CHAPTER 2

Combinational Logic Design

2.1 INTRODUCTION

A digital circuit whose output value depends solely on the *present combination of the circuit inputs' values* is called a ***combinational circuit***. For example, Figure 2.1(a) shows a doorbell system; if the button is pressed, the bell sounds. Figure 2.1(b) shows a motion-in-the-dark lamp; if there is motion and it is dark, the lamp turns on. In contrast, in a *sequential* circuit, the output value depends on the present *and past* input values, such as the toggle lamp in Figure 2.1(c); pressing the button turns the lamp on, which stays on even after the button is released. Pressing the button again would turn the lamp off.

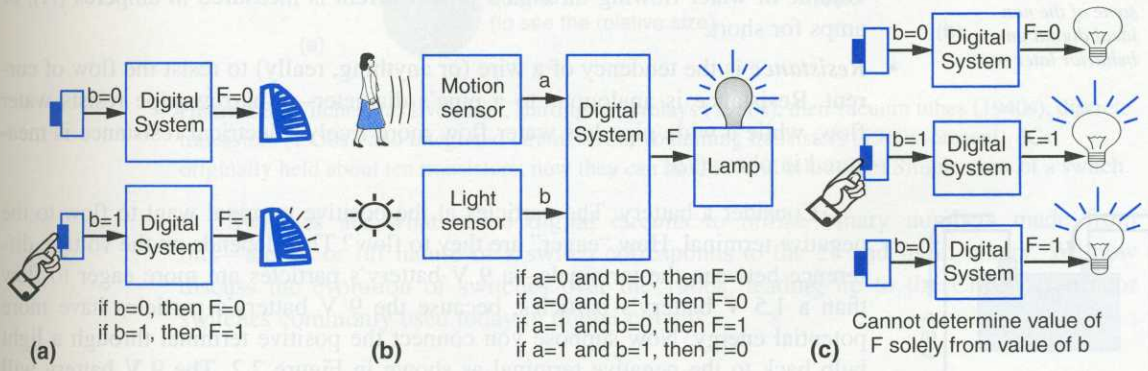


Figure 2.1 Combinational systems like (a) and (b) are such that the output value can be determined solely from the present input values. In contrast, a sequential system as in (c) has some internal "memory" that also impacts the output.

Combinational circuits are a basic class of digital circuits that are able to implement some systems, but that importantly make up part of more complex circuits. This chapter introduces the basic building blocks of combinational circuits, known as logic gates, and also introduces a form of mathematics known as Boolean algebra that is useful for working with combinational circuits. The chapter will describe a process for designing basic combinational circuits, wherein a designer first captures the desired circuit behavior, and then converts that behavior into a circuit of logic gates. Chapter 3 will introduce sequential circuits and a process for their design, and Chapter 4 will describe more complex combinational components.

▶ 2.2 SWITCHES

Electronic switches form the basis of all digital circuits, so make a good starting point for the discussion of digital circuits. You use a type of switch, a light switch, whenever you turn lights on or off. To understand a switch, it helps to understand some basic electronics.

Electronics 101

You're probably familiar with the idea of electrons, or let's just say charged particles, flowing through wires and causing lights to illuminate or stereos to blast music. An analogous situation is water flowing through pipes and causing sprinklers to pop up or turbines to turn. We now describe three basic electrical terms:

Although understanding the electronics underlying digital logic gates is optional, many people find that a basic understanding satisfies much curiosity and also helps in understanding some of the non-ideal digital gate behavior later on.

- ***Voltage*** is the difference in electric potential between two points. Voltage is measured in volts (V). Convention says that the earth, or *ground*, is 0 V. Informally, voltage tells us how "eager" the charged particles on one side of a wire are to get to ground (or any lower voltage) on the wire's other side. Voltage is analogous to the pressure of water trying to flow through a pipe—water under higher pressure is more eager to flow, even if the water can't actually flow, perhaps because of a closed faucet.
- ***Current*** is a measure of the flow of charged particles. Informally, current indicates the rate that particles are actually flowing. Current is analogous to the volume of water flowing through a pipe. Current is measured in amperes (A), or amps for short.
- ***Resistance*** is the tendency of a wire (or anything, really) to resist the flow of current. Resistance is analogous to a pipe's diameter—a narrow pipe resists water flow, while a wide pipe lets water flow more freely. Electrical resistance is measured in ohms (Ω).

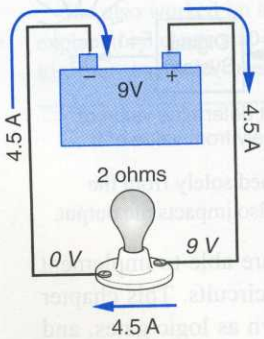


Figure 2.2 9V battery connected to a light bulb.

Consider a battery. The particles at the positive terminal want to flow to the negative terminal. How "eager" are they to flow? That depends on the voltage difference between the terminals—a 9 V battery's particles are more eager to flow than a 1.5 V battery's particles, because the 9 V battery's particles have more potential energy. Now suppose you connect the positive terminal through a light bulb back to the negative terminal as shown in Figure 2.2. The 9 V battery will result in more current flowing, and thus a brighter lit light, than the 1.5 V battery. Precisely how much current will flow is determined using the equation

$$V = IR \text{ (known as Ohm's Law)}$$

where V is voltage, I is current, and R is resistance (in this case, of the light bulb). So if the resistance were 2 ohms, a 9 V battery would result in 4.5 A (since $9 = I*2$) of current, while a 1.5 V battery would result in 0.75 A.

Rewriting the equation as $I = V/R$ might make more intuitive sense—the higher the voltage, the more current; the higher the resistance, the less current. Ohm's Law is perhaps the most fundamental equation in electronics.

The Amazing Shrinking Switch

Now back to switches. Figure 2.3(b) shows that a switch has three parts—let's call them the source input, the output, and the control input. The source input has higher voltage than the output, so current tries to flow from the source input through the switch to the output. The whole purpose of a switch is to block that current when the control sets the switch "off," and to allow that current to flow when control sets the switch "on." For example, when you flip a light switch up to turn the switch on, the switch causes the source input wire to physically touch the output wire, so current flows. When you flip the switch down to turn the switch off, the switch physically separates the source input from the output. In our water analogy, the control input is like a faucet valve that determines whether water flows through a pipe.

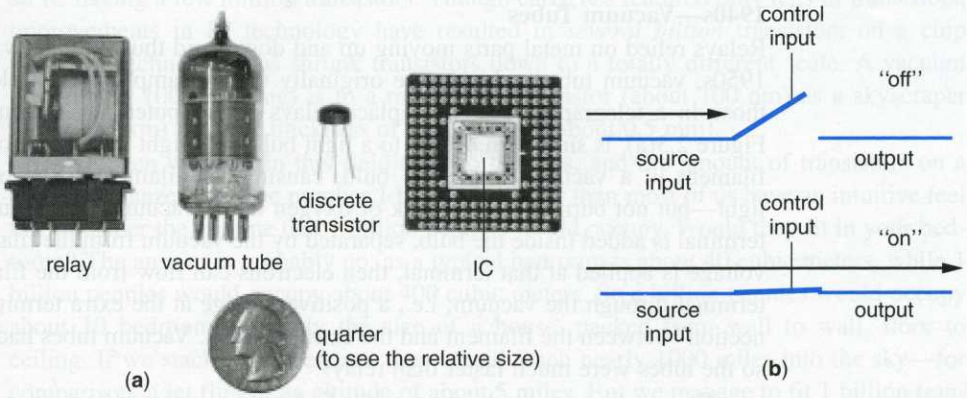


Figure 2.3 Switches: (a) Evolution, starting with relays (1930s), then vacuum tubes (1940s), discrete transistors (1950s), and integrated circuits (ICs) containing transistors (1960s–present). ICs originally held about ten transistors; now they can hold several billion. (b) Simple view of a switch.

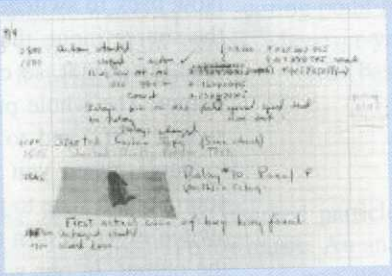
Switches are what cause digital circuits to utilize binary numbers made from bits—the on or off nature of a switch corresponds to the `1`s and `0`s in binary. We now discuss the evolution of switches over the 1900s, leading up to the CMOS transistor switches commonly used today in digital circuits.

1930s—Relays

Engineers in the 1930s tried to devise ways to compute using electronically controlled switches—switches whose control input was another voltage. One such switch, an electromagnetic relay like that in Figure 2.3(a), was already being used by the telephone industry for switching telephone calls. A relay has a control input that is a type of magnet, which becomes magnetized when the control has a positive voltage. In one type of relay, that magnet pulls a piece of metal down, resulting in a connection from the source input to the output—akin to pulling down a drawbridge to connect one road to another. When the control input returns to 0 V, the piece of metal returns up again (perhaps pushed by a small spring), disconnecting the source input from the output. In telephone systems, relays enabled calls to be routed from one phone to another, without the need for those nice human operators that previously would manually connect one phone's line to another.

▶ *"DEBUGGING"*

In 1945, a moth got stuck in one of the relays of the Mark II computer at Harvard. To get the computer working properly again, technicians found and removed the bug. Though the term "bug" had been used for decades before by engineers to indicate a defect in mechanical or electrical equipment, the removal of that moth in 1945 is considered to be the origin of the term "debugging" in computer programming. Technicians taped that moth to their written log (shown in the picture to the right), and that moth is now on display at the National Museum of American History in Washington, D.C.



Jack Kilby of Texas Instrum and Robert N of Fairchild Semiconducto are often cred with each hav independently invented the

1940s—Vacuum Tubes

Relays relied on metal parts moving up and down, and thus were slow. In the 1940s and 1950s, vacuum tubes, which were originally used to amplify weak electric signals like those in a telegraph, began to replace relays in computers. A vacuum tube, shown in Figure 2.3(a), is similar in design to a light bulb. In a light bulb, electrons flow through a filament in a vacuum inside the bulb, causing the filament to become hot and emit light—but not burn due to the lack of oxygen in the vacuum. In a vacuum tube, an extra terminal is added inside the bulb, separated by the vacuum from the filament. If a positive voltage is applied at that terminal, then electrons can flow from the filament to the extra terminal through the vacuum; i.e., a positive voltage at the extra terminal causes a "connection" between the filament and the extra terminal. Vacuum tubes had no moving parts, so the tubes were much faster than relays.

The machine said to be the world's first general-purpose computer, the ENIAC (electronic numerical integrator and computer), was completed in the U.S. in 1946. ENIAC contained about 18,000 vacuum tubes and 1500 relays, weighed over 30 tons, was 100 feet long and 8 feet high (so it likely would not fit in any room of your house, unless you have an absurdly big house), and consumed 174,000 watts of power. Imagine the heat generated by a room full of 1740 100-watt light bulbs. That's hot. For all that, ENIAC could compute about 5000 operations per second—compare that to the billions of operations per second of today's personal computers, and even the tens of millions of computations per second by a handheld cell phone.

Although vacuum tubes were faster than relays, they consumed a lot of power, generated a lot of heat, and failed frequently.

Vacuum tubes were commonplace in many electronic appliances in the 1960s and 1970s. I remember taking trips to the store with my dad in the early 1970s to buy replacement tubes for our television set. Vacuum tubes still live today in a few electronic devices. One place you might still find tubes is in electric guitar amplifiers, where the tube's unique-sounding audio amplification is still demanded by rock guitar enthusiasts who want their versions of classic rock songs to sound just like the originals.

1950s—Discrete Transistors

The invention of the transistor in 1947, credited to William Shockley, John Bardeen, and Walter Brattain of Bell Laboratories (the research arm of AT&T), resulted in smaller and lower-power computers. A solid-state (discrete) transistor, shown in Figure 2.3(a), uses a

▶ *A S*

We now start of revoluti century, on TV, implicat

small piece of silicon, "doped" with some extra materials, to create a switch. Since these switches used "solid" materials rather than a vacuum or even moving parts in a relay, they were commonly referred to as solid-state transistors. Solid-state transistors were smaller, cheaper, faster, and more reliable than tubes, and became the dominant computer switch in the 1950s and 1960s.

1960s—Integrated Circuits

Jack Kilby of Texas Instruments and Robert Noyce of Fairchild Semiconductors are often credited with each having independently invented the IC.

The invention of the ***integrated circuit (IC)*** in 1958 revolutionized computing. An IC, a.k.a. a chip, packs numerous tiny transistors on a fingernail-sized piece of silicon. So instead of 10 transistors requiring 10 discrete electronic components on a board, 10 transistors can be implemented on one component, the chip. Figure 2.3(a) shows a picture of an IC having a few million transistors. Though early ICs featured only tens of transistors, improvements in IC technology have resulted in *several billion* transistors on a chip today. IC technology has shrunk transistors down to a totally different scale. A vacuum tube (about 100 mm long) is to a modern IC transistor (about 100 nm) as a skyscraper (about 0.5 km) is to the thickness of a credit card (about 0.5 mm).

I've been working in this field for two decades, and the amount of transistors on a chip still amazes me. The number 1 billion is bigger than most of us have an intuitive feel for. Consider the volume that 1 billion pennies would occupy. Would they fit in your bedroom? The answer is probably no, as a typical bedroom is about 40 cubic meters, while 1 billion pennies would occupy about 400 cubic meters. So a billion pennies would occupy about 10 bedrooms, roughly the size of a house, packed from wall to wall, floor to ceiling. If we stacked the pennies, they would reach nearly 1000 miles into the sky—for comparison, a jet flies at an altitude of about 5 miles. But we manage to fit 1 billion transistors onto silicon chips of just a few square centimeters. Truly amazing. The wires that connect all those transistors on a chip, if straightened into one straight wire, would be several miles long.

IC transistors are smaller, more reliable, faster, and less power-hungry than discrete transistors. IC transistors are now by far the most commonly used switch in computing.

