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

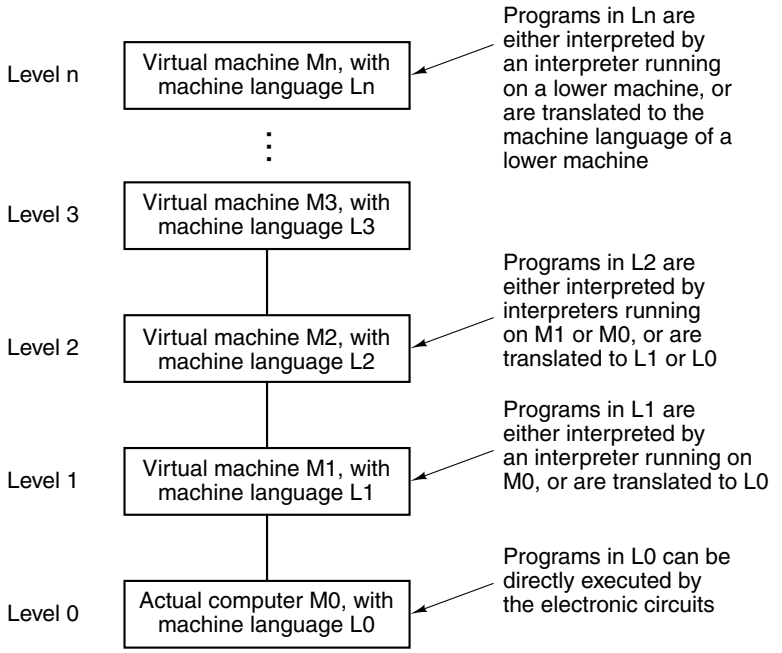

Figure 1-1. A multilevel machine.

"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.

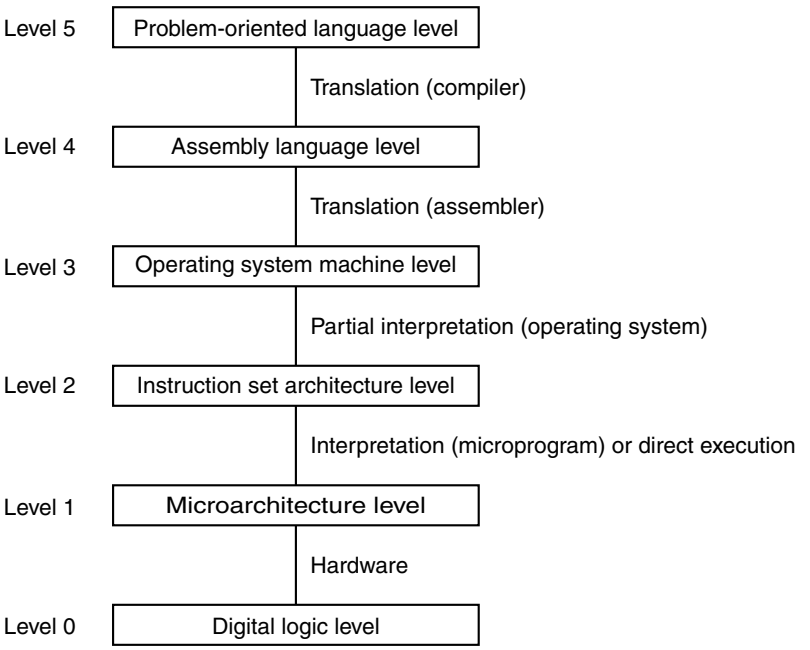

Figure 1-2. A six-level computer. The support method for each level is indicated below it (along with the name of the supporting program).

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

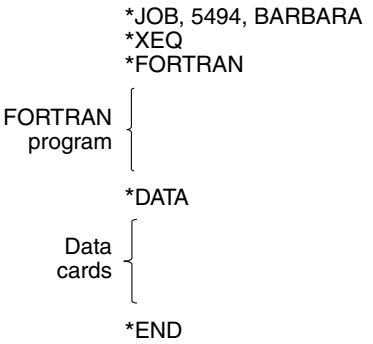

Figure 1-3. A sample job for the FMS operating system.

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.

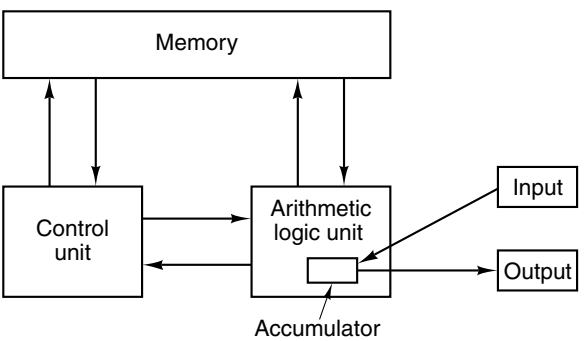

Figure 1-5. The original von Neumann machine.

