>> second edition

introduction to computing systems

from bits and gates to C and beyond

Yale N. Patt
The University of Texas at Austin

Sanjay J. Patel
University of Illinois at Urbana-Champaign

Mc Graw Hill Higher Education

Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis
Bangkok Bogotá Caracas Kuala Lumpur Lisbon London Madrid Mexico City
Milan Montreal New Delhi Santiago Seoul Singapore Sydney Taipei Toronto

contents

1.2 How We Will Get There

We will start (in Chapter 2) by noting that the computer is a piece of electronic equipment and, as such, consists of electronic parts interconnected by wires. Every wire in the computer, at every moment in time, is either at a high voltage or a low voltage. We do not differentiate exactly how high. For example, we do not distinguish voltages of 115 volts from voltages of 118 volts. We only care whether there is or is not a large voltage relative to 0 volts. That absence or presence of a large voltage relative to 0 volts is represented as 0 or 1.

We will encode all information as sequences of 0s and 1s. For example, one encoding of the letter *a* that is commonly used is the sequence 01100001. One encoding of the decimal number *35* is the sequence 00100011. We will see how to perform operations on such encoded information.

Once we are comfortable with information represented as codes made up of 0s and 1s and operations (addition, for example) being performed on these representations, we will begin the process of showing how a computer works. In Chapter 3, we will see how the transistors that make up today's microprocessors work. We will further see how those transistors are combined into larger structures that perform operations, such as addition, and into structures that allow us to save information for later use. In Chapter 4, we will combine these larger structures into the Von Neumann machine, a basic model that describes how a computer works. In Chapter 5, we will begin to study a simple computer, the LC-3. *LC-3* stands for Little Computer 3; we started with LC-1 but needed two more shots at it before we got it right! The LC-3 has all the important characteristics of the microprocessors that you may have already heard of, for example, the Intel 8088, which was used in the first IBM PCs back in 1981. Or the Motorola 68000, which was used in the Macintosh, vintage 1984. Or the Pentium IV, one of the high-performance microprocessors of choice in the PC of the year 2003. That is, the LC-3 has all the important characteristics of these "real" microprocessors, without being so complicated that it gets in the way of your understanding.

Once we understand how the LC-3 works, the next step is to program it, first in its own language (Chapter 6), then in a language called *assembly language* that is a little bit easier for humans to work with (Chapter 7). Chapter 8 deals with the problem of getting information into (input) and out of (output) the LC-3. Chapter 9 covers two sophisticated LC-3 mechanisms, TRAPs and subroutines.

We conclude our introduction to programming the LC-3 in Chapter 10 by first introducing two important concepts (stacks and data conversion), and then by showing a sophisticated example: an LC-3 program that carries out the work of a handheld calculator.

In the second half of the book (Chapters 11–19), we turn our attention to a high-level programming language, C. We include many aspects of C that are usually not dealt with in an introductory textbook. In almost all cases, we try to tie high-level C constructs to the underlying LC-3, so that you will understand what you demand of the computer when you use a particular construct in a C program.

Our treatment of C starts with basic topics such as variables and operators (Chapter 12), control structures (Chapter 13), and functions (Chapter 14). We

then move on to the more advanced topics of debugging C programs (Chapter 15), recursion (Chapter 16), and pointers and arrays (Chapter 17).

We conclude our introduction to C by examining two very common high-level constructs, input/output in C (Chapter 18) and the linked list (Chapter 19).

1.3 Two Recurring Themes

Two themes permeate this book that we have previously taken for granted, assuming that everyone recognized their value and regularly emphasized them to students of engineering and computer science. Lately, it has become clear to us that from the git-go, we need to make these points explicit. So, we state them here up front. The two themes are (a) the notion of abstraction and (b) the importance of not separating in your mind the notions of hardware and software. Their value to your development as an effective engineer or computer scientist goes well beyond your understanding of how a computer works and how to program it.

The notion of abstraction is central to all that you will learn and expect to use in practicing your craft, whether it be in mathematics, physics, any aspect of engineering, or business. It is hard to think of any body of knowledge where the notion of abstraction is not central. The misguided hardware/software separation is directly related to your continuing study of computers and your work with them. We will discuss each in turn.

1.3.1 The Notion of Abstraction

The use of abstraction is all around us. When we get in a taxi and tell the driver, "Take me to the airport," we are using abstraction. If we had to, we could probably direct the driver each step of the way: "Go down this street ten blocks, and make a left turn." And, when he got there, "Now take this street five blocks and make a right turn." And on and on. You know the details, but it is a lot quicker to just tell the driver to take you to the airport.

Even the statement "Go down this street ten blocks . . ." can be broken down further with instructions on using the accelerator, the steering wheel, watching out for other vehicles, pedestrians, etc.

Our ability to abstract is very much a productivity enhancer. It allows us to deal with a situation at a higher level, focusing on the essential aspects, while keeping the component ideas in the background. It allows us to be more efficient in our use of time and brain activity. It allows us to not get bogged down in the detail when everything about the detail is working just fine.

There is an underlying assumption to this, however: "when everything about the detail is just fine." What if everything about the detail is not just fine? Then, to be successful, our ability to abstract must be combined with our ability to *un*-abstract. Some people use the word *deconstruct*—the ability to go from the abstraction back to its component parts.

Two stories come to mind.