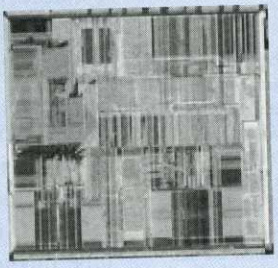
ICs of the early 1960s could hold tens of transistors, and are known today as small-scale integration (***SSI***). As transistor sizes shrank, in the late 1960s and early 1970s, ICs could hold hundreds of transistors, known as medium-scale integration (***MSI***). The 1970s saw the development of large-scale integration (***LSI***) ICs with thousands of transistors, while very large-scale integration (***VLSI***) chips evolved in the 1980s. Since then, ICs have continued to increase in their capacity, to several billion transistors. To calibrate your understanding of this number, a processor in a 2009 laptop computer, like an Intel

▶ *A SIGNIFICANT INVENTION*

We now know that the invention of the transistor was the start of the amazing computation and communication revolutions that occurred in the latter half of the 20th century, enabling us today to do things like see the world on TV, surf the web, and talk on cell phones. But the implications of the transistor were not known by most people at the time of its invention. Newspapers did not headline the news, and most stories that did appear predicted simply that transistors would improve things like radios and hearing aids. One may wonder what recently invented but unnoticed technology might significantly change the world once again.

▶ *HOW TRANSISTORS ARE MADE SO SMALL—USING PHOTOGRAPHIC METHODS*

If you took a pencil and made the smallest dot that you could on a sheet of paper, that dot's area would hold many thousands of transistors on a modern silicon chip. How can chip makers create such tiny transistors? The key lies in photographic methods. Chip makers lay a special chemical onto the chip—special because the chemical changes when exposed to light. Chip makers then shine light through a lens that focuses the light down to extremely small regions on the chip—similar to how a microscope's lens lets us see tiny things by focusing light, but in reverse. The chemical in the small illuminated region changes, and then a solvent washes away the chemical—but some regions stay because of the light that changed that region. Those remaining regions form parts of transistors. Repeating this process over and over again, with different chemicals at different steps, results not only in transistors, but also wires connecting the transistors, and insulators preventing crossing wires from touching.



Photograph of a Pentium processor's silicon chip, having millions of transistors. Actual size is about 1 cm each side.

Atom or Celeron processor, requires only about 50 million transistors, and the processor in a cell phone, like an ARM processor, may have only a few million transistors. Many of today's high-end chips, like chips inside Internet routers, contain tens or hundreds of such microprocessors, and can conceivably contain thousands of even smaller microprocessors (or just a few very big microprocessors).

IC density has been doubling roughly every 18 months since the 1960s. The doubling of IC density every 18 months is widely known as ***Moore's Law***, named after Gordon Moore, a co-founder of Intel Corporation, who made predictions back in 1965 that the number of components per IC would double every year or so. At some point, chip makers won't be able to shrink transistors any further. After all, the transistor has to at least be wide enough to let electrons pass through. People have been predicting the end of Moore's Law for two decades now, but transistors keep shrinking, though in 2009 many observers noted a slowdown.

Not only do smaller transistors and wires provide for more functionality in a chip, but they also provide for faster circuits, in part because electrons need not travel as far to get from one transistor to the next. This increased speed is the main reason why personal computer clock speeds have improved so drastically over the past few decades, from kilohertz frequencies in the 1970s to gigahertz frequencies in the 2000s.

▶ 2.3 THE CMOS TRANSISTOR

The most popular type of IC transistor is the CMOS transistor. A detailed explanation of how a CMOS transistor works is beyond the scope of this book, but nevertheless a simplified explanation may satisfy much curiosity.

A chip is made primarily from the element silicon. A chip, also known as an integrated circuit, or IC, is typically about the size of a fingernail. Even if you open up a computer or other chip-based device, you would not actually see the silicon chip, since chips are actually inside a larger, usually black, protective package. But you certainly

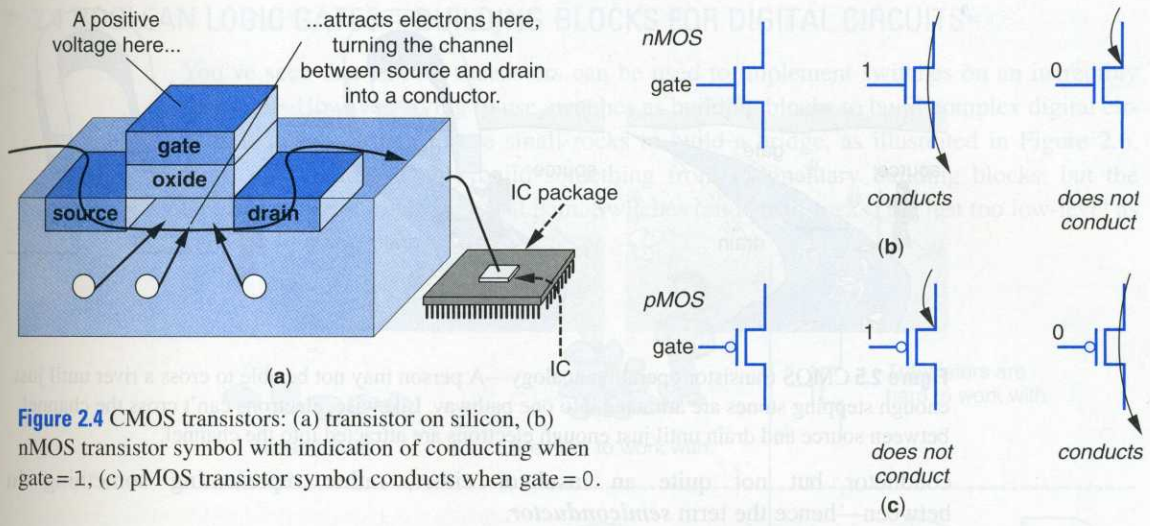


Figure 2.4 CMOS transistors: (a) transistor on silicon, (b) nMOS transistor symbol with indication of conducting when gate = 1, (c) pMOS transistor symbol conducts when gate = 0.

should be able to see those black packages, mounted on a printed circuit board, inside a variety of household electronic devices.

Figure 2.4(a) illustrates a cross section of a tiny part of silicon chip, showing the side view of one type of CMOS transistor—an nMOS transistor. The transistor has the three parts of a switch: (1) the ***source*** input; (2) the output, which is called the ***drain***, perhaps because electric particles flow to the drain like water flows to a drain; and (3) the control input, which is called the ***gate***, perhaps because the gate blocks the current flow like a gate blocks a dog from escaping the backyard. A chip maker creates the source and drain by injecting certain elements into the silicon. The region between the source and drain is the ***channel***. The gate is separated from the channel by an insulation layer made from silicon dioxide, known as ***oxide***. Figure 2.4(b) shows the electronic symbol of an nMOS transistor.

Suppose the drain was connected to a small positive voltage (modern technologies use about 1 or 2 V) known as the "power supply," and the source was connected through a resistor to ground. Current would thus try to flow from drain to source, and on to ground. (Unfortunately, convention is that current flow is defined using positive charge, even though negatively charged electrons are actually flowing—so notice we say current flows from drain to source, even though electrons flow from source to drain.) However, the silicon channel between source and drain is not normally a ***conductor***, which is a substance that allows electrons to flow readily. Instead, the channel is normally an ***insulator***, which is a substance that resists the flow of electrons. Think of an insulator as an extremely large resistance. Since $I = V/R$, then I will essentially be 0. The switch is off.

A feature of silicon is that the channel can be changed from an insulator to a conductor just *by applying a small positive voltage to the gate*. That gate voltage doesn't result in current flow from the gate to channel, because the insulating oxide layer between the gate and the channel blocks such flow. But that gate voltage does create a positive electric field that passes through the oxide and attracts electrons, which have a negative charge, from the larger silicon region into the channel region—akin to how you can move paper clips on a tabletop by moving a magnet under the table, whose magnetic field passes through the table. When enough electrons gather into the channel, the channel suddenly becomes a conductor. A conductor has extremely low resistance, so current flows almost freely between drain and source. The switch is now on. Thus, silicon is not quite a

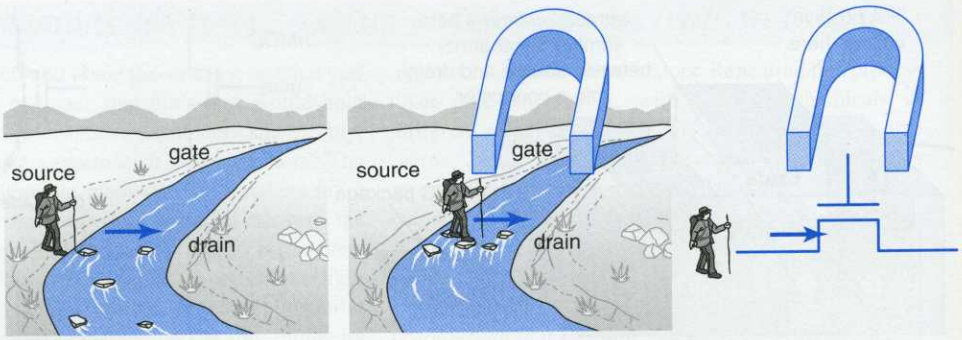


Figure 2.5 CMOS transistor operation analogy—A person may not be able to cross a river until just enough stepping stones are attracted into one pathway. Likewise, electrons can't cross the channel between source and drain until just enough electrons are attracted into the channel.

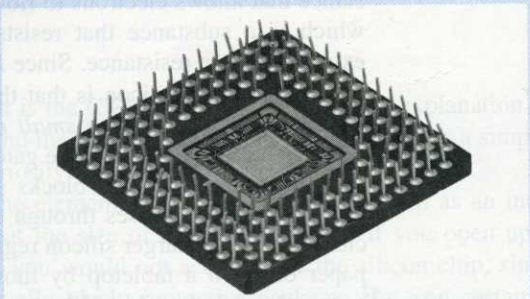
conductor but not quite an insulator either, rather representing something in between—hence the term ***semiconductor***.

An analogy to the current trying to cross the channel is a person trying to cross a river. Normally, the river might not have enough stepping stones for the person to to walk across. But if we could attract stones from other parts of the river into one pathway (the channel), the person could easily walk across the river (Figure 2.5).

nMOS is one type of CMOS transistor. The other type is pMOS, which is similar, except that the channel has the opposite functionality—the channel is a conductor normally, and *doesn't* conduct when the gate has a positive voltage. Figure 2.4(c) shows the electronic symbol for a pMOS transistor. The use of these two "complementary" types of transistors is where the C comes from in CMOS. The MOS stands for metal oxide semiconductor; the reasons for that name should be clear from above, as MOS transistors use metal (to connect transistors), oxide (to insulate), and semiconductor material.

▶ *SILICON VALLEY, AND THE SHAPE OF SILICON*

Silicon Valley is not a city, but refers to an area in Northern California, about an hour south of San Francisco, that includes several cities like San Jose, Mountain View, Sunnyvale, Milpitas, Palo Alto, and others. The area is heavily populated by computer and other high-technology companies, and to a large extent is the result of Stanford University's (located in Palo Alto) efforts to attract and create such companies. **What shape is silicon?** Once, as my plane arrived in Silicon Valley, the person next to me asked "What shape is a silicon, anyway?" I realized he thought silicon was a type of polygon, like a pentagon or an octagon. Well, the words do sound similar. Silicon is not a shape, but an element, like carbon or aluminum or silver. Silicon has an atomic number of 14, has a chemical symbol of "Si," and is the second most abundant element (next to oxygen) in the earth's crust, found in items like sand and clay. Silicon is used to make mirrors and glass, in addition to chips. In fact, to the naked eye, a silicon chip actually looks like a small mirror.



A chip package with its chip cover removed—you can see the mirror-like silicon chip in the center.

▶ 2.4 BOOLEAN LOGIC GATES—BUILDING BLOCKS FOR DIGITAL CIRCUITS

You've seen that CMOS transistors can be used to implement switches on an incredibly tiny scale. However, trying to use switches as building blocks to build complex digital circuits is akin to trying to use small rocks to build a bridge, as illustrated in Figure 2.6. Sure, you could probably build something from rudimentary building blocks, but the building process would be a real pain. Switches (and small rocks) are just too low-level as building blocks.

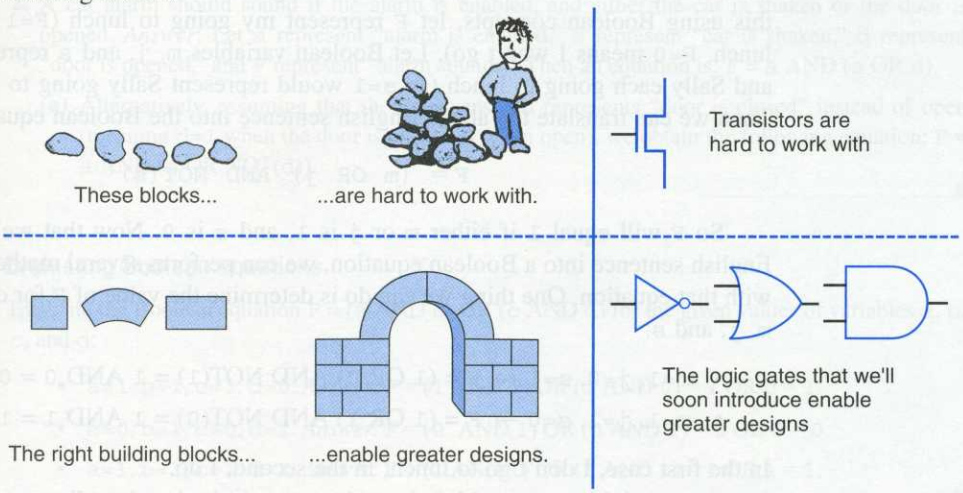


Figure 2.6 Having the right building blocks can make all the difference when building things.

Boolean Algebra and its Relation to Digital Circuits

Fortunately, Boolean logic gates aid the design task by representing digital circuit building blocks that are much easier to work with than switches. Boolean logic was developed in the mid-1800s by the mathematician George Boole, not to build digital circuits (which weren't even a glimmer in anyone's eye back then), but rather as a scheme for using algebraic methods to formalize human logic and thought.

Algebra is a branch of mathematics that uses letters or symbols to represent numbers or values, where those letters/symbols can be combined according to a set of known rules. ***Boolean algebra*** uses variables (known as Boolean variables) whose values can only be `1` or `0` (representing true or false, respectively). Boolean algebra's operators, like AND, OR, and NOT, operate on such variables and return `1` or `0`. So we might declare Boolean variables `x`, `y`, and `z`, and then say that `z = x` OR `y`, meaning `z` will equal `1` if `x` is `1` or `y` is `1`, else `z` will equal `0`. Contrast Boolean algebra with the regular algebra you're familiar with, perhaps from high school, in which variable values could be integers (for example), and operators could be addition, subtraction, and multiplication.

