THE PATTERN ON THE STONE

The Simple Ideas That
Make Computers Work

W. DANIEL HILLIS

BASIC
BOOKS
A Member of the
Perseus Books Group

PREFACE: MAGIC IN THE STONE

I etch a pattern of geometric shapes onto a stone. To the uninitiated, the shapes look mysterious and complex, but I know that when arranged correctly they will give the stone a special power, enabling it to respond to incantations in a language no human being has ever spoken. I will ask the stone questions in this language, and it will answer by showing me a vision: a world created by my spell, a world imagined within the pattern on the stone.

A few hundred years ago in my native New England, an accurate description of my occupation would have gotten me burned at the stake. Yet my work involves no witchcraft; I design and program computers. The stone is a wafer of silicon, and the incantations are software. The patterns etched on the chip and the programs that instruct the computer may look complicated and mysterious, but they are generated according to a few basic principles that are easily explained.

Computers are the most complex objects we human beings have ever created, but in a fundamental sense they are remarkably simple. Working with teams of only a few dozen people, I have designed and built computers containing billions of active parts. The wiring diagram of one of these machines, if it were ever to be drawn, would fill all the books in a good-sized public library, and nobody would have the patience to scan the whole of it. Fortunately, such a diagram is unnecessary, because of the regularity of a computer's design. Computers

are built up in a hierarchy of parts, with each part repeated many times over. All you need to understand a computer is an understanding of this hierarchy.

Another principle that makes computers easy to understand is the nature of the interactions among the parts. These interactions are simple and well-defined. They are also usually one-directional, so that the actions of the computer can be sorted neatly into causes and effects, making the inner workings of a computer more comprehensible than, say, the inner workings of an automobile engine or a radio. A computer has a lot more parts than a car or a radio does, but it's much simpler in the way the parts work together. A computer is not dependent so much on technology as on ideas.

Moreover, the ideas have almost nothing to do with the electronics out of which computers are built. Present-day computers are built of transistors and wires, but they could just as well be built, according to the same principles, from valves and water pipes, or from sticks and strings. The principles are the essence of what makes a computer compute. One of the most remarkable things about computers is that their essential nature transcends technology. That nature is what this book is about.

This is the book I wish I had read when I first started learning about the field of computing. Unlike most books on computers—which are either about how to use them or about the technology out of which they're built (ROM, RAM, disk drives, and so on)—this is a book about *ideas*. It explains, or at least introduces, most of the important ideas in the field of computer science, including Boolean logic, finite-state machines, programming languages, compilers and interpreters, Turing universality, information theory, algorithms and algorithmic complexity, heuristics, uncomutable functions, parallel computing, quantum computing, neural networks, machine learning, and self-organizing systems. Anyone interested enough in computers to be reading this book will probably have encountered many of these ideas before, but

outside of a formal education in computer science there are few opportunities to see how they all fit together. This book makes the connections—all the way from simple physical processes like the closing of a switch to the learning and adaptation exhibited by self-organizing parallel computers.

A few general themes underlie an exposition of the nature of computers: the first is the principle of *functional abstraction,* which leads to the aforementioned hierarchy of causes and effects. The structure of the computer is an example of the application of this principle—over and over again, at many levels. Computers are understandable because you can focus on what is happening at one level of the hierarchy without worrying about the details of what goes on at the lower levels. Functional abstraction is what decouples the ideas from the technology.

The second unifying theme is the principle of the *universal computer*—the idea that there is really only one kind of computer, or, more precisely, that all kinds of computers are alike in what they can and cannot do. As near as we can tell, any computing device, whether it's built of transistors, sticks and strings, or neurons, can be simulated by a universal computer. This is a remarkable hypothesis: as I will explain, it suggests that making a computer think like a brain is just a matter of programming it correctly.

The third theme in this book, which won't be fully addressed until the last chapter, is in some sense the antithesis of the first. There may be an entirely new way of designing and programming computers—a way not based on the standard methods of engineering. This would be exciting, because the way we normally design systems begins to break down when the systems become too complicated. The very principles that enable us to design computers lead ultimately to a certain fragility and inefficiency. This weakness has nothing to do with any fundamental limitations of information-processing machines—it's a limitation of the hierarchical method of design. But what if instead we were to use a

design process analogous to biological evolution—that is, a process in which the behaviors of the system *emerge* from the accumulation of many simple interactions, without any "top-down" control? A computing device designed by such an evolutionary process might exhibit some of the robustness and flexibility of a biological organism—at least, that's the hope. This approach is not yet well understood, and it may turn out to be impractical. It is the topic of my current research.

In an explanation of the nature of computers, there are some fundamentals that have to be dealt with before we can move on to the good stuff. The first two chapters introduce the fundamentals: Boolean logic, bits, and finite-state machines. The payoff is that by the end of chapter 3 you'll understand how computers work, top to bottom. This sets the stage for the exciting ideas about universal computing machines, which begin in chapter 4.

The philosopher Gregory Bateson once defined information as "the difference that makes a difference." Another way of saying this is that information is in the distinctions we choose to make significant. In a primitive electrical calculator, say, information is indicated by light bulbs that go on or off depending on whether a current is flowing or not. The voltage of the signal doesn't matter, nor does the direction of current flow. All that matters is that a wire carries one of two possible signals, one of which causes a bulb to light. The distinction that we choose to make significant—the difference that makes a difference, in Bateson's phrase—is between current flowing and not flowing. Bateson's definition is a good one, but the phrase has always meant something more to me. In my lifetime of four decades, the world has been transformed. Most of the changes we've seen in business, politics, science, and philosophy in that time have been caused by, or enabled by, developments in information technology. A lot of things are different in the world today, but the difference that has made the difference has been computers.

These days, computers are popularly thought of as multimedia devices, capable of incorporating and combining all previous forms of media—text, graphics, moving pictures, sound. I think this point of view leads to an underestimation of the computer's potential. It is certainly true that a computer can incorporate and manipulate all other media, but the true power of the computer is that it is capable of manipulating not just the *expression* of ideas but also the ideas themselves. The amazing thing to me is not that a computer can hold the contents of all the books in a library but that it can notice relationships between the concepts described in the books—not that it can display a picture of a bird in flight or a galaxy spinning but that it can imagine and predict the consequences of the physical laws that create these wonders. The computer is not just an advanced calculator or camera or paintbrush; rather, it is a device that accelerates and extends our processes of thought. It is an imagination machine, which starts with the ideas we put into it and takes them farther than we ever could have taken them on our own.

CHAPTER 1

NUTS AND BOLTS

When I was a child, I read a story about a boy who built a robot out of parts he found lying around a junkyard. The boy's robot could move, talk, and think, just like a person, and it became his friend. For some reason, I found the idea of building a robot very appealing, so I decided to build one myself. I remember collecting body parts—tubes for the arms and legs, motors for the muscles, lightbulbs for the eyes, and a big paint can for the head—in the full and optimistic expectation that after they were assembled and the contraption was plugged in, I would end up with a working mechanical man.

After nearly electrocuting myself a few times, I began to get my parts to move, light up, and make noises. I felt I was making progress. I began to understand how to construct movable joints for the arms and legs. But something even more important was beginning to dawn on me: I didn't have the slightest idea how to control the motors and the lights, and I realized that something was missing in my knowledge of how robots worked. I now have a name for what was missing: it's called *computation*. Back then, I called it "thinking," and I saw that I didn't have a clue about how to get something to think. It seems obvious to me now that computation is the hardest part of building a mechanical man, but as a child this came as a surprise.

BOOLEAN LOGIC

Fortunately, the first book I ever read on the subject of computation was a classic. My father was an epidemiologist, and we were living in Calcutta at the time. Books in English were hard to come by, but in the library of the British consulate I found a dusty copy of a book written by the nineteenth-century logician George Boole. The title of the book was what attracted me: *An Investigation of the Laws of Thought.* This grabbed my imagination. Could there really be laws that governed thought? In the book, Boole tried to reduce the logic of human thought to mathematical operations. Although he did not really explain human thinking, Boole demonstrated the surprising power and generality of a few simple types of logical operations. He invented a language for describing and manipulating logical statements and determining whether or not they are true. The language is now called *Boolean algebra.*

Boolean algebra is similar to the algebra you learned in high school, except that the variables in the equations represent logic statements instead of numbers. Boole's variables stand for propositions that are either true or false, and the symbols $\wedge$, $\vee$, and $\neg$ represent the logical operations **And**, **Or**, and **Not**. For example, the following is a Boolean algebraic equation

$$\neg(A \vee B) = (\neg A) \wedge (\neg B)$$

This particular equation, called De Morgan's theorem (after Boole's colleague Augustus De Morgan), says that if neither A nor B is true, then both A and B must be false. The variables A and B can represent any logical (that is, true or false) statement. This particular equation is obviously correct, but Boolean algebra also allows much more complex logical statements to be written down and proved or disproved.

