, making it possible to read back the previously stored bits. Thus as the platter rotates under the head, a stream of bits can be written and later read back. The geometry of a disk track is shown in Fig. 2-19.

The circular sequence of bits written as the disk makes a complete rotation is called a **track**. Each track is divided up into some number of fixed-length **sectors**, typically containing 512 data bytes, preceded by a **preamble** that allows the head to be synchronized before reading or writing. Following the data is an Error-Correcting Code (ECC), either a Hamming code or, more commonly, a code that can correct multiple errors called a **Reed-Solomon code**. Between consecutive sectors is a small **intersector gap**. Some manufacturers quote their disks' capacities in unformatted state (as if each track contained only data), but a more honest measurement is the formatted capacity, which does not count the preambles, ECCs, and gaps as data. The formatted capacity is typically about 15 percent lower than the unformatted capacity.

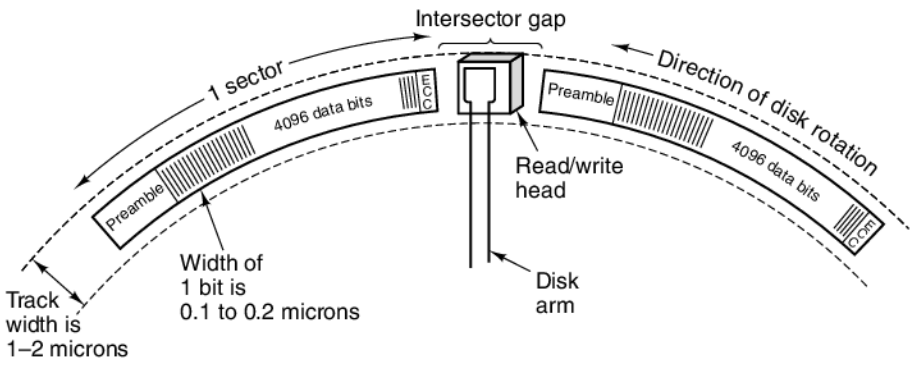


Figure 2-19. A portion of a disk track. Two sectors are illustrated.

All disks have movable arms that are capable of moving in and out to different radial distances from the spindle about which the platter rotates. At each radial distance, a different track can be written. The tracks are thus a series of concentric circles about the spindle. The width of a track depends on how large the head is and how accurately the head can be positioned radially. With current technology, disks have around 50,000 tracks per centimeter, giving track widths in the 200-nanometer range (1 nanometer = 1/1,000,000 mm). It should be noted that a track is not a physical groove in the surface, but simply an annulus (ring) of magnetized material, with small guard areas separating it from the tracks inside and outside it.

The linear bit density around the circumference of the track is different from the radial one. In other words, the number of bits per millimeter measured going around a track is different from the number of bits per millimeter starting from the center and moving outward. The density along a track is determined largely by the purity of the surface and air quality. Current disks achieve densities of 25 gigabits/cm. The radial density is determined by how accurately the arm can be made to seek to a track. Thus a bit is many times larger in the radial direction as compared to the circumference, as suggested by Fig. 2-19.

Ultrahigh density disks utilize a recording technology in which the "long" dimension of the bits is not along the circumference of the disk, but vertically, down into the iron oxide. This technique is called **perpendicular recording**, and it has been demonstrated to provide data densities of up to 100 gigabits/cm. It is likely to become the dominant technology in the coming years.

In order to achieve high surface and air quality, most disks are sealed at the factory to prevent dust from getting in. Such drives were originally called **Winchester disks** because the first such drives (created by IBM) had 30 MB of sealed, fixed storage and 30 MB of removable storage. Supposedly, these 30-30 disks reminded people of the Winchester 30-30 rifles that played a great role in opening the American frontier, and the name "Winchester" stuck. Now they are just called

hard disks to differentiate them from the long-vanished **floppy disks** used on the very first personal computers. In this business, it is difficult to pick a name for anything and not have it be ridiculous 30 years later.

Most disks consist of multiple platters stacked vertically, as depicted in Fig. 2-20. Each surface has its own arm and head. All the arms are ganged together so they move to different radial positions all at once. The set of tracks at a given radial position is called a **cylinder**. Current PC and server disks typically have 1 to 12 platters per drive, giving 2 to 24 recording surfaces. High-end disks can store 1 TB on a single platter and that limit is sure to grow with time.

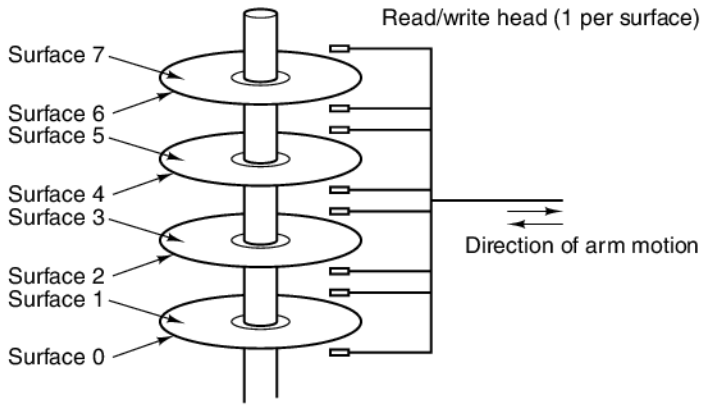


Figure 2-20. A disk with four platters.

Disk performance depends on a variety of factors. To read or write a sector, first the arm must be moved to the right radial position. This action is called a **seek**. Average seek times (between random tracks) range in the 5- to 10-msec range, although seeks between consecutive tracks are now down below 1 msec. Once the head is positioned radially, there is a delay, called the **rotational latency**, until the desired sector rotates under the head. Most disks rotate at 5400 RPM, 7200 RPM, or 10,800 RPM, so the average delay (half a rotation) is 3 to 6 msec. Transfer time depends on the linear density and rotation speed. With typical internal transfer rate of 150 MB/sec, a 512-byte sector takes about 3.5 μsec. Consequently, the seek time and rotational latency dominate the transfer time. Reading random sectors all over the disk is clearly an inefficient way to operate.

It is worth mentioning that on account of the preambles, the ECCs, the intersector gaps, the seek times, and the rotational latencies, there is a big difference between a drive's maximum burst rate and its maximum sustained rate. The maximum burst rate is the data rate once the head is over the first data bit. The computer must be able to handle data coming in this fast. However, the drive can keep up that rate only for one sector. For some applications, such as multimedia, what

matters is the average sustained rate over a period of seconds, which has to take into account the necessary seeks and rotational delays as well.

A little thought and the use of that old high-school math formula for the circumference of a circle, $c = 2\pi r$, will reveal that the outer tracks have more linear distance around them than the inner ones do. Since all magnetic disks rotate at a constant angular velocity, no matter where the heads are, this observation creates a problem. In older drives, manufacturers used the maximum possible linear density on the innermost track, and successively lowered linear bit densities on tracks further out. If a disk had 18 sectors per track, for example, each one occupied 20 degrees of arc, no matter which cylinder it was in.

Nowadays, a different strategy is used. Cylinders are divided into zones (typically 10 to 30 per drive), and the number of sectors per track is increased in each zone moving outward from the innermost track. This change makes keeping track of information harder but increases the drive capacity, which is viewed as more important. All sectors are the same size. A disk with five zones is shown in Fig. 2-21.

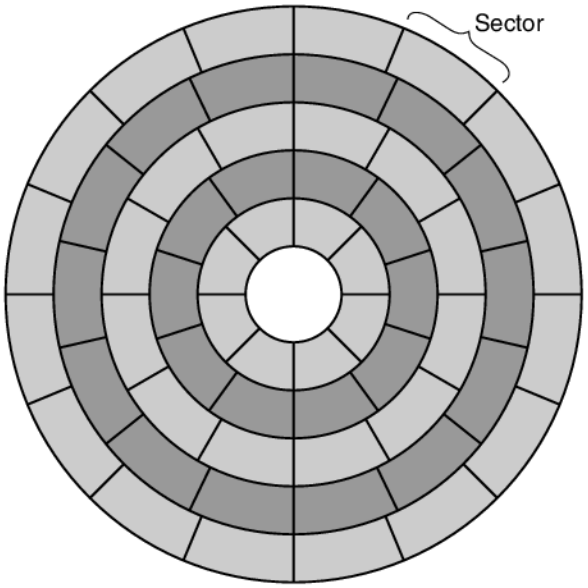


Figure 2-21. A disk with five zones. Each zone has many tracks.

Associated with each drive is a **disk controller**, a chip that controls the drive. Some controllers contain a full CPU. The controller's tasks include accepting commands from the software, such as READ, WRITE, and FORMAT (writing all the preambles), controlling the arm motion, detecting and correcting errors, and converting 8-bit bytes read from memory into a serial bit stream and vice versa. Some controllers also handle buffering of multiple sectors, caching sectors read for

potential future use, and remapping bad sectors. This latter function is caused by the existence of sectors with a bad (permanently magnetized) spot. When the controller discovers a bad sector, it replaces it by one of the spare sectors reserved for this purpose within each cylinder or zone.

2.3.3 IDE Disks

Modern personal computer disks evolved from the one in the IBM PC XT, which was a 10-MB Seagate disk controlled by a Xebec disk controller on a plug-in card. The Seagate disk had 4 heads, 306 cylinders, and 17 sectors/track. The controller was capable of handling two drives. The operating system read from and wrote to a disk by putting parameters in CPU registers and then calling the **BIOS** (**Basic Input Output System**), located in the PC's built-in read-only memory. The BIOS issued the machine instructions to load the disk controller registers that initiated transfers.

The technology evolved rapidly from having the controller on a separate board, to having it closely integrated with the drives, starting with **IDE** (**Integrated Drive Electronics**) drives in the mid 1980s. However, the BIOS calling conventions were not changed for reasons of backward compatibility. These calling conventions addressed sectors by giving their head, cylinder, and sector numbers, with the heads and cylinders numbered starting at 0 and the sectors starting at 1. This choice was probably due to a mistake on the part of the original BIOS programmer, who wrote his masterpiece in 8088 assembler. With 4 bits for the head, 6 bits for the sector, and 10 bits for the cylinder, the maximum drive could have 16 heads, 63 sectors, and 1024 cylinders, for a total of 1,032,192 sectors. Such a maximum drive has a capacity of 504 MB, which probably seemed like infinity at the time but certainly does not today. (Would you fault a new machine today that could not handle drives bigger than 1000 TB?)

Unfortunately, before too long, drives below 504 MB appeared but with the wrong geometry (e.g., 4 heads, 32 sectors, 2000 cylinders is 256,000 sectors). There was no way for the operating system to address them due to the long-frozen BIOS calling conventions. As a result, disk controllers began to lie, pretending that the geometry was within the BIOS limits but actually remapping the virtual geometry onto the real geometry. Although this approach worked, it wreaked havoc with operating systems that carefully placed data to minimize seek times.

Eventually, IDE drives evolved into **EIDE** drives (**Extended IDE**), which also support a second addressing scheme called **LBA** (**Logical Block Addressing**), which just numbers the sectors starting at 0 up until a maximum of $2^{28} - 1$. This scheme requires the controller to convert LBA addresses to head, sector, and cylinder addresses, but at least it does get beyond the 504-MB limit. Unfortunately, it created a new bottleneck at $2^{28} \times 2^9$ bytes (128 GB). In 1994, when the EIDE standard was adopted, nobody in their wildest imagination could imagine 128-GB

disks. Standards committees, like politicians, have a tendency to push problems forward in time so the next committee has to solve them.

EIDE drives and controllers also had other improvements as well. For example, EIDE controllers could have two channels, each with a primary and a secondary drive. This arrangement allowed a maximum of four drives per controller. CD-ROM and DVD drives were also supported, and the transfer rate was increased from 4 MB/sec to 16.67 MB/sec.

As disk technology continued to improve, the EIDE standard continued to evolve, but for some reason the successor to EIDE was called **ATA-3** (**AT Attachment**), a reference to the IBM PC/AT (where AT referred to the then-Advanced Technology of a 16-bit CPU running at 8 MHz). In the next edition, the standard was called **ATAPI-4** (**ATA Packet Interface**) and the speed was increased to 33 MB/sec. In ATAPI-5 it went to 66 MB/sec.

By this time, the 128-GB limit imposed by the 28-bit LBA addresses was looming larger and larger, so ATAPI-6 changed the LBA size to 48 bits. The new standard will run into trouble when disks reach $2^{48} \times 2^9$ bytes (128 PB). With a 50% annual increase in capacity, the 48-bit limit will probably last until about 2035. To find out how the problem was solved, please consult the 11th edition of this book. The smart money is betting on increasing the LBA size to 64 bits. The ATAPI-6 standard also increased the transfer rate to 100 MB/sec and addressed the issue of disk noise for the first time.