The first involves a trip through Arizona the first author made a long time ago in the hottest part of the summer. At the time I was living in Palo Alto, California, where the temperature tends to be mild almost always. I knew enough to take

the car to a mechanic before making the trip, and I told him to check the cooling system. That was the abstraction: cooling system. What I had not mastered was that the capability of a cooling system for Palo Alto, California is not the same as the capability of a cooling system for the summer deserts of Arizona. The result: two days in Deer Lodge, Arizona (population 3), waiting for a head gasket to be shipped in.

The second story (perhaps apocryphal) is supposed to have happened during the infancy of electric power generation. General Electric Co. was having trouble with one of its huge electric power generators and did not know what to do. On the front of the generator were lots of dials containing lots of information, and lots of screws that could be rotated clockwise or counterclockwise as the operator wished. Something on the other side of the wall of dials and screws was malfunctioning and no one knew what to do. So, as the story goes, they called in one of the early giants in the electric power industry. He looked at the dials and listened to the noises for a minute, then took a small pocket screwdriver out of his geek pack and rotated one screw 35 degrees counterclockwise. The problem immediately went away. He submitted a bill for $1,000 (a lot of money in those days) without any elaboration. The controller found the bill for two minutes' work a little unsettling, and asked for further clarification. Back came the new bill:

```
Turning a screw 35 degrees counterclockwise:  $  0.75
Knowing which screw to turn and by how much:    999.25
```

In both stories the message is the same. It is more efficient to think of entities as abstractions. One does not want to get bogged down in details unnecessarily. And as long as nothing untoward happens, we are OK. If I had never tried to make the trip to Arizona, the abstraction "cooling system" would have been sufficient. If the electric power generator never malfunctioned, there would have been no need for the power engineering guru's deeper understanding.

When one designs a logic circuit out of gates, it is much more efficient to not have to think about the internals of each gate. To do so would slow down the process of designing the logic circuit. One wants to think of the gate as a component. But if there is a problem with getting the logic circuit to work, it is often helpful to look at the internal structure of the gate and see if something about its functioning is causing the problem.

When one designs a sophisticated computer application program, whether it be a new spreadsheet program, word processing system, or computer game, one wants to think of each of the components one is using as an abstraction. If one spent time thinking about the details of a component when it is not necessary, the distraction could easily prevent the total job from ever getting finished. But when there is a problem putting the components together, it is often useful to examine carefully the details of each component in order to uncover the problem.

The ability to abstract is a most important skill. In our view, one should try to keep the level of abstraction as high as possible, consistent with getting everything to work effectively. Our approach in this book is to continually raise the level of abstraction. We describe logic gates in terms of transistors. Once we understand the abstraction of gates, we no longer think in terms of transistors. Then we build

larger structures out of gates. Once we understand these larger abstractions, we no longer think in terms of gates.

The Bottom Line

Abstractions allow us to be much more efficient in dealing with all kinds of situations. It is also true that one can be effective without understanding what is below the abstraction as long as everything behaves nicely. So, one should not pooh-pooh the notion of abstraction. On the contrary, one should celebrate it since it allows us to be more efficient.

In fact, if we never have to combine a component with anything else into a larger system, and if nothing can go wrong with the component, then it is perfectly fine to understand this component only at the level of its abstraction.

But if we have to combine multiple components into a larger system, we should be careful not to allow their abstractions to be the deepest level of our understanding. If we don't know the components below the level of their abstractions, then we are at the mercy of them working together without our intervention. If they don't work together, and we are unable to go below the level of abstraction, we are stuck. And that is the state we should take care not to find ourselves in.

1.3.2 Hardware versus Software

Many computer scientists and engineers refer to themselves as hardware people or software people. By hardware, they generally mean the physical computer and all the specifications associated with it. By software, they generally mean the programs, whether operating systems like UNIX or Windows, or database systems like Oracle or DB-terrific, or application programs like Excel or Word. The implication is that the person knows a whole lot about one of these two things and precious little about the other. Usually, there is the further implication that it is OK to be an expert at one of these (hardware OR software) and clueless about the other. It is as if there were a big wall between the hardware (the computer and how it actually works) and the software (the programs that direct the computer's bidding), and that one should be content to remain on one side of that wall or the other.

As you approach your study and practice of computing, we urge you to take the opposite approach—that hardware and software are names for components of two parts of a computing system that work best when they are designed by someone who took into account the capabilities and limitations of both.

Microprocessor designers who understand the needs of the programs that will execute on that microprocessor they are designing can design much more effective microprocessors than those who don't. For example, Intel, Motorola, and other major producers of microprocessors recognized a few years ago that a large fraction of future programs would contain video clips as part of e-mail, video games, and full-length movies. They recognized that it would be important for such programs to execute efficiently. The result: most microprocessors today contain special hardware capability to process these video clips. Intel defined additional instructions, collectively called their MMX instruction set, and developed

special hardware for it. Motorola, IBM, and Apple did essentially the same thing, resulting in the AltaVec instruction set and special hardware to support it.

A similar story can be told about software designers. The designer of a large computer program who understands the capabilities and limitations of the hardware that will carry out the tasks of that program can design the program more efficiently than the designer who does not understand the nature of the hardware. One important task that almost all large software systems have to carry out is called sorting, where a number of items have to be arranged in some order. The words in a dictionary are arranged in alphabetical order. Students in a class are often arranged in numeric order, according to their scores on the final exam. There are a huge number of fundamentally different programs one can write to arrange a collection of items in order. Donald Knuth devoted 391 pages to the task in *The Art of Computer Programming*, vol. 3. Which sorting program works best is often very dependent on how much the software designer is aware of the characteristics of the hardware.