Boole's work found its way into computer science through the master's thesis of a young engineering student at the Massachusetts Institute of Technology named Claude Shannon. Shannon is best known for having invented a branch of mathematics called *information theory*, which defines the measure of information we call a *bit.* Inventing the bit was an impressive accomplishment, but what Shannon did with Boolean logic was at least as important to the science of computation. With these two pieces of work, Shannon laid the foundation for the developments that were to occur in the field of computing for the next fifty years.

Shannon was interested in building a machine that could play chess—and more generally in building mechanisms that imitated thought. In 1940, he published his master's thesis, which was titled "A Symbolic Analysis of Relay Switching Circuits." In it, he showed that it was possible to build electrical circuits equivalent to expressions in Boolean algebra. In Shannon's circuits, switches that were open or closed corresponded to logical variables of Boolean algebra that were true or false. Shannon demonstrated a way of converting any expression in Boolean algebra into an arrangement of switches. The circuit would establish a connection if the statement was true and break the connection if it was false. The implication of this construction is that any function capable of being described as a precise logical statement can be implemented by an analogous system of switches.

Rather than presenting the detailed formalisms developed by Boole and Shannon, I will give an example of their application in the design of a very simple kind of computing device, a machine that plays the game of tic-tac-toe. This machine is much simpler than a general-purpose computer, but it demonstrates two principles that are important in any type of computer. It shows how a task can be reduced to *logical functions* and how such functions can be implemented as

a circuit of connected switches. I actually built a tic-tac-toe machine out of lights and switches shortly after I read Boole's book in Calcutta, and this was my introduction to computer logic. Later, when I was an undergraduate at MIT, Claude Shannon became a friend and teacher, and I discovered that he, too, had used lights and switches to build a machine that could play tic-tac-toe.

As most readers know, the game is played on a 3 x 3 square grid. Players take turns marking the squares, one player using an *X*, the other an *O*. The first player to place three symbols in a row (horizontally, vertically, or diagonally) wins the game. Young children enjoy tic-tac-toe because it seems to offer limitless possible strategies for winning. Eventually they realize that only a small number of patterns can occur, and the game consequently loses its charm: once both players learn the patterns, each game invariably ends in a tie. Tic-tac-toe is a good example of a computation precisely because it wavers on this line between the complex and the simple. Crossing that line is what computation is all about. Computation is about performing tasks that seem to be complex (like winning a game of tic-tac-toe) by breaking them down into simple operations (like closing a switch).

In tic-tac-toe, the situations that occur are few enough so that it's practical to write them all down, and therefore to build the correct response in every case into the machine. We can use a simple two-step process for designing the machine: *first*, reduce the play to a series of cases defining the correct response to each pattern of moves; *second*, convert those cases into electrical circuits by wiring the switches to recognize the pattern and indicate the appropriate response.

One way to proceed would be to write down every conceivable arrangment of *X's* and *O's* which could be placed on the grid and then decide how the computer would play in each instance. Since each of the nine squares has three possible states (*X*, *O*, and blank), there are 3^9 (or 19,683) ways to

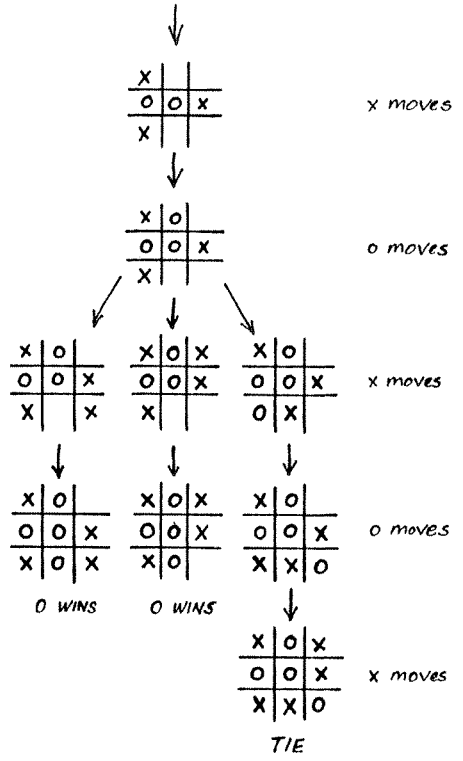

FIGURE 1

Part of a game tree for tic-tac-toe

fill the grid. But most of these patterns would never occur in the course of a game. A better method of listing the possibilities is to draw up a *game tree*—a configuration that traces every possible line of play. The game tree starts with a blank grid at the root and has a branch for every possible alternative line of play, determined by the move of the human player. (The tree does not need to branch when the machine plays, because the response of the machine to any given move is always predetermined.) Figure 1 shows a small part of such a tree. For every possible move made by *X*, the human player, there is a predetermined *O* response to be made by the

machine. (For some strange reason, computer scientists always draw trees upside-down, with the "root" at the top.)

The tree in Figure 1 illustrates the strategy that I always use in tic-tac-toe: I play in the center whenever I can. The machine's moves are determined by the human player's moves, which vastly reduces the number of possibilities to be considered. A full game tree, showing what the machine should do in every situation, has about five hundred or six hundred branches, the exact number depending on the details of strategy. Following the tree will cause the machine to win, or at least tie, every game. The rules of the game are built into the responses, so by following the tree the machine will always obey the rules. From this game tree, we can write down specifications that say exactly when the machine should play in any particular position. These specifications constitute the Boolean logic of the machine.

Once we have defined the desired behavior, we can translate that behavior into electrical circuits built out of batteries, wires, switches, and lights. The basic circuit in the machine is the same circuit used in a flashlight: when the switch is pressed down—that is, closed—the light goes on, because a complete path has been formed between the bulb and the battery. (The connections to the battery are indicated by the + and – signs.) Most important, these switches can be wired either *in series* or *in parallel*. For instance, we can put two switches together in series to make a light that works only when both switches are closed. This circuit implements one of the basic switching functions of the computer—the "logic block" known as the **And** function, so called because the bulb lights only when the first **and** the second switches are closed. Switches connected in parallel form the **Or** function, which connects the circuit (and thus lights the bulb) whenever either **or** both of the switches are closed (see Figure 2).

These simple patterns of serial and parallel wiring can be used in combinations to form connections that follow various logical rules. In the tic-tac-toe machine, chains of

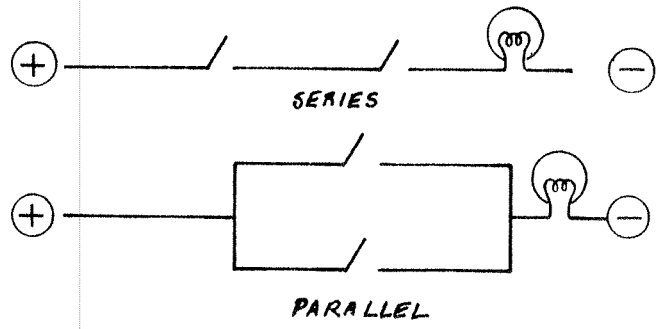

FIGURE 2

Switches in series and parallel

switches connected in *series* are used to detect patterns, and these chains are connected in *parallel* to lights, so that several patterns can light the same bulb—that is, produce the same response from the machine.

The tic-tac-toe machine I built has four banks of nine switches each, and each switch corresponds to one of the nine squares on the tic-tac-toe grid. It also has nine lightbulbs, arranged in the pattern of a tic-tac-toe board. The machine, which always plays first, makes its moves by lighting a bulb. The human player moves by closing a switch—using the first bank of switches to make his first move, the second bank for his second move, and so on. In my version, the machine always begins by playing in the upper left corner of the board, a scheme that reduces the number of cases considerably. The human player responds by closing one of the switches in the first bank (say, the one corresponding to the center square in the grid), and the game proceeds. The machine's strategy is embodied in the wiring between the switches and the lights.