The ATAPI-7 standard is a radical break with the past. Instead of increasing the size of the drive connector (to increase the data rate), this standard uses what is called **serial ATA** to transfer 1 bit at a time over a 7-pin connector at speeds starting at 150 MB/sec and expected to rise over time to 1.5 GB/sec. Replacing the old 80-wire flat cable with a round cable only a few mm thick improves airflow within the computer. Also, serial ATA uses 0.5 volts for signaling (compared to 5 volts on ATAPI-6 drives), which reduces power consumption. It is likely that within a few years, all computers will use serial ATA. The issue of power consumption by disks is an increasingly important one, both at the high end, where data centers have vast disk farms, and at the low end, where notebooks are power limited (Gurumurthi et al., 2003).

2.3.4 SCSI Disks

SCSI disks are not different from IDE disks in terms of how their cylinders, tracks, and sectors are organized, but they have a different interface and much higher transfer rates. SCSI traces its history back to Howard Shugart, the inventor of the floppy disk, which was used on the first personal computers in the 1980s. His company introduced the SASI (Shugart Associates System Interface) disk in 1979. After some modification and quite a bit of discussion, ANSI standardized it in 1986 and changed the name to **SCSI** (**Small Computer System Interface**). SCSI is pronounced "scuzzy." Since then, increasingly higher bandwidth versions

have been standardized under the names Fast SCSI (10 MHz), Ultra SCSI (20 MHz), Ultra2 SCSI (40 MHz), Ultra3 SCSI (80 MHz), Ultra4 SCSI (160 MHz), and Ultra5 SCSI (320 MHz). Each of these has a wide (16-bit) version as well. In fact, the recent ones have only a wide version. The main combinations are shown in Fig. 2-22.

Name	Data bits	Bus MHz	MB/sec
SCSI-1	8	5	5
Fast SCSI	8	10	10
Wide Fast SCSI	16	10	20
Ultra SCSI	8	20	20
Wide Ultra SCSI	16	20	40
Ultra2 SCSI	8	40	40
Wide Ultra2 SCSI	16	40	80
Wide Ultra3 SCSI	16	80	160
Wide Ultra4 SCSI	16	160	320
Wide Ultra5 SCSI	16	320	640

Figure 2-22. Some of the possible SCSI parameters.

Because SCSI disks have high transfer rates, they are the standard disk in many high-end workstations and servers, especially those that run RAID configurations (see below).

SCSI is more than just a hard-disk interface. It is a bus to which a SCSI controller and up to seven devices can be attached. These can include one or more SCSI hard disks, CD-ROMs, CD recorders, scanners, tape units, and other SCSI peripherals. Each SCSI device has a unique ID, from 0 to 7 (15 for wide SCSI). Each device has two connectors: one for input and one for output. Cables connect the output of one device to the input of the next one, in series, like a string of cheap Christmas tree lamps. The last device in the string must be terminated to prevent reflections from the ends of the SCSI bus from interfering with other data on the bus. Typically, the controller is on a plug-in card and the start of the cable chain, although this configuration is not strictly required by the standard.

The most common cable for 8-bit SCSI has 50 wires, 25 of which are grounds paired one-to-one with the other 25 wires to provide the excellent noise immunity needed for high-speed operation. Of the 25 wires, 8 are for data, 1 is for parity, 9 are for control, and the remainder are for power or are reserved for future use. The 16-bit (and 32-bit) devices need a second cable for the additional signals. The cables may be several meters long, allowing for external drives, scanners, etc.

SCSI controllers and peripherals can operate either as initiators or as targets. Usually, the controller, acting as initiator, issues commands to disks and other peripherals acting as targets. These commands are blocks of up to 16 bytes telling the target what to do. Commands and responses occur in phases, using various

control signals to delineate the phases and arbitrate bus access when multiple devices are trying to use the bus at the same time. This arbitration is important because SCSI allows all the devices to run at once, potentially greatly improving performance in an environment with multiple processes active at once. IDE and EIDE allow only one active device at a time.

2.3.5 RAID

CPU performance has been increasing exponentially over the past decade, roughly doubling every 18 months. Not so with disk performance. In the 1970s, average seek times on minicomputer disks were 50 to 100 msec. Now seek times are 10 msec. In most technical industries (say, automobiles or aviation), a factor of 5 to 10 performance improvement in two decades would be major news, but in the computer industry it is an embarrassment. Thus the gap between CPU performance and disk performance has become much larger over time.

As we have seen, parallel processing is often used to speed up CPU performance. It has occurred to various people over the years that parallel I/O might be a good idea, too. In their 1988 paper, Patterson et al. suggested six specific disk organizations that could be used to improve disk performance, reliability, or both (Patterson et al., 1988). These ideas were quickly adopted by industry and have led to a new class of I/O device called a **RAID**. Patterson et al. defined **RAID** as **Redundant Array of Inexpensive Disks**, but industry redefined the I to be "Independent" rather than "Inexpensive" (maybe so they could use expensive disks?). Since a villain was also needed (as in RISC versus CISC, also due to Patterson), the bad guy here was the **SLED** (**Single Large Expensive Disk**).

The idea behind a RAID is to install a box full of disks next to the computer, typically a large server, replace the disk controller card with a RAID controller, copy the data over to the RAID, and then continue normal operation. In other words, a RAID should look like a SLED to the operating system but have better performance and better reliability. Since SCSI disks have good performance, low price, and the ability to have up to 7 drives on a single controller (15 for wide SCSI), it is natural that many RAIDs consist of a RAID SCSI controller plus a box of SCSI disks that appear to the operating system as a single large disk. In this way, no software changes are required to use the RAID, a big selling point for many system administrators.

In addition to appearing like a single disk to the software, all RAIDs have the property that the data are distributed over the drives, to allow parallel operation. Several different schemes for doing this were defined by Patterson et al., and they are now known as RAID level 0 through RAID level 5. In addition, there are a few other minor levels that we will not discuss. The term "level" is something of a misnomer since there is no hierarchy involved; there are simply six different organizations, each with a different mix of reliability and performance characteristics.

RAID level 0 is illustrated in Fig. 2-23(a). It consists of viewing the virtual disk simulated by the RAID as being divided up into strips of k sectors each, with sectors 0 to $k - 1$ being strip 0, sectors k to $2k - 1$ as strip 1, and so on. For $k = 1$, each strip is a sector; for $k = 2$ a strip is two sectors, etc. The RAID level 0 organization writes consecutive strips over the drives in round-robin fashion, as depicted in Fig. 2-23(a) for a RAID with four disk drives. Distributing data over multiple drives like this is called **striping**. For example, if the software issues a command to read a data block consisting of four consecutive strips starting at a strip boundary, the RAID controller will break this command up into four separate commands, one for each of the four disks, and have them operate in parallel. Thus we have parallel I/O without the software knowing about it.

RAID level 0 works best with large requests, the bigger the better. If a request is larger than the number of drives times the strip size, some drives will get multiple requests, so that when they finish the first request they start the second one. It is up to the controller to split the request up and feed the proper commands to the proper disks in the right sequence and then assemble the results in memory correctly. Performance is excellent and the implementation is straightforward.

RAID level 0 works worst with operating systems that habitually ask for data one sector at a time. The results will be correct, but there is no parallelism and hence no performance gain. Another disadvantage of this organization is that the reliability is potentially worse than having a SLED. If a RAID consists of four disks, each with a mean time to failure of 20,000 hours, about once every 5000 hours a drive will fail and all the data will be completely lost. A SLED with a mean time to failure of 20,000 hours would be four times more reliable. Because no redundancy is present in this design, it is not really a true RAID.

The next option, RAID level 1, shown in Fig. 2-23(b), is a true RAID. It duplicates all the disks, so there are four primary disks and four backup disks in this example, although any other even number of disks is also possible. On a write, every strip is written twice. On a read, either copy can be used, distributing the load over more drives. Consequently, write performance is no better than for a single drive, but read performance can be up to twice as good. Fault tolerance is excellent: if a drive crashes, the copy is simply used instead. Recovery consists of simply installing a new drive and copying the entire backup drive to it.

Unlike levels 0 and 1, which work with strips of sectors, RAID level 2 works on a word basis, possibly even a byte basis. Imagine splitting each byte of the single virtual disk into a pair of 4-bit nibbles, then adding a Hamming code to each one to form a 7-bit word, of which bits 1, 2, and 4 were parity bits. Further imagine that the seven drives of Fig. 2-23(c) were synchronized in terms of arm position and rotational position. Then it would be possible to write the 7-bit Hamming coded word over the seven drives, one bit per drive.

The Thinking Machines CM-2 computer used this scheme, taking 32-bit data words and adding 6 parity bits to form a 38-bit Hamming word, plus an extra bit for word parity, and spread each word over 39 disk drives. The total throughput

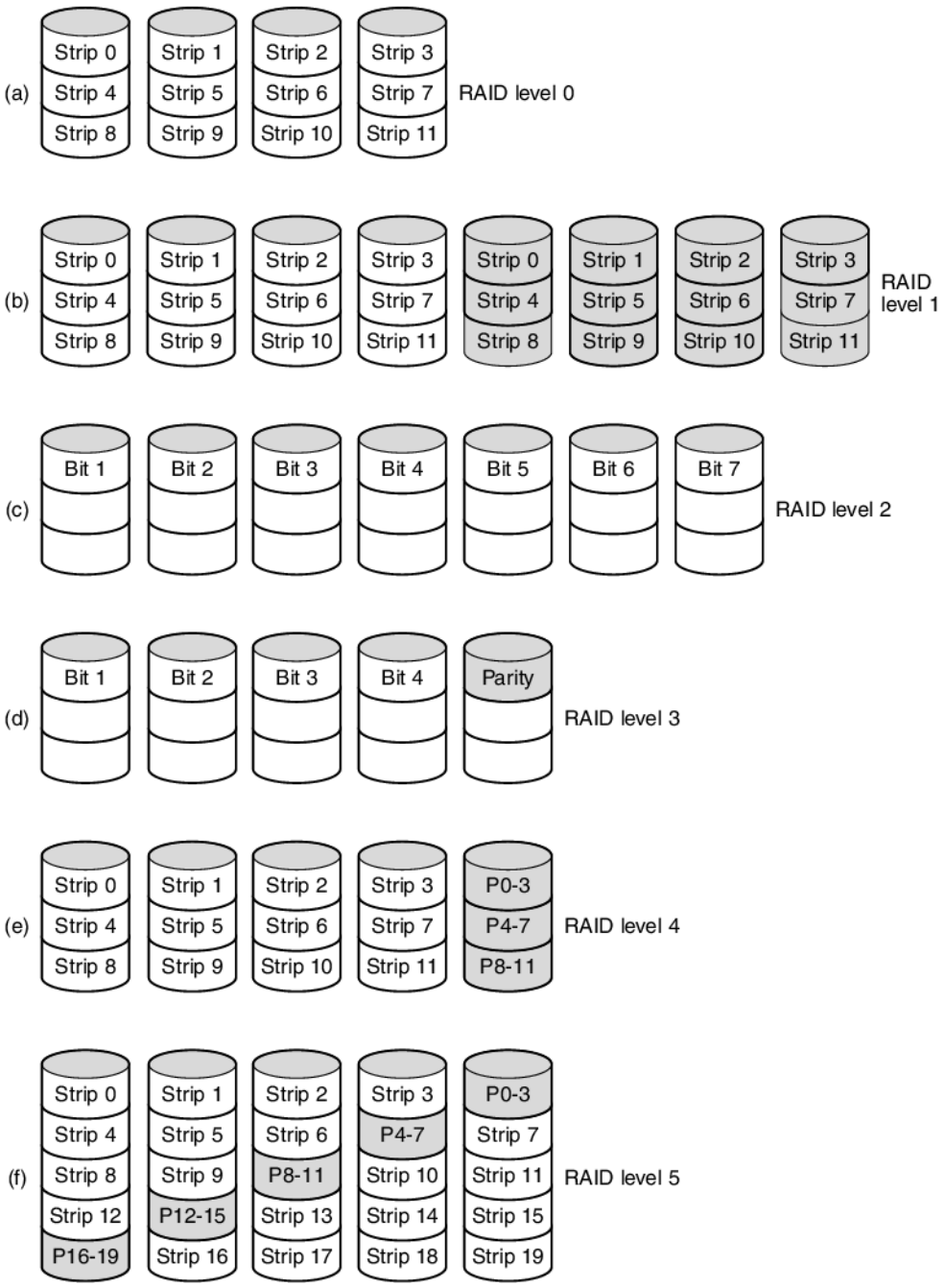


Figure 2-23. RAID levels 0 through 5. Backup and parity drives are shown shaded.

was immense, because in one sector time it could write 32 sectors worth of data. Also, losing one drive did not cause problems, because loss of a drive amounted to losing 1 bit in each 39-bit word read, something the Hamming code could handle on the fly.