The Bottom Line

We believe that whether your inclinations are in the direction of a computer hardware career or a computer software career, you will be much more capable if you master both. This book is about getting you started on the path to mastering both hardware and software. Although we sometimes ignore making the point explicitly when we are in the trenches of working through a concept, it really is the case that each sheds light on the other.

When you study data types, a software concept (in C, Chapter 12), you will understand how the finite word length of the computer, a hardware concept, affects our notion of data types.

When you study functions (in C, Chapter 14), you will be able to tie the *rules* of calling a function with the hardware implementation that makes those rules necessary.

When you study recursion (a powerful algorithmic device, in Chapter 16), you will be able to tie it to the hardware. If you take the time to do that, you will better understand when the additional time to execute a procedure recursively is worth it.

When you study pointer variables (in C, in Chapter 17), your knowledge of computer memory will provide a deeper understanding of what pointers provide, when they should be used, and when they should be avoided.

When you study data structures (in C, in Chapter 19), your knowledge of computer memory will help you better understand what must be done to manipulate the actual structures in memory efficiently.

We understand that most of the terms in the preceding five short paragraphs are not familiar to you *yet*. That is OK; you can reread this page at the end of the semester. What is important to know right now is that there are important topics in the software that are very deeply interwoven with topics in the hardware. Our contention is that mastering either is easier if you pay attention to both.

Most importantly, most computing problems yield better solutions when the problem solver has the capability of both at his or her disposal.

1.4 A Computer System

We have used the word *computer* many times in the preceding paragraphs, and although we did not say so explicitly, we used it to mean a mechanism that does two things: It directs the processing of information and it performs the actual processing of information. It does both of these things in response to a computer program. When we say "directing the processing of information," we mean figuring out which task should get carried out next. When we say "performing the actual processing," we mean doing the actual additions, multiplications, and so forth that are necessary to get the job done. A more precise term for this mechanism is a central processing unit (CPU), or simply a processor. This textbook is primarily about the processor and the programs that are executed by the processor.

Twenty years ago, the processor was constructed out of ten or more 18-inch electronic boards, each containing 50 or more electronic parts known as integrated circuit packages (see Figure 1.1). Today, a processor usually consists of a single microprocessor chip, built on a piece of silicon material, measuring less than an inch square, and containing many millions of transistors (see Figure 1.2).

However, when most people use the word *computer*, they usually mean more than the processor. They usually mean the collection of parts that in combination

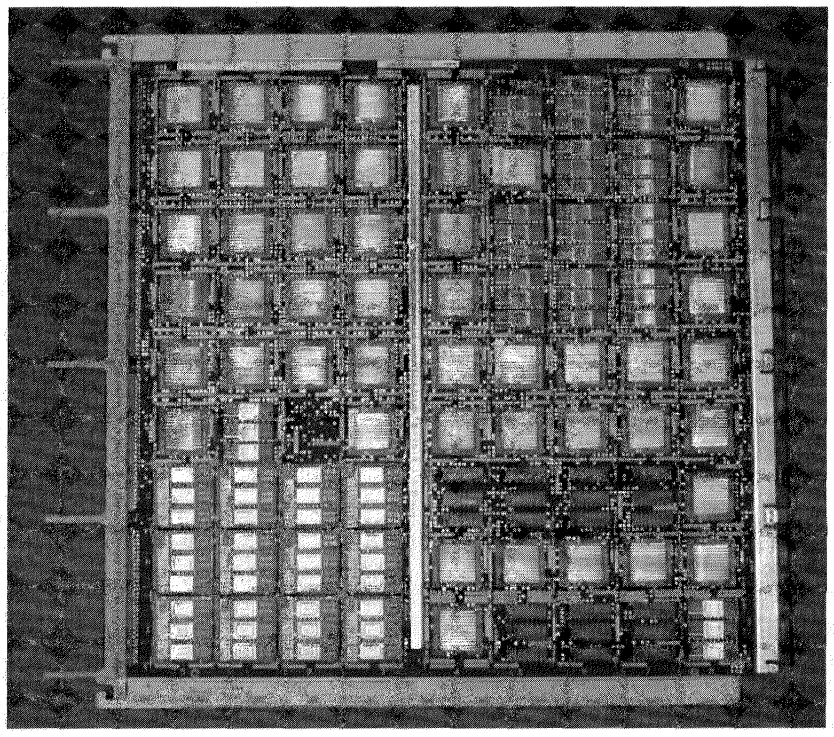

Figure 1.1 A processor board, vintage 1980s (Courtesy of Emilio Salgueiro, Unisys Corporation.)

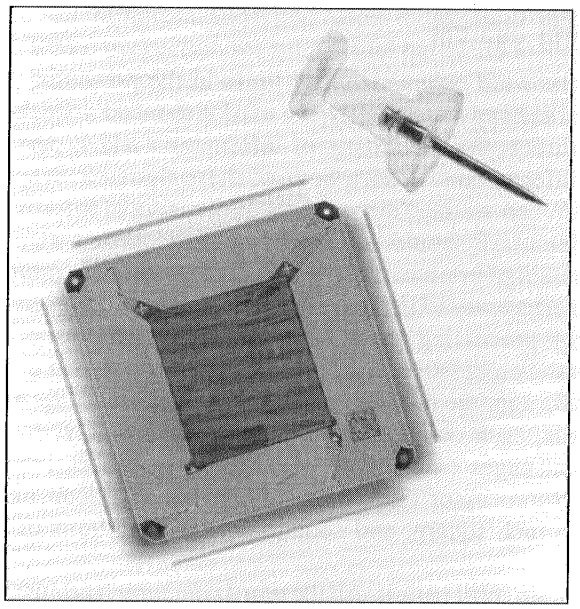

Figure 1.2 A microprocessor, vintage 1998 (Courtesy of Intel Corporation.)