The wiring that produces the machine's first response is easy (see Figure 3). Each switch in the first bank is connected to a light that corresponds to the machine's reply. For instance, a play in the center causes a response in the lower

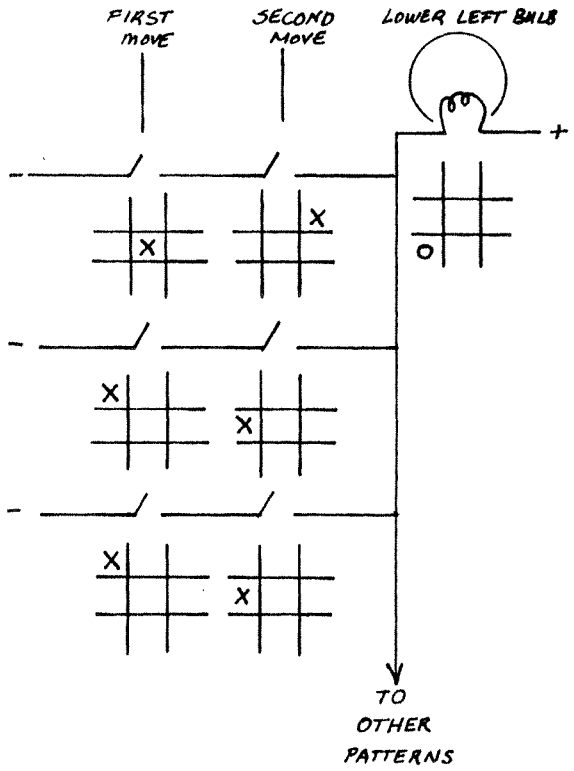

FIGURE 3

Several different patterns that produce the same response

right, so the center switch is wired to the lower-right light. Since my machine always responds in the center square if it can, most of the first bank of switches is wired in parallel to the middle light.

Each pattern for the second round of play depends on the human player's first and second moves. To recognize this combination of human moves, the corresponding switches are wired in series. For example, if the player's first move is in the center and second move in the upper right, the machine is then supposed to respond by playing in the lower

left. This pattern is accomplished by wiring the center switch in the first bank in series with the upper-right switch in the second bank ("if center **and** upper-right squares are filled, then . . . "), with the chain of two switches being connected to the lightbulb in the lower left. Each parallel connection to a bulb specifies a different combination that will cause the bulb to light ("this move or that move will provoke this response"). Whenever it was necessary to use the same switch in two different circuits, I used a "double throw" switch—two switches mechanically linked to the same button, so that they switch together—which allows the same move to be part of two different patterns. The wiring of the third and fourth banks of switches follows the same principle, but there are even more combinations. As you can imagine, the wiring gets complicated, even though the principles are simple. There are fewer choices open on the grid, but the chains of switches are longer.

The tic-tac-toe machine I built has about a hundred and fifty switches. This seemed like a lot to me at the time (I made the switches out of wood and nails), but the computer chips I design today have millions of switches, most of them connected in patterns very similar to those used in the tic-tac-toe machine. Most modern computers use a different kind of electrical switch—a transistor, which I will describe later—but the basic notion of connecting switches in series to produce the **And** function and connecting switches in parallel to produce the **Or** function is exactly the same.

While the logic of the tic-tac-toe machine is similar to the logic of a computer, there are several important differences. One is that the tic-tac-toe machine has no notion of events happening sequentially in time; therefore, the entire sequence of the game—that is, the entire game tree—must be determined in advance. This is cumbersome enough where tic-tac-toe is concerned and practically impossible for a more complicated game, like chess, or even checkers. Modern computers are very good at playing checkers and pretty good

at playing chess (see chapter 5), because in place of the predetermined game tree they use a different method—one that involves examining patterns sequentially in time.

Another difference between the tic-tac-toe machine and a general-purpose computer is that the tic-tac-toe machine can perform only one function. The "program" of the machine is built into its wiring. The tic-tac-toe machine has no software.

BITS AND LOGIC BLOCKS

As I noted in the Introduction, there is no reason the tic-tac-toe machine (or any other computer) has to be built out of electrical switches. A computer can represent information using electrical currents, fluid pressures, or even chemical reactions. Whether you build a computer out of transistors, hydraulic valves, or a chemistry set, the principles on which it operates are much the same. The key idea of the tic-tac-toe machine is that the **And** function is implemented by connecting two switches in series and the **Or** function is implemented by connecting two switches in parallel, but there are many other ways to implement **And** and **Or**.

Here I must pause to mention the *bit*. The smallest "difference that makes a difference" (to use Bateson's phrase again) is a difference that splits all signals into two distinct classes. In the tic-tac-toe machine, the two classes are "current flowing" and "no current flowing." By convention, we call the two possible classes 1 and 0. These are just names; we could as easily call them **True** and **False**, or **Alice** and **Bob**. Even the choice of which class is called 0 and which is called 1 is arbitrary. A signal that can carry one of two different messages (like 1 or 0) is called a *binary* signal, or a *bit*. A computer uses combinations of bits to represent all kinds of sets of alternatives—different moves in tic-tac-toe, say, or different colors to be displayed on a screen. Since the convention is to designate the bits by 1's and 0's, people often think of these bit patterns as numbers, hence the old chestnut "The computer does everything with numbers." But this convention is simply a way of thinking about what's going on. If we had named the two possible messages conveyed by the bit the letters X and Y, people would be saying, "The computer does everything with letters." The more accurate statement is "The computer represents numbers, letters, and everything else with patterns of bits."

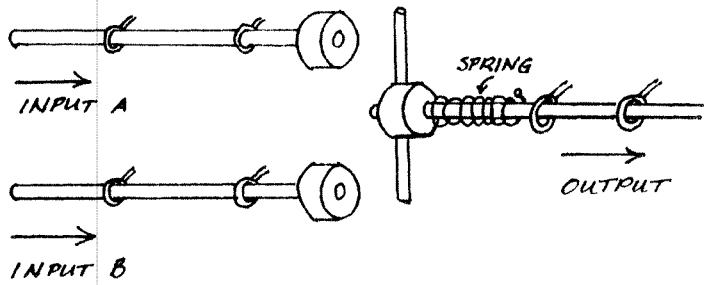

FIGURE 4

Mechanical implementation of the OR function

Instead of using the flow of electricity to represent a bit, we could have used mechanical motion. Figure 4 shows how the **Or** function is implemented using a technology that represents 1 by sliding a stick to the right. As long as both the *A* and the *B* input sticks stay to the left, representing 0, then the spring will keep the output stick pushed to the left, but if either input stick slides to the right, then the output stick will slide to the right also. The object in Figure 5 computes another useful function, that of inversion: The inverter turns every signal into its opposite: for example, it turns a push to the right into a pull to the left, and vice versa.

These **And**, **Or**, and **Invert** functions are *logic blocks*, and they can be connected in order to create other functions. For instance, the output of an **Or** block can be connected to an **Invert** block to create a **Nor** function: the **Nor**

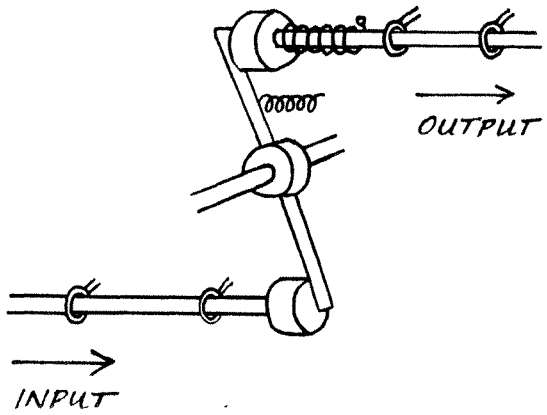

FIGURE 5

Mechanical inverter

output will be a 1 when neither of its inputs is 1. In another example (using De Morgan's theorem), we can make an **And** block by connecting two **Invert** blocks to the inputs of an **Or** block and connecting a third **Invert** block to the output (see Figure 6). These four work together to implement the **And** function, so the final output is 1 only when both the inputs are 1.