On the down side, this scheme requires all the drives to be rotationally synchronized, and it makes sense only with a substantial number of drives (even with 32 data drives and 6 parity drives, the overhead is 19 percent). It also asks a lot of the controller, since it must do a Hamming checksum every bit time.

RAID level 3 is a simplified version of RAID level 2. It is illustrated in Fig. 2-23(d). Here a single parity bit is computed for each data word and written to a parity drive. As in RAID level 2, the drives must be exactly synchronized, since individual data words are spread over multiple drives.

At first thought, it might appear that a single parity bit gives only error detection, not error correction. For the case of random undetected errors, this observation is true. However, for the case of a drive crashing, it provides full 1-bit error correction since the position of the bad bit is known. If a drive crashes, the controller just pretends that all its bits are 0s. If a word has a parity error, the bit from the dead drive must have been a 1, so it is corrected. Although both RAID levels 2 and 3 offer very high data rates, the number of separate I/O requests per second they can handle is no better than for a single drive.

RAID levels 4 and 5 work with strips again, not individual words with parity, and do not require synchronized drives. RAID level 4 [see Fig. 2-23(e)] is like RAID level 0, with a strip-for-strip parity written onto an extra drive. For example, if each strip is k bytes long, all the strips are EXCLUSIVE ORed together, resulting in a parity strip k bytes long. If a drive crashes, the lost bytes can be recomputed from the parity drive.

This design protects against the loss of a drive but performs poorly for small updates. If one sector is changed, all the drives must be read in order to recalculate the parity, which then must be rewritten. Alternatively, the old user data and the old parity data can be read and the new parity recomputed from them. Even with this optimization, a small update requires two reads and two writes, clearly a bad arrangement.

As a consequence of the heavy load on the parity drive, it may become a bottleneck. This bottleneck is eliminated in RAID level 5 by distributing the parity bits uniformly over all the drives, round robin fashion, as shown in Fig. 2-23(f). However, in the event of a drive crash, reconstructing the contents of the failed drive is a complex process.

2.3.6 Solid-State Disks

Disks made from nonvolatile flash memory, often called **solid-state disks** (**SSDs**), are growing in popularity as a high-speed alternative to traditional magnetic disk technologies. The invention of the SSD is a classic tale of "When they

give you lemons, make lemonade." While modern electronics may seem totally reliable, the reality is that transistors slowly wear out as they are used. Every time they switch, they wear out a little bit more and get closer to no longer working. One likely way that a transistor will fail is due to "hot carrier injection," a failure mechanism in which an electron charge gets embedded inside a once-working transistor, leaving it in a state where it is permanently stuck on or off. While generally thought of as a death sentence for a (likely) innocent transistor, Fujio Masuoka while working for Toshiba discovered a way to harness this failure mechanism to create a new nonvolatile memory. In the early 1980s, he invented the first flash memory.

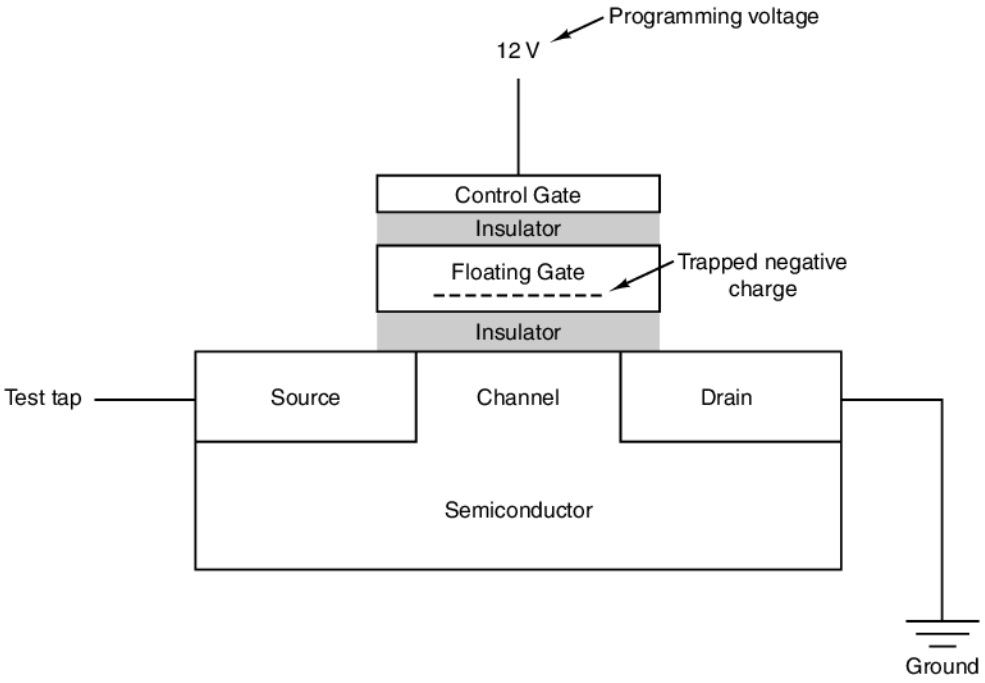


Figure 2-24. A flash memory cell.

Flash disks are made of many solid-state flash memory cells. The flash memory cells are made from a single special flash transistor. A flash memory cell is shown in Fig. 2-24. Embedded inside the transistor is a floating gate that can be charged and discharged using high voltages. Before being programmed, the floating gate does not affect the operation of the transistor, essentially acting as an extra insulator between the control gate and the transistor channel. If the flash cell is tested, it will act like a simple transistor.

To program the flash bit cell, a high voltage (in the computer world 12 V is a high voltage) is applied to the control gate, which accelerates the process of hot

carrier injection into the floating gate. Electrons become embedded into the floating gate, which places a negative charge internal to the flash transistor. The embedded negative charge increases the voltage necessary to turn on the flash transistor, and by testing whether or not the channel turns on with a high or low voltage, it is possible to determine whether the floating gate is charged or not, resulting in a 0 or 1 value for the flash cell. The embedded charge remains in the transistor, even if power is removed from the system, making the flash memory cell nonvolatile.

Because SSDs are essentially memory, they have superior performance to spinning disks and have zero seek time. While a typical magnetic disk can access data up to 100 MB/sec, a SSD can operate two to three times faster. And because the device has no moving parts, it is particularly suited for use in notebook computers, where jarring and movement will not affect its ability to access data. The downside of SSDs, compared to magnetic disks, is their cost. While magnetic disks cost pennies/gigabyte, a typical SSD will cost one to three dollars/gigabyte, making their use appropriate only for smaller drive applications or situations that are not cost sensitive. The cost of SSDs is dropping, but they still have a long way to go to catch up to cheap magnetic disks. So while SSDs are replacing magnetic disks in many computers, it will likely be a long time before the magnetic disk goes the way of the dinosaur (unless another big meteorite strikes the earth, in which cases the SSDs are probably not going to survive either).

Another disadvantage of SSDs compared with magnetic disks is their failure rate. A typical flash cell can be written only about 100,000 times before it will no longer function. The process of injecting electrons into the floating gate slowly damages it and the surrounding insulators, until it can no longer function. To increase the lifetime of SSDs, a technique called **wear leveling** is used to spread writes out to all flash cells in the disk. Every time a new disk block is written, the destination block is reassigned to a new SSD block that has not been recently written. This requires the use of a logical block map inside the flash drive, which is one of the reasons that flash drives have high internal storage overheads. Using wear leveling, a flash drive can support a number of writes equal to the number of writes a cell can sustain times the number of blocks on the disk.

Some SSDs are able to encode multiple bits per byte using multilevel flash cells. The technology carefully controls the amount of charge placed into the floating gate. An increasing sequence of voltages is then applied to the control gate to determine how much charge is stored in the floating gate. Typical multilevel cells will support four charge levels, yielding two bits per flash cell.

2.3.7 CD-ROMs

Optical disks were originally developed for recording television programs, but they can be put to more esthetic use as computer storage devices. Due to their large capacity and low price optical disks are widely used for distributing software, books, movies, and data of all kinds, as well as making backups of hard disks.

First-generation optical disks were invented by the Dutch electronics conglomerate Philips for holding movies. They were 30 cm across and marketed under the name LaserVision, but they did not catch on, except in Japan.

In 1980, Philips, together with Sony, developed the CD (Compact Disc), which rapidly replaced the 33 1/3 RPM vinyl record for music. The precise technical details for the CD were published in an official International Standard (IS 10149), popularly called the **Red Book**, after to the color of its cover. (International Standards are issued by the International Organization for Standardization, which is the international counterpart of national standards groups like ANSI, DIN, etc. Each one has an IS number.) The point of publishing the disk and drive specifications as an International Standard is to allow CDs from different music publishers and players from different electronics manufacturers to work together. All CDs are 120 mm across and 1.2 mm thick, with a 15-mm hole in the middle. The audio CD was the first successful mass-market digital storage medium. Audio CDs are supposed to last 100 years. Please check back in 2080 for an update on how well the first batch did.

A CD is prepared by using a high-power infrared laser to burn 0.8-micron diameter holes in a coated glass master disk. From this master, a mold is made, with bumps where the laser holes were. Into this mold, molten polycarbonate is injected to form a CD with the same pattern of holes as the glass master. Then a thin layer of reflective aluminum is deposited on the polycarbonate, topped by a protective lacquer and finally a label. The depressions in the polycarbonate substrate are called **pits**; the unburned areas between the pits are called **lands**.

When a CD is played back, a low-power laser diode shines infrared light with a wavelength of 0.78 micron on the pits and lands as they stream by. The laser is on the polycarbonate side, so the pits stick out in the direction of the laser as bumps in the otherwise flat surface. Because the pits have a height of one-quarter the wavelength of the laser light, light reflecting off a pit is half a wavelength out of phase with light reflecting off the surrounding surface. As a result, the two parts interfere destructively and return less light to the player's photodetector than light bouncing off a land. This is how the player tells a pit from a land. Although it might seem simplest to use a pit to record a 0 and a land to record a 1, it is more reliable to use a pit/land or land/pit transition for a 1 and its absence as a 0, so this scheme is used.

The pits and lands are written in a single continuous spiral starting near the hole and working out a distance of 32 mm toward the edge. The spiral makes 22,188 revolutions around the disk (about 600 per mm). If unwound, it would be 5.6 km long. The spiral is illustrated in Fig. 2-25.

To make the music play at a uniform rate, it is necessary for the pits and lands to stream by at a constant *linear* velocity. Consequently, the rotation rate of the CD must be continuously reduced as the reading head moves from the inside of the CD to the outside. At the inside, the rotation rate is 530 RPM to achieve the desired streaming rate of 120 cm/sec; at the outside it has to drop to 200 RPM to give

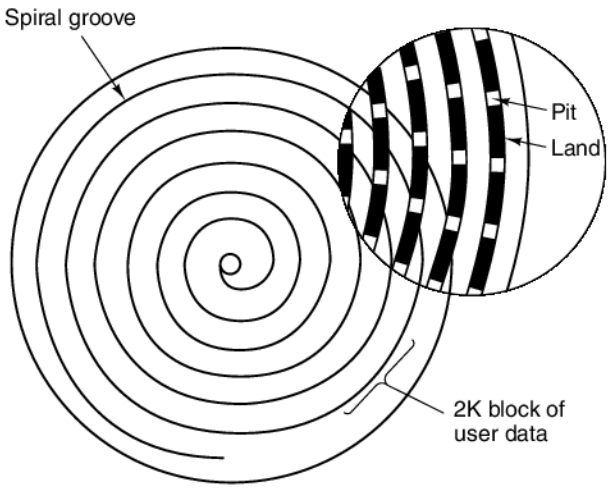


Figure 2-25. Recording structure of a Compact Disc or CD-ROM.

the same linear velocity at the head. A constant-linear-velocity drive is quite different than a magnetic-disk drive, which operates at a constant *angular* velocity, independent of where the head is currently positioned. Also, 530 RPM is a far cry from the 3600 to 7200 RPM that most magnetic disks whirl at.

