CHAPTER I
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

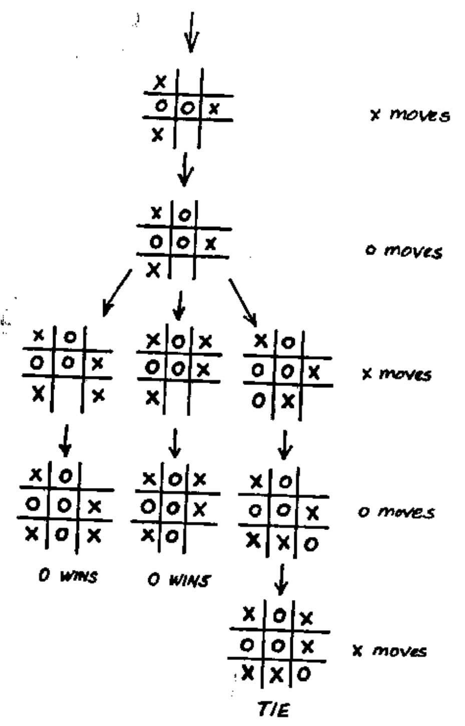

FIGURE 1

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

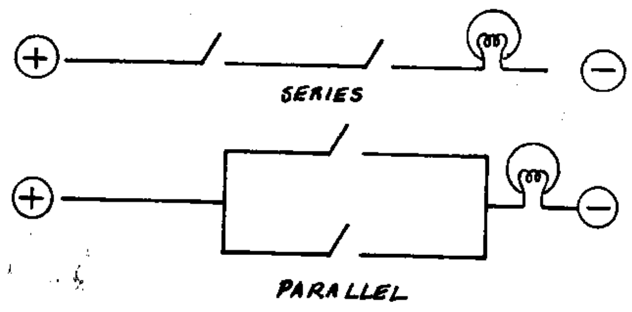

FIGURE 2

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

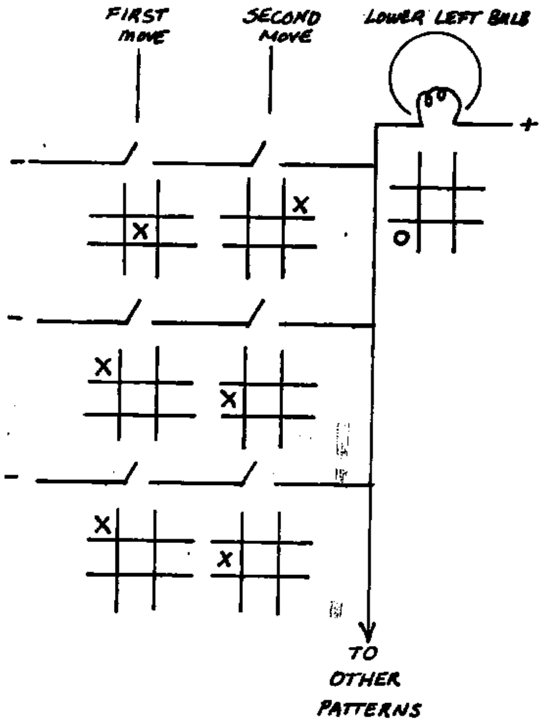

FIGURE 3

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."

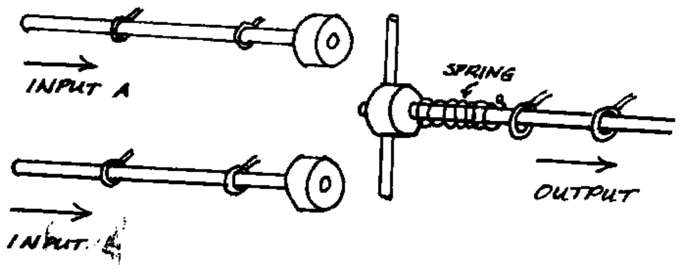

FIGURE 4

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**

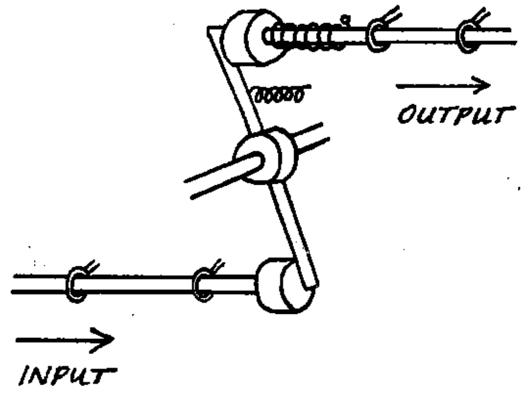

FIGURE 5

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

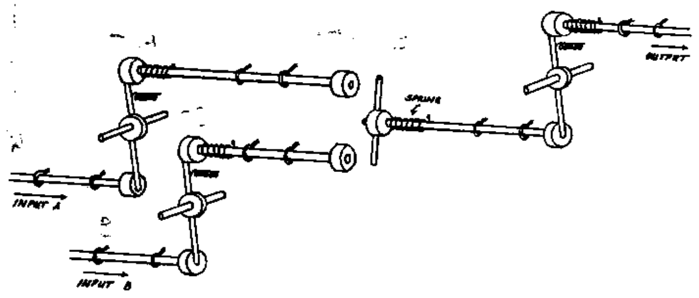

FIGURE 6

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

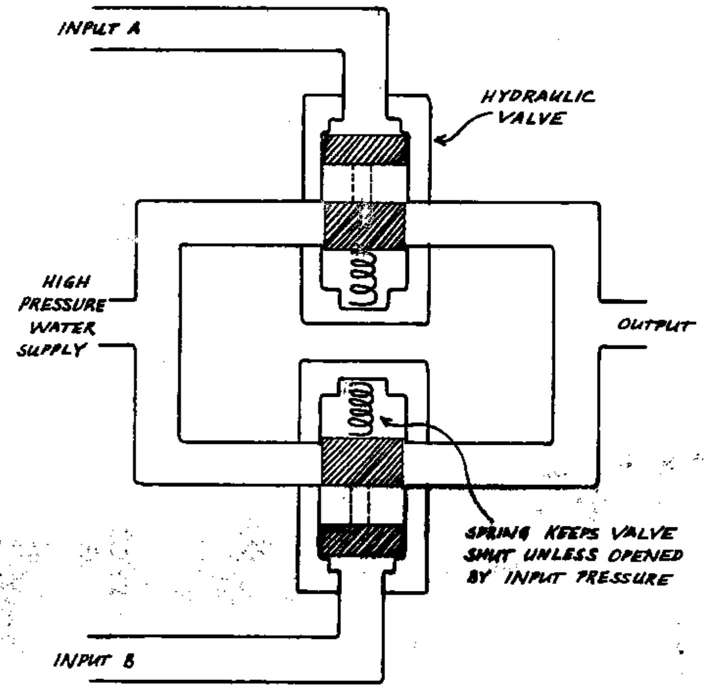
INPUT A
HYDRAULIC VALVE
HIGH PRESSURE WATER SUPPLY
OUTPUT
SPRING KEEPS VALVE SHUT UNLESS OPENED BY INPUT PRESSURE
INPUT B

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 **Ands** and **Ors** 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.