Early computing devices were made with mechanical components. In the seventeenth century, Blaise Pascal built a mechanical adding machine, which inspired both Gottfried Wilhelm Leibniz and the English polymath Robert Hooke to build improved machines that could multiply, divide, and even take square roots. These machines were not programmable, but in 1833 another Englishman, the mathematician and inventor Charles Babbage, designed and partially constructed a programmable mechanical computer. Even as late as my own childhood in the sixties, most arithmetic calculators were mechanical. I've always liked these mechanical machines, because you can see what's happening, which is not the case with electronic

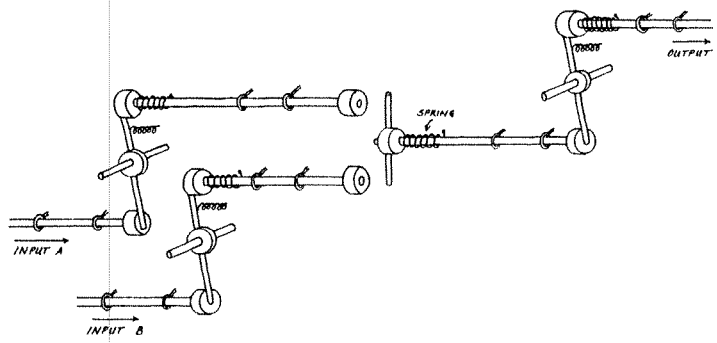

FIGURE 6

An And block constructed by connecting an Or block to inverters

computers. When I'm designing an electronic computer chip, I imagine the operation of the circuits as moving mechanical parts.

THE FLUID COMPUTER

The picture I have in my mind when I design a logic circuit is of hydraulic valves. A hydraulic valve is like a switch that controls and is controlled by the flow of water. Each valve has three connections: the input, the output, and the control. Pressure on the control connection pushes on a piston that turns off the water flow from input to output. Figure 7 shows a circuit for the **Or** function, built out of hydraulic valves.

In this circuit, water pressure is used to distinguish between the two possible signals. Notice that in a hydraulic valve the control pipe can affect the output pipe but the output pipe cannot affect the control pipe. This restriction establishes a forward flow of information through the switch; in a sense, it establishes a direction in time. Also, since the valve is

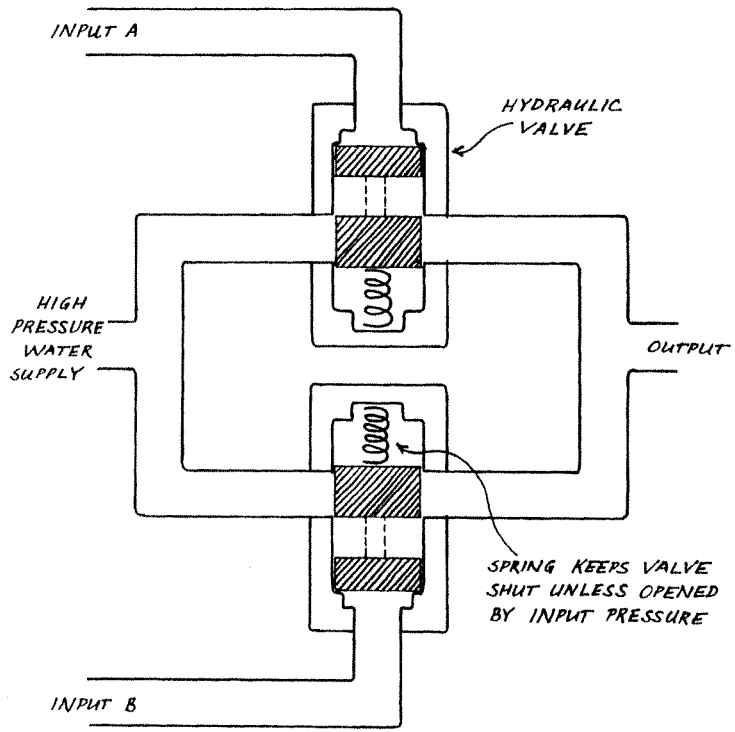

FIGURE 7

An Or block built with hydraulic valves

either open or closed, it serves an additional function of *amplification*, which allows the strength of the signal to be restored to its maximum value at every stage. Even if the input is a little low on pressure—because it goes through a long, thin pipe, say, or because of a leak—the output will always be at full pressure thanks to the on/off operation of the valve. This is the fundamental difference between *digital* and *analog*: A digital valve is either on or off; an analog valve, like your kitchen faucet, can be anything in between. In the hydraulic computer, all that is required of the input signal is that it be

strong enough to move the valve. In this case, the difference that makes a difference is the difference in water pressure sufficient to switch the valve on. And since a weakened signal entering an input will still produce a full-strength output, we can connect thousands of layers of logic, the output of one layer controlling the next, without worrying about a gradual decrease in pressure. The output of each gate will always be at full pressure.

This type of design is called *restoring logic*, and the example in hydraulic technology is particularly interesting, because it corresponds almost exactly to the logic used in modern electronic computers. The water pressure in the pipes is analogous to the voltage on the wires, and the hydraulic valve is analogous to the metal-oxide transistor. The control, input, and output connections on the valve correspond closely to the three connections (called *gate*, *source*, and *drain*) on a transistor. The analogy between water valves and transistors is so exact that you could translate the design for a modern microprocessor directly into a design for a hydraulic computer. To do so, you would need to look at the pattern of wires on the silicon chip under a microscope and then bend a set of pipes into the same shapes as the wires on the chip and connect them in exactly the same pattern. In place of each transistor, you would use a hydraulic valve. The pipe that corresponds to the power-supply voltage on your chip would be connected to a pressurized water supply, and the pipe that corresponds to the ground connection could empty down a drain.

To use the hydraulic computer, you would have to connect hydraulic equivalents of its inputs and outputs—you would need to build a hydraulic keyboard, a hydraulic display, hydraulic memory chips, and so on—but if you did all this, it would go through exactly the same switching events as the electronic chip. Of course, the hydraulic computer would be much slower than your latest microprocessor (to say nothing of larger), because water pressure travels down pipes much more slowly than electricity travels down wires. As to the

size: Since the modern microchip has several million transistors, its hydraulic equivalent would require several million valves. A transistor in a chip is about a millionth of a meter across; a hydraulic valve is about 10 centimeters on a side. If the pipes scale proportionally, then the hydraulic computer would cover about a square kilometer with pipes and valves. From an airplane, it would look roughly the same as the electronic chip does under a microscope.

When I design a computer chip, I draw lines on a computer screen, and the pattern is reduced (in a process analogous to photographic reduction) and etched onto a chip of silicon. The lines on the screen are my pipes and valves. Actually, most computer designers don't even bother drawing lines; instead, they specify the connections between **And**s and **Or**s and let a computer work out the details of placement and geometry of the switches. Most of time, they forget about the technology and concentrate on the function. I do this, too, sometimes, but I still prefer to draw my own shapes. Whenever I design a chip, the first thing I want to do is look at it under a microscope—not because I think I can learn something new by looking at it but because I am always fascinated by how a pattern can create reality.

TINKER TOYS

Except for the miracle of reduction, there is no special reason to build computers with silicon technology. Building a computer out of any technology requires a large supply of only two kinds of elements: *switches* and *connectors*. The switch is a steering element (the hydraulic valve, or the transistor), which can combine multiple signals into a single signal. Ideally, the switch should be asymmetrical, so that the input signal affects the output signal but not vice versa, and it should have a restoring quality, so that a weak or degraded

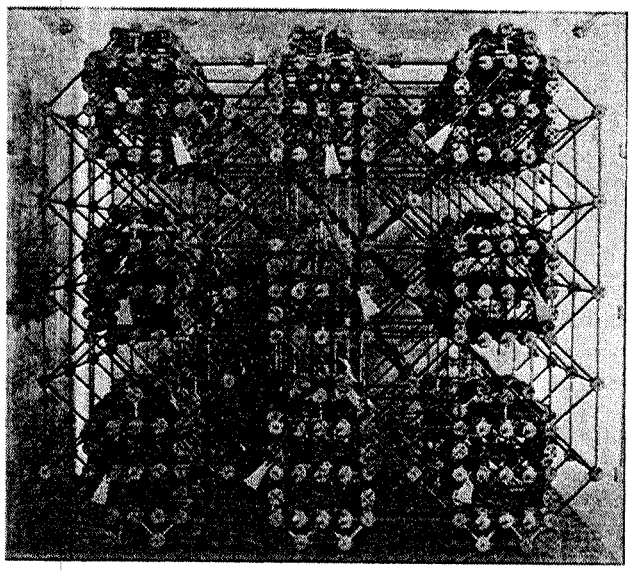

FIGURE 8

Tinker Toy computer

input signal will not result in a degraded output. The second element, the connector, is the wire or pipe that carries a signal between switches. This connecting element must have the ability to branch, so that a single output can feed many inputs. These are the only two elements necessary to build a computer. Later we will introduce one more element—a register, for storing information—but this can be constructed of the same steering and connecting components.