The basic Boolean operators are AND, OR, and NOT:

- AND returns `1` if *both* its operands are `1`. So the result of `a` AND `b` is `1` if both `a=1` and `b=1`, otherwise the result is `0`.

"ab=01" is shorthand for "a=0, b=1."

- OR returns 1 if *either or both* of its operands are 1. So the result of a OR b is 1 in any of the following cases: ab=01, ab=10, ab=11. Thus, the only time a OR b is 0 is when ab=00.
- NOT returns 1 if its operand is 0. So NOT(a) returns 1 if a is 0, and returns 0 if a is 1.

We use Boolean logic operators frequently in everyday thought, such as in the statement "I'll go to lunch if Mary goes OR John goes, AND Sally does not go." To represent this using Boolean concepts, let F represent my going to lunch (F=1 means I'll go to lunch, F=0 means I won't go). Let Boolean variables m, j, and s represent Mary, John, and Sally each going to lunch (so s=1 would represent Sally going to lunch, else s=0). Then we can translate the above English sentence into the Boolean equation:

```
F = (m OR j) AND NOT(s)
```

So F will equal 1 if either m or j is 1, and s is 0. Now that we've translated the English sentence into a Boolean equation, we can perform several mathematical activities with that equation. One thing we can do is determine the value of F for different values of m, j, and s:

- m=1, j=0, s=1 → F = (1 OR 0) AND NOT(1) = 1 AND 0 = 0
- m=1, j=1, s=0 → F = (1 OR 1) AND NOT(0) = 1 AND 1 = 1

In the first case, I don't go to lunch; in the second, I do.

A second thing we could do is apply some algebraic rules (discussed later) to modify the original equation to the equivalent equation:

```
F = (m and NOT(s)) OR (j and NOT(s))
```

In other words, I'll go to lunch if Mary goes AND Sally does not go, OR if John goes AND Sally does not go. That statement, as different as it may look from the earlier statement, is nevertheless equivalent to the earlier statement.

A third thing we could do is formally prove properties about the equation. For example, we could prove that if Sally goes to lunch (s=1), then I don't go to lunch (F=0) no matter who else goes, using the equation:

```
F = (m OR j) AND NOT(1) = (m OR j) AND 0 = 0
```

No matter what the values of m and j, F will equal 0.

Noting all the mathematical activities we can do using Boolean equations, you can start to see what Boole was trying to accomplish in formalizing human reasoning.

Example 2.1 Converting a problem statement to a Boolean equation

Convert the following problem statements to Boolean equations using AND, OR, and NOT operators. F should equal 1 only if:

1. a is 1 and b is 1. *Answer:* F = a AND b
2. either of a or b is 1. *Answer:* F = a OR b

3. both a and b are not 0. *Answer:* F = NOT(a) AND NOT(b)
4. a is 1 and b is 0. *Answer*: F = a AND NOT(b)

Convert the following English problem statements to Boolean equations:

1. A fire sprinkler system should spray water if high heat is sensed and the system is set to enabled. *Answer*: Let Boolean variable h represent "high heat is sensed," e represent "enabled," and F represent "spraying water." Then an equation is: F = h AND e.
2. A car alarm should sound if the alarm is enabled, and either the car is shaken or the door is opened. *Answer*: Let a represent "alarm is enabled," s represent "car is shaken," d represent "door is opened," and F represent "alarm sounds." Then an equation is: F = a AND (s OR d).
   (a) Alternatively, assuming that the door sensor d represents "door is closed" instead of open (meaning d=1 when the door is closed, 0 when open), we obtain the following equation: F = a AND (s OR NOT(d)).

Example 2.2 Evaluating Boolean equations

Evaluate the Boolean equation F = (a AND b) OR (c AND d) for the given values of variables a, b, c, and d:

- a=1, b=1, c=1, d=0. *Answer*: F = (1 AND 1) OR (1 AND 0) = 1 OR 0 = 1.
- a=0, b=1, c=0, d=1. *Answer*: F = (0 AND 1) OR (0 AND 1) = 0 OR 0 = 0.
- a=1, b=1, c=1, d=1. *Answer*: F = (1 AND 1) OR (1 AND 1) = 1 OR 1 = 1.

One might now be wondering what Boolean algebra has to do with building circuits using switches. In 1938, an MIT graduate student named Claude Shannon wrote a paper (based on his master's thesis) describing how Boolean algebra could be applied to switch-based circuits, by showing that "on" switches could be treated as a 1 (or true), and "off" switches as a 0 (or false), by connecting those switches in a certain way (Figure 2.7). His thesis is widely considered as the seed that developed into modern digital design. Since Boolean algebra comes with a rich set of axioms, theorems, postulates, and rules, we can use all those things to manipulate digital circuits using algebra. In other words:

Shannon, by the way, is also considered the father of information theory, due to his later work on digital communication.

We can build circuits by doing math.

That's an extremely powerful concept. We'll be building circuits by doing math throughout this chapter.

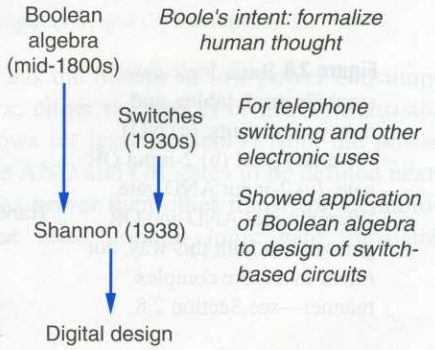


Figure 2.7 Shannon applied Boolean algebra to switch-based circuits, providing a formal basis to digital circuit design.

AND, OR, & NOT Gates

Earlier we said a "gate" was the switch control input of a CMOS transistor, but now we're talking about "logic gates." In an unfortunate naming similarity, the same word (gate) refers to two different things. Don't worry, though; after the next section, we'll be using the word "gate" to refer only to a logic gate.

To build digital circuits that can be manipulated using Boolean algebra, we first implement the Boolean operators AND, OR, and NOT using small circuits of switches, and call those circuits Boolean ***logic gates***. Then, *we forget about switches*, and instead use Boolean logic gates as building blocks. Suddenly, the power of Boolean algebra is at our fingertips for designing more complex circuits! This is like first assembling rocks into three shapes of bricks, and then building structures like a bridge from those bricks, as in Figure 2.6. Trying to build a bridge from small rocks is harder than building a bridge from the three basic brick shapes. Likewise, trying to build a motion-in-the-dark circuit (or any circuit) from switches is harder than building a circuit from Boolean logic gates.

Let's first implement Boolean logic gates using CMOS transistors, shown in Figure 2.8 and soon to be described, and then a later section will show how Boolean algebra helps build better circuits. You really don't *have* to understand the underlying transistor implementations of logic gates to learn the digital design methods in the rest of this book, and in fact many textbooks omit the transistor discussion entirely. But an understanding of the underlying transistor implementation can be quite satisfying to a student, leaving no "mysteries." Such an understanding can also help in understanding the nonideal behavior of logic gates that one may later have to learn to deal with in digital design.

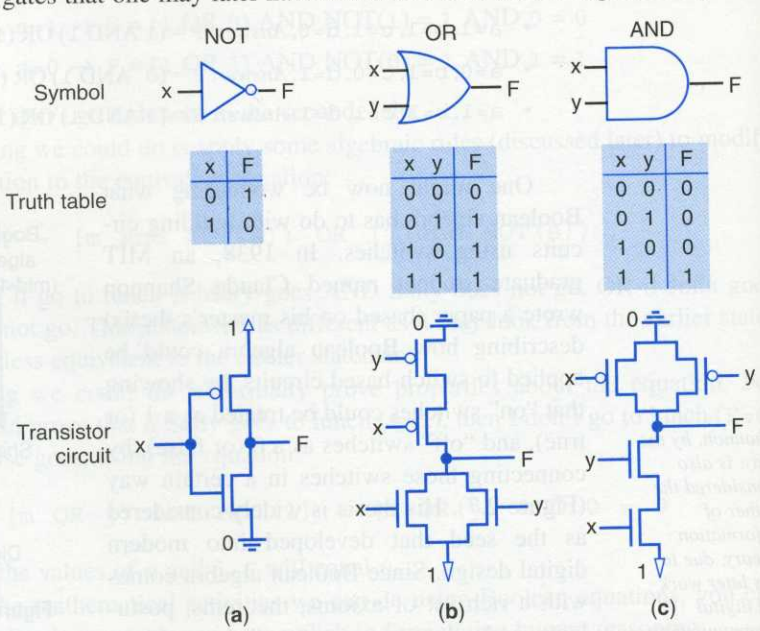


Figure 2.8 Basic logic gates' symbols, truth tables, and transistor circuits: (a) NOT (inverter) gate, (b) 2-input OR gate, (c) 2-input AND gate. Warning: real AND and OR gates *aren't* built this way, but rather in a more complex manner—see Section 2.8.

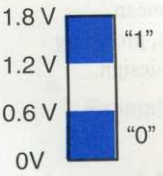
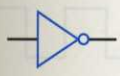


Figure 2.9 Sample voltage ranges for 1s and 0s.

`1` will represent the power supply's voltage level, which today is usually around 1 V to 2 V for CMOS technology (e.g., 0.9 V, or 1.6 V). `0` will represent ground. Note that any two symbols or words could be used rather than `1` and `0` to represent power and ground voltage levels. For example, alternatives could be `true` and `false`, or `H` and `L`. Furthermore, `1` and `0` typically each represents a voltage *range*, such as `1` representing any voltage between 1.2 V to 1.8 V and `0` representing between 0 V and 0.6 V, as in Figure 2.9.

NOT Gate



A ***NOT gate*** has an input x and an output F. F should always be the opposite, or inverse, of x—for this reason, a NOT gate is commonly called an ***inverter***. We can build a NOT gate using one pMOS and one nMOS transistor, as shown in Figure 2.8(a). The triangle at the top of the transistor circuit represents the positive voltage of the power supply, which we represent as 1. The series of lines at the bottom of the circuit represents ground, which we represent as 0. When the input x is 0, the pMOS transistor will conduct, but the nMOS will not, as shown in (a). In that case, we can think of the circuit as a wire from 1 to F, so when x = 0, then F = 1. On the other hand, when x is 1, the nMOS will conduct, but the pMOS will not, as shown in (b). In that case, we can think of the circuit as a wire from 0 to F, so when x=1, then F=0. The table in Figure 2.8, called a ***truth table***, summarizes the NOT gate's behavior by listing the gate's output for every possible input.

Figure 2.10 shows a timing diagram for an inverter (See Section 1.3 for an introduction to timing diagrams.) When the input is 0, the output is 1; when the input is 1, the output is 0.

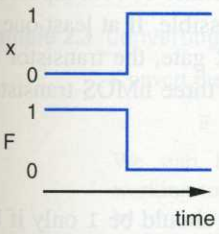


Figure 2.10 Inverter timing diagram.

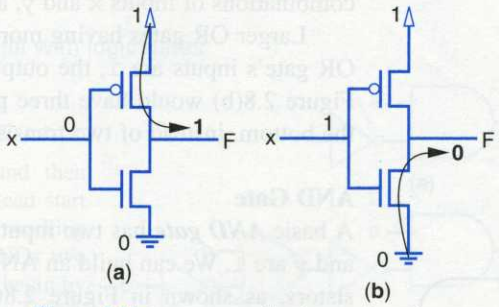
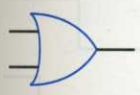


Figure 2.11 Inverter conduction paths when: (a) the input is 0, and (b) the input is 1.

Combining pMOS and nMOS in this way has the benefit of low power consumption. Figure 2.11 shows that for any value of x, either the pMOS or nMOS transistor will be nonconducting. Thus, current never flows (at least in theory) from the power source to ground, which will also be true for the AND and OR gates to be defined next. This feature makes CMOS circuits consume less power than other transistor technologies, and partly explains why CMOS is the most popular logic gate transistor technology today.

OR Gate



A basic ***OR gate*** has two inputs x and y and an output F. F should be 1 only if at least one of x or y is 1. We can build an OR gate using two pMOS transistors and two nMOS transistors, as shown in Figure 2.8(b). (Section 2.8 explains that OR gates are actually built in a more complex manner.) If at least one of x or y is 1, then a connection occurs from 1 to F, but no connection from 0 to F, so F is 1, as shown in Figure 2.12(a). If both x and y are 0, then a connection occurs from 0 to F, but no connection from 1 to F, so F is 0, as shown in Figure 2.12(b). The truth table for the OR gate appears in Figure 2.8(b).

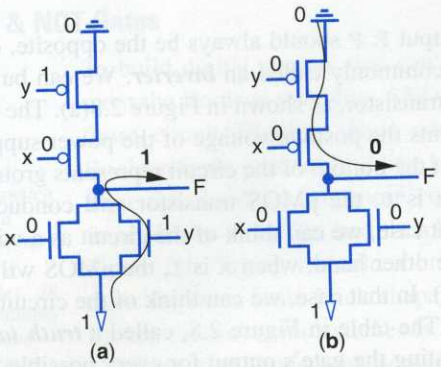


Figure 2.12 OR gate conduction paths: (a)when one input is 1, and (b) when both inputs are 0.

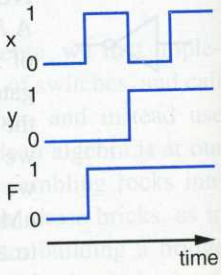
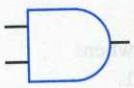


Figure 2.13 OR gate timing diagram.

Figure 2.13 shows a timing diagram for an OR gate. The table lists all possible value combinations of inputs x and y, and shows that F will be 1 if either or both inputs is a 1.

Larger OR gates having more than two inputs are also possible. If at least one of the OR gate's inputs are 1, the output is 1. For a three-input OR gate, the transistor circuit Figure 2.8(b) would have three pMOS transistors on top and three nMOS transistors on the bottom, instead of two transistors of each kind.

AND Gate



A basic ***AND gate*** has two inputs x and y and an output F. F should be 1 only if both x and y are 1. We can build an AND gate using two pMOS transistors and two nMOS transistors, as shown in Figure 2.8(c) (again, Section 2.8 will show that AND gates are actually built in a more complex manner). If both x and y are 1, then a connection occurs from 1 to F, but no connection from ground to F, so F is 1, as shown in Figure 2.14(a). If at least one of x or y is 0, then a connection occurs from 0 to F, but no connection from 1 to F, so F is 0, as shown in Figure 2.14(b). The truth table for the AND gate appears in Figure 2.8(c).