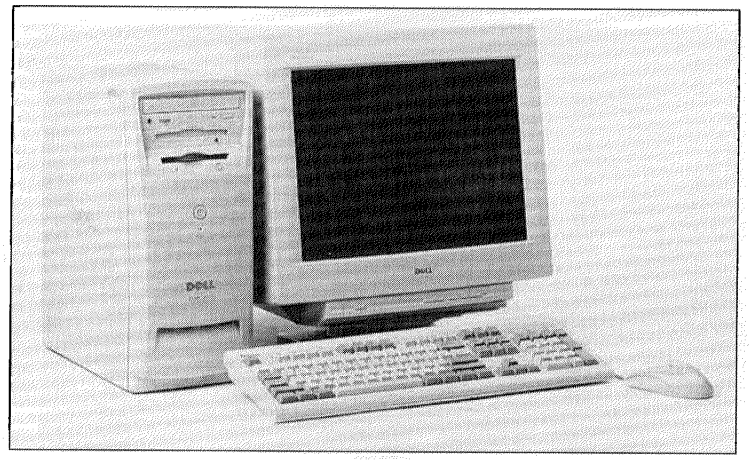

Figure 1.3 A personal computer (Courtesy of Dell Computer.)

form their *computer system* (see Figure 1.3). A computer system usually includes, in addition to the processor, a keyboard for typing commands, a mouse for clicking on menu entries, a monitor for displaying information that the computer system has produced, a printer for obtaining paper copies of that information, memory for temporarily storing information, disks and CD-ROMs of one sort or another for storing information for a very long time, even after the computer has been turned off, and the collection of programs (the software) that the user wishes to execute.

These additional items are useful in helping the computer user do his or her job. Without a printer, for example, the user would have to copy by hand what is displayed on the monitor. Without a mouse, the user would have to type each command, rather than simply clicking on the mouse button.

So, as we begin our journey, which focuses on how we get less than 1 square inch of silicon to do our bidding, we note that the computer systems we use contain a lot of other components to make our life more comfortable.

1.5 Two Very Important Ideas

Before we leave this first chapter, there are two very important ideas that we would like you to understand, ideas that are at the core of what computing is all about.

Idea 1: All computers (the biggest and the smallest, the fastest and the slowest, the most expensive and the cheapest) are capable of computing exactly the same things if they are given enough time and enough memory. That is, anything a fast computer can do, a slow computer can do also. The slow computer just does it more slowly. A more expensive computer cannot figure out something that a cheaper computer is unable to figure out as long as the cheap computer can access enough memory. (You may have to go to the store to buy disks whenever it runs out of memory in order to keep increasing memory.) **All** computers can do **exactly** the same things. Some computers can do things faster, but none can do **more** than any other.

Idea 2: We describe our problems in English or some other language spoken by people. Yet the problems are solved by electrons running around inside the computer. It is necessary to transform our problem from the language of humans to the voltages that influence the flow of electrons. This transformation is really a sequence of systematic transformations, developed and improved over the last 50 years, which combine to give the computer the ability to carry out what appears to be some very complicated tasks. In reality, these tasks are simple and straightforward.

The rest of this chapter is devoted to discussing these two ideas.

1.6 Computers as Universal Computational Devices

It may seem strange that an introductory textbook begins by describing how computers work. After all, mechanical engineering students begin by studying physics, not how car engines work. Chemical engineering students begin by studying chemistry, not oil refineries. Why should computing students begin by studying computers?

The answer is that computers are different. To learn the fundamental principles of computing, you must study computers or machines that can do what

computers can do. The reason for this has to do with the notion that computers are *universal computational devices*. Let's see what that means.

Before modern computers, there were many kinds of calculating machines. Some were *analog machines*—machines that produced an answer by measuring some physical quantity such as distance or voltage. For example, a slide rule is an analog machine that multiplies numbers by sliding one logarithmically graded ruler next to another. The user can read a logarithmic "distance" on the second ruler. Some early analog adding machines worked by dropping weights on a scale. The difficulty with analog machines is that it is very hard to increase their accuracy.

This is why *digital machines*—machines that perform computations by manipulating a fixed finite set of digits or letters—came to dominate computing. You are familiar with the distinction between analog and digital watches. An analog watch has hour and minute hands, and perhaps a second hand. It gives the time by the positions of its hands, which are really angular measures. Digital watches give the time in digits. You can increase accuracy just by adding more digits. For example, if it is important for you to measure time in hundredths of a second, you can buy a watch that gives a reading like 10:35.16 rather than just 10:35. How would you get an analog watch that would give you an accurate reading to one one-hundredth of a second? You could do it, but it would take a mighty long second hand! When we talk about computers in this book, we will always mean digital machines.

Before modern digital computers, the most common digital machines in the West were adding machines. In other parts of the world another digital machine, the abacus, was common. Digital adding machines were mechanical or electromechanical devices that could perform a specific kind of computation: adding integers. There were also digital machines that could multiply integers. There were digital machines that could put a stack of cards with punched names in alphabetical order. The main limitation of all of these machines is that they could do only one specific kind of computation. If you owned only an adding machine and wanted to multiply two integers, you had some pencil and paper work to do.