I have never built a hydraulic computer, but once, with some friends, I did construct a computer out of sticks and strings. The pieces came from a children's construction set called Tinker Toys. Readers may remember this as a set of cylindrical wooden sticks that fit into fat little wooden hubs with holes in them. The logic of my Tinker Toy computer worked much like that shown in Figure 8. Like the switches-and-lights computer, the Tinker Toy computer played tic-tac-toe. It never lost. The computer was a lot of trouble to make,

requiring tens of thousands of pieces from more than a hundred Tinker Toy "Giant Engineer" construction sets, and the finished product (now sitting in the Computer Museum in Boston, Massachusetts) looks incomprehensibly complex. Yet the principles on which it operates are just the simple combination of **And** and **Or** functions described above.

The big mistake I made in designing the Tinker Toy computer is that I did not use *restoring logic*—that is, there was no amplification from one stage of logic to the next. The implementation of the logic was based on sticks pressing against sticks, in a design similar to the one illustrated in figure 4. Because of this design choice, all the force required to move the hundreds of elements in the machine had to be supplied by the press of the input switch. The accumulated force tended to stretch the strings that transmitted the motion, and because there was no restoration at each stage, the errors caused by the stretching accumulated from one logic element to the next. Unless the strings were constantly tuned, the machine would make mistakes.

I constructed a later version of the Tinker Toy computer which fixed the problem, but I never forgot the lesson of that first machine: the implementation technology must produce perfect outputs from imperfect inputs, nipping small errors in the bud. This is the essence of digital technology, which restores signals to near perfection at every stage. It is the only way we know—at least, so far—for keeping a complicated system under control.

FREE TO WORRY ABOUT THE DIFFERENCE THAT MAKES A DIFFERENCE

Naming the two signals in computer logic 0 and 1 is an example of functional abstraction. It lets us manipulate information without worrying about the details of its underlying representation. Once we figure out how to accomplish a given function, we can put the mechanism inside a "black box," or a "building block" and stop thinking about it. The function embodied by the building block can be used over and over, without reference to the details of what's inside. This process of functional abstraction is a fundamental in computer design—not the only way to design complicated systems but the most common way (later, I'll describe an alternate method). Computers are built up of a hierarchy of such functional abstractions, each one embodied in a building block. The blocks that perform functions are hooked together to implement more complex functions, and these collections of blocks in turn become the new building blocks for the next level.

This hierarchical structure of abstraction is our most powerful tool in understanding complex systems, because it lets us focus on a single aspect of a problem at a time. For instance, we can talk about Boolean functions like **And** and **Or** in the abstract, without worrying about whether they are built out of electrical switches or sticks and strings or water-operated valves. For most purposes, we can forget about technology. This is wonderful, because it means that almost everything we say about computers will be true even when transistors and silicon chips become obsolete.

CHAPTER 2

UNIVERSAL BUILDING BLOCKS

From now on, we can forget about wires and switches and work with the abstraction of logic blocks operating on 1's and 0's, a simple step that allows us to pass from the realm of engineering into the realm of mathematics. This is the most abstract chapter in the book; it will show you how the methods used to construct a tic-tac-toe machine can be used to construct almost any function. In it, we'll define a powerful set of building blocks: logical functions and finite-state machines. With these elements, it's easy to build a computer.

LOGICAL FUNCTIONS

In constructing the tic-tac-toe machine, we began by writing the game tree, whch gave us a set of rules for generating the outputs from the inputs. This turns out to be a generally useful method of attack. Once we write down the rules that specify what outputs we want for each combination of inputs, we can build a device that implements these rules using **And**, **Or**, and **Invert** functions. The logic blocks **And**,

Or, and **Invert** form a *universal construction set,* which can be used to implement any set of rules. (These primitive types of logic blocks are sometimes also called *logic gates.*)

This idea of a universal set of blocks is important: it means that the set is general enough to build anything. My favorite toy when I was a child was a set of interlocking plastic bricks called Lego blocks, with which I built all kinds of toys: cars, houses, spaceships, dinosaurs. I loved to play with these blocks, but they were not quite universal, since the only objects you could build with them had a certain squarish, stair-steppy look. Building something with a different shape—a cylinder or a sphere, for example—would require a new type of block. Eventually, I had to switch to another medium in order to build the things I wanted. But the **And, Or,** and **Invert** blocks of Boolean logic are a universal construction set for converting inputs to outputs. The best way to see how they form a universal set is to understand a general method for using them to implement rules. To start, we will consider *binary* rules—rules that specify inputs and outputs that are either 1 or 0. The tic-tac-toe machine is a good example of a function specified by binary rules, because the input switches and the output lights are either on or off—that is, either 1 or 0. (Later, we will discuss rules for handling letters, numbers, or even pictures and sounds as inputs and outputs.) Any set of binary rules can be completely specified by showing a table of the outputs for each possible combination of 1's and 0's on the inputs. For example, the rules for the **Or** function are specified by the following table:

	Input A	Input B	Output
	0	0	0
OR Function	0	1	1
	1	0	1
	1	1	1

The **Invert** function is specified by an even simpler table:

	Input	Output
Invert Function	0	1
	1	0

For a binary function with n inputs, there are 2^n possible combinations of input signals. Sometimes we won't bother to specify all of them, because we don't care about certain combinations of inputs. For example, in specifying the function performed by the tic-tac-toe machine, we don't care what happens if the human player plays in all squares simultaneously. This move would be disallowed, and we don't need to specify the function's output for this combination of inputs.

Complex logic blocks are constructed by connecting **And, Or,** and **Invert** blocks. In drawings of the connection pattern, the three blocks are conventionally represented by boxes of different shape (see Figure 9); the lines connecting on the left side represent inputs to the blocks, and the lines connecting on the right represent the output. Figure 10 shows how a pair of two-input **Or** blocks can be connected to form a three-input **Or** function; the output of this function will be 1 if *any one* of its three inputs is 1. It's also possible to string several **And** blocks together in a similar manner to make an **And** block with any number of inputs.

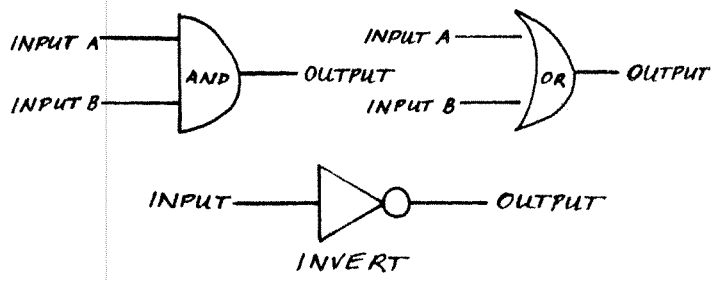

FIGURE 9

And, Or, **and** Invert **Blocks**

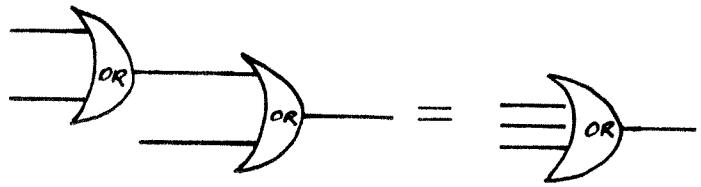

FIGURE 10

A three-input Or block made from a pair of two-input Or blocks

Figure 11 shows how an **And** block can be constructed by connecting an Inverter to the inputs and output of an **Or** block. (Here is De Morgan's theorem again.) The best way to get a feeling for how this works is to trace through the 1's and 0's for every combination of inputs. Notice that this illustration is essentially the same as Figure 6 in the previous chapter. It points up an interesting fact: we don't really need **And** blocks in our universal building set, because we can always construct them out of **Or** blocks and **Inverters.**