Figure 2.15 shows a timing diagram for an AND gate. We set inputs x and y to each possible combination of values, and show that F will be 1 only if both inputs are a 1.

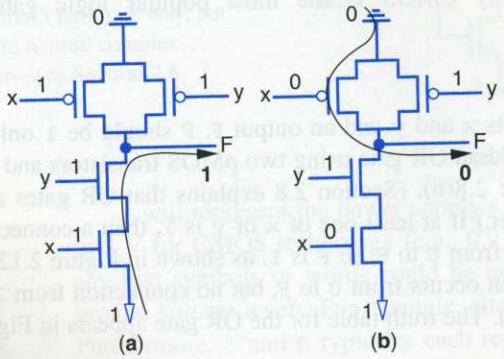


Figure 2.14 AND gate conduction paths: (a) when all inputs are 1, and (b) when any input is 0.

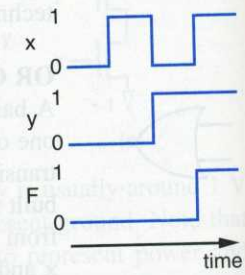


Figure 2.15 AND gate timing diagram.

Larger AND gates having more than two inputs are also possible. The output is 1 only if all the inputs are 1. For a three-input AND gate, the transistor circuit in Figure 2.8(b) would have three pMOS transistors on top and three nMOS transistors on the bottom, instead of two transistors of each kind.

Building Simple Circuits Using Gates

Having built logic gate building blocks from transistors, we now show how to build useful circuits from those building blocks. Recall the digital system example of Chapter 1, the motion-in-the-dark detector. a=1 meant motion, and b=0 meant dark, so we wanted F = a AND NOT(b). We can connect b through an inverter to get NOT(b), and connect the result along with a into an AND gate, whose output is F. The resulting circuit appears in Chapter 1, shown again in Figure 2.16 for convenience. We now provide more examples.

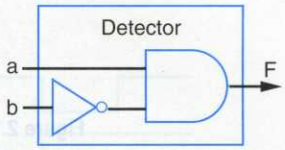


Figure 2.16 Motion-in-the-dark detector circuit.

Example 2.3 Converting a Boolean equation to a circuit with logic gates

Convent the following equation to a circuit:

```
F = a AND NOT( b OR NOT(c) )
```

We start by drawing F on the right, and then working toward the inputs. (We could instead start by drawing the inputs on the left and working toward the output.) The equation for F ANDs two items: a, and the output of a NOT. We thus begin by drawing the circuit of Figure 2.17(a). The NOT's input comes from an OR of two items: b, and NOT(c). We complete the drawing in Figure 2.17(b) by including an OR gate and NOT gate as shown.

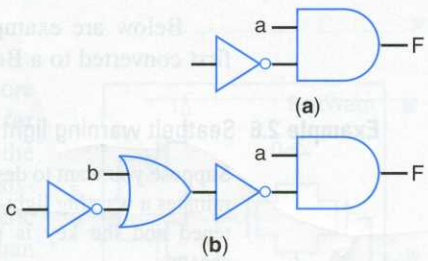


Figure 2.17 Building the circuit for F: (a) partial, (b) complete.

Example 2.4 More examples converting Boolean equations to gates

Figure 2.18 shows two more examples converting Boolean equations to circuits using logic gates. We again start from the output and work back to the inputs. The figure shows the correspondence between equation operators and gates, and the order in which we added each gate to the circuit.

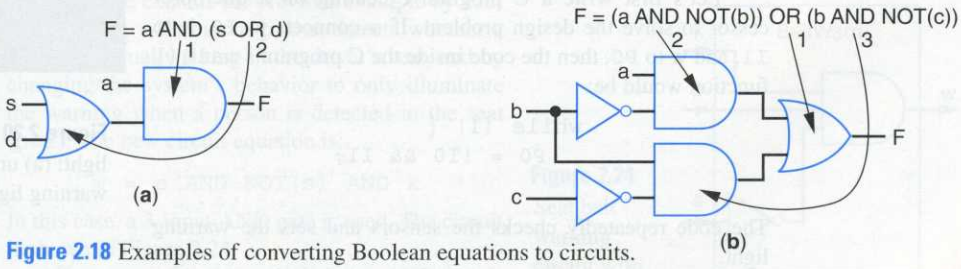


Figure 2.18 Examples of converting Boolean equations to circuits.

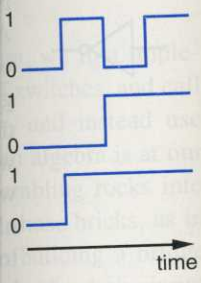


Figure 2.13 OR gate timing diagram.

sts all possible value
or both inputs is a 1.
If at least one of the
the transistor circuit
nMOS transistors on

d be 1 only if both x
and two nMOS tran-
that AND gates are
n a connection occurs
n in Figure 2.14(a). If
ut no connection from
AND gate appears in

inputs x and y to each
oth inputs are a 1.

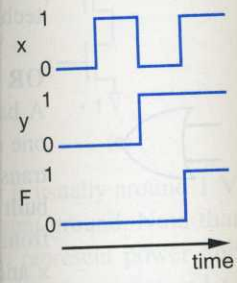


Figure 2.15 AND gate timing diagram.

Example 2.5 Using AND and OR gates with more than two inputs

Figure 2.19(a) shows an implementation of the equation `F = a AND b AND c`, using two-input AND gates. However, designers would typically instead implement such an equation using a single three-input AND gate, shown in (b). The function is the same, but the three-input AND gate uses fewer transistors, 6 rather than 4+4=8 (as well as having less delay—more on delay later). Likewise, `F = a AND b AND c AND d` would typically be implemented using a four-input AND gate.

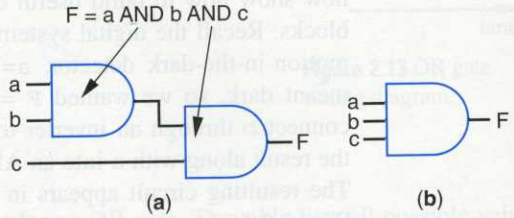


Figure 2.19 Using multiple-input AND gates: (a) using 2-input AND gates, (b) using a 3-input AND gate.

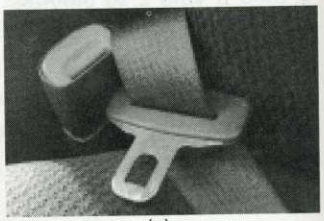
The same approach applies to OR gates. For example, `F = a OR b OR c` would typically be implemented using a single three-input OR gate.

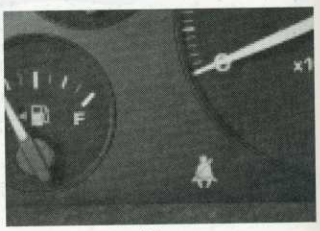
Below are examples starting from English problem descriptions, each of which is first converted to a Boolean equation, and then implemented as a circuit.

Example 2.6 Seatbelt warning light

Suppose you want to design a system for an automobile that illuminates a warning light whenever the driver's seatbelt is not fastened and the key is in the ignition. Assume the following sensors:

- a sensor with output `s` indicates whether the driver's belt is fastened (`s = 1` means the belt is fastened), and
- a sensor with output `k` indicates whether the key is in the ignition (`k = 1` means the key is in).


(a)



(b)

Figure 2.20 Seatbelt warning light: (a) unfastened, (b) warning light illuminated.

Assume the warning light has a single input `w` that illuminates the light when `w` is `1`. So the inputs to our digital system are `s` and `k`, and the output is `w`. `w` should equal 1 when both of the following occur: `s` is `0` and `k` is `1`.

Let's first write a C program executing on a microprocessor to solve the design problem. If `s` connects to `I0`, `k` to `I1`, and `w` to `P0`, then the code inside the C program's `main()` function would be:

```
while (1) {
  P0 = !I0 && I1;
}
```

The code repeatedly checks the sensors and sets the warning light.

A Boolean equation describing a circuit implementing the design is:

```
w = NOT(s) AND k
```

Using the AND and NOT logic gates introduced earlier, the design can be completed by connecting s to a NOT gate, and connecting the resulting NOT(s) and k to the inputs of a 2-input AND gate, as shown in Figure 2.21.

Figure 2.22 provides a timing diagram for the circuit. In a timing diagram, we can set the inputs to whatever values we want, but then we must draw the output line to match the circuit's function. In the figure, we set s and k to 00, then 01, then 10, then 11. The only time that the output w will be 1 is when s is 0 and k is 1, as shown in the figure.

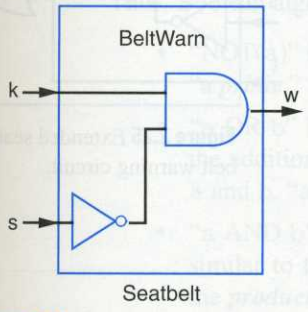


Figure 2.21 Seatbelt warning circuit.

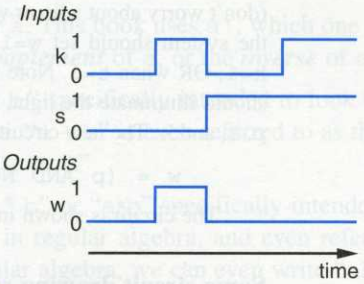


Figure 2.22 Timing diagram for seatbelt warning circuit.

We stated earlier that logic gates are more appropriate than transistors as building blocks for designing digital circuits. Note, however, that the logic gates are ultimately implemented using transistors, as shown in Figure 2.23. For C programmers, an analogy is that writing software in C is easier than writing in assembly, even though the C ultimately gets implemented using assembly. Notice how much less intuitive and less descriptive is the transistor-based circuit in Figure 2.23 than the equivalent logic gate-based circuit in Figure 2.21.

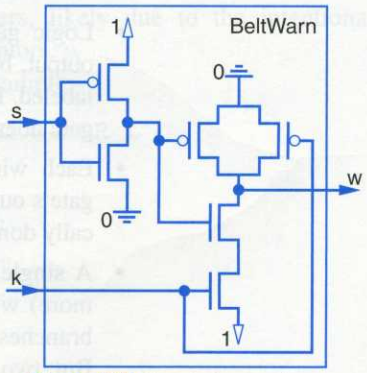


Figure 2.23 Seat belt warning circuit using transistors.

Example 2.7 Seat belt warning light with driver sensor

This example extends the previous example by adding a sensor, with output p, that detects whether a person is actually sitting in the driver's seat, and by changing the system's behavior to only illuminate the warning when a person is detected in the seat (p=1). The new circuit equation is:

```
w = p AND NOT(s) AND k
```

In this case, a 3-input AND gate is used. The circuit is shown in Figure 2.24.

Be aware that the order of the AND gate's inputs does not matter.

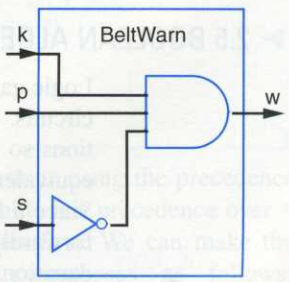


Figure 2.24 Seat belt warning circuit with person sensor.

Example 2.8 Seat belt warning light with initial illumination

Let's further extend the previous example. Automobiles typically light up all their warning lights when you first turn the key, so that you can check that all the warning lights are working. Assume that the system receives an input `t` that is `1` for the first 5 seconds after a key is inserted into the ignition, and `0` afterward (don't worry about who or what sets `t` in that way). So the system should set `w=1` when `p=1` and `s=0` and `k=1`, OR when `t=1`. Note that when `t=1`, the circuit should illuminate the light, regardless of the values of `p`, `s`, and `k`. The new circuit equation is:

```
w = (p AND NOT(s) AND k) OR t
```

The circuit is shown in Figure 2.25.

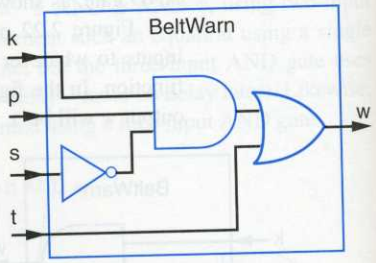


Figure 2.25 Extended seat belt warning circuit.

Some circuit drawing rules and conventions

There are some rules and conventions that designers commonly follow when drawing circuits of logic gates, as shown in Figure 2.26.

- Logic gates have one or more inputs and one output, but each input and output is typically not labeled. Remember: the order of the inputs into a gate doesn't affect the gate's logical behavior.
- Each wire has an implicit direction, from one gate's output to another gate's input, but we typically don't draw arrows showing each direction.
- A single wire can be branched out into two (or more) wires going to multiple gate inputs—the branches have the same value as the single wire. But two wires can NOT be merged into one wire—what would be the value of that one wire if the incoming two wires had different values?

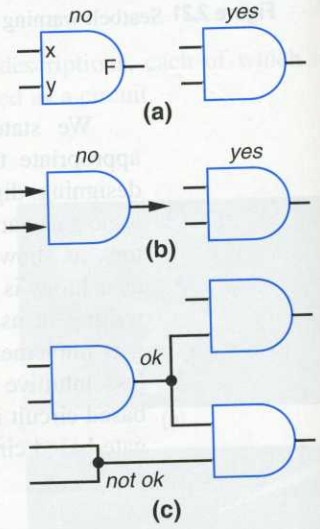


Figure 2.26 Circuit drawing rules.

2.5 BOOLEAN ALGEBRA

Logic gates are useful for implementing circuits, but equations are better for manipulating circuits. The algebraic tools of Boolean algebra enable us to manipulate Boolean equations so we can do things like simplify the equations, check whether two equations are equivalent, find the inverse of an equation, prove properties about the equations, etc. Since a Boolean equation consisting of AND, OR, and NOT operations can be straightforwardly transformed into a circuit of AND, OR, and NOT gates, manipulating Boolean equations can be considered as manipulating digital circuits. We'll informally introduce some of the most useful algebraic tools of Boolean algebra. Appendix A provides a formal definition of Boolean algebra.

Notation and Terminology

This section defines notation and terminology for describing Boolean equations. These definitions will be used extensively throughout the book.