In 1984, Philips and Sony realized the potential for using CDs to store computer data, so they published the **Yellow Book** defining a precise standard for what are now called **CD-ROM**s (**Compact Disc-Read Only Memory**). To piggyback on the by-then already substantial audio CD market, CD-ROMs were to be the same physical size as audio CDs, mechanically and optically compatible with them, and produced using the same polycarbonate injection molding machines. The consequences of this decision were that slow variable-speed motors were required, but also that the manufacturing cost of a CD-ROM would be well under one dollar in moderate volume.

What the Yellow Book defined was the formatting of the computer data. It also improved the error-correcting abilities of the system, an essential step because although music lovers do not mind losing a bit here and there, computer lovers tend to be Very Picky about that. The basic format of a CD-ROM consists of encoding every byte in a 14-bit symbol. As we saw above, 14 bits is enough to Hamming encode an 8-bit byte with 2 bits left over. In fact, a more powerful encoding system is used. The 14-to-8 mapping for reading is done in hardware by table lookup.

At the next level up, a group of 42 consecutive symbols forms a 588-bit **frame**. Each frame holds 192 data bits (24 bytes). The remaining 396 bits are for error correction and control. This scheme is identical for audio CDs and CD-ROMs.

What the Yellow Book adds is the grouping of 98 frames into a **CD-ROM sector**, as shown in Fig. 2-26. Every CD-ROM sector begins with a 16-byte preamble, the first 12 of which are 00FFFFFFFFFFFFFFFFFFFF00 (hexadecimal), to allow the player to recognize the start of a CD-ROM sector. The next 3 bytes contain the sector number, needed because seeking on a CD-ROM with its single data spiral is much more difficult than on a magnetic disk with its uniform concentric tracks. To seek, the software in the drive calculates approximately where to go, moves the head there, and then starts hunting around for a preamble to see how good its guess was. The last byte of the preamble is the mode.

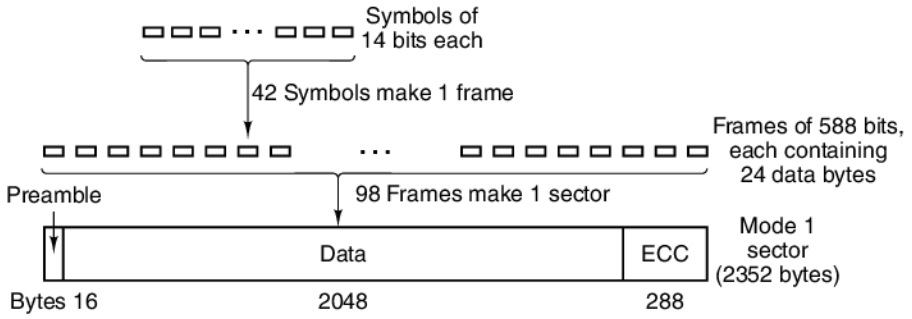


Figure 2-26. Logical data layout on a CD-ROM.

The Yellow Book defines two modes. Mode 1 uses the layout of Fig. 2-26, with a 16-byte preamble, 2048 data bytes, and a 288-byte error-correcting code (a cross-interleaved Reed-Solomon code). Mode 2 combines the data and ECC fields into a 2336-byte data field for those applications that do not need (or cannot afford the time to perform) error correction, such as audio and video. Note that to provide excellent reliability, three separate error-correcting schemes are used: within a symbol, within a frame, and within a CD-ROM sector. Single-bit errors are corrected at the lowest level, short burst errors are corrected at the frame level, and any residual errors are caught at the sector level. The price paid for this reliability is that it takes 98 frames of 588 bits (7203 bytes) to carry a single 2048-byte payload, an efficiency of only 28 percent.

Single-speed CD-ROM drives operate at 75 sectors/sec, which gives a data rate of 153,600 bytes/sec in mode 1 and 175,200 bytes/sec in mode 2. Double-speed drives are twice as fast, and so on up to the highest speed. A standard audio CD has room for 74 minutes of music, which, if used for mode 1 data, gives a capacity of 681,984,000 bytes. This figure is usually reported as 650 MB because 1 MB is 2^{20} bytes (1,048,576 bytes), not 1,000,000 bytes.

As usual, whenever a new technology comes out, some people try to push the envelope. When designing the CD-ROM, Philips and Sony were cautious and had the writing process stop well before the outer edge of the disc was reached. It did

not take long before some drive manufacturers allowed their drives to go beyond the official limit and come perilously close to the physical edge of the medium, giving about 700 MB instead of 650 MB. But as the technology improved and the blank discs were manufactured to a higher standard, 703.12 MB (360,000 2048-byte sectors instead of 333,000 sectors) became the new norm.

Note that even a 32x CD-ROM drive (4,915,200 bytes/sec) is no match for a fast SCSI-2 magnetic-disk drive at 10 MB/sec. When you realize that the seek time is often several hundred milliseconds, it should be clear that CD-ROM drives are not at all in the same performance category as magnetic-disk drives, despite their large capacity.

In 1986, Philips struck again with the **Green Book**, adding graphics and the ability to interleave audio, video, and data in the same sector, a feature essential for multimedia CD-ROMs.

The last piece of the CD-ROM puzzle is the file system. To make it possible to use the same CD-ROM on different computers, agreement was needed on CD-ROM file systems. To get this agreement, representatives of many computer companies met at Lake Tahoe in the High Sierras on the California-Nevada boundary and devised a file system that they called **High Sierra**. It later evolved into an International Standard (IS 9660). It has three levels. Level 1 uses file names of up to 8 characters optionally followed by an extension of up to 3 characters (the MS-DOS file naming convention). File names may contain only uppercase letters, digits, and the underscore. Directories may be nested up to eight deep, but directory names may not contain extensions. Level 1 requires all files to be contiguous, which is not a problem on a medium written only once. Any CD-ROM conformant to IS 9660 level 1 can be read using MS-DOS, an Apple computer, a UNIX computer, or just about any other computer. CD-ROM publishers regard this property as a big plus.

IS 9660 level 2 allows names up to 32 characters, and level 3 allows noncontiguous files. The Rock Ridge extensions (whimsically named after the town in the Mel Brooks film *Blazing Saddles*) allow very long names (for UNIX), UIDs, GIDs, and symbolic links, but CD-ROMs not conforming to level 1 will not be readable on old computers.

2.3.8 CD-Recordables

Initially, the equipment needed to produce a master CD-ROM (or audio CD, for that matter) was extremely expensive. But in the computer industry nothing stays expensive for long. By the mid 1990s, CD recorders no bigger than a CD player were a common peripheral available in most computer stores. These devices were still different from magnetic disks because once written, CD-ROMs could not be erased. Nevertheless, they quickly found a niche as a backup medium for large magnetic hard disks and also allowed individuals or startup companies to

manufacture their own small-run CD-ROMs (hundreds, not thousands) or make masters for delivery to high-volume commercial CD duplication plants. These drives are known as **CD-R**s (**CD-Recordables**).

Physically, CD-Rs start with 120-mm polycarbonate blanks that are like CD-ROMs, except that they contain a 0.6-mm-wide groove to guide the laser for writing. The groove has a sinusoidal excursion of 0.3 mm at a frequency of exactly 22.05 kHz to provide continuous feedback so the rotation speed can be accurately monitored and adjusted if need be. The first CD-Rs looked like regular CD-ROMs, except that they were colored gold on top instead of silver. The gold color came from the use of real gold instead of aluminum for the reflective layer. Unlike silver CDs, which have physical depressions, on CD-Rs the differing reflectivity of pits and lands has to be simulated. This is done by adding a layer of dye between the polycarbonate and the reflective layer, as shown in Fig. 2-27. Two kinds of dye are used: cyanine, which is green, and pthalocyanine, which is a yellowish orange. Chemists can argue endlessly about which one is better. Eventually, an aluminum reflective layer replaced the gold one.

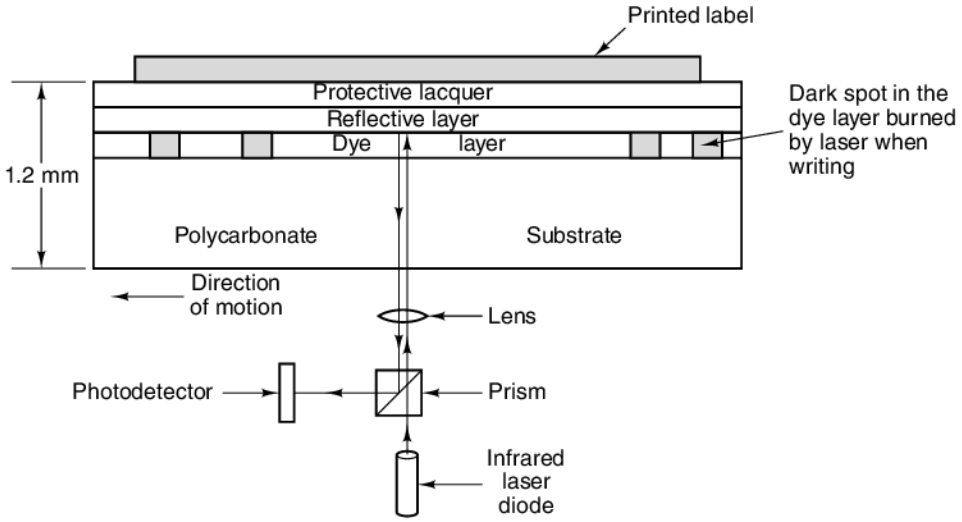


Figure 2-27. Cross section of a CD-R disk and laser (not to scale). A CD-ROM has a similar structure, except without the dye layer and with a pitted aluminum layer instead of a reflective layer.

In its initial state, the dye layer is transparent and lets the laser light pass through and reflect off the reflective layer. To write, the CD-R laser is turned up to high power (8–16 mW). When the beam hits a spot of dye, it heats up, breaking a chemical bond. This change to the molecular structure creates a dark spot. When read back (at 0.5 mW), the photodetector sees a difference between the dark spots where the dye has been hit and transparent areas where it is intact. This difference

is interpreted as the difference between pits and lands, even when read back on a regular CD-ROM reader or even on an audio CD player.

No new kind of CD could hold up its head with pride without a colored book, so CD-R has the **Orange Book**, published in 1989. This document defines CD-R and also a new format, **CD-ROM XA**, which allows CD-Rs to be written incrementally, a few sectors today, a few tomorrow, and a few next month. A group of consecutive sectors written at once is called a **CD-ROM track**.

One of the first uses of CD-R was for the Kodak PhotoCD. In this system the customer brings a roll of exposed film and his old PhotoCD to the photo processor and gets back the same PhotoCD with the new pictures added after the old ones. The new batch, which is created by scanning in the negatives, is written onto the PhotoCD as a separate CD-ROM track. Incremental writing is needed because the CD-R blanks are too expensive to provide a new one for every film roll.

However, incremental writing creates a new problem. Prior to the Orange Book, all CD-ROMs had a single **VTOC** (**Volume Table of Contents**) at the start. That scheme does not work with incremental (i.e., multitrack) writes. The Orange Book's solution is to give each CD-ROM track its own VTOC. The files listed in the VTOC can include some or all of the files from previous tracks. After the CD-R is inserted into the drive, the operating system searches through all the CD-ROM tracks to locate the most recent VTOC, which gives the current status of the disk. By including some, but not all, of the files from previous tracks in the current VTOC, it is possible to give the illusion that files have been deleted. Tracks can be grouped into **sessions**, leading to **multisession** CD-ROMs. Standard audio CD players cannot handle multisession CDs since they expect a single VTOC at the start.

CD-R makes it possible for individuals and companies to easily copy CD-ROMs (and audio CDs), possibly in violation of the publisher's copyright. Several schemes have been devised to make such piracy harder and to make it difficult to read a CD-ROM using anything other than the publisher's software. One of them involves recording all the file lengths on the CD-ROM as multigigabyte, thwarting any attempts to copy the files to hard disk using standard copying software. The true lengths are embedded in the publisher's software or hidden (possibly encrypted) on the CD-ROM in an unexpected place. Another scheme uses intentionally incorrect ECCs in selected sectors, in the expectation that CD copying software will "fix" the errors. The application software checks the ECCs itself, refusing to work if they are "correct." Nonstandard gaps between the tracks and other physical "defects" are also possibilities.

2.3.9 CD-Rewritables

Although people are used to other write-once media such as paper and photographic film, there is a demand for a rewritable CD-ROM. One technology now available is **CD-RW** (**CD-ReWritable**), which uses the same size media as CD-R.

However, instead of cyanine or pthalocyanine dye, CD-RW uses an alloy of silver, indium, antimony, and tellurium for the recording layer. This alloy has two stable states: crystalline and amorphous, with different reflectivities.