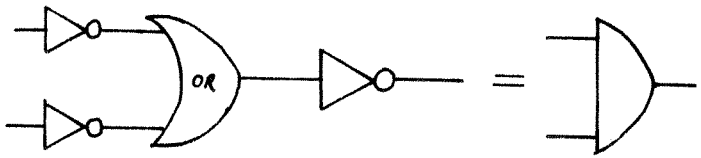

FIGURE 11

Making And out of Or

As in the tic-tac-toe playing machine, **And** blocks are used to detect each possible combination of inputs for which the output is 1, while **Or** blocks provide a roster of these combinations. For example, let's start with a simple function of three inputs. Imagine that we want to build a block that allows the three inputs to vote on the output. In this new

block, majority wins—that is, the output will be 1 only if two or more of the inputs are 1.

Inputs			Majority Output
A	B	C	
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figure 12A shows how this function is implemented.An **And** block with the appropriate **Invert** blocks as input is used to recognize each combination of inputs for which the output is 1; these blocks are connected by an **Or** block, which produces the output. This strategy can be used to create any transformation of inputs to outputs:

Of course, this particular method of using a separate **And** gate to recognize each combination of inputs is not the only way to implement the function, and it is often not the simplest way. Figure 12B shows a simpler way to produce the majority function. The great thing about the method described is not that it produces the best implementation but that it always produces an implementation that works. The important conclusion to draw is that it is possible to combine **And, Or,** and **Invert** blocks to implement *any* binary function—that is, any function that can be specified by an input/output table of 0's and 1's.

Restricting the inputs and output to binary numbers is not really much of a restriction, because the combinations of 1's and 0's can be used to represent other things—letters,

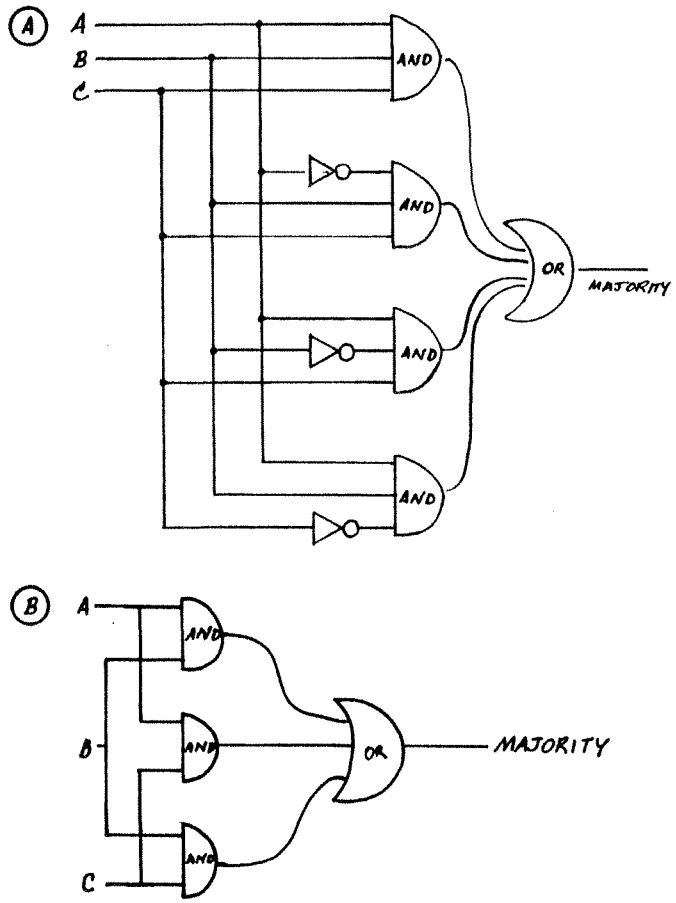

FIGURE 12

How the voting function is implemented by And, Or, **and** Invert **Blocks**

larger numbers, any entity that can be encoded. As an example of a nonbinary function, suppose we want to build a machine to act as a judge of the children's game of Scissors/Paper/Rock. This is a game for two players in which each chooses, in secret, one of three "weapons"—scissors,

paper, or rock. The rules are simple: scissors cuts paper, paper covers rock, rock crushes scissors. If the two children choose the same weapon, they tie. Rather than building a machine that plays the game (which would involve guessing which weapon the opponent is going to choose), we will build a machine that judges who wins. Here's the input/output table for the function that takes the choices as inputs and declares the winner as output. The table encodes the rules of the game:

Input A	**Input B**	**Output**
Scissors	Scissors	Tie
Scissors	Paper	A wins
Scissors	Rock	B wins
Paper	Scissors	B wins
Paper	Paper	Tie
Paper	Rock	A wins
Rock	Scissors	A wins
Rock	Paper	B wins
Rock	Rock	Tie

The Scissors-Paper-Rock judging function is a combinational function, but it is not a binary function, since its inputs and output have more than two possible values. To implement this function as a combinational logic block, we must convert it to a function of 1's and 0's. This requires us to establish some convention for representing the inputs and outputs. A simple way to do this would be to use a separate bit for each of the possibilities. There would be three input signals for each weapon: a 1 on the first input represents **Scissors,** a 1 on the second input represents **Rock**, and a 1 on the third input represents **Paper**. Similarly, we could use separate output lines to represent a win for player A, a win for player B, or a tie. So the box would have six inputs and three outputs.

Using three input signals for each weapon is a perfectly good way to build the function, but if we were doing it inside a computer we would probably use some kind of

encoding that required a smaller number of inputs and outputs. For example, we could use two bits for each input and use the combination 01 to represent **Scissors**, 10 to represent **Paper**, and 11 to represent **Rock**. We could similarly encode each of the possible outputs using two bits. This encoding would result in the simpler three-input/two-output table shown below:

	A Inputs	B Inputs	Outputs
	01	01	00
	01	10	10
Scissors = 01	01	11	01
Paper = 10	10	01	01
Rock = 11	10	10	00
A wins = 10	10	11	10
B wins = 01	11	01	10
Tie = 00	11	10	01
	11	11	00

Computers can use combinations of bits to represent anything; the number of bits depends on the number of messages that need to be distinguished. Imagine, for example, a computer that works with the letters of the alphabet. Five-bit input signals can represent thirty-two different possibilities ($2^5 = 32$). Functions within the computer that work on letters sometimes use such a code, although they more often use an encoding with seven or eight bits, to allow representation of capitals, punctuation marks, numerals, and so on. Most modern computers use the standard representation of alphabet letters called ASCII (an acronym for American Standard Code for Information Interchange). In ASCII, the sequence 1000001 represents the capital letter **A**, and 1000010 represents the capital **B**, and so on. The convention, of course, is arbitrary.

Most computers have one or more conventions for representing numbers. One of the most common is the *base 2* representation of numbers, in which the bit sequence 0000000

represents zero, the sequence 0000001 represents the number 1, the sequence 0000010 represents 2, and so on. The description of computers as "64-bit" or "32-bit" indicates the number of bit positions in the representation used by the computer's circuits: a 32-bit computer uses a combination of thirty-two bits to represent a base-2 number. The base-2 number system is a common convention, but there is nothing that requires its use. Some computers don't use it at all, and most computers that do also represent numbers in other ways for various purposes. For instance, many computers use a slightly different convention for representing negative numbers and also have a convention called a *floating point* to represent numbers that have decimal points. (The position of the decimal point "floats" relative to the digits, so that a fixed number of digits can be used to represent a wide range of numbers.) The particular representation schemes are often chosen in such a way as to simplify the logic of the circuits that perform arithmetical operations, or to make it easy to convert from one representation to another.