Operators

Writing out the AND, OR, and NOT operators as words in equations is cumbersome. Thus, Boolean algebra uses simpler notation for those operators:

- "NOT(a)" is typically written as `a'` or $\overline{a}$. This book uses `a'`, which one speaks as "a ***prime***." `a'` is also known as the ***complement*** of `a`, or the ***inverse*** of `a`.
- "`a` OR `b`" is typically written as "`a + b`," specifically intended to look similar to the addition operator in regular algebra. "`a + b`" is even referred to as the ***sum*** of `a` and `b`. "`a + b`" is usually spoken as "a or b."
- "`a` AND `b`" is typically written as "`a * b`" or "`a•b`" specifically intended to look similar to the multiplication operator in regular algebra, and even referred to as the ***product*** of `a` and `b`. Just as in regular algebra, we can even write "`ab`" for the product of `a` and `b`, as long as the fact that `a` and `b` are separate variables is clear. "`a*b`" is usually spoken as "a and b" or even just as "a b."

Mathematicians often use other notations for Boolean operators, but the above notations seem to be the most popular among engineers, likely due to the intentional similarity of those operators with regular algebra operators.

Using the simpler notation, the earlier seat belt example:

```
w = (p AND NOT(s) AND k) OR t
```

could be rewritten more concisely as:

```
w = ps'k + t
```

which would be spoken as "w equals p s prime k, or t."

Example 2.9 Speaking Boolean equations

Speak the following equations:

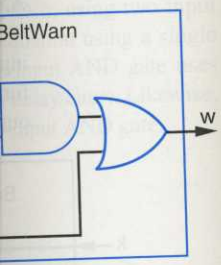
1. `F = a'b' + c`. *Answer:* "F equals a prime b prime or c."
2. `F = a + b * c'`. *Answer:* "F equals a or b and c prime."

Convert the following spoken equations into written equations:

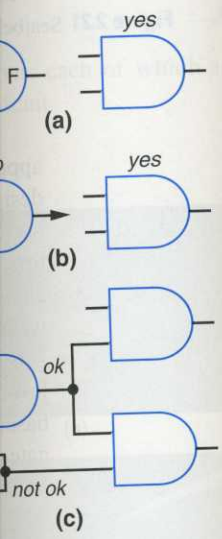
1. "F equals a b prime c prime." *Answer:* `F = ab'c'`.
2. "F equals a b c or d e prime." *Answer:* `F = abc + de'`.

The rules of Boolean algebra require that we evaluate expressions using the precedence rule that * has precedence over +, that complementing a variable has precedence over * and +, and that we of course compute what is in parentheses first. We can make the earlier equation's order of evaluation explicit using parentheses as follows: `w = (p * (s') * k) + t`.

Table 2.1 summarizes Boolean algebra precedence rules.



5 Extended seat
ng circuit.



26 Circuit drawing rules.

e better for manipulating
anipulate Boolean equa-
ether two equations are
bout the equations, etc.
erations can be straight-
s, manipulating Boolean
e'll informally introduce
Appendix A provides a

TABLE 2.1 Boolean algebra precedence, highest precedence first.

Symbol	Name	Description
( )	Parentheses	Evaluate expressions nested in parentheses first
'	NOT	Evaluate from left to right
*	AND	Evaluate from left to right
+	OR	Evaluate from left to right

Conventions

Although we borrowed the multiplication and addition operations from regular algebra and even use the terms sum and product, we *don't* say "times" for AND or "plus" for OR.

Digital design textbooks typically name each variable using a single character, because using a single character makes for concise equations like the equations above. We'll be writing many equations, so conciseness will aid understanding by preventing equations that wrap across multiple lines or pages. Thus, we'll usually follow the convention of using single characters. However, when you describe digital systems using a hardware description language or a programming language like C, you should probably use much more descriptive names so that your code is readable. So instead of using "`s`" to represent the output of a seat-belt-fastened sensor, you might instead use "`SeatBeltFastened`."

Example 2.10 Evaluating Boolean equations using precedence rules

Evaluate the following Boolean equations, assuming `a=1, b=1, c=0, d=1`.

1. `F = a * b + c`. *Answer:* * has precedence over +, so we evaluate the equation as `F = (1 * 1) + 0 = (1) + 0 = 1 + 0 = 1`.
2. `F = ab + c`. *Answer:* the problem is identical to the previous problem, using the shorthand notation for *.
3. `F = ab'`. *Answer:* we first evaluate `b'` because NOT has precedence over AND, resulting in `F = 1 * (1') = 1 * (0) = 1 * 0 = 0`.
4. `F = (ac)'`. *Answer:* we first evaluate what is inside the parentheses, then we NOT the result, yielding `(1*0)' = (0)' = 0' = 1`.
5. `F = (a + b') * c + d'`. *Answer:* The parentheses have highest precedence. Inside the parentheses, NOT has highest precedence. So we evaluate the parentheses part as `(1 + (1'))) = (1 + (0)) = (1 + 0) = 1`. Next, * has precedence over +, yielding `(1 * 0) + 1' = (0) + 1'`. The NOT has precedence over the OR, giving `(0) + (1') = (0) + (0) = 0 + 0 = 0`.

Variables, Literals, Terms, and Sum of Products

This section defines a few more concepts, using the example equation: `F(a,b,c) = a'bc + abc' + ab + c`.

- ***Variable:*** A variable represents a quantity (`0` or `1`). The above equation has three variables: `a`, `b`, and `c`. We typically use variables in Boolean equations to repre-

sent the inputs of our systems. Sometimes we explicitly list a function's variables as above ("`F(a,b,c) = ...`"). Other times we omit the explicit list ("`F = ...`").

- ***Literal:*** A literal is the appearance of a variable, in either true or complemented form. The above equation has 9 literals: `a'`, `b`, `c`, `a`, `b`, `c'`, `a`, `b`, and `c`.
- ***Product term:*** A product term is a product of literals. The above equation has four terms: `a'bc`, `abc'`, `ab`, and `c`.
- ***Sum of products:*** An equation written as an ORing of product terms is known as being in sum-of-products form. The above example equation for `F` is in sum-of-products form. The following equations are also in sum-of-products form:

```
abc + abc'
ab + a'c + abc
a + b' + ac  (note that a product term can have just one literal).
```

The following equations are NOT in sum-of-products form:

```
(a + b)c
(ab + bc)(b + c)
(a')' + b
a(b + c(d + e))
(ab + bc)'
```

People seem to prefer working with Boolean equations in sum-of-products form, and thus that form is very common.

Some Properties of Boolean Algebra

This section lists some of the key rules of Boolean algebra. Assume `a`, `b`, and `c` are Boolean variables, which each holds the value of either `0` or `1`.

Basic Properties

The following properties, known as postulates, are assumed to be true:

- ***Commutative***

```
a + b = b + a
a * b = b * a
```

This property should be obvious. Just try it for different values of `a` and `b`.

- ***Distributive***

```
a * (b + c) = a * b + a * c
a + (b * c) = (a + b) * (a + c)  (This one is tricky!)
```

Careful, the second one may not be obvious. It's different than regular algebra. But you can verify that both of the distributive properties hold simply by evaluating both sides for all possible values of `a`, `b`, and `c`.

- ***Associative***

```
(a + b) + c = a + (b + c)
(a * b) * c = a * (b * c)
```

Again, try it for different values of `a` and `b` to see that this holds.

- ***Identity***

```
0 + a = a + 0 = a
1 * a = a * 1 = a
```

This one should be intuitive. ORing a with 0 (a+0) just means that the result will be whatever a is. After all, 1+0 is 1, while 0+0 is 0. Likewise, ANDing a with 1 (a*1) results in a. 1*1 is 1, while 0*1 is 0.

- ***Complement***

```
a + a' = 1
a * a' = 0
```

This also makes intuitive sense. Regardless of the value of a, a' is the opposite, so you get a 0 and a 1, or you get a 1 and a 0. One of (a, a') will always be a 1, so ORing them (a+a') must yield a 1. Likewise, one of (a, a') will always be a 0, so ANDing them (a*a') must yield a 0.

The following examples apply these basic properties to some digital design examples to see how the properties can help.

Example 2.11 Applying the basic properties of Boolean algebra

Use the properties of Boolean algebra for the following problems:

- Show that abc' is equivalent to c'ba.
  The commutative property allows swapping the operands being ANDed, so a*b*c' = a*c'*b = c'*a*b = c'*b*a = c'ba.
- Show that abc + abc' = ab.
  The first distributive property allows factoring out the ab term: abc + abc' = ab(c+c'). Then, the complement property allows replacing the c+c' by 1: ab(c+c') = ab(1). Finally, the identity property allows removal of the 1 from the AND term: ab(1) = ab*1 = ab.
- Show that the equation x + x'z is equivalent to x + z.
  The second distributive property (the tricky one) allows replacing x+x'z by (x+x')*(x+z). The complement property allows replacing (x+x') by 1, and the identity property allows replacing 1*(x+z) by x+z.
- Show that (a+a')bc is just bc.
  The complement property states that (a+a') is 1, yielding 1*bc. The identity property then results in bc.
- Multiply out (w + x)(y + z) into sum-of-products form.
  First writing (w + x) as A will make clear that the distributive property can be applied: A(y+z). The first distributive property yields Ay + Az. Expanding A back yields (w+x)y + (w+x)z. Applying the first distributive property again yields wy + xy + wz + xz, which is in sum-of-products form.

Example 2.12 Simplification of an automatic sliding door system

Suppose you wish to design a system to control an automatic sliding door, like one that might be found at a grocery store's entrance. An input `p` to the system indicates whether a sensor detects a person in front of the door (`p=1` means a person is detected). An input `h` indicates whether the door should be manually held open (`h=1`) regardless of whether a person is detected. An input `c` indicates whether the door should be forced to stay closed (like when the store is closed for business)—`c = 1` means the door should stay closed. The latter two would normally be set by a manager with the proper keys. An output `f` opens the door when `f` is `1`.

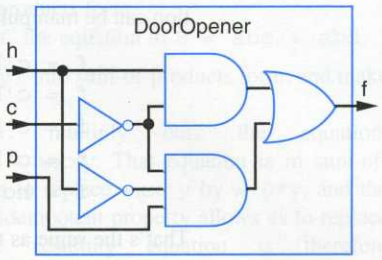


Figure 2.27 Initial door opener circuit.

The door should be opened if the door is set to be manually held open, OR if the door is not set to be manually held open but a person is detected. However, in either case, the door should only be opened if the door is not set to stay closed. These requirements can be translated into a Boolean equation:

```
f = hc' + h'pc'
```

A circuit to implement this equation could then be created, as in Figure 2.27.

The equation can be manipulated using the properties described earlier. Looking at the equation, we might try to factor out the `c'`. We might then be able to simplify the remaining `h+h'p` part too. Let's try some transformations, first factoring out `c'`:

```
f = hc' + h'pc'
f = c'h + c'h'p          (by the commutative property)
f = c'(h + h'p)          (by the first distributive property)
f = c'((h+h')*(h+p))     (by the 2nd distributive property—the tricky one)
f = c'((1)*(h + p))      (by the complement property)
f = c'(h+p)              (by the identity property)
```

Note that the simpler equation still makes intuitive sense—the door should be opened only if the door is not set to stay closed (`c'`), AND either the door is set to be manually held open (`h`) OR a person is detected (`v`). A circuit implementing this simpler equation is shown in Figure 2.28. Applying the algebraic properties led to a simpler circuit. In other words, we used math to simplify the circuit.

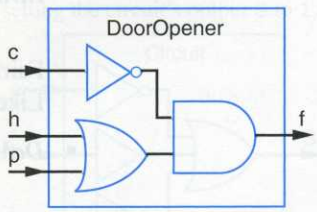


Figure 2.28 Simplified door opener circuit.

Simplification of logic circuits is the focus of Section 2.11.

Example 2.13 Equivalence of two automatic sliding door systems

Suppose you found a really cheap device for automatic sliding door systems. The device had inputs `c`, `h`, and `p` and output `f`, as in Example 2.12, but the device's documentation said that:

```
f = c'hp + c'hp' + c'h'p
```

Does that device do the same as that in Example 2.12? One way to check is to see if the above equation can be manipulated into the equation in Example 2.12:

```
f = c'hp + c'hp' + c'h'p
f = c'h(p + p') + c'h'p          (by the distributive property)
f = c'h(1) + c'h'p               (by the complement property)
f = c'h + c'h'p                  (by the identity property)
f = hc' + h'pc'                  (by the commutative property)
```

That's the same as the original equation in Example 2.12, so the device should work.

Additional Properties

This section introduces some additional properties, which happen to be known as theorems because they can be proven using the above postulates:

- ***Null elements***

```
a + 1 = 1
a * 0 = 0
```

These should be fairly obvious. `1` OR anything is going to be `1`, while `0` AND anything is going to be `0`.

- ***Idempotent Law***

```
a + a = a
a * a = a
```

Again, this should be fairly obvious. If `a` is `1`, `1+1=1` and `1*1=1`, while if `a` is `0`, `0+0=0` and `0*0=0`.

- ***Involution Law***

```
(a')' = a
```

Fairly obvious. If `a` is `1`, the first negation gives `0`, while the second gives `1` again. Likewise, if `a` is `0`, the first negation gives `1`, while the second gives `0` again.

- ***DeMorgan's Law***

```
(a + b)' = a'b'
(ab)' = a' + b'
```

These are not as obvious. Their proofs are in Appendix A. Let's consider both equations intuitively here. Consider `(a + b)' = a'b'`. The left side will only be `1` if `(a + b)` evaluates to `0`, which only occurs when both `a` AND `b` are `0`, meaning `a'b'` — the right side. Likewise, consider `(ab)' = a' + b'`. The left side will only be `1` if `(ab)` evaluates to `0`, meaning at least one of `a` OR `b` must be `0`, meaning `a' + b'` — the right side. DeMorgan's Law can be stated in English as follows: The complement of a sum equals the product of the complements; the complement of a product equals the sum of the complements. DeMorgan's Law is widely used, so take the time now to understand it and to remember it.

The following examples apply some of these additional properties.