CD-RW drives use lasers with three different powers. At high power, the laser melts the alloy, converting it from the high-reflectivity crystalline state to the low-reflectivity amorphous state to represent a pit. At medium power, the alloy melts and reforms in its natural crystalline state to become a land again. At low power, the state of the material is sensed (for reading), but no phase transition occurs.

The reason CD-RW has not replaced CD-R is that the CD-RW blanks are more expensive than the CD-R blanks. Also, for applications consisting of backing up hard disks, the fact that once written, a CD-R cannot be accidentally erased is a feature, not a bug.

2.3.10 DVD

The basic CD/CD-ROM format has been around since 1980. By the mid-1990s optical media technology had improved dramatically, so higher-capacity video disks were becoming economically feasible. At the same time Hollywood was looking for a way to replace analog video tapes with an optical disk technology that had higher quality, was cheaper to manufacture, lasted longer, took up less shelf space in video stores, and did not have to be rewound. It was looking as if the wheel of progress for optical disks was about to turn once again.

This combination of technology and demand by three immensely rich and powerful industries has led to **DVD**, originally an acronym for **Digital Video Disk**, but now officially **Digital Versatile Disk**. DVDs use the same general design as CDs, with 120-mm injection-molded polycarbonate disks containing pits and lands that are illuminated by a laser diode and read by a photodetector. What is new is the use of

1. Smaller pits (0.4 microns versus 0.8 microns for CDs).
2. A tighter spiral (0.74 microns between tracks versus 1.6 microns for CDs).
3. A red laser (at 0.65 microns versus 0.78 microns for CDs).

Together, these improvements raise the capacity sevenfold, to 4.7 GB. A 1x DVD drive operates at 1.4 MB/sec (versus 150 KB/sec for CDs). Unfortunately, the switch to the red lasers used in supermarkets means that DVD players require a second laser to read existing CDs and CD-ROMs, which adds a little to the complexity and cost.

Is 4.7 GB enough? Maybe. Using MPEG-2 compression (standardized in IS 13346), a 4.7-GB DVD disk can hold 133 minutes of full-screen, full-motion video at high resolution (720×480), as well as soundtracks in up to eight languages and subtitles in 32 more. About 92 percent of all the movies Hollywood has ever made

are under 133 minutes. Nevertheless, some applications such as multimedia games or reference works may need more, and Hollywood would like to put multiple movies on the same disk, so four formats have been defined:

1. Single-sided, single-layer (4.7 GB).
2. Single-sided, dual-layer (8.5 GB).
3. Double-sided, single-layer (9.4 GB).
4. Double-sided, dual-layer (17 GB).

Why so many formats? In a word: politics. Philips and Sony wanted single-sided, dual-layer disks for the high-capacity version, but Toshiba and Time Warner wanted double-sided, single-layer disks. Philips and Sony did not think people would be willing to turn the disks over, and Time Warner did not believe putting two layers on one side could be made to work. The compromise: all combinations, with the market deciding which ones will survive. Well, the market has spoken. Philips and Sony were right. Never bet against technology.

The dual-layering technology has a reflective layer at the bottom, topped with a semireflective layer. Depending on where the laser is focused, it bounces off one layer or the other. The lower layer needs slightly larger pits and lands to be read reliably, so its capacity is slightly smaller than the upper layer's.

Double-sided disks are made by taking two 0.6-mm single-sided disks and gluing them together back to back. To make the thicknesses of all versions the same, a single-sided disk consists of a 0.6-mm disk bonded to a blank substrate (or perhaps in the future, one consisting of 133 minutes of advertising, in the hope that people will be curious as to what is down there). The structure of the double-sided, dual-layer disk is illustrated in Fig. 2-28.

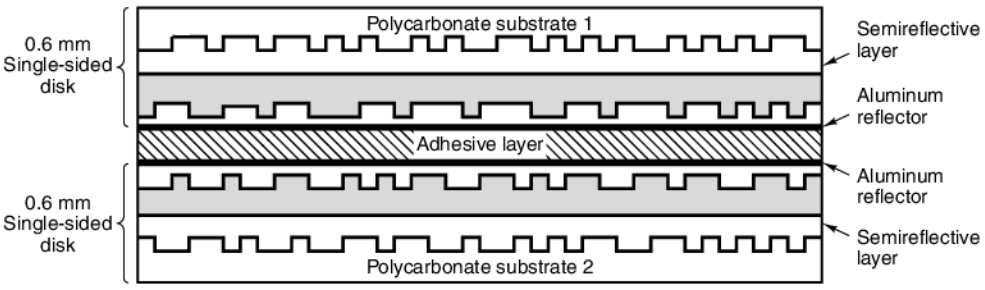


Figure 2-28. A double-sided, dual layer DVD disk.

DVD was devised by a consortium of 10 consumer electronics companies, seven of them Japanese, in close cooperation with the major Hollywood studios (some of which are owned by the Japanese electronics companies in the consortium). The computer and telecommunications industries were not invited to the

picnic, and the resulting focus was on using DVD for movie rentals. For example, standard features include real-time skipping of dirty scenes (to allow parents to turn a film rated NC17 into one safe for toddlers), six-channel sound, and support for Pan-and-Scan. The latter feature allows the DVD player to dynamically decide how to crop the left and right edges off movies (whose width:height ratio is 3:2) to fit on then-current television sets (whose aspect ratio was 4:3).

Another item the computer industry probably would not have thought of is an intentional incompatibility between disks intended for the United States and disks intended for Europe and yet other standards for other continents. Hollywood demanded this "feature" because new films are often released first in the United States and then physically shipped to Europe when the videos come out in the United States. The idea was to make sure European video stores could not buy videos in the U.S. too early, thereby reducing new movies' European theater sales. If Hollywood had been running the computer industry, we would have had 3.5-inch floppy disks in the United States and 9-cm floppy disks in Europe.

2.3.11 Blu-ray

Nothing stands still in the computer business, certainly not storage technology. DVD was barely introduced before its successor threatened to make it obsolete. The successor to DVD is **Blu-ray**, so called because it uses a blue laser instead of the red one used by DVDs. A blue laser has a shorter wavelength than a red one, which allows it to focus more accurately and thus support smaller pits and lands. Single-sided Blu-ray disks hold about 25 GB of data; double-sided ones hold about 50 GB. The data rate is about 4.5 MB/sec, which is good for an optical disk, but still insignificant compared to magnetic disks (cf. ATAPI-6 at 100 MB/sec and wide Ultra5 SCSI at 640 MB/sec). It is expected that Blu-ray will eventually replace CD-ROMs and DVDs, but this transition will take some years.

2.4 INPUT/OUTPUT

As we mentioned at the start of this chapter, a computer system has three major components: the CPU, the memories (primary and secondary), and the **I/O** (**Input/Output**) equipment such as printers, scanners, and modems. So far we have looked at the CPU and the memories. Now it is time to examine the I/O equipment and how it is connected to the rest of the system.

2.4.1 Buses

Physically, most personal computers and workstations have a structure similar to the one shown in Fig. 2-29. The usual arrangement is a metal box with a large printed circuit board at the bottom or side, called the **motherboard** (parentboard,

for the politically correct). The motherboard contains the CPU chip, some slots into which DIMM modules can be clicked, and various support chips. It also contains a bus etched along its length, and sockets into which the edge connectors of I/O boards can be inserted.

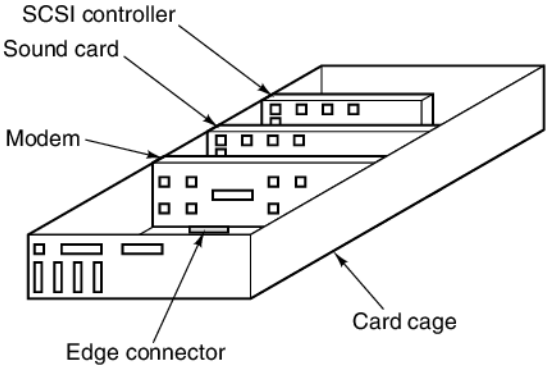


Figure 2-29. Physical structure of a personal computer.

The logical structure of a simple personal computer is shown in Fig. 2-30. This one has a single bus used to connect the CPU, memory, and I/O devices; most systems have two or more buses. Each I/O device consists of two parts: one containing most of the electronics, called the **controller**, and one containing the I/O device itself, such as a disk drive. The controller is usually integrated directly onto the motherboard or sometimes contained on a board plugged into a free bus slot. Even though the display (monitor) is not an option, the video controller is sometimes located on a plug-in board to allow the user to choose between boards with or without graphics accelerators, extra memory, and so on. The controller connects to its device by a cable attached to a connector on the back of the box.

The job of a controller is to control its I/O device and handle bus access for it. When a program wants data from the disk, for example, it gives a command to the disk controller, which then issues seeks and other commands to the drive. When the proper track and sector have been located, the drive begins outputting the data as a serial bit stream to the controller. It is the controller's job to break the bit stream up into units and write each unit into memory, as it is assembled. A unit is typically one or more words. A controller that reads or writes data to or from memory without CPU intervention is said to be performing **Direct Memory Access**, better known by its acronym **DMA**. When the transfer is completed, the controller normally causes an **interrupt**, forcing the CPU to immediately suspend running its current program and start running a special procedure, called an **interrupt handler**, to check for errors, take any special action needed, and inform the operating system that the I/O is now finished. When the interrupt handler is finished, the CPU continues with the program that was suspended when the interrupt occurred.

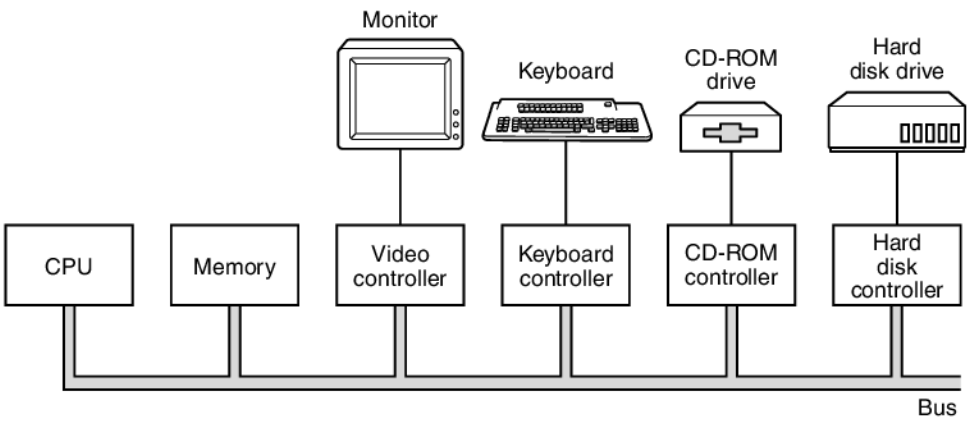


Figure 2-30. Logical structure of a simple personal computer.

The bus is used not only by the I/O controllers but also by the CPU for fetching instructions and data. What happens if the CPU and an I/O controller want to use the bus at the same time? The answer is that a chip called a **bus arbiter** decides who goes next. In general, I/O devices are given preference over the CPU, because disks and other moving devices cannot be stopped, and forcing them to wait would result in lost data. When no I/O is in progress, the CPU can have all the bus cycles for itself to reference memory. However, when some I/O device is also running, that device will request and be granted the bus when it needs it. This process is called **cycle stealing** and it slows down the computer.

This design worked fine for the first personal computers, since all the components were roughly in balance. However, as the CPUs, memories, and I/O devices got faster, a problem arose: the bus could no longer handle the load presented. On a closed system, such as an engineering workstation, the solution was to design a new and faster bus for the next model. Because nobody ever moved I/O devices from an old model to a new one, this approached worked fine.

However, in the PC world, people often upgraded their CPU but wanted to move their printer, scanner, and modem to the new system. Also, a huge industry had grown up around providing a vast range of I/O devices for the IBM PC bus, and this industry had exceedingly little interest in throwing out its entire investment and starting over. IBM learned this the hard way when it brought out the successor to the IBM PC, the PS/2 range. The PS/2 had a new, faster bus, but most clone makers continued to use the old PC bus, now called the **ISA** (**Industry Standard Architecture**) bus. Most disk and I/O device makers also continued to make controllers for it, so IBM found itself in the peculiar situation of being the only PC maker that was no longer IBM compatible. Eventually, it was forced back to supporting the ISA bus. Today the ISA bus has been relegated to ancient systems and computer museums, since it has been replaced by newer and faster standard bus

architectures. As an aside, please note that ISA stands for Instruction Set Architecture in the context of machine levels, whereas it stands for Industry Standard Architecture in the context of buses.