This is why computers are different. You can tell a computer how to add numbers. You can tell it how to multiply. You can tell it how to alphabetize a list or perform any computation you like. When you think of a new kind of computation, you do not have to buy or design a new computer. You just give the old computer a new set of instructions (or program) to carry out the computation. This is why we say the computer is a *universal computational device*. Computer scientists believe that *anything that can be computed, can be computed by a computer* provided it has enough time and enough memory. When we study computers, we study the fundamentals of all computing. We learn what computation is and what can be computed.

The idea of a universal computational device is due to Alan Turing. Turing proposed in 1937 that all computations could be carried out by a particular kind of machine, which is now called a Turing machine. He gave a mathematical description of this kind of machine, but did not actually build one. Digital computers were not operating until 1946. Turing was more interested in solving a philosophical problem: defining computation. He began by looking at the kinds of actions that people perform when they compute; these include making marks

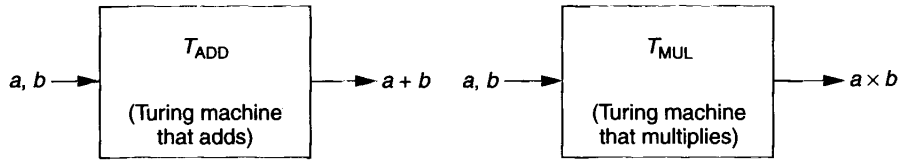

Figure 1.4 Black box models of Turing machines

on paper, writing symbols according to certain rules when other symbols are present, and so on. He abstracted these actions and specified a mechanism that could carry them out. He gave some examples of the kinds of things that these machines could do. One Turing machine could add two integers; another could multiply two integers.

Figure 1.4 provides what we call "black box" models of Turing machines that add and multiply. In each case, the operation to be performed is described in the box. The data on which to operate is shown as input to the box. The result of the operation is shown as output from the box. A black box model provides no information as to exactly how the operation is performed, and indeed, there are many ways to add or multiply two numbers.

Turing proposed that every computation can be performed by some Turing machine. We call this *Turing's thesis*. Although Turing's thesis has never been proved, there does exist a lot of evidence to suggest it is true. We know, for example, that various enhancements one can make to Turing machines do not result in machines that can compute more.

Perhaps the best argument to support Turing's thesis was provided by Turing himself in his original paper. He said that one way to try to construct a machine more powerful than any particular Turing machine was to make a machine U that could simulate *all* Turing machines. You would simply describe to U the particular Turing machine you wanted it to simulate, say a machine to add two integers, give U the input data, and U would compute the appropriate output, in this case the sum of the inputs. Turing then showed that there was, in fact, a Turing machine that could do this, so even this attempt to find something that could not be computed by Turing machines failed.

Figure 1.5 further illustrates the point. Suppose you wanted to compute $g \cdot (e + f)$. You would simply provide to U descriptions of the Turing machines to add and to multiply, and the three inputs, e, f, and g. U would do the rest.

In specifying U, Turing had provided us with a deep insight: He had given us the first description of what computers do. In fact, both a computer (with as much

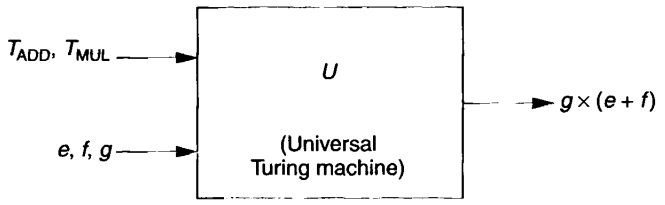

Figure 1.5 Black box model of a universal Turing machine

memory as it wants) and a universal Turing machine can compute exactly the same things. In both cases you give the machine a description of a computation and the data it needs, and the machine computes the appropriate answer. Computers and universal Turing machines can compute anything that can be computed because they are *programmable*.

This is the reason that a big or expensive computer cannot do more than a small, cheap computer. More money may buy you a faster computer, a monitor with higher resolution, or a nice sound system. But if you have a small, cheap computer, you already have a universal computational device.

1.7 How Do We Get the Electrons to Do the Work?

Figure 1.6 shows the process we must go through to get the electrons (which actually do the work) to do our bidding. We call the steps of this process the

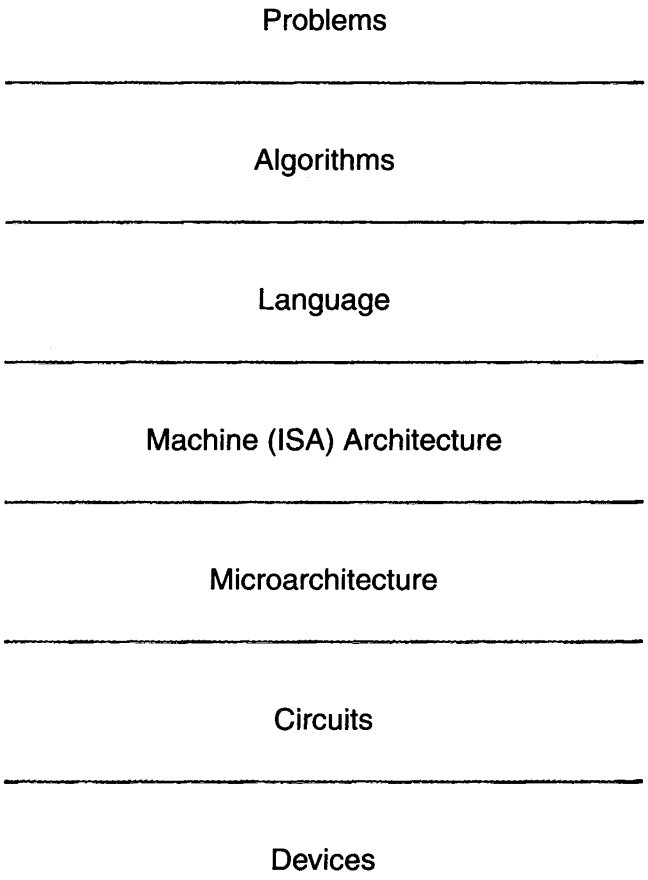

Figure 1.6 Levels of transformation

"Levels of Transformation." As we will see, at each level we have choices. If we ignore any of the levels, our ability to make the best use of our computing system can be very adversely affected.

1.7.1 The Statement of the Problem

We describe the problems we wish to solve with a computer in a "natural language." Natural languages are languages that people speak, like English, French, Japanese, Italian, and so on. They have evolved over centuries in accordance with their usage. They are fraught with a lot of things unacceptable for providing instructions to a computer. Most important of these unacceptable attributes is ambiguity. Natural language is filled with ambiguity. To infer the meaning of a sentence, a listener is often helped by the tone of voice of the speaker, or at the very least, the context of the sentence.

An example of ambiguity in English is the sentence, "Time flies like an arrow." At least three interpretations are possible, depending on whether (1) one is noticing how fast time passes, (2) one is at a track meet for insects, or (3) one is writing a letter to the Dear Abby of Insectville. In the first case, a simile, one is comparing the speed of time passing to the speed of an arrow that has been released. In the second case, one is telling the timekeeper to do his/her job much like an arrow would. In the third case, one is relating that a particular group of flies (time flies, as opposed to fruit flies) are all in love with the same arrow.

Such ambiguity would be unacceptable in instructions provided to a computer. The computer, electronic idiot that it is, can only do as it is told. To tell it to do something where there are multiple interpretations would cause the computer to not know which interpretation to follow.

1.7.2 The Algorithm

The first step in the sequence of transformations is to transform the natural language description of the problem to an algorithm, and in so doing, get rid of the objectionable characteristics. An algorithm is a step-by-step procedure that is guaranteed to terminate, such that each step is precisely stated and can be carried out by the computer. There are terms to describe each of these properties.

We use the term *definiteness* to describe the notion that each step is precisely stated. A recipe for excellent pancakes that instructs the preparer to "stir until lumpy" lacks definiteness, since the notion of lumpiness is not precise.

We use the term *effective computability* to describe the notion that each step can be carried out by a computer. A procedure that instructs the computer to "take the largest prime number" lacks effective computability, since there is no largest prime number.

We use the term *finiteness* to describe the notion that the procedure terminates.

For every problem there are usually many different algorithms for solving that problem. One algorithm may require the fewest number of steps. Another algorithm may allow some steps to be performed concurrently. A computer that allows more than one thing to be done at a time can often solve the problem in

less time, even though it is likely that the total number of steps to be performed has increased.

1.7.3 The Program

The next step is to transform the algorithm into a computer program, in one of the programming languages that are available. Programming languages are "mechanical languages." That is, unlike natural languages, mechanical languages did not evolve through human discourse. Rather, they were invented for use in specifying a sequence of instructions to a computer. Therefore, mechanical languages do not suffer from failings such as ambiguity that would make them unacceptable for specifying a computer program.

There are more than 1,000 programming languages. Some have been designed for use with particular applications, such as Fortran for solving scientific calculations and COBOL for solving business data-processing problems. In the second half of this book, we will use C, a language that was designed for manipulating low-level hardware structures.

Other languages are useful for still other purposes. Prolog is the language of choice for many applications that require the design of an expert system. LISP was for years the language of choice of a substantial number of people working on problems dealing with artificial intelligence. Pascal is a language invented as a vehicle for teaching beginning students how to program.

There are two kinds of programming languages, high-level languages and low-level languages. High-level languages are at a distance (a high level) from the underlying computer. At their best, they are independent of the computer on which the programs will execute. We say the language is "machine independent." All the languages mentioned thus far are high-level languages. Low-level languages are tied to the computer on which the programs will execute. There is generally one such low-level language for each computer. That language is called the *assembly language* for that computer.

1.7.4 The ISA

The next step is to translate the program into the instruction set of the particular computer that will be used to carry out the work of the program. The instruction set architecture (ISA) is the complete specification of the interface between programs that have been written and the underlying computer hardware that must carry out the work of those programs.

The ISA specifies the set of instructions the computer can carry out, that is, what operations the computer can perform and what data is needed by each operation. The term *operand* is used to describe individual data values. The ISA specifies the acceptable representations for operands. They are called *data types.* A *data type* is a legitimate representation for an operand such that the computer can perform operations on that representation. The ISA specifies the mechanisms that the computer can use to figure out where the operands are located. These mechanisms are called *addressing modes.*

The number of operations, data types, and addressing modes specified by an ISA vary among the different ISAs. Some ISAs have as few as a half dozen operations, whereas others have as many as several hundred. Some ISAs have only one data type, while others have more than a dozen. Some ISAs have one or two addressing modes, whereas others have more than 20. The x86, the ISA used in the PC, has more than 100 operations, more than a dozen data types, and more than two dozen addressing modes.