Example 2.14 Applying the additional properties

- Convert the equation `F = ab(c+d)` into sum-of-products form.
  The distributive property allows us to "multiply out" the equation to `F = abc + abd`.
- Convert the equation `F = wx(x'y + zy' + xy)` into sum-of-products form, and make any obvious simplifications.
  The distributive property allows us to "multiply out" the equation: `wx(x'y + zy' + xy) = wxx'y + wxzy' + wxxy`. That equation is in sum-of-products form. The complement property allows us to replace `wxx'y` by `w*0*y`, and the null element property means that `w*0*y = 0`. The idempotent property allows us to replace `wxxy` by `wxy` (because `xx = x`). The resulting equation is therefore `0 + wxzy' + wxy = wxzy' + wxy`.
- Prove that `x(x' + y(x'+y'))` can never evaluate to `1`.
  Repeated application of the first distributive property yields: `xx' + xy(x'+y') = xx' + xyx' + xyy'`. The complement property tells us that `xx'=0` and `yy'=0`, yielding `0 + 0*y + x*0`. The null element property leads to `0 + 0 + 0`, which equals `0`. So the equation always evaluates to `0`, regardless of the actual values of `x` and `y`.
- Determine the opposite function of `F = (ab' + c)`.
  The desired function is `G = F' = (ab'+c)'`. DeMorgan's Law yields `G = (ab')' * c'`. Applying DeMorgan's Law again to the first term yields `G = (a'+(b')') * c'`. The involution property yields `(a' + b) * c'`. Finally, the distributive property yields `G = a'c' + bc'`.

Example 2.15 Applying DeMorgan's Law in an aircraft lavatory sign

Figure 2.29 Aircraft lavatory sign block diagram.

Commercial aircraft typically have an illuminated sign indicating whether a lavatory (bathroom) is available. Suppose an aircraft has three lavatories. Each lavatory has a sensor outputting `1` if the lavatory door is locked, `0` otherwise. A circuit will have three inputs, `a`, `b`, and `c`, coming from those sensors, as shown in Figure 2.29. If *any* lavatory door is unlocked (whether one, two, or all three doors are unlocked), the circuit should illuminate the "Available" sign by setting the circuit's output `S` to `1`.

With this understanding, we recognize that the OR function suits the problem, as OR outputs `1` if any of its inputs are `1`, regardless of how many inputs are `1`. We begin writing an equation for `S`. `S` should be `1` if `a` is `0` OR `b` is `0` OR `c` is `0`. Saying `a` is `0` is the same as saying `a'`. Thus, the equation for `S` is:

```
S = a' + b' + c'
```

We translate the equation to the circuit in Figure 2.30.

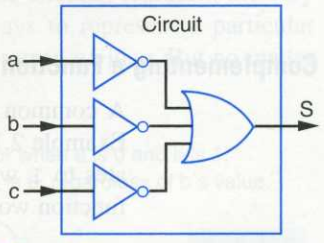


Figure 2.30 Aircraft lavatory sign circuit.

DeMorgan's Law can be applied (in reverse) to the equation by noting that `(abc)' = a'+b'+c'`, so we can replace the equation by:

```
S = (abc)'
```

The circuit for that equation appears in Figure 2.31.

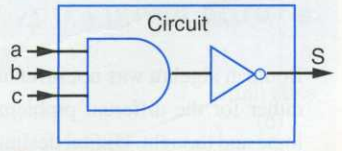


Figure 2.31 Circuit after applying DeMorgan's Law.

Example 2.16 Proving a property of the automatic sliding door system

A famous digital circuit error was the error found in the floating point unit of Intel's Pentium processor. It was found after the processor was already being widely sold in 1994, ultimately costing Intel $475 million. Thus, using Boolean techniques to prove correct behavior of circuits is a growing trend.

Your boss wants you to *prove* that the automatic sliding door circuit of Example 2.12 ensures that the door will stay closed when the door is supposed to be forced to stay closed, namely, when `c=1`. If the function `f = c'(h+p)` describes the sliding door, you can prove the door will stay closed (`f=0`) using properties of Boolean algebra:

```
f = c'(h+p)
Let c = 1                (door forced closed)
f = 1'(h+p)
f = 0(h+p)
f = 0h + 0p              (by the distributive property)
f = 0 + 0                (by the null elements property)
f = 0
```

Therefore, no matter what the values of `h` and `p`, if `c=1`, `f` will equal `0`—the door will stay closed.

Example 2.17 Automatic sliding door with opposite polarity

Example 2.12 computed the function to open an automatic sliding door as:

```
f = c'( h + p )
```

Suppose our function will control an automatic door control that has the opposite polarity: the function should output `0` to open the door, and `1` to close the door. The function `g` that opens the door can be computed and simplified as follows:

```
g = f'
g = (c'(h+p))'           (by substituting the equation for f)
g = (c')' + (h+p)'       (by DeMorgan's Law)
g = c + (h+p)'           (by the Involution Law)
g = c + h'p'             (by DeMorgan's Law)
```

Complementing a Function

A common task is to compute the complement of a Boolean function, as was done in Example 2.17. A function's ***complement***, also known as the ***inverse*** of a function, evaluates to `1` whenever the function would evaluate to `0`, and evaluates to `1` whenever the function would evaluate to `0`.

▶ *YOUR PROBLEM IS MY PROBLEM*

Boolean algebra was not invented for digital design, but rather for the different problem of formalizing human logic and thought. Digital design progressed slowly until Claude Shannon showed how Boolean algebra could be applied. In other words, powerful techniques from some other problem were applied to the digital design problem. Such borrowing of techniques from different problem domains is common in various fields engineering, and can lead to major breakthroughs.

The equation for a function's complement can be simplified by repeated use of DeMorgan's Law followed by other simplficiations. Note that DeMorgan's Law applies to any number of variables, not just two. Specifically, for three variables:

```
(a + b + c)' = (abc)'
(abc)' = (a' + b' + c')
```

Likewise for four variables, five variables, and so on.

For example, the complement of the function `f = w'xy + wx'y'z'` is `f' = (w'xy + wx'y'z')'`. DeMorgan's Law can then be applied as follows:

```
f' = (w'xy + wx'y'z')'
f' = (w'xy)'(wx'y'z')'          (by DeMorgan's Law)
f' = (w+x'+y')(w'+x+y+z)        (by DeMorgan's Law)
```

The equation can then be expanded into sum-of-products form as follows:

```
f' = w(w'+x+y+z) + x'(w'+x+y+z) + y'(w'+x+y+z)
f' = ww' + wx + wy + wz + x'w' + x'x + x'y + x'z +
     y'w' + y'x + y'y + y'z
f' = wx + wy + wz + w'x' + x'y + x'z + w'y' + xy' + y'z
```

2.6 REPRESENTATIONS OF BOOLEAN FUNCTIONS

A ***Boolean function*** is a *mapping* of each possible combination of values for the function's variables (the inputs) to either a `0` or `1` (the output). Figure 2.32(a) provides two alternative English descriptions of a particular Boolean function. There are several better representations than English for describing a Boolean function, including equations, circuits, and truth tables, as shown in Figure 2.32(b), (c), and (d). Each representation has its own advantages and disadvantages, and each is useful at different times during design. Yet all the representations, as different as they look from one another, represent the very same function. Such is akin to how there are different ways to represent a particular recipe for chocolate chip cookies: written words, pictures, or even a video. But no matter how the recipe is represented, it's the same recipe.

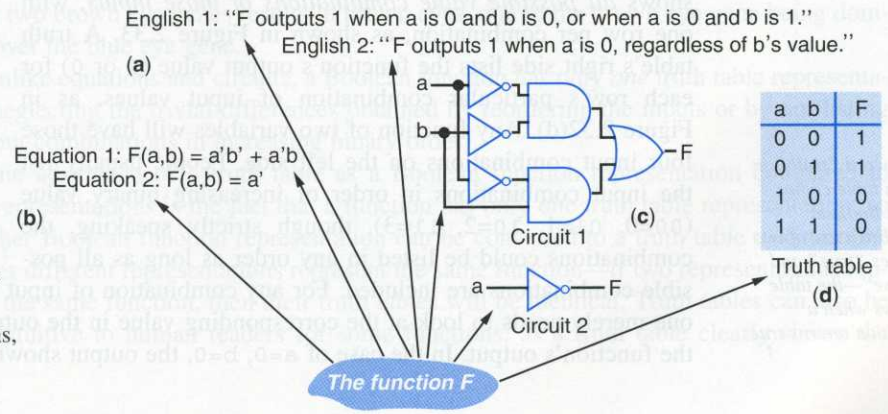


Figure 2.32 Seven representations of the same function F(a,b): (a) Two English descriptions, (b) two equations, (c) two circuits, (d) a truth table.

Equations

One way to represent a Boolean function is by using an equation. An ***equation*** is a mathematical statement equating one expression with another. `F(a,b) = a'b' + a'b` is an example of an equation. The right-hand side of the equation is often referred to as an ***expression***, which evaluates to either `0` or `1`.

Different equations can represent the same function. The two equations in Figure 2.32(b), `F(a,b) = a'b' + a'b` and `F(a,b) = a'`, represent the same function. Both equations perform exactly the same mapping of the input values to output values—pick any input values (e.g., `a=0` and `b=0`), and both equations map those input values to the same output value (e.g., `a=0` and `b=0` would be mapped to `F=1` by either equation).

One advantage of an equation as a Boolean function representation compared to other representations (such as English) is that equations can be manipulated using properties of Boolean algebra, enabling simplification of an equation, or proving that two equations represent the same function, or proving properties about a function, and more.

Circuits

Another way to represent a Boolean function is using a circuit of logic gates. A ***circuit*** is an interconnection of components. Because each logic gate component has a predefined mapping of input values to output values, and because wires just transmit their values unchanged, a circuit describes a function.

Different circuits can represent the same function. The two circuits in Figure 2.32(c) both represent the same function `F`. The bottom circuit uses fewer gates, but the function is exactly the same as the function of the top circuit.

One advantage of a circuit as a Boolean function representation compared to other representations is that a circuit may represent an actual physical implementation of a Boolean function. Another advantage is that a circuit drawn graphically can enable quick and easy comprehension of a function by humans.

Truth Tables

Another way to represent a Boolean function is using a ***truth table***. A truth table's left side lists the input variables, and shows *all possible value combinations of those inputs*, with one row per combination, as shown in Figure 2.33. A truth table's right side lists the function's output value (`1` or `0`) for each row's particular combination of input values, as in Figure 2.32(d). Any function of two variables will have those four input combinations on the left side. People usually list the input combinations in order of increasing binary value (`00`=0, `01`=1, `10`=2, `11`=3), though strictly speaking, the combinations could be listed in any order as long as all possible combinations are included. For any combination of input values (e.g., `a=0`, `b=0`), one merely needs to look at the corresponding value in the output column to determine the function's output. In the case of `a=0`, `b=0`, the output shown in Figure 2.32(d) is `1`.

Inputs		Output
a	b	F
0	0	
0	1	
1	0	
1	1	

Figure 2.33 Truth table structure for a two-input function `F(a,b)`.

The word "truth" in truth table comes from Boolean algebra's use of two values "true" and "false"—the table shows when a function returns true.

a	b	F
0	0	
0	1	
1	0	
1	1	

(a)

a	b	c	F
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

(b)

a	b	c	d	F
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

(c)

Figure 2.34 Truth table structures for: (a) a two-input function `F(a,b)`, (b) a three-input function `F(a,b,c)`, and (c) a four-input function `F(a,b,c,d)`. Defining a specific function involves filling in the rightmost column for `F` with a `0` or a `1` for each row.

Figure 2.34 shows the truth table structures for a two-input function, a three-input function, and a four-input function.

Truth tables are not only found in digital design. If you've studied basic biology, you've likely seen a type of truth table describing the outcome of various gene pairs. For example, the table in Figure 2.35 shows outcomes for different eye color genes. Each person has two genes for eye color, one (labeled `M`) from the mom, one (labeled `D`) from the dad. Assuming only two possible values for each gene, `blue` and `brown`, the table lists all possible combinations of eye color gene pairs that a person may have. For each combination, the table lists the outcome. Only when a person has two blue eye genes will they have blue eyes; having one or two brown eye genes results in brown eyes, due to the brown eye gene being dominant over the blue eye gene.

Gene pair		Outcome
M	D	F
blue	blue	blue
blue	brown	brown
brown	blue	brown
brown	brown	brown

Figure 2.35 Truth table used to describe outcomes for gene pairs.

Unlike equations and circuits, a Boolean function has only *one* truth table representation (neglecting the trivial differences obtained by reordering the inputs or by not listing the input combinations in increasing binary order).

One advantage of a truth table as a Boolean function representation compared to other representations is the fact that a function has only one truth table representation, so any other Boolean function representation can be converted to a truth table to determine whether different representations represent the same function—if two representations represent the same function, then their truth tables will be identical. Truth tables can also be quite intuitive to human readers for some functions, as a truth table clearly shows the

output for every possible input. Thus, notice that truth tables were used in Figure 2.8 to describe in an intuitive manner the behavior of basic logic gates.

A drawback of truth tables is that for a large number of inputs, the number of truth table rows can be very large. Given a function with n inputs, the number of input combinations is 2^n. A function with 10 inputs would have $2^{10} = 1024$ possible input combinations—you can't easily see much of anything in a table having 1024 rows. A function with 16 inputs would have 65,536 rows in its truth table.

Example 2.18 Capturing a function as a truth table

Create a truth table describing a function that detects whether a three-bit inputs' value, representing a binary number, is 5 or greater. Table 2.2 shows a truth table for the function. We first list all possible combinations of the three input bits, which we've labeled `a`, `b`, and `c`. We then enter a `1` in the output row if the inputs represent 5, 6, or 7 in binary, meaning the last three rows. We enter `0`s in all the other rows.

TABLE 2.2 Truth table for 5-or-greater function.

a	b	c	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Converting among Boolean Function Representations

Given the above representations, converting from one representation to another is sometimes necessary or useful. For the three representations discussed so far (equations, circuits, and truth tables), there are six possible conversions from one representation to another, as shown in Figure 2.36, which will now be described.

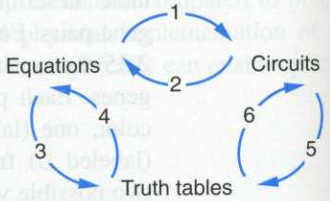


Figure 2.36 Possible conversions from one Boolean function representation to another.

1. Equations to Circuits

Converting an equation to a circuit can be done straightforwardly by using an AND gate for every AND operator, an OR gate for every OR operator, and a NOT gate for every NOT operator. Several examples of such conversions appear in Section 2.4.

2. Circuits to Equations

Converting a circuit into an equation can be done by starting from the circuit's inputs, and then writing the output of each gate as an expression involving the gate's inputs. The expression of the last gate before the output represents the expression for the circuit's function.

For example, consider the circuit in Figure 2.37. To convert to an equation, we start with the inverter, whose output will represent `c'`. We continue with the OR gate—note that we can't determine the output for the AND gate yet until we create expressions for all that gate's inputs. The OR gate's output represents `h+p`. Finally, we write the output of the AND as `c'(h+p)`. Thus, the equation `F(c,h,p) = c'(h+p)` represents the same function as the circuit.