The PCI and PCIe Buses

Nevertheless, despite the market pressure not to change anything, the old bus really was too slow, so something had to be done. This situation led to other companies developing machines with multiple buses, one of which was the old ISA bus, or its backward-compatible successor, the **EISA** (**Extended ISA**) bus. The winner was the **PCI** (**Peripheral Component Interconnect**) bus. It was designed by Intel, but Intel decided to put all the patents in the public domain, to encourage the entire industry (including its competitors) to adopt it.

The PCI bus can be used in many configurations, but a typical one is illustrated in Fig. 2-31. Here the CPU talks to a memory controller over a dedicated high-speed connection. The controller talks to the memory and to the PCI bus directly, so CPU-memory traffic does not go over the PCI bus. Other peripherals connect to the PCI bus directly. A machine of this design would typically contain two or three empty PCI slots to allow customers to plug in PCI I/O cards for new peripherals.

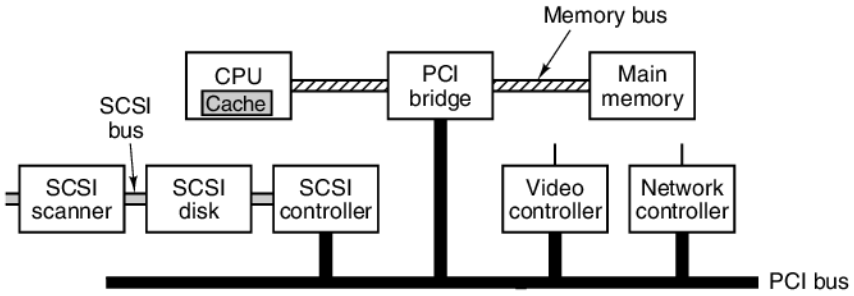


Figure 2-31. A typical PC built around the PCI bus. The SCSI controller is a PCI device.

No matter how fast something is in the computer world, a lot of people think it is too slow. This fate also befell the PCI bus, which is being replaced by **PCI Express**, abbreviated as **PCIe**. Most modern computers support both, so users can attach new, fast devices to the PCIe bus and older, slower ones to the PCI bus.

While the PCI bus was just an upgrade to the older ISA bus with higher speeds and more bits transferred in parallel, PCIe represents a radical change from the PCI bus. In fact, it is not even a bus at all. It is point-to-point network using bit-serial lines and packet switching, more like the Internet than like a traditional bus. It's architecture is shown in Fig. 2-32.

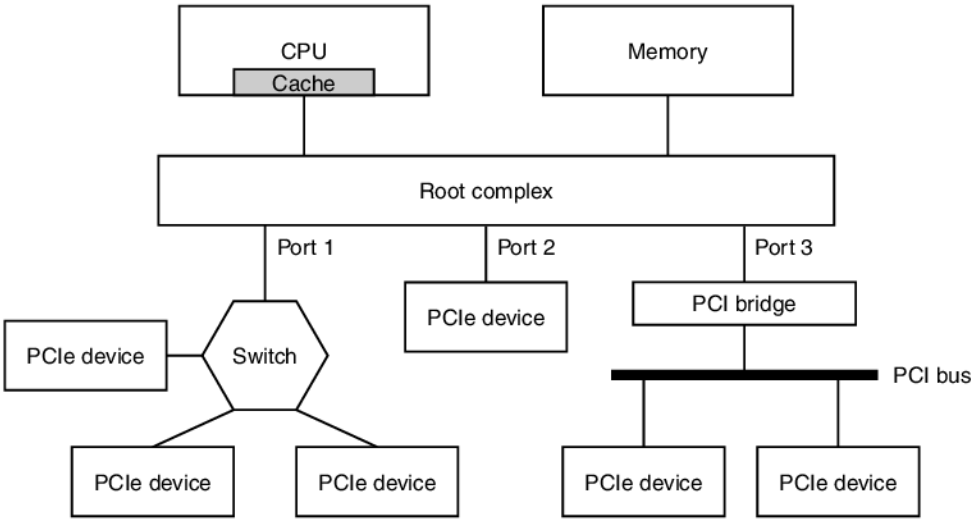


Figure 2-32. Sample architecture of a PCIe system with three PCIe ports.

Several things stand out immediately about PCIe. First, the connections between the devices are serial, that is, 1-bit wide rather than 8-, 16-, 32-, or 64-bits wide. While one might think that a 64-bit-wide connection would have a higher bandwidth than a 1-bit wide connection, in practice, differences in propagation time of the 64 bits, called the **skew**, means relatively low speeds have to be used. With a serial connection much higher speeds can be used and this more than offsets the loss of parallelism. PCI buses run at a maximum clock rate of 66 MHz. With 64 bits transferred per cycle, the data rate is 528 MB/sec. With a clock rate of 8 Gbps, even with serial transfer, the data rate of PCIe is 1 GB/sec. Furthermore, devices are not limited to a single wire pair to communicate with the root complex or a switch. A device can have up to 32 wire pairs, called **lanes**. These lanes are not synchronous, so skew is not important here. Most motherboards have a 16-lane slot for the graphics card, which in PCIe 3.0 will give the graphics card a bandwidth of 16 GB/sec, about 30 times faster than what a PCI graphics card can get. This bandwidth is necessary for increasingly demanding applications, such as 3D.

Second, all communication is point to point. When the CPU wants to talk to a device, it sends a packet to the device and generally later gets an answer. The packet goes through the root complex, which is on the motherboard, and then on to the device, possibly through a switch (or if the device is a PCI device, through the PCI bridge). This evolution from a system in which all devices listened to the same bus to one using point-to-point communications parallels the development of Ethernet (a popular local area network), which also started with a broadcast channel but now uses switches to enable point-to-point communication.

2.4.2 Terminals

Many kinds of I/O devices are available today. A few of the common ones are discussed below. Computer terminals consist of two parts: a keyboard and a monitor. In the mainframe world, these parts are often integrated into a single device and attached to the main computer by a serial line or over a telephone line. In the airline reservation, banking, and other mainframe-oriented industries, these devices are still in use. In the personal computer world, the keyboard and monitor are independent devices. Either way, the technology of the two parts is the same.

Keyboards

Keyboards come in several varieties. The original IBM PC came with a keyboard that had a snap-action switch under each key that gave tactile feedback and made a click when the key was depressed far enough. Nowadays, the cheaper keyboards have keys that just make mechanical contact when depressed. Better ones have a sheet of elastometric material (a kind of rubber) between the keys and the underlying printed circuit board. Under each key is a small dome that buckles when depressed far enough. A small spot of conductive material inside the dome closes the circuit. Some keyboards have a magnet under each key that passes through a coil when struck, thus inducing a current that can be detected. Various other methods, both mechanical and electromagnetic, are also in use.

On personal computers, when a key is depressed, an interrupt is generated and the keyboard interrupt handler (a piece of software that is part of the operating system) is started. The interrupt handler reads a hardware register inside the keyboard controller to get the number of the key (1 through 102) that was just depressed. When a key is released, a second interrupt is caused. Thus if a user depresses the SHIFT key, then depresses and releases the M key, then releases the SHIFT key, the operating system can see that the user wants an uppercase "M" rather than a lowercase "m." Handling of multikey sequences involving SHIFT, CTRL, and ALT is done entirely in software (including the infamous CTRL-ALT-DEL key sequence that is used to reboot PCs).

Touch Screens

While keyboards are in no danger of going the way of the manual typewriter, there is a new kid on the block when it comes to computer input: the touch screen. While these devices only became mass-market items with the introduction of Apple's iPhone in 2007, they go back much further. The first touch screen was developed at the Royal Radar Establishment in Malvern, U.K. in 1965. Even the much-heralded pinching capability of the iPhone dates back to work at the University of Toronto in 1982. Since then, many different technologies have been developed and marketed.

Touch devices fall into two categories: opaque and transparent. A typical opaque touch device is the touchpad on a notebook computer. A typical transparent device is the screen of a smart phone or tablet. We will only consider the latter here. They are usually called **touch screens**. The major types of touch screens are infrared, resistive, and capacitive.

Infrared screens have infrared transmitters, such as infrared light emitting diodes or lasers on (for example) the left and top edges of the bezel around the screen and detectors on the right and bottom edges. When a finger, stylus, or any opaque object blocks one or more beams, the corresponding detector senses the drop in signal and the hardware of the device can tell the operating system which beams have been blocked, allowing it to compute the (x, y) coordinates of the finger or stylus. While these devices have a long history and are still in use in kiosks and other applications, they are not used for mobile devices.

Another old technology consists of **resistive touch screens**. These consist of two layers, the top one of which is flexible. It contains a large number of horizontal wires. The one under it contains vertical wires. When a finger or other object depresses a point on the screen, one or more of the upper wires comes in contact with (or close to) the perpendicular wires in the lower layer. The electronics of the device make it possible to read out which area has been depressed. These screens can be built very inexpensively and are widely used in price-sensitive applications.

Both of these technologies are fine when the screen is pressed by one finger but have a problem when two fingers are used. To describe the problem, we will use the terminology of the infrared touch screen but the resistive one has the same problem. Imagine that the two fingers are at (3, 3) and (8, 8). As a result, the $x = 3$ and $x = 8$ vertical beams are interrupted as are the $y = 3$ and $y = 8$ horizontal beams. Now consider a different scenario with the fingers at (3, 8) and (8, 3), which are the opposite corners of the rectangle whose corners are (3, 3), (8, 3), (8, 8), and (3, 8). Precisely the same beams are blocked, so the software has no way of telling which of the two scenarios holds. This problem is called **ghosting**.

To be able to detect multiple fingers at the same time—a property required for pinching and expanding gestures—a new technology was needed. The one used on most smart phones and tablets (but not on digital cameras and other devices) is the **projected capacitive touch screen**. There are various types but the most common one is the **mutual capacitance** type. All touch screens that can detect two or more points of contact at the same time are known as **multitouch screens**. Let us now briefly see how they work.

For readers who are a bit rusty on their high-school physics, a **capacitor** is a device that can store electric charge. A simple one has two conductors separated by an insulator. In modern touch screens, a grid-like pattern of thin "wires" running vertically is separated from a horizontal grid by a thin insulating layer. When a finger touches the screen, it changes the capacitance at all the intersections touched (possibly far apart). This change can be measured. As a demonstration that a modern touch screen is not like the older infrared and resistive screens, try touching

one with a pen, pencil, paper clip, or gloved finger and you will see that nothing happens. The human body is good at storing electric charge, as anyone who has shuffled across a rug on a cold, dry day and then touched a metal doorknob can painfully testify. Plastic, wooden, and metal instruments are not nearly as good as people in terms of their capacitance.

The "wires" in a touch screen are not the usual copper wires found in normal electrical devices since they would block the light from the screen. Instead they are thin (typically 50 micron) strips of transparent, conducting indium tin oxide bonded to opposite sides of a thin glass plate, which together form the capacitors. In some newer designs, the insulating glass plate is replaced by a thin layer of silicon dioxide (sand!), with the three layers sputtered (sprayed, atom by atom) onto some substrate. Either way, the capacitors are protected from dirt and scratching by a glass plate placed above, to form the surface of the screen to be touched. The thinner the upper glass plate, the more sensitive the performance but the more fragile the device is.

In operation, voltages are applied alternately to the horizontal and vertical "wires" while the voltage values, which are affected by the capacitance of each intersection, are read off the other ones. This operation is repeated many times per second with the coordinates touched fed to the device driver as a stream of (x, y) pairs. Further processing, such as determining whether pointing, pinching, expanding, or swiping is taking place is done by the operating system. If you use all 10 fingers, and bring a friend to add some more, the operating system will have its hands full, but the multitouch hardware will be up to the job.

Flat Panel Displays

The first computer monitors used **cathode ray tubes** (**CRT**s), just like old television sets. They were far too bulky and heavy to be used in notebook computers, so a more compact display technology had to be developed for their screens. The development of flat panel displays provided the compact form factor necessary for notebooks, and these devices also used less power. Today the size and power benefits of the flat panel display have all but eliminated the use of CRT monitors.

The most common flat panel display technology is the **LCD** (**Liquid Crystal Display**). It is highly complex, has many variations, and is changing rapidly, so this description will, of necessity, be brief and greatly simplified.

Liquid crystals are viscous organic molecules that flow like a liquid but also have spatial structure, like a crystal. They were discovered by an Austrian botanist, Friedrich Reinitzer, in 1888 and first applied to displays (e.g., calculators, watches) in the 1960s. When all the molecules are lined up in the same direction, the optical properties of the crystal depend on the direction and polarization of the incoming light. Using an applied electric field, the molecular alignment, hence the optical properties, can be changed. In particular, by shining a light through a liquid

crystal, the intensity of the light exiting from it can be controlled electrically. This property can be exploited to construct flat panel displays.