The ISA also specifies the number of unique locations that comprise the computer's memory and the number of individual 0s and 1s that are contained in each location.

Many ISAs are in use today. The most common example is the x86, introduced by Intel Corporation in 1979 and currently also manufactured by AMD and other companies. Other ISAs are the Power PC (IBM and Motorola), PA-RISC (Hewlett Packard), and SPARC (Sun Microsystems).

The translation from a high-level language (such as C) to the ISA of the computer on which the program will execute (such as x86) is usually done by a translating program called a *compiler*. To translate from a program written in C to the x86 ISA, one would need an x86 C compiler. For each high-level language and each desired target computer, one must provide a corresponding compiler.

The translation from the unique assembly language of a computer to its ISA is done by an assembler.

1.7.5 The Microarchitecture

The next step is to transform the ISA into an implementation. The detailed organization of an implementation is called its *microarchitecture*. So, for example, the x86 has been implemented by several different microprocessors over the years, each having its own unique microarchitecture. The original implementation was the 8086 in 1979. More recently, in 2001, Intel introduced the Pentium IV microprocessor. Motorola and IBM have implemented the Power PC ISA with more than a dozen different microprocessors, each having its own microarchitecture. Two of the more recent implementations are the Motorola MPC 7455 and the IBM Power PC 750FX.

Each implementation is an opportunity for computer designers to make different trade-offs between the cost of the microprocessor and the performance that microprocessor will provide. Computer design is always an exercise in trade-offs, as the designer opts for higher (or lower) performance at greater (or lesser) cost.

The automobile provides a good analogy of the relationship between an ISA and a microarchitecture that implements that ISA. The ISA describes what the driver sees as he/she sits inside the automobile. All automobiles provide the same interface (an ISA different from the ISA for boats and the ISA for airplanes). Of the three pedals on the floor, the middle one is always the brake. The one on the right is the accelerator, and when it is depressed, the car will move faster. The ISA is about basic functionality. All cars can get from point A to point B, can move forward and backward, and can turn to the right and to the left.

The implementation of the ISA is about what goes on under the hood. Here all automobile makes and models are different, depending on what cost/performance trade-offs the automobile designer made before the car was manufactured. So, some automobiles come with disc brakes, others (in the past, at least) with drums. Some automobiles have eight cylinders, others run on six cylinders, and still others have four. Some are turbocharged, some are not. In each case, the "microarchitecture" of the specific automobile is a result of the automobile designers' decisions regarding cost and performance.

1.7.6 The Logic Circuit

The next step is to implement each element of the microarchitecture out of simple logic circuits. Here, also, there are choices, as the logic designer decides how to best make the trade-offs between cost and performance. So, for example, even for the simple operation of addition, there are several choices of logic circuits to perform this operation at differing speeds and corresponding costs.

1.7.7 The Devices

Finally, each basic logic circuit is implemented in accordance with the requirements of the particular device technology used. So, CMOS circuits are different from NMOS circuits, which are different, in turn, from gallium arsenide circuits.

1.7.8 Putting It Together

In summary, from the natural language description of a problem to the electrons running around that actually solve the problem, many transformations need to be performed. If we could speak electron, or the electrons could understand English, perhaps we could just walk up to the computer and get the electrons to do our bidding. Since we can't speak electron and they can't speak English, the best we can do is this systematic sequence of transformations. At each level of transformation, there are choices as to how to proceed. Our handling of those choices determines the resulting cost and performance of our computer.

In this book, we describe each of these transformations. We show how transistors combine to form logic circuits, how logic circuits combine to form the microarchitecture, and how the microarchitecture implements a particular ISA, in our case, the LC-3. We complete the process by going from the English-language description of a problem to a C program that solves the problem, and we show how that C program is translated (i.e., compiled) to the ISA of the LC-3.

We hope you enjoy the ride.

Exercises

1.1 Explain the first of the two important ideas stated in Section 1.5.

1.2 Can a higher-level programming language instruct a computer to compute more than a lower-level programming language?

1.3 What difficulty with analog computers encourages computer designers to use digital designs?

1.4 Name one characteristic of natural languages that prevents them from being used as programming languages.

1.5 Say we had a "black box," which takes two numbers as input and outputs their sum. See Figure 1.7a. Say we had another box capable of multiplying two numbers together. See Figure 1.7b. We can connect these boxes together to calculate $p \times (m + n)$. See Figure 1.7c. Assume we have an unlimited number of these boxes. Show how to connect them together to calculate:

a. $ax + b$
b. The average of the four input numbers w, x, y, and z
c. $a^2 + 2ab + b^2$ (Can you do it with one add box and one multiply box?)

1.6 Write a statement in a natural language and offer two different interpretations of that statement.

1.7 The discussion of abstraction in Section 1.3.1 noted that one does not need to understand the makeup of the components as long as "everything about the detail is just fine." The case was made that when everything is not fine, one must be able to deconstruct the components, or be at the mercy of the abstractions. In the taxi example, suppose you did not understand the component, that is, you had no clue how to get to the airport. Using the notion of abstraction, you simply tell the driver,

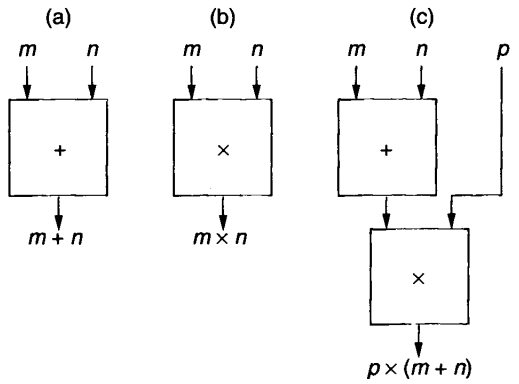

Figure 1.7 "Black boxes" capable of (a) addition, (b) multiplication, and (c) a combination of addition and multiplication

"Take me to the airport." Explain when this is a productivity enhancer, and when it could result in very negative consequences.