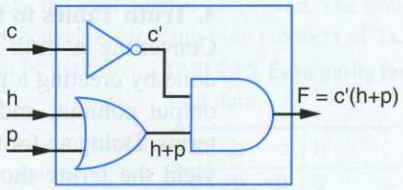


Figure 2.37 Converting a circuit to an equation.

3. Equations to Truth Tables

Converting an equation to a truth table can be done by first creating a truth table structure appropriate for the number of function input variables, and then evaluating the right-hand side of the equation for each combination of input values. For example, to convert the equation `F(a,b) = a'b' + a'b` to a truth table, we would first create the truth table structure for a two-input function, as in Figure 2.34(a). We would then evaluate the right-hand side of the equation for each row's combination of input values, as follows:

Inputs		Output
a	b	F
0	0	1
0	1	1
1	0	0
1	1	0

Figure 2.38 Truth table for F(a,b)=a'b'+a'b.

- `a=0` and `b=0`, `F = 0'*0' + 0'*0 = 1*1 + 1*0 = 1 + 0 = 1`
- `a=0` and `b=1`, `F = 0'*1' + 0'*1 = 1*0 + 1*1 = 0 + 1 = 1`
- `a=1` and `b=0`, `F = 1'*0' + 1'*0 = 0*1 + 0*0 = 0 + 0 = 0`
- `a=1` and `b=1`, `F = 1'*1' + 1'*1 = 0*0 + 0*1 = 0 + 0 = 0`

We would therefore fill in the table's right column as in Figure 2.38. Note that we applied properties of Boolean algebra (mostly the identity property and null elements property) to evaluate the equations.

Notice that converting the equation `F(a,b)=a'` to a truth table results in exactly the same truth table as in Figure 2.38. In particular, evaluating the right-hand side of the equation for each row's combination of input values yields:

- `a=0` and `b=0`, `F = 0' = 1`
- `a=0` and `b=1`, `F = 0' = 1`
- `a=1` and `b=0`, `F = 1' = 0`
- `a=1` and `b=1`, `F = 1' = 0`

Inputs				Output
a	b	a'b'	a'b	F
0	0	1	0	1
0	1	0	1	1
1	0	0	0	0
1	1	0	0	0

Figure 2.39 Truth table for `F(a,b) = a'b'+a'b` with intermediate columns.

Some people find it useful to create intermediate columns in the truth table to compute the equation's intermediate values, thus filling each column of the table from left to right, moving to the next column only after filling all rows of the current column. An example for the equation `F(a,b) = a'b' + a'b` is shown in Figure 2.39.

4. Truth Tables to Equations

Converting a truth table to an equation can be done by creating a product term for each `1` in the output column, and then ORing all the product terms. Doing so for the table of Figure 2.40 would yield the terms shown in the rightmost column. ORing those terms yields `F = a'b' + a'b`. This conversion will be very frequently used; take the time to understand it now.

Inputs		Outputs	Term
a	b	F	F = sum of
0	0	1	a' b'
0	1	1	a' b
1	0	0	
1	1	0	

Thus: F = a' b' + a' b

Figure 2.40 Converting a truth table to an equation.

5. Circuits to Truth Tables

A combinational circuit can be converted to a truth table by first converting the circuit to an equation (described earlier), and then converting the equation to a truth table (described earlier).

6. Truth Tables to Circuits

A truth table can be converted to a circuit by first converting the truth table to an equation (described earlier), and then converting the equation to a circuit (described earlier).

Example 2.19 Parity generator circuit design starting from a truth table

Nothing is perfect, and digital circuits are no exception. Sometimes a bit on a wire changes even though it is not supposed to change. So a `1` becomes a `0`, or a `0` becomes a `1`, accidentally. For example, a `0` may be traveling along a wire, when suddenly some electrical noise comes out of nowhere and changes the `0` to a `1`. The likelihood of such errors can be reduced by methods such as using well-insulated wires, but such errors can't be completely prevented. Nor can all such errors be detected and corrected—but we can detect *some* of them. Designers typically look for situations where errors are likely to occur, such as data being transmitted between two chips over long wires—like from a computer over a printer cable to a printer, or from a keyboard over a wireless channel to a computer. For those situations, designers add circuits that at least try to detect that an error has occurred, in which case the receiving circuit can request that the sending circuit resend the data.

One common method of detecting an error is called ***parity***. Say we have 7 data bits to transmit. We add an extra bit, called the parity bit, to make 8 bits total. The sender sets the parity bit to a `1` if that would make the total number of `1`s even—that's called ***even parity***. For example, if the 7 data bits were `0000001`, then the parity bit would be `1`, making the total number of `1`s equal to 2 (an even number). The complete 8 bits would be `00000011`, where we've italicized the parity bit. If the 7 data bits were `1011111`, then the parity bit would be `0`, making the total number of `1`s equal to 6 (an even number). The complete 8 bits would be `10111110`.

The receiver now can detect whether a bit has changed during transmission by checking whether there is an even number of `1`s in the 8 bits received. If there is an even number of `1`s, the transmission is assumed correct. If not even, an error occurred during transmission. For example, if `00000011` is received, the transmission is assumed to be correct, and the parity bit can be discarded, leaving `0000001`. Suppose instead that `10000011` is received. Seeing the odd number of `1`s, the receiver knows that an error occurred—note that the receiver does *not* know which bit is erroneous. Likewise, `00000010` would represent an error too.

For this example, starting from a truth table is a more natural choice than an equation.

Let's describe a function that generates an even parity bit `P` for 3 data bits `a`, `b`, and `c`. Starting from an equation is hard—what's the equation? For this example, starting with a truth table is the natural choice, as shown in Table 2.3. For each configuration of data bits (i.e., for each row in the

truth table), the value of the parity bit is set such as to make the total number of 1s even. The rows labeled 1, 2, 3, and 4 have two, two, two, and four 1s, respectively—all being even numbers of 1s.

From the truth table, the following equation for P can be derived:

```
P = a'b'c(1) + a'bc'(2) + ab'c'(3) + abc(4)
```

We used the numbers *1*, *2*, *3*, and *4* to show the correspondence between each 1 in the table and each term in the equation. For example, the input values for the row numbered *(3)* in the table are 1 0 0, which means ab'c'. This equation could then be converted to a circuit having four 3-input AND gates and one 4-input OR gate.

TABLE 2.3 Even parity for 3-bit data.

a	b	c	P	
0	0	0	0	
0	0	1	1	*(1)*
0	1	0	1	*(2)*
0	1	1	0	
1	0	0	1	*(3)*
1	0	1	0	
1	1	0	0	
1	1	1	1	*(4)*

Undetected incorrect transmissions are sometimes why an email or webpage is received with garbled text, or why a computer, printer, or mobile phone might execute incorrectly or freeze up.

Note that receiving data that has an even number of 1s and that is supposed to have even parity doesn't mean for sure that the received data is really correct (note that we were careful to say earlier that the transmission was "assumed" to be correct if the parity was correct). In particular, if two errors occur on different bits, then the parity will still be even. For example, the sender may send 0110, but the receiver may receive 1111. 1111 has even parity and thus looks correct.

More powerful error detection methods are possible to detect multiple errors, but at the price of adding extra bits.

Odd parity is also a common kind of parity—the parity bit value makes the total number of 1s odd. There's no quality difference between even parity and odd parity—the key is simply that the sender and receiver must both use the same kind of parity, even or odd.

A popular representation of letters and numbers is known as ASCII, which encodes each character into 7 bits. ASCII adds one bit for parity, for a total of 8 bits per character.

Example 2.20 Converting a combinational circuit to a truth table

Convert the circuit depicted in Figure 2.41(a) into a truth table.

We begin by converting the circuit to an equation. Starting from the gates closest to the inputs—the leftmost AND gate and the inverter in this case—we label each gate's output as an expression of the gate's inputs. We label the leftmost AND gate's output, for example, as ab. Likewise, we label the leftmost inverter's output as c'. Continuing through the circuit's gates, we label the rightmost inverter's output as (ab)'. Finally, we label the rightmost AND gate's output as (ab)'c', which corresponds to the Boolean equation for F. The fully labeled circuit is shown in Figure 2.41(b).

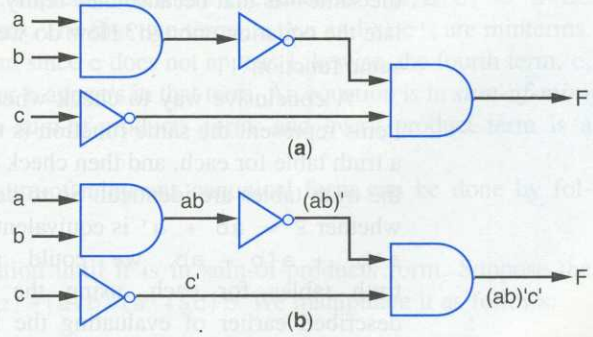


Figure 2.41 Converting a circuit to an equation: (a) original circuit, and (b) circuit with gates' output expressions labeled.

Inputs						Outputs
a	b	c	ab	(ab)'	c'	F
0	0	0	0	1	1	1
0	0	1	0	1	0	0
0	1	0	0	1	1	1
0	1	1	0	1	0	0
1	0	0	0	1	1	1
1	0	1	0	1	0	0
1	1	0	1	0	1	0
1	1	1	1	0	0	0

Figure 2.42 Truth table for the circuit's equation.

From the Boolean equation, we can now construct the truth table for the combinational circuit. Since our circuit has three inputs—a, b, and c—there are $2^3 = 8$ possible combinations of inputs (i.e., `abc = 000, 001, 010, 011, 100, 101, 110, 111`), so the truth table has the eight rows shown in Figure 2.42. For each input, we compute the value of F and fill in the corresponding entry in the truth table. For example, when `a=0`, `b=0`, and `c=0`, F is `(00)'*0' = (0)'*1 = 1*1 = 1`. We compute the circuit's output for the remaining combinations of inputs using a truth table with intermediate values, shown in Figure 2.42.

Standard Representation and Canonical Form

Standard Representation—Truth Tables

As stated earlier, although there are many equation representations and circuit representations of a Boolean function, there is only one possible truth table representation of a Boolean function. Truth tables therefore represent a ***standard representation*** of a function—for any function, there may be many possible equations, and many possible circuits, but there is only *one* truth table. The truth table representation is unique.

One use of a standard representation of a Boolean function is for comparing two functions to see if they are equivalent. Suppose you wanted to check whether two Boolean equations represented the same function. One way would be to try to manipulate one equation to be the same as the other equation, like we did in the automatic sliding door example of Example 2.13. But suppose we were not successful in getting them to be the same—is that because they really are not the same, or because we just didn't manipulate the equation enough? How do we really know the two equations do not represent the same function?

A conclusive way to check whether two items represent the same function is to create a truth table for each, and then check whether the truth tables are identical. So to determine whether `F = ab + a'` is equivalent to `F = a'b' + a'b + ab`, we could generate truth tables for each, using the method described earlier of evaluating the function for each output row, as Figure 2.43.

F = ab + a′

a	b	F
0	0	1
0	1	1
1	0	0
1	1	1

F = a′b′ + a′b + ab

a	b	F
0	0	1
0	1	1
1	0	0
1	1	1

Figure 2.43 Truth tables showing equivalence.

We see that the two equations are indeed equivalent, because the outputs are identical for each input combination. Now let's check whether `F = ab + a'` is equivalent to `F = (a+b)'` by comparing truth tables.

As seen in Figure 2.44, those two equations are clearly not equivalent. Comparing truth tables leaves no doubt.

F = ab + a'		
a	b	F
0	0	1
0	1	1
1	0	0
1	1	1

F = (a+b)'		
a	b	F
0	0	1
0	1	0
1	0	0
1	1	0

Figure 2.44 Non-equivalence proven.

While comparing truth tables works fine when a function has only 2 inputs, what if a function has 5 inputs, or 10, or 32? Creating truth tables becomes increasingly cumbersome, and in many cases unrealistic, since a truth table's number of rows equals 2^n, where n is the number of inputs. 2^n grows very quickly. 2^{32} is approximately 4 billion, for example. We can't realistically expect to compare 2 tables of 4 billion rows each.

However, in many cases, the number of output `1`s in a truth table may be very small compared to the number of output `0`s. For example, consider a function `G` of 5 variables `a`, `b`, `c`, `d`, and `e`: `G = abcd + a'bcde`. A truth table for this function would have 32 rows, but only three `1`s in the output column—one `1` from `a'bcde`, and two `1`s from `abcd` (which covers rows corresponding to `abcde` and `abcde'`). This leads to the question:

Is there a more compact but still *standard* representation of a Boolean function?

Canonical Form—Sum-of-Minterms Equation

The answer to the above question is yes. The key is to create a standard representation that only describes the situations where the function outputs `1`, with the other situations assumed to output `0`. An equation, such as `G = abcd + a'bcde`, is indeed a representation that only describes the situations where `G` is `1`, but that representation is not unique, that is, the representation is not standard. We therefore want to define a standard form of a Boolean equation, known as a ***canonical form***.

You've seen canonical forms in regular algebra. For example, the canonical form of a polynomial of degree two is: $ax^2 + bx + c$. To check whether the equation $9x^2 + 3x + 2 + 1$ is equivalent to the equation $3*(3x^2 + 1 + x)$, we convert each to canonical form, resulting in $9x^2 + 3x + 3$ for both equations.

One canonical form for a Boolean function is known as a sum of minterms. A ***minterm*** of a function is a product term whose literals include every variable of the function *exactly once*, in either true or complemented form. The function `F(a,b,c) = a'bc + abc' + ab + c` has four terms. The first two terms, `a'bc` and `abc'`, are minterms. The third term, `ab`, is not a minterm since `c` does not appear. Likewise, the fourth term, `c`, is not a minterm, since neither `a` nor `b` appears in that term. An equation is in ***sum-of-minterms form*** if the equation is in sum-of-products form, and every product term is a minterm.