Because any logical function can be implemented as a Boolean logic block, it is possible to build blocks that perform arithmetical operations like addition or multiplication by using numbers with any sort of representation. For instance, imagine that we want to build a functional block that will add numbers on an eight-bit computer. An eight-bit adder block must have sixteen input signals (eight for each of the numbers to be added), and eight output signals for the sum. Since each number is represented by eight bits, there are 256 possible combinations, and each can represent a different number. For example, we could use these combinations to represent the numbers between 0 and 255, or between –100 and +154. Defining the function of the block would be just a matter of writing down the addition table and then converting it to 1's and 0's, using the chosen representation. The table of 1's and 0's could then be converted to **And** and **Or** blocks by the methods described above.

By adding two more inputs to the block, we could use similar techniques to build a block that not only adds but also subtracts, multiplies, and divides. The two extra control inputs would specify which of these operations was to take place. For instance, on every line of the table where the control inputs were 01, we would specify the output to be the sum of the input numbers, whereas in every combination where the control inputs were 10, we would specify the outputs to be the product, and so on. Most computers have logical blocks of this type inside them called *arithmetic units.*

Combining **Ands** and **Ors** according to this strategy is one way to build any logical function, but it is not always the most efficient way. Often, by clever design, you can implement a circuit using far fewer building blocks than the preceding strategy requires. It may also be desirable to use other types of building blocks or to design circuits that minimize the delay from input to output. Here are some typical puzzles in logic design: How do you use **And** blocks and **Inverters** to construct **Or** blocks? (Easy.) How do you use a collection of **And** and **Or** blocks, plus only two **Inverters,** to construct the function of three **Inverters**? (Hard, but possible.) Puzzles like this come up in the course of designing a computer, which is part of what makes the process fun.

FINITE-STATE MACHINES

The methods I've described can be used to implement any function that stays constant in time, but a more interesting class of functions are those that involve sequences in time. To handle such functions, we use a device called a *finite-state machine.* Finite-state machines can be used to implement time-varying functions—functions that depend not just on the current input but also on the previous history of inputs. Once you learn to recognize a finite-state machine, you'll notice

them everywhere—in combination locks, ballpoint pens, even legal contracts. The basic idea of a finite-state machine is to combine a look-up table, constructed using Boolean logic, with a memory device. The memory is used to store a summary of the past, which is the *state* of the finite-state machine.

A combination lock is a simple example of a finite-state machine. The state of a combination lock is a summary of the sequence of numbers dialed into the lock. The lock doesn't remember all the numbers that have ever been dialed into it, but it does remember enough about the most recent numbers to know when they form the sequence that will open the lock. An even simpler example of a finite-state machine is the retractable ballpoint pen. This finite-state machine has two possible states—extended and retracted—and the pen remembers whether its button has been pressed an odd or an even number of times. All finite-state machines have a fixed set of possible states, a set of allowable inputs that change the state (clicking a pen's button, or dialing a number into a combination lock), and a set of possible outputs (retracting or extending the ballpoint, opening the lock). The outputs depend only on the state, which in turn depends only on the history of the sequence of inputs.

Another simple example of a finite-state machine is a counter, such as the tally counter on a turnstile indicating the number of people who have passed through. Each time a new person goes through, the counter's state is advanced by one. The counter is a *finite* state because it can only count up to a certain number of digits. When it reaches its maximum count—say, 999—the next advance will cause it to return to zero. Odometers on automobiles work like this. I once drove an old Checker cab with an odometer that read 70,000, but I never knew if the cab had traveled 70,000 miles, 170,000 miles, or 270,000 miles, because the odometer had only 100,000 states; all those histories were equivalent as far as the odometer was concerned. This is why mathematicians often define a state as "a set of equivalent histories."

Other familiar examples of finite-state machines include traffic lights and elevator-button panels. In these machines, the sequence of states is controlled by some combination of an internal clock and input buttons such as the "Walk" button at the crosswalk and the elevator call and floor-selection buttons. The next state of the machine depends not only on the previous state but also on the signals that come from the input button. The transition from one state to another is determined by a fixed set of rules, which can be summarized by a simple state diagram showing the transition between states. Figure 13 shows a state diagram for a traffic-light

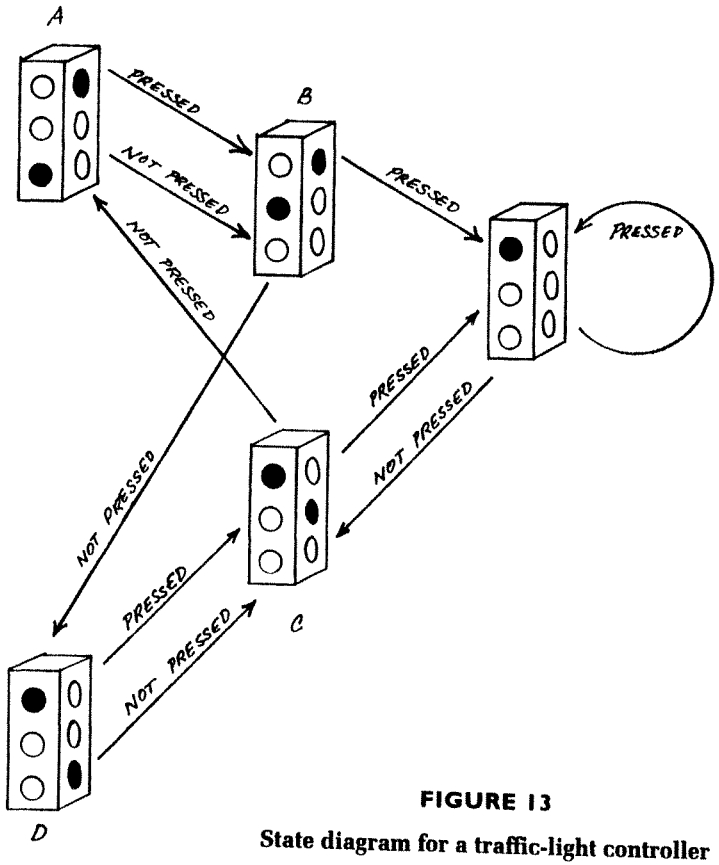

FIGURE 13

State diagram for a traffic-light controller

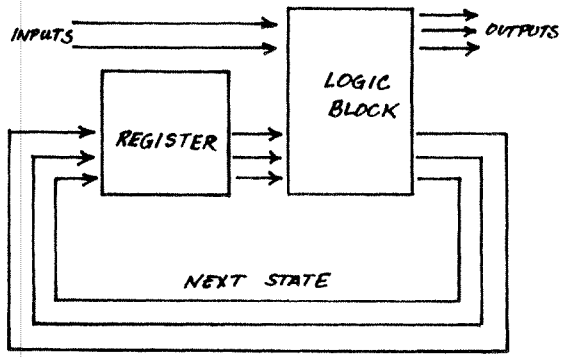

FIGURE 14

Finite-state machine, with logic block feeding register

controller at an intersection where the light turns red in both directions after the Walk button is pressed. Each drawing of light represents a state and each arrow represents a transition between states. The transition depends on whether or not the "walk" button is pressed.

To store the state of the finite-state machine, we need to introduce one last building block—a device called a *register,* which can be used to store bits. An *n*-bit register has *n* inputs and *n* outputs, plus an additional timing input that tells the register when to change state. Storing new information is called "writing" the state of the register. When the timing signal tells the register to write a new state, the register changes its state to match the inputs. The outputs of the register always indicate its current state. Registers can be implemented in many ways, one of which is to use a Boolean logic block to steer the state information around in a circle. This type of register is often used in electronic computers, which is why they lose track of what they're doing if their power is interrupted.

A finite-state machine consists of a Boolean logic block connected to a register, as shown in Figure 14. The finite-state machine advances its state by writing the output of the

Boolean logic block into the register; the logic block then computes the next state, based on the input and the current state. This next state is then written into the register on the next cycle. The process repeats in every cycle.