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.

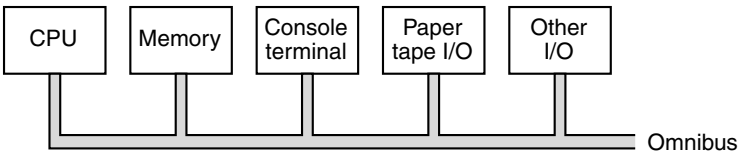

Figure 1-6. The PDP-8 omnibus.

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

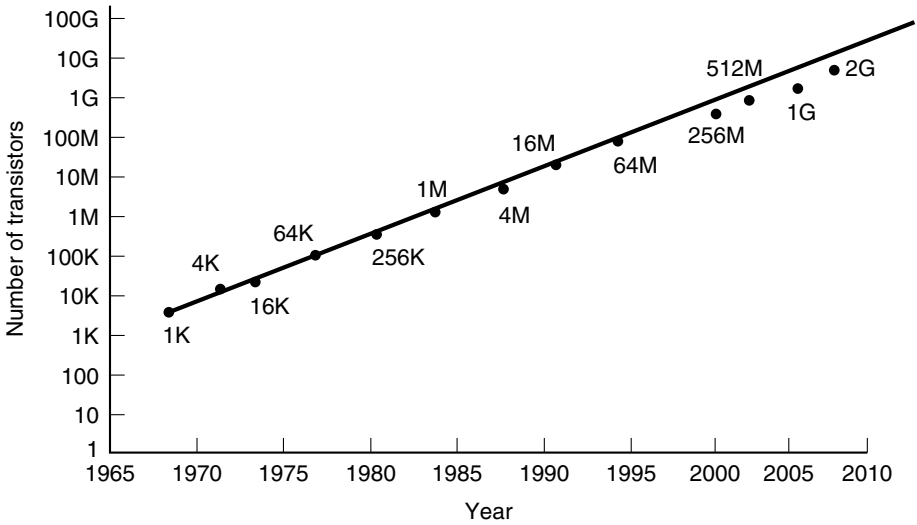

Figure 1-8. Moore's law predicts a 60 percent annual increase in the number of transistors that can be put on a chip. The data points given above and below the line are memory sizes, in bits.

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.

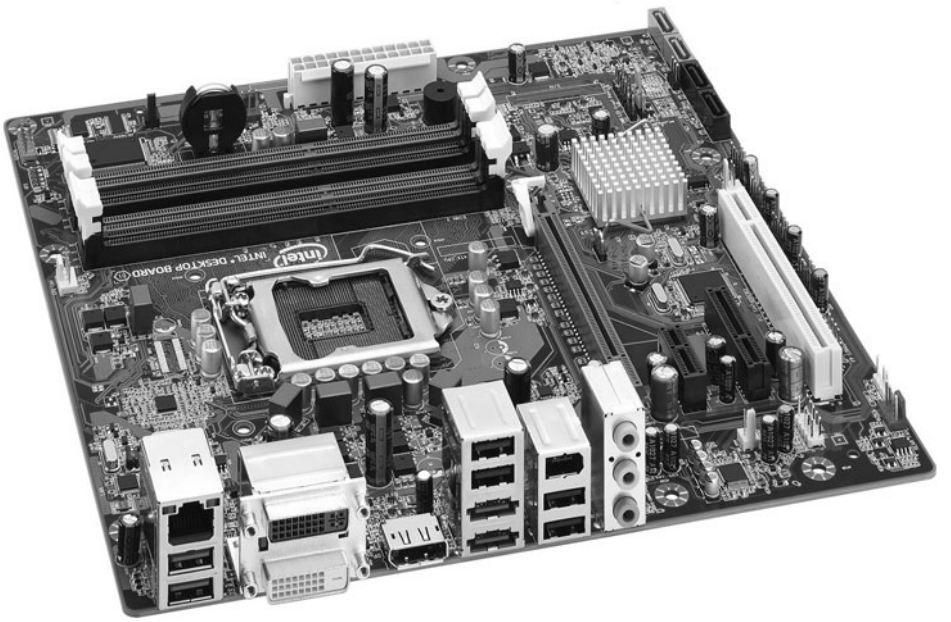

Figure 1-10. A printed circuit board is at the heart of every personal computer. This one is the Intel DQ67SW board. © 2011 Intel Corporation. Used by permission.