Converting any equation to sum-of-minterms canonical form can be done by following just a few steps:

1. First, manipulate the equation until it is in sum-of-products form. Suppose the given equation is `F(a,b,c)=(a+b)(a'+ac)b`. We manipulate it as follows:

```
F = (a+b)(a'+ac)b
F = (a+b)(a'b+acb)                (by the distributive property)
F = a(a'b+acb) + b(a'b+acb)       (distributive property)
F = aa'b + aacb + ba'b + bacb     (distributive property)
F = 0*b + acb + a'b + acb         (complement, commutative,
                                   idempotent)
F = acb + a'b + acb               (null elements)
F = acb + a'b                     (idempotent)
```

2. Second, expand each term until every term is a minterm:

```
F = acb + a'b
F = acb + a'b*1                   (identity)
F = acb + a'b*(c+c')              (complement)
F = acb + a'bc + a'bc'            (distributive)
```

3. (Optional step.) For neatness, arrange the literals within each term to a consistent order (say, alphabetical), and also arrange the terms in the order they would appear in a truth table:

```
F = a'bc' + a'bc + abc
```

The equation is now in sum-of-minterms form. The equation is in sum-of-products form, and every product term is a minterm because each term includes every variable exactly once.

An alternative canonical form is known as product of maxterms. A ***maxterm*** is a sum term in which every variable appears exactly once in either true or complemented form, such as `(a + b + c')` for a function of three variables `a`, `b`, and `c`. An equation is in ***product-of-maxterms form*** if the equation is the product of sum terms, and every sum term is a maxterm. An example of a function (different from that above) in product-of-maxterms form is `J(a,b,c) = (a + b + c')(a' + b' + c')`. To avoid confusing the reader, we will not discuss the product-of-maxterms form further, as sum-of-minterms form is more common in practice, and sufficient for our purposes.

Example 2.21 Comparing two functions using canonical form

Suppose we want to determine whether the functions `G(a,b,c,d,e) = abcd + a'bcde` and `H(a,b,c,d,e) = abcde + abcde' + a'bcde + a'bcde(a' + c)` are equivalent. We first convert `G` to sum-of-minterms form:

```
G = abcd + a'bcde
G = abcd(e+e') + a'bcde
G = abcde + abcde' + a'bcde
G = a'bcde + abcde' + abcde
```

We then convert H to sum-of-minterms form:

```
H = abcde + abcde' + a'bcde + a'bcde(a' + c)
H = abcde + abcde' + a'bcde + a'bcdea' + a'bcdec
H = abcde + abcde' + a'bcde + a'bcde + a'bcde
H = abcde + abcde' + a'bcde
H = a'bcde + abcde' + abcde
```

Clearly, G and H are equivalent.

Note that checking the equivalence using truth tables would have resulted in two rather large truth tables having 32 rows each. Using sum of minterms was probably more appropriate here.

Compact sum-of-minterms representation

A more compact representation of sum-of-minterms form involves listing each minterm as a number, with each minterm's number determined from the binary representation of its variables' values. For example, `a'bcde` corresponds to `01111`, or 15; `abcde'` corresponds to `11110`, or 30; and `abcde` corresponds to `11111`, or 31. Thus, we can say that the function `H` represented by the equation

```
H = a'bcde + abcde' + abcde
```

is the sum of the minterms 15, 30, and 31, which can be compactly written as:

$$H = \Sigma m(15, 30, 31)$$

The summation symbol means the sum, and then the numbers inside the parentheses represent the minterms being summed on the right side of the equation.

Multiple-Output Combinational Circuits

The examples above showed combinational circuits with only one output, but many circuits have multiple outputs. The simplest approach to handling a multiple-output circuit is to treat each output separately, leading to a separate circuit for each output. Actually, the circuits need not be completely separate—they could share common gates. The following examples show how to handle multiple-output circuits.

Example 2.22 Two-output combinational circuit

Design a circuit to implement the following two equations of three inputs `a`, `b`, and `c`:

```
F = ab + c'     G = ab + bc
```

We can design the circuit by simply creating two separate circuits, as in Figure 2.45(a).

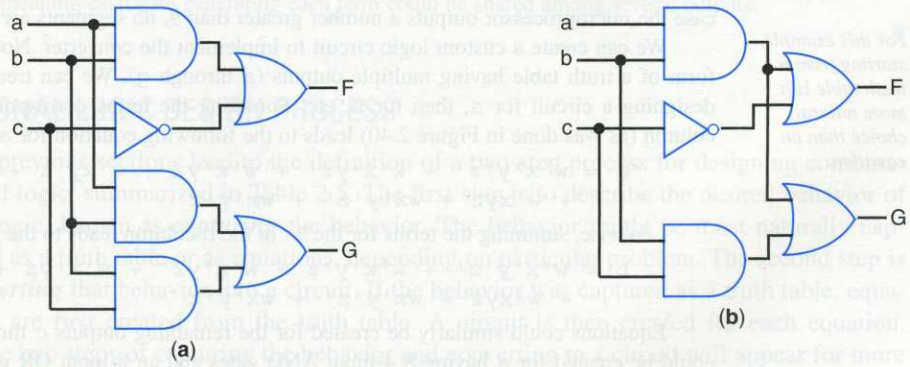


Figure 2.45 Multiple-output circuit: (a) treated as two separate circuits, and (b) with gate sharing.

We might instead notice that the term `ab` is common to both equations. Thus, the two circuits can share the gate that computes `ab`, as shown in Figure 2.45(b).

Example 2.23 Binary number to seven-segment display converter

Many electronic appliances display a number for us to read. Example appliances include a clock, a microwave oven, and a telephone answering machine. A popular and simple device for displaying a single digit number is a ***seven-segment display***, illustrated in Figure 2.46.

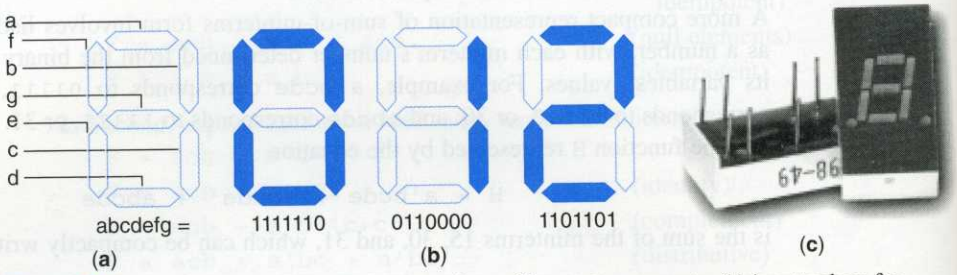


Figure 2.46 Seven-segment display: (a) connections of inputs to segments, (b) input values for numbers 0, 1, and 2, and (c) a pair of real seven-segment display components.

The display consists of seven light segments, each of which can be illuminated independently of the others. A desired digit can be displayed by setting the signals `a`, `b`, `c`, `d`, `e`, `f`, and `g` appropriately. So to display the digit 8, all seven signals must be set to `1`. To display the digit `1`, `b` and `c` are each set to `1`. A few letters can be displayed too, like a lower case "b."

Commonly, a microprocessor outputs a 4-bit binary number intended to be shown on a seven-segment display as a decimal (base ten) digit. Outputing four rather than seven signals conserves scarce pins on the microprocessor. Thus, a useful combinational circuit converts four bits `w`, `x`, `y`, and `z` of a binary number to the seven-segment display signals `a`–`g`, as in Figure 2.47.

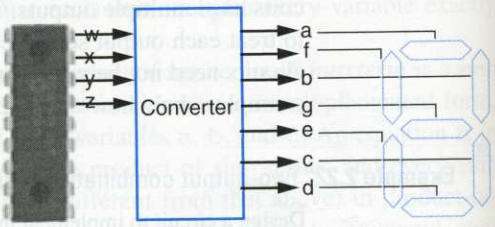


Figure 2.47 Binary to seven-segment converter.

The desired circuit behavior is easily captured as a table, shown in Table 2.4. In case the microprocessor outputs a number greater than 9, no segments are activated.

For this example, starting from a truth table is a more natural choice than an equation.

We can create a custom logic circuit to implement the converter. Note that Table 2.4 is in the form of a truth table having multiple outputs (`a` through `g`). We can treat each output separately, designing a circuit for `a`, then for `b`, etc. Summing the terms corresponding to the `1`s in the `a` column (as was done in Figure 2.40) leads to the following equation for `a`:

```
a = w'x'y'z' + w'x'yz' + w'x'yz + w'xy'z + w'xyz'
  + w'xyz + wx'y'z' + wx'y'z
```

Likewise, summing the terms for the `1`s in the `b` column leads to the following equation for `b`:

```
b = w'x'y'z' + w'x'y'z + w'x'yz' + w'x'yz + w'xy'z'
  + w'xyz + wx'y'z' + wx'y'z
```

Equations could similarly be created for the remaining outputs `c` through `g`. Finally, a circuit could be created for `a` having 8 4-input AND gates and an 8-input OR gate, another circuit for `b` having 8 4-input AND gates and an 8-input OR gate, and so on for `c` through `g`. We could, of course, have minimized the logic for each equation before creating each of the circuits.

You may notice that the equations for `a` and `b` have several terms in common. For example, the term `w'x'y'z'` appears in both equations. So it would make sense for both outputs to share one

TABLE 2.4 4-bit binary number to seven-segment display truth table.

w	x	y	z	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

AND gate generating that term. Looking at the truth table, we see that the term `w'x'y'z'` is in fact needed for outputs `a`, `b`, `c`, `e`, `f`, and `g`, and thus the one AND gate generating that term could be shared by all six of those outputs. Likewise, each of the other required terms is shared by several outputs, meaning each gate generating each term could be shared among several outputs.

2.7 COMBINATIONAL LOGIC DESIGN PROCESS

The previous sections lead to the definition of a two-step process for designing combinational logic, summarized in Table 2.5. The first step is to describe the desired behavior of the logic, known as ***capturing*** the behavior. The behavior might be most naturally captured as a truth table or as equations, depending on particular problem. The second step is ***converting*** that behavior into a circuit. If the behavior was captured as a truth table, equations are first created from the truth table. A circuit is then created for each equation. These two steps of capturing the behavior and converting to a circuit will appear for more complex circuits in subsequent chapters too, though their details will differ.

TABLE 2.5 Combinational logic design process.

Step		Description
Step 1: Capture behavior	***Capture*** *the function*	Create a truth table or equations, whichever is most natural for the given problem, to describe the desired behavior of each output of the combinational logic.
Step 2: Convert to circuit	2A ***Create*** *equations*	This substep is only necessary if you captured the function using a truth table instead of equations. Create an equation for each output by ORing all the minterms for that output. Simplify the equations if desired.
	2B ***Implement*** *as a gate-based circuit*	For each output, create a circuit corresponding to the output's equation. (Sharing gates among multiple outputs is OK optionally.)

Below are several examples demonstrating the combinational logic design process. We normally create equations that are in sum-of-products form. Directly converting a sum-of-products equation into a circuit results in a column of AND gates (possibly preceded by some NOT gates) that feeds into a single OR gate, which is known as a ***two-level circuit*** or ***two-level logic***.

Example 2.24 Three 1s pattern detector

This example implements a circuit that can detect whether a pattern of at least three adjacent `1`s occur anywhere in an 8-bit input, and that outputs a `1` in that case. The inputs are `a`, `b`, `c`, `d`, `e`, `f`, `g`, and `h`, and the output is `y`. So for an input of `abcdefgh = 00011101`, `y` should be `1`, since there are three adjacent `1`s, on inputs `d`, `e`, and `f`. For an input of `10101011`, the output should be `0`, since there are not three adjacent `1`s anywhere. An input of `11110000` should result in `y = 1`, since having more than three adjacent `1`s should still output a `1`. Such a circuit is an extremely simple example of a general class of circuits known as pattern detectors. Pattern detectors are widely used in image processing to detect objects like humans or tanks in a digitized video image, or to detect specific spoken words in a digitized audio stream, for example.

For this example, starting from an equation is a more natural choice than a truth table.

Step 1: ***Capture behavior.*** We could capture the function as a rather large truth table, listing out all 256 combinations of inputs, and entering a `1` for `y` in each row where at least three `1`s occur. However, a simpler method for capturing this particular function is to create an equation that lists the possible occurrences of three `1`s in a row. One possibility is that of `abc=111`. Another is that of `bcd=111`. Likewise, if `cde=111`, `def=111`, `efg=111`, or `fgh=111`, we should output a `1`. For each possibility, the values of the other inputs don't matter. So if `abc=111`, we output a `1`, regardless of the values of `d`, `e`, `f`, `g`, and `h`. Thus, an equation describing `y` is simply:

```
y = abc + bcd + cde + def + efg + fgh
```

Step 2A: ***Create equations.*** We skip this substep because an equation was already created above.

Step 2B: ***Implement as a gate-based circuit.*** No simplification of the equation is possible. The resulting circuit is shown in Figure 2.48.

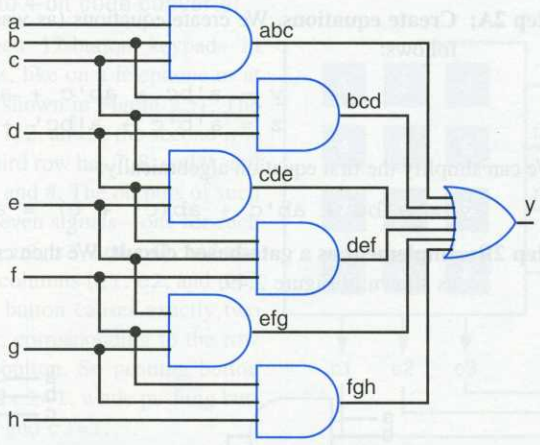


Figure 2.48 Three 1s pattern detector.

Example 2.25 Number-of-1s counter

For this example, starting from a truth table is a more natural choice than an equation.

This example designs a circuit that counts the number of 1s present on three inputs a, b, c, and outputs that number in binary using two outputs, y and z. An input of 110 has two 1s, so the circuit should output 10. The number of 1s on three inputs can range from 0 to 3, so a 2-bit output is sufficient, since 2 bits can represent 0 to 3. A number-of-1s counter circuit is useful in various situations, such as detecting the density of electronic particles hitting a collection of sensors by counting how many sensors are activated.

Step 1: **Capture behavior.** Capturing the behavior for this example is most naturally achieved using a truth table. We list all the possible input combinations, and the desired output number, as in Table 2.6.

TABLE 2.6 Truth table for number-of-1s counter.

Inputs			(# of 1s)	Outputs	
a	b	c		y	z
0	0	0	(0)	0	0
0	0	1	(1)	0	1
0	1	0	(1)	0	1
0	1	1	(2)	1	0
1	0	0	(1)	0	1
1	0	1	(2)	1	0
1	1	0	(2)	1	0
1	1	1	(3)	1	1

Step 2A: Create equations. We create equations (as was done in Figure 2.40) