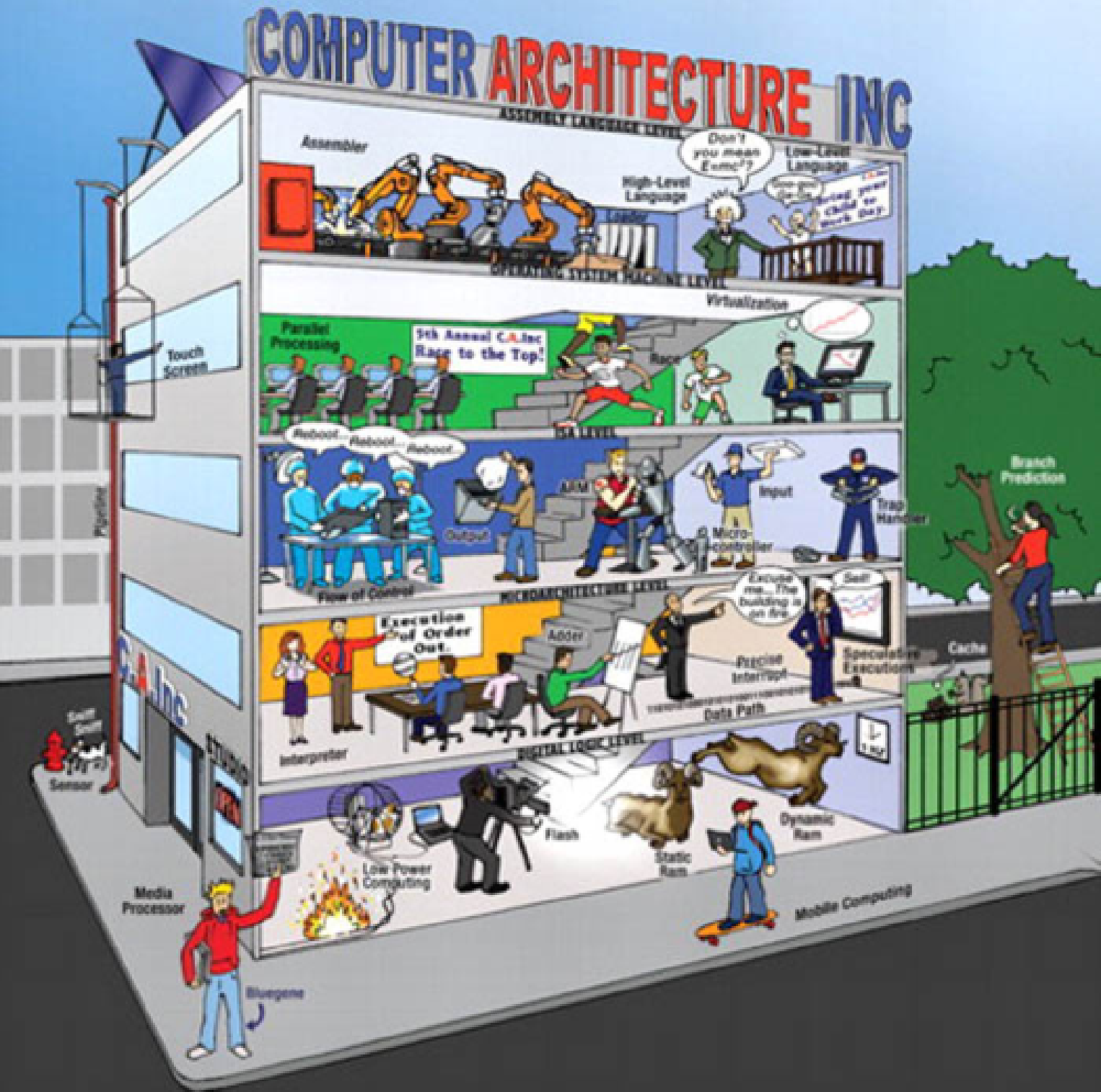
Structured Computer Organization

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

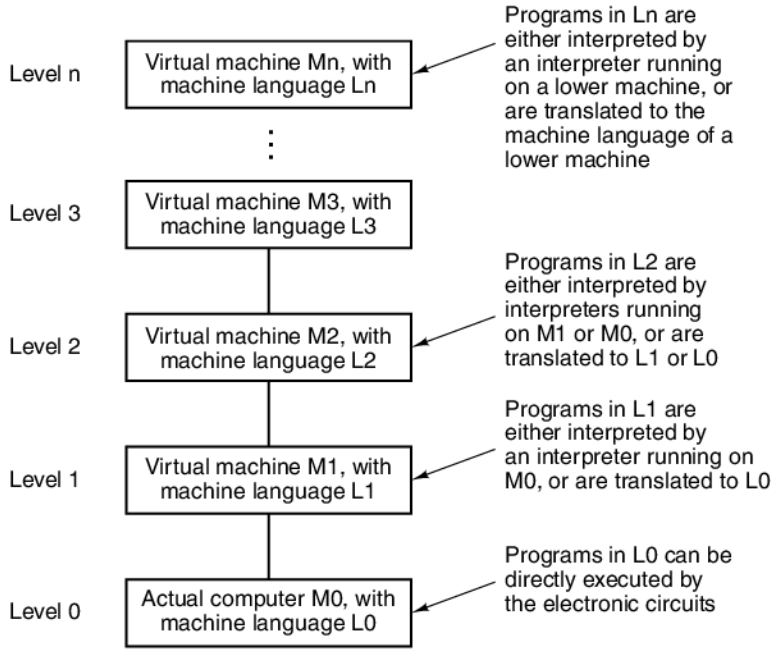


Figure 1-1. A multilevel machine.

"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.

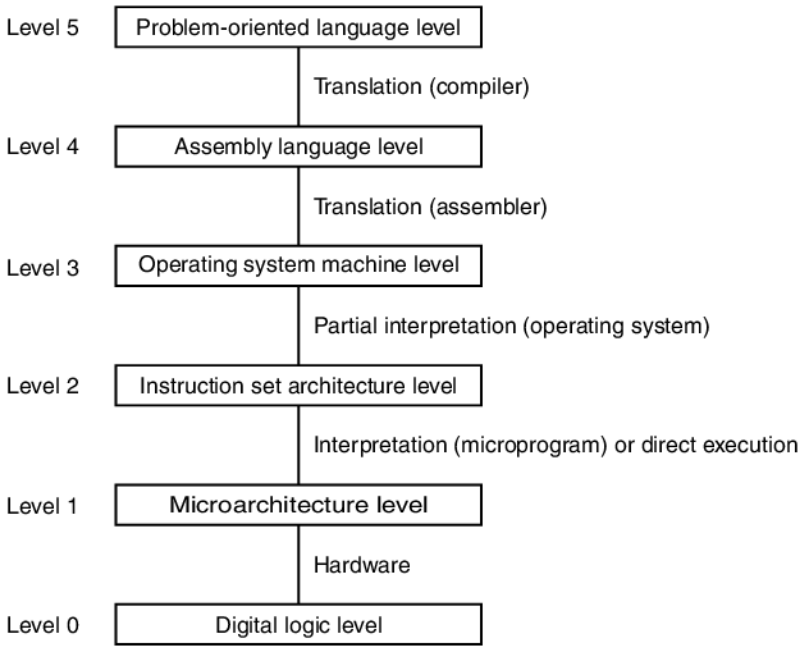


Figure 1-2. A six-level computer. The support method for each level is indicated below it (along with the name of the supporting program).

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

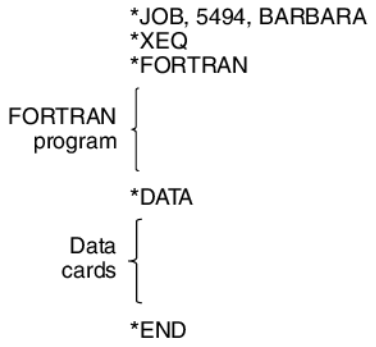


Figure 1-3. A sample job for the FMS operating system.

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.

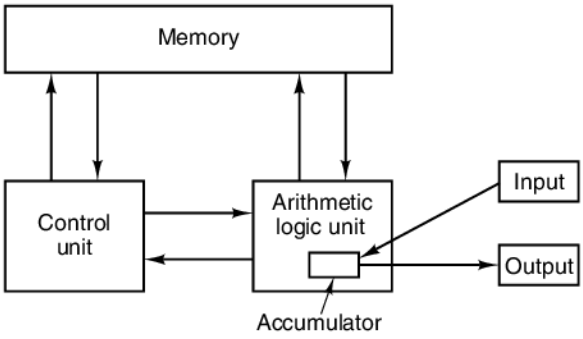


Figure 1-5. The original von Neumann machine.

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.

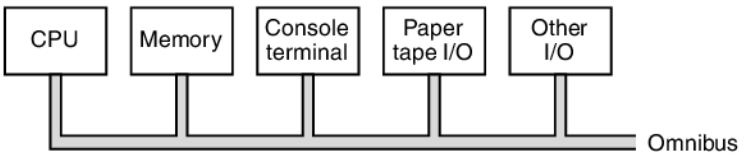


Figure 1-6. The PDP-8 omnibus.

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