The function of a finite-state machine can be specified by a table that shows, for every state and every input, the state that follows. For example, we can summarize the operation of the traffic-light controller by the following table:

Inputs:			*Outputs:*	
Walk Button	**Current State**	**Main Road**	**Cross Road**	**Next State**
Not Pressed	A	Red	Green	B
Not Pressed	B	Red	Yellow	D
Not Pressed	C	Yellow	Red	A
Not Pressed	D	Green	Red	C
Not Pressed	Walk	Walk	Walk	D
Pressed	A	Red	Green	B
Pressed	B	Red	Yellow	Walk
Pressed	C	Yellow	Red	Walk
Pressed	D	Green	Red	C
Pressed	Walk	Walk	Walk	Walk

The first step in implementing a finite-state machine is to generate such a table. The second step is to assign a different pattern of bits to each state. The five states of the traffic-light controller will require three bits. (Since each bit doubles the number of possible patterns, it is possible to store up to 2^n states using n bits.) By consistently replacing each word in the preceding table with a binary pattern, we can convert the table to a function that can be implemented with Boolean logic.

In the traffic-light system, a timer controls the writing of the register, which causes the state to change at regular intervals. Another example of a finite-state machine that advances its state at regular intervals is a digital clock. A digital clock with a seconds indicator can be in one of 24 x 60 x 60 = 86,400

possible display states—one for each second of the day. The timing mechanism within the clock causes it to advance its state exactly once per second. Many other types of digital computing devices, including most general-purpose computers, also advance their state at regular intervals, and the rate at which they advance is called the *clock rate* of the machine. Within a computer, time is not a continuous flow but a fixed sequence of transitions between states. The clock rate of the computer determines the rate of these transitions, hence the correspondence between physical and computational time. For instance, the laptop computer on which I am writing this book has a clock rate of 33 megahertz, which means that it advances its state at a rate of 33 million times per second. The computer would be faster if the clock rate were higher, but its speed is limited by the time required for information to propagate through the logic blocks to compute the next state. As technology improves, the logic tends to become faster and the clock rate increases. As I write these words, my computer is state-of-the-art, but by the time you read this book computers with 33 megahertz clock rates will probably be considered slow. This is one of the wonders of silicon technology: as we learn to make computers smaller and smaller, the logic becomes faster and faster.

One reason finite-state machines are so useful is that they can recognize sequences. Consider a combination lock that opens only when it is given the sequence 0–5–2. Such a lock, whether it is mechanical or electronic, is a finite-state machine with the state diagram shown in Figure 15.

A similar machine can be constructed to recognize any finite sequence. Finite-state machines can also be made to recognize sequences that match certain patterns. Figure 16 shows one that recognizes any sequence starting with a 1, followed by a sequence of any number of 0s, followed by a 3. Such a combination will unlock the door with the combination 1–0–3, or a combination such as 1–0–0–0–3, but *not* with the combination 1–0–2–3, which doesn't fit the pattern. A more complex

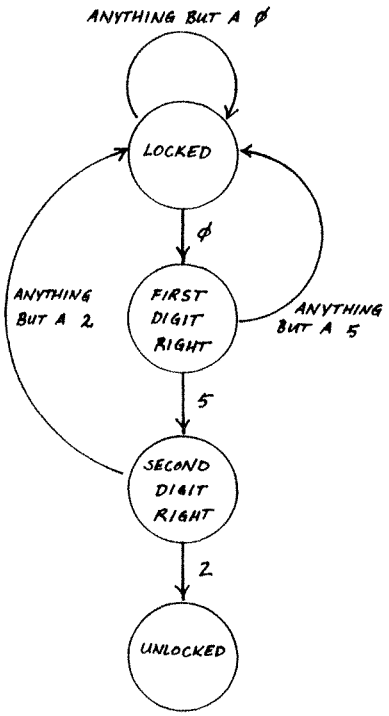

FIGURE 15

State diagram for a lock with combination 0-5-2

finite-state machine could recognize a more complicated pattern, such as a misspelled word within a stream of text.

As powerful as they are, finite-state machines are not capable of recognizing all types of patterns in a sequence. For instance, it is impossible to build a finite-state machine that will unlock a lock whenever you enter any palindrome—a sequence that is the same forward and backward, like 3–2–1–1–2–3. This is because palindromes can be of any length, and to recognize the second half of a palindrome you need to remember every character in the first half. Since there are infinitely many possible first halves, this would require a machine with an infinite number of states.

A similar argument demonstrates the impossibility of building a finite-state machine that recognizes whether a given English sentence is grammatically correct. Consider the sim-

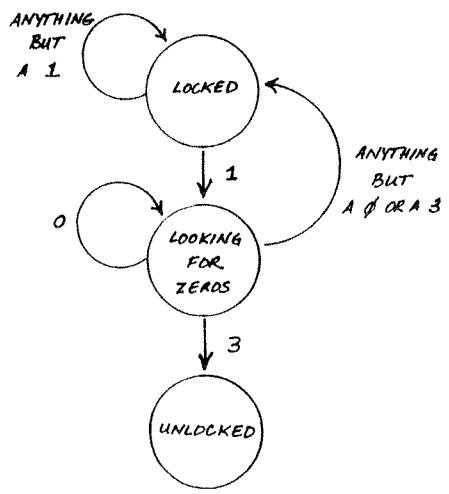

FIGURE 16

State diagram to recognize sequences like 1,0,3 and 1,0,0,0,3

ple sentence "Dogs bite." The meaning of this sentence can be changed by putting a qualifier between the noun and the verb; for instance, "Dogs that people annoy bite." This sentence can in turn be modified by putting another phrase in the middle: "Dogs that people with dogs annoy bite." Although the meaning of such sentences might be expressed more clearly, and although they become increasingly difficult to understand, they are grammatically correct. In principle, this process of nesting phrases inside of one another can go on forever, producing absurd sentences like "Dogs that dogs that dogs that dogs annoy ate bit bite." Recognizing such a sentence as grammatically correct is impossible for a finite-state machine, and for exactly the same reason it's difficult for a person: you need a lot of memory to keep track of all those dogs. The fact that human beings seem to have trouble with the same kinds of sentences that stump finite-state machines has caused some people to speculate that we may have something like a finite-state machine inside our head for understanding language. As you will see in the next chapter, there are other types of computing devices that seem to fit even more naturally with the recursive structure of human grammar.

I was introduced to finite-state machines by my mentor Marvin Minsky. He presented me with the following famous puzzle, called the *firing squad problem*: You are a general in charge of an extremely long line of soldiers in a firing squad. The line is too long for you to shout the order to "fire," and so you must give your order to the first soldier in the line, and ask him to repeat to the next soldier and so on. The hard part is that all the soldiers in the line are supposed to fire at the same time. There is a constant drumbeat in the background; however, you can't even specify that the men should all fire after a certain number of beats, because you don't know how many soldiers are in the line. The problem is to get the entire line to fire simultaneously; you can solve it by issuing a complex set of orders which tells each soldier what to say to the soldiers on either side of him. In this problem, the soldiers are equivalent to a line of finite-state machines with each machine advancing its state by the same clock (the drumbeat), and each receiving input from the output of its immediate neighbors. The problem is therefore to design a line of identical finite-state machines that will produce the "fire" output at the same time in response to a command supplied at one end. (The finite-state machines at either end of the line are allowed to be different from the others.) I won't spoil the puzzle by giving away the solution, but it can be solved using finite-state machines that have only a few states.

Before showing you how Boolean logic and finite-state machines are combined to produce a computer, I'll skip ahead in this bottom-up description and tell you where we're going. The next chapter starts by setting out one of the highest levels of abstraction in the function of a computer, which is also the level at which most programmers interact with the machine.

CHAPTER 3

PROGRAMMING

The magic of a computer lies in its ability to become almost anything you can imagine, as long as you can explain exactly what that is. The hitch is in explaining what you want. With the right programming, a computer can become a theater, a musical instrument, a reference book, a chess opponent. No other entity in the world except a human being has such an adaptable, universal nature. Ultimately all these functions are implemented by the Boolean logic blocks and finite-state machines described in the previous chapter, but the human computer programmer rarely thinks about these elements; instead, programmers work with a more convenient tool called a *programming language.*

Just as Boolean logic and finite-state machines are the building blocks of computer hardware, a programming language is a set of building blocks for constructing computer software. Like a human language, a programming language has a vocabulary and a grammar, but unlike a human language there is an exact meaning in the programming language for every word and sentence. Most programming languages are universal, in the same sense that Boolean logic is universal: they can be used to describe anything a computer can do. Anyone who has ever written a program—or debugged a program—knows that telling a computer what