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

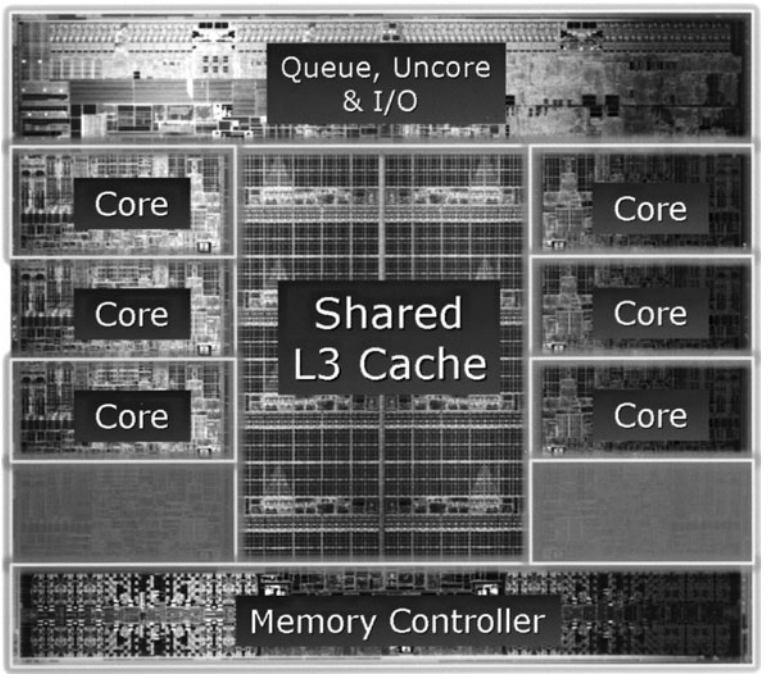

Figure 1-12. The Intel Core i7-3960X die. The die is 21 by 21 mm and has 2.27 billion transistors. © 2011 Intel Corporation. Used by permission.

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.

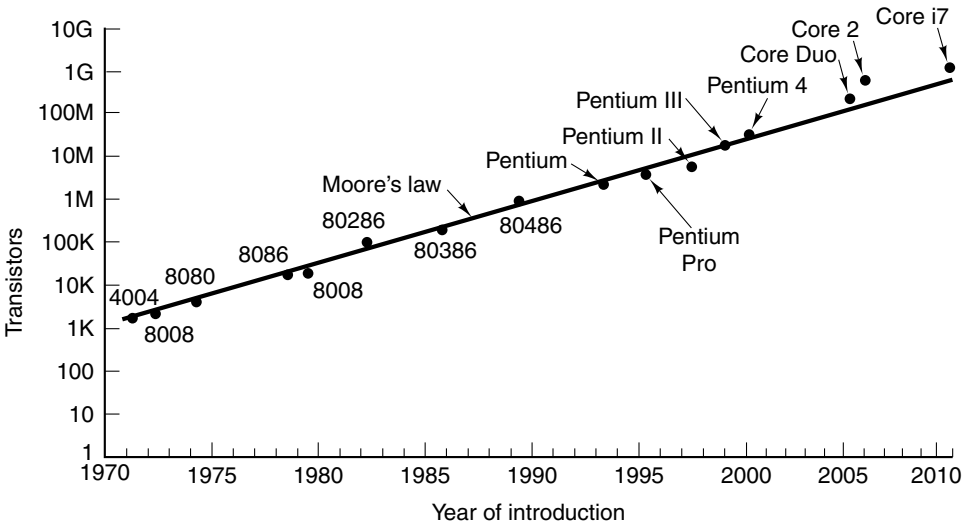

Figure 1-13. Moore's law for (Intel) CPU chips.

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

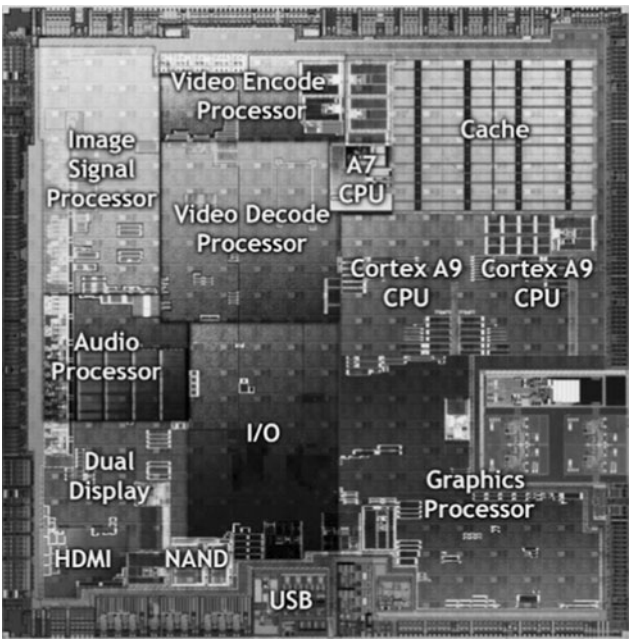

Figure 1-14. The Nvidia Tegra 2 system on a chip. © 2011 Nvidia Corporation. Used by permission.

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*.