An LCD display screen consists of two parallel glass plates between which is a sealed volume containing a liquid crystal. Transparent electrodes are attached to both plates. A light behind the rear plate (either natural or artificial) illuminates the screen from behind. The transparent electrodes attached to each plate are used to create electric fields in the liquid crystal. Different parts of the screen get different voltages, to control the image displayed. Glued to the front and rear of the screen are polarizing filters because the display technology requires the use of polarized light. The general setup is shown in Fig. 2-33(a).

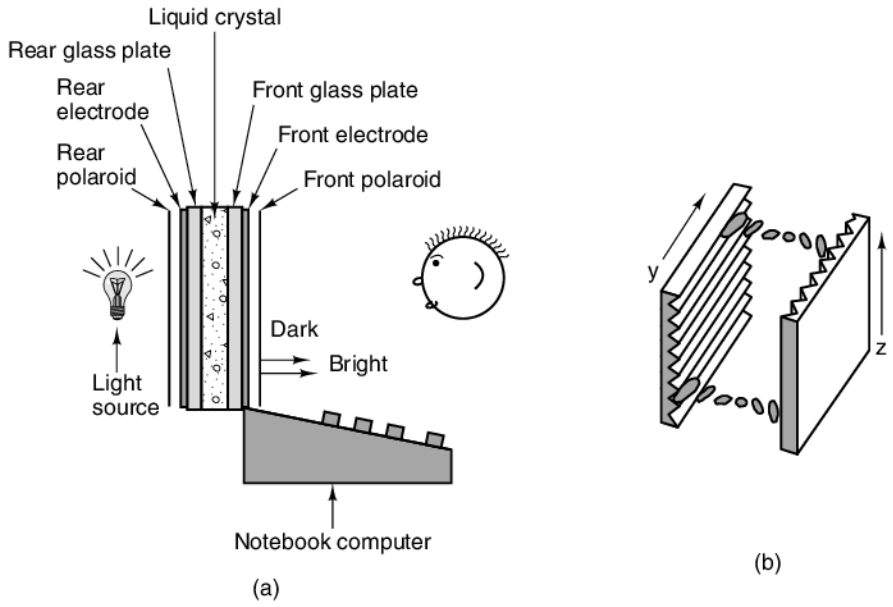


Figure 2-33. (a) The construction of an LCD screen. (b) The grooves on the rear and front plates are perpendicular to one another.

Although many kinds of LCD displays are in use, we will now consider one particular kind, the **TN** (**Twisted Nematic**) display, as an example. In this display, the rear plate contains tiny horizontal grooves and the front plate contains tiny vertical grooves, as illustrated in Fig. 2-33(b). In the absence of an electric field, the LCD molecules tend to align with the grooves. Since the front and rear alignments differ by 90 degrees, the molecules (and thus the crystal structure) twist from rear to front.

At the rear of the display is a horizontal polarizing filter. It allows in only horizontally polarized light. At the front of the display is a vertical polarizing filter. It allows only vertically polarized light to pass through. If there were no liquid pres-

ent between the plates, horizontally polarized light let in by the rear polarizing filter would be blocked by the front polarizing filter, making the screen black.

However the twisted crystal structure of the LCD molecules guides the light as it passes and rotates its polarization, making it come out vertically. Thus in the absence of an electric field, the LCD screen is uniformly bright. By applying a voltage to selected parts of the plate, the twisted structure can be destroyed, blocking the light in those parts.

Two schemes can be used for applying the voltage. In a (low-cost) **passive matrix display**, both electrodes contain parallel wires. In a 1920×1080 display (the size for full high-definition video), for example, the rear electrode might have 1920 vertical wires and the front one 1080 horizontal ones. By putting a voltage on one of the vertical wires and then pulsing one of the horizontal ones, the voltage at one selected pixel position can be changed, making it go dark briefly. A **pixel** (originally a picture element, is a colored dot from which all digital images are built). By repeating this pulse with the next pixel and then the next one, a dark scan line can be painted. Normally, the entire screen is painted 60 times a second to fool the eye into thinking there is a constant image there.

The other scheme in widespread use is the **active matrix display**. It is more expensive but it gives a better image. Instead of just having two sets of perpendicular wires, it has a tiny switching element at each pixel position on one of the electrodes. By turning these on and off, an arbitrary voltage pattern can be created across the screen, allowing for an arbitrary bit pattern. The switching elements are called **thin film transistors** and the flat panel displays using them are often called **TFT displays**. Most notebook computers and stand-alone flat panel displays for desktop computers use TFT technology now.

So far we have described how a monochrome display works. Suffice it to say that color displays use the same general principles as monochrome displays but the details are a great deal more complicated. Optical filters are used to separate the white light into red, green, and blue components at each pixel position so these can be displayed independently. Every color can be built up from a linear superposition of these three primary colors.

Still new screen technologies are on the horizon. One of the more promising is the **Organic Light Emitting Diode** (**OLED**) display. It consists of layers of electrically charged organic molecules sandwiched between two electrodes. Voltage changes cause the molecules to get excited and move to higher energy states. When they drop back to their normal state, they emit light. More detail is beyond the scope of this book (and its authors).

Video RAM

Most monitors are refreshed 60–100 times per second from a special memory, called a **video RAM**, on the display's controller card. This memory has one or more bit maps that represent the screen image. On a screen with, say, 1920×1080

pixels, the video RAM would contain 1920×1080 values, one for each pixel. In fact, it might contain many such bit maps, to allow rapid switching from one screen image to another.

On a garden-variety display, each pixel would be represented as a 3-byte RGB value, one each for the intensity of the red, green, and blue components of the pixel's color (high-end displays use 10 or more bits per color). From the laws of physics, it is known that any color can be constructed from a linear superposition of red, green, and blue light.

A video RAM with 1920×1080 pixels at 3 bytes/pixel requires over 6.2 MB to store the image and a fair amount of CPU time to do anything with it. For this reason, some computers compromise by using an 8-bit number to indicate the color desired. This number is then used as an index into a hardware table called the **color palette** that contains 256 entries, each holding a 24-bit RGB value. Such a design, called **indexed color**, reduces the video RAM memory requirements by 2/3, but allows only 256 colors on the screen at once. Usually, each window on the screen has its own mapping, but with only one hardware color palette, often when multiple windows are present on the screen, only the current one has its colors rendered correctly. Color palettes with 2^{16} entries are also used, but the gain here is only 1/3.

Bit-mapped video displays require a lot of bandwidth. To display full-screen, full-color multimedia on a 1920×1080 display requires copying 6.2 MB of data to the video RAM for every frame. For full-motion video, a rate of at least 25 frame/sec is needed, for a total data rate of 155 MB/sec. This load is more than the original PCI bus could handle (132 MB/sec) but PCIe can handle it with ease.

2.4.3 Mice

As time goes on, computers are being used by people with less expertise in how computers work. Computers of the ENIAC generation were used only by the people who built them. In the 1950s, computers were only used by highly skilled professional programmers. Now, computers are widely used by people who need to get some job done and do not know (or even want to know) much about how computers work or how they are programmed.

In the old days, most computers had command line interfaces, to which users typed commands. Since people who are not computer specialists often perceived command line interfaces as user-unfriendly, if not downright hostile, many computer vendors developed point-and-click interfaces, such as the Macintosh and Windows. Using this model requires having a way to point at the screen. The most common way of allowing users to point at the screen is with a mouse.

A **mouse** is a small plastic box that sits on the table next to the keyboard. When it is moved around on the table, a little pointer on the screen moves too, allowing users to point at screen items. The mouse has one, two, or three buttons on top, to allow users to select items from menus. Much blood has been spilled as a

result of arguments about how many buttons a mouse ought to have. Naive users prefer one (it is hard to push the wrong button if there is only one), but sophisticated ones like the power of multiple buttons to do fancy things.

Three kinds of mice have been produced: mechanical mice, optical mice, and optomechanical mice. The first mice had two rubber wheels protruding through the bottom, with their axles perpendicular to one another. When the mouse was moved parallel to its main axis, one wheel turned. When it is moved perpendicular to its main axis, the other one turned. Each wheel drove a variable resistor or potentiometer. By measuring changes in the resistance, it was possible to see how much each wheel had rotated and thus calculate how far the mouse had moved in each direction. Later, this design was been replaced by one in which a ball that protruded slightly from the bottom was used instead of wheels. It is shown in Fig. 2-34.

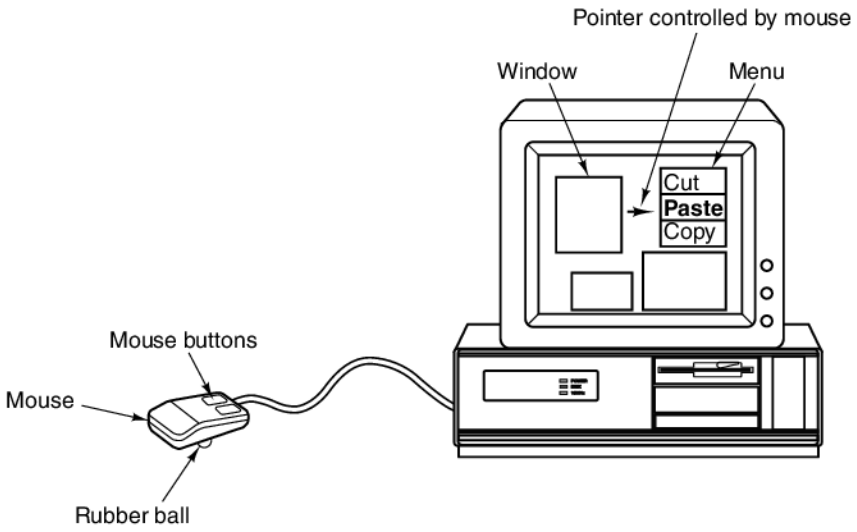


Figure 2-34. A mouse being used to point to menu items.

The second kind of mouse is the optical mouse. This kind has no wheels or ball. Instead, it has an **LED** (**Light Emitting Diode**) and a photodetector on the bottom. Early optical mice required a mouse pad with closely spaced lines on it to detect how many lines had been crossed and thus how far the mouse had moved. Modern optical mice contain an LED that illuminates the imperfections of the underlying surface along with a tiny video camera that records a small image (typically 18×18 pixels) up to 1000 times/sec. Consecutive images are compared to see how far the mouse has moved. Some optical mice use a laser instead of an LED for illumination. They are more accurate, but also more expensive.

The third kind of mouse is optomechanical. Like the newer mechanical mouse, it has a rolling ball that turns two shafts aligned at 90 degrees to each other.

The shafts are connected to encoders that have slits through which light can pass. As the mouse moves, the shafts rotate, and light pulses strike the detectors whenever a slit comes between an LED and its detector. The number of pulses detected is proportional to the amount of motion.

Although mice can be set up in various ways, a common arrangement is to have the mouse send a sequence of 3 bytes to the computer every time the mouse moves a certain minimum distance (e.g., 0.01 inch), sometimes called a **mickey**. Usually, these characters come in on a serial line, one bit at time. The first byte contains a signed integer telling how many units the mouse has moved in the x-direction since the last time. The second byte gives the same information for y motion. The third byte contains the current state of the mouse buttons. Sometimes 2 bytes are used for each coordinate.

Low-level software in the computer accepts this information as it comes in and converts the relative movements sent by the mouse to an absolute position. It then displays an arrow on the screen at the position corresponding to where the mouse is. When the arrow points at the proper item, the user clicks a mouse button, and the computer can then figure out which item has been selected from its knowledge of where the arrow is on the screen.

2.4.4 Game Controllers

Video games typically have heavy user I/O demands, and in the video console market specialized input devices have been developed. In this section we look at two recent developments in video game controllers, the Nintendo Wiimote and the Microsoft Kinect.

Wiimote Controller

First released in 2006 with the Nintendo Wii game console, the Wiimote controller contains traditional gamepad buttons plus a dual motion-sensing capability. All interactions with the Wiimote are sent in real time to the game console using an internal Bluetooth radio. The motion sensors in the Wiimote allow it to sense its own movement in three dimensions, plus when pointed at the television it provides a fine-grained pointing capability.