1.8 John said, "I saw the man in the park with a telescope." What did he mean? How many reasonable interpretations can you provide for this statement? List them. What property does this sentence demonstrate that makes it unacceptable as a statement in a program.

1.9 Are natural languages capable of expressing algorithms?

1.10 Name three characteristics of algorithms. Briefly explain each of these three characteristics.

1.11 For each characteristic of an algorithm, give an example of a procedure that does not have the characteristic, and is therefore not an algorithm.

1.12 Are items *a* through *e* in the following list algorithms? If not, what qualities required of algorithms do they lack?

a. Add the first row of the following matrix to another row whose first column contains a nonzero entry. (*Reminder*: Columns run vertically; rows run horizontally.)

$$\begin{bmatrix} 1 & 2 & 0 & 4 \\ 0 & 3 & 2 & 4 \\ 2 & 3 & 10 & 22 \\ 12 & 4 & 3 & 4 \end{bmatrix}$$

b. In order to show that there are as many prime numbers as there are natural numbers, match each prime number with a natural number in the following manner. Create pairs of prime and natural numbers by matching the first prime number with 1 (which is the first natural number) and the second prime number with 2, the third with 3, and so forth. If, in the end, it turns out that each prime number can be paired with each natural number, then it is shown that there are as many prime numbers as natural numbers.

c. Suppose you're given two vectors each with 20 elements and asked to perform the following operation. Take the first element of the first vector and multiply it by the first element of the second vector. Do the same to the second elements, and so forth. Add all the individual products together to derive the dot product.

d. Lynne and Calvin are trying to decided who will take the dog for a walk. Lynne suggests that they flip a coin and pulls a quarter out of her pocket. Calvin does not trust Lynne and suspects that the quarter may be weighted (meaning that it might favor a particular outcome when tossed) and suggests the following procedure to fairly determine who will walk the dog.

1. Flip the quarter twice.
2. If the outcome is heads on the first flip and tails on the second, then I will walk the dog.
3. If the outcome is tails on the first flip, and heads on the second, then you will walk the dog.

4. If both outcomes are tails or both outcomes are heads, then we flip twice again.

Is Calvin's technique an algorithm?

e. Given a number, perform the following steps in order:
1. Multiply it by four
2. Add four
3. Divide by two
4. Subtract two
5. Divide by two
6. Subtract one
7. At this point, add one to a counter to keep track of the fact that you performed steps 1 through 6. Then test the result you got when you subtracted one. If 0, write down the number of times you performed steps 1 through 6 and stop. If not 0, starting with the result of subtracting 1, perform the above 7 steps again.

1.13 Two computers, A and B, are identical except for the fact that A has a subtract instruction and B does not. Both have add instructions. Both have instructions that can take a value and produce the negative of that value. Which computer is able to solve more problems, A or B? Prove your result.

1.14 Suppose we wish to put a set of names in alphabetical order. We call the act of doing so *sorting*. One algorithm that can accomplish that is called the bubble sort. We could then program our bubble sort algorithm in C, and compile the C program to execute on an x86 ISA. The x86 ISA can be implemented with an Intel Pentium IV microarchitecture. Let us call the sequence "Bubble Sort, C program, x86 ISA, Pentium IV microarchitecture" one *transformation process*.

Assume we have available four sorting algorithms and can program in C, C++, Pascal, Fortran, and COBOL. We have available compilers that can translate from each of these to either x86 or SPARC, and we have available three different microarchitectures for x86 and three different microarchitectures for SPARC.

a. How many transformation processes are possible?
b. Write three examples of transformation processes.
c. How many transformation processes are possible if instead of three different microarchitectures for x86 and three different microarchitectures for SPARC, there were two for x86 and four for SPARC?

1.15 Identify one advantage of programming in a higher-level language compared to a lower-level language. Identify one disadvantage.

1.16 Name at least three things specified by an ISA.

1.17 Briefly describe the difference between an ISA and a microarchitecture.

1.18 How many ISAs are normally implemented by a single microarchitecture? Conversely, how many microarchitectures could exist for a single ISA?

1.19 List the levels of transformation and name an example for each level.

1.20 The levels of transformation in Figure 1.6 are often referred to as levels of abstraction. Is that a reasonable characterization? If yes, give an example. If no, why not?

1.21 Say you go to the store and buy some word processing software. What form is the software actually in? Is it in a high-level programming language? Is it in assembly language? Is it in the ISA of the computer on which you'll run it? Justify your answer.

1.22 Suppose you were given a task at one of the transformation levels shown in Figure 1.6, and required to tranform it to the level just below. At which level would it be most difficult to perform the transformation to the next lower level? Why?

1.23 Why is an ISA unlikely to change between successive generations of microarchitectures that implement it? For example, why would Intel want to make certain that the ISA implemented by the Pentium III is the same as the one implemented by the Pentium II? *Hint:* When you upgrade your computer (or buy one with a newer CPU), do you need to throw out all your old software?