Figure 2-35 illustrates how the Wiimote implements this motion-sensing function. Tracking of the Wiimote's movement in three dimensions is accomplished with an internal 3-axis accelerometer. This device contains three small masses, each of which can move in the x, y, and z axis (with respect to the accelerometer chip). These masses move in proportion to the degree of acceleration in their particular axis, which changes the capacitance of the mass with respect to a fixed metal wall. By measuring these three changing capacitances, it becomes possible to sense acceleration in three dimensions. Using this technology and some classic calculus, the Wii console can track the Wiimote's movement in space. As you

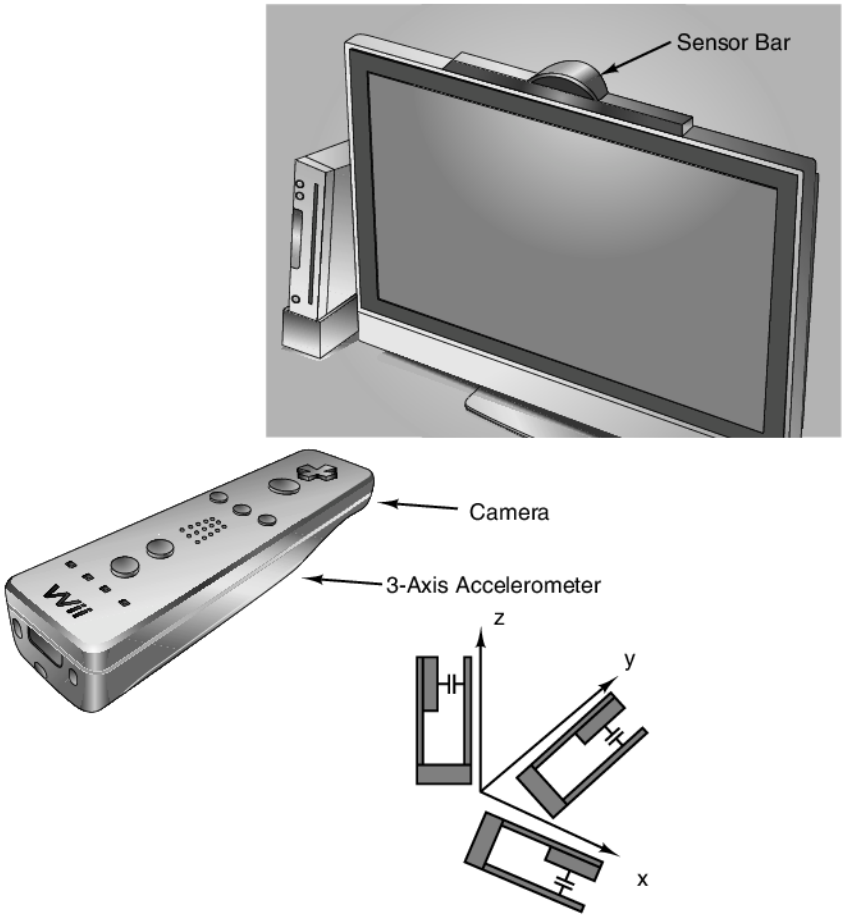


Figure 2-35. The Wiimote video game controller motion sensors.

swing the Wiimote to hit a virtual tennis ball, the motion of the Wiimote is tracked as you swing toward the ball, and if you twist your wrist at the last moment to put topspin on the ball, the Wiimote accelerometers will sense that motion as well.

While the accelerometers perform well at tracking the motion of the Wiimote as it moves in three dimensions, they cannot provide the fine-grained motion sensing necessary to control a pointer on the television screen. The accelerometers suffer from unavoidable tiny errors in their acceleration measurements, thus over time the exact location of the Wiimote (based on integration of its accelerations) will become increasingly inaccurate.

To provide fine-grained motion sensing, the Wiimote utilizes a clever computer vision technology. Sitting atop the television is a "sensor bar" which contains LEDs a fixed width apart. Contained in the Wiimote is a camera that when pointed

at the sensor bar can deduce the distance and orientation of the Wiimote with respect to the television. Since the sensor bar's LEDs are a fixed distance apart, their distance as viewed by the Wiimote is proportional to the Wiimote's distance from the sensor bar. The location of the sensor bar in the Wiimote's field of view indicates the direction that the Wiimote is pointing with respect to the television. By continuously sensing this orientation, it is possible to support a fine-grained pointing capability without the positional errors inherent to accelerometers.

Kinect Controller

The Microsoft Kinect takes the computer vision capabilities of game controllers to a whole new level. This device uses computer vision alone to determine the user's interactions with the game console. It works by sensing the user's position in the room, plus the orientation and motion of their body. Games are controlled by making predetermined motions with your hands, arms, and whatever else the game designers think you should flail in an effort to control their game.

The sensing capability of the Kinect is based on a depth camera combined with a video camera. The depth camera computes the distance of object in the Kinect's field of view. It does this by emitting a two-dimensional array of infrared laser dots, then capturing their reflections with an infrared camera. Using a computer vision technique called "structured lighting," the Kinect can determine the distance of the objects in its view based on how the stipple of infrared dots is disturbed by the lighted surfaces.

Depth information is combined with the texture information returned from the video camera to produce a textured depth map. This map can then be processed by computer vision algorithms to locate the people in the room (even recognizing their faces) and the orientation and motion of their body. After processing, information about the persons in the room is sent to the game console which uses this data to control the video game.

2.4.5 Printers

Having prepared a document or fetched a page from the World Wide Web, users often want to print it, so many computers can be equipped with a printer. In this section we will describe some of the more common kinds of printers.

Laser Printers

Probably the most exciting development in printing since Johann Gutenberg invented movable type in the fifteenth century is the **laser printer**. This device combines a high-quality image, excellent flexibility, great speed, and moderate cost into a single compact peripheral. Laser printers use almost the same technology as

photocopy machines. In fact, many companies make devices that combine copying and printing (and sometimes fax as well).

The basic technology is illustrated in Fig. 2-36. The heart of the printer is a rotating precision drum (or in some high-end systems, a belt). At the start of each page cycle, it is charged up to about 1000 volts and coated with a photosensitive material. Then light from a laser is scanned along the length of the drum by reflecting it of a rotating octagonal mirror. The light beam is modulated to produce a pattern of light and dark spots. The spots where the beam hits lose their charge.

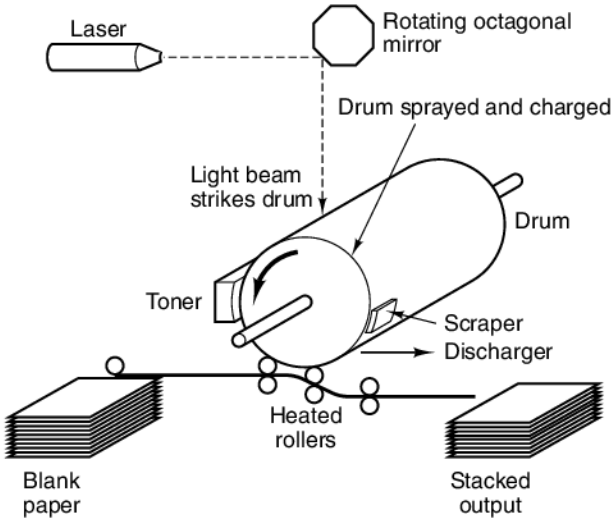


Figure 2-36. Operation of a laser printer.

After a line of dots has been painted, the drum rotates a fraction of a degree to allow the next line to be painted. Eventually, the first line of dots reaches the toner, a reservoir of an electrostatically sensitive black powder. The toner is attracted to those dots that are still charged, thus forming a visual image of that line. A little later in the transport path, the toner-coated drum is pressed against the paper, transferring the black powder to the paper. The paper is then passed through heated rollers to fuse the toner to the paper permanently, fixing the image. Later in its rotation, the drum is discharged and scraped clean of any residual toner, preparing it for being charged and coated again for the next page.

That this process is an exceedingly complex combination of physics, chemistry, mechanical engineering, and optical engineering hardly needs to be said. Nevertheless, complete assemblies, called **print engines**, are available from several vendors. Laser printer manufacturers combine the print engines with their own electronics and software to make a complete printer. The electronics consist of a fast embedded CPU along with megabytes of memory to hold a full-page bit map and numerous fonts, some of them built in and some of them downloadable. Most

printers accept commands that describe the pages to be printed (as opposed to simply accepting bit maps prepared by the main CPU). These commands are given in languages such as HP's PCL and Adobe's PostScript or PDF, which are complete, albeit specialized, programming languages.

Laser printers at 600 dpi and up can do a reasonable job of printing black and white photographs but the technology is trickier than it might at first appear. Consider a photograph scanned in at 600 dpi that is to be printed on a 600-dpi printer. The scanned image contains 600×600 pixels/inch, each one consisting of a gray value from 0 (white) to 255 (black). The printer can also print 600 dpi, but each printed pixel is either black (toner present) or white (no toner present). Gray values cannot be printed.

The usual solution to printing images with gray values is to use **halftoning**, the same as commercially printed posters. The image is broken up into halftone cells, each typically 6×6 pixels. Each cell can contain between 0 and 36 black pixels. The eye perceives a cell with many pixels as darker than one with fewer pixels. Gray values in the range 0 to 255 are represented by dividing this range into 37 zones. Values from 0 to 6 are in zone 0, values from 7 to 13 are in zone 1, and so on (zone 36 is slightly smaller than the others because 37 does not divide 256 exactly). Whenever a gray value in zone 0 is encountered, its halftone cell on the paper is left blank, as illustrated in Fig. 2-37(a). A zone-1 value is printed as 1 black pixel. A zone-2 value is printed as 2 black pixels, as shown in Fig. 2-37(b). Other zone values are shown in Fig. 2-37(c)–(f). Of course, taking a photograph scanned at 600 dpi and halftoning this way reduces the effective resolution to 100 cells/inch, called the **halftone screen frequency**, conventionally measured in **lpi** (**lines per inch**).

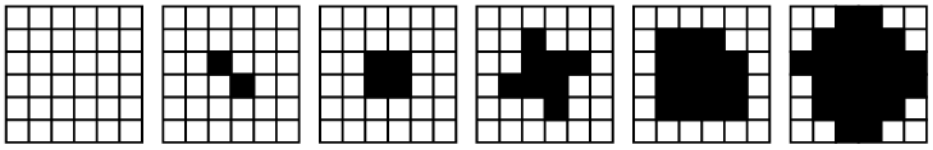


Figure 2-37. Halftone dots for various grayscale ranges. (a) 0–6. (b) 14–20. (c) 28–34. (d) 56–62. (e) 105–111. (f) 161–167.

Color Printing

Although most laser printers are monochrome, color laser printers are starting to become more common, so some explanation of color printing (also applicable to inkjet and other printers) is perhaps useful here. As you might imagine, it is not trivial. Color images can be viewed in one of two ways: transmitted light and reflected light. Transmitted-light images, such as those produced on monitors, are

built up from the linear superposition of the three additive primary colors, which are red, green, and blue.

In contrast, reflected-light images, such as color photographs and pictures in glossy magazines, absorb certain wavelengths of light and reflect the rest. These are built up from a linear superposition of the three subtractive primary colors, cyan (all red absorbed), magenta (all green absorbed), and yellow (all blue absorbed). In theory, every color can be produced by mixing cyan, yellow, and magenta ink. In practice it is difficult to get the inks pure enough to absorb all light and produce a true black. For this reason, nearly all color printing systems use four inks: cyan, magenta, yellow, and black. These systems are called **CMYK printers**. The K is sometimes attributed to blacK but it really stands for the Key plate with which the color plates are aligned in conventional four-color printing presses. Monitors, in contrast, use transmitted light and the RGB system for producing colors.

The complete set of colors that a display or printer can produce is called its **gamut**. No device has a gamut that matches the real world, since typically each color comes in 256 intensities, giving only 16,777,216 discrete colors. Imperfections in the technology reduce the total more, and the remaining ones are not always uniformly spaced over the color spectrum. Furthermore, color perception has a lot to do with how the rods and cones in the human retina work, and not just the physics of light.

As a consequence of the above observations, converting a color image that looks fine on the screen to an identical printed one is far from trivial. Among the problems are

1. Color monitors use transmitted light; color printers use reflected light.
2. Monitors have 256 intensities per color; color printers must halftone.
3. Monitors have a dark background; paper has a light background.
4. The RGB gamut of a monitor and the CMYK gamut of a printer are different.

Getting printed color images to match real life (or even to match screen images) requires hardware device calibration, sophisticated software for building and using International Color Consortium profiles, and considerable expertise on the part of the user.

Inkjet Printers

For low-cost home printing, **inkjet printers** are a favorite. The movable print head, which holds the ink cartridges, is swept horizontally across the paper by a belt while ink is sprayed from tiny nozzles. The ink droplets have a volume of about 1 picoliter, so 100 million of them fit in a single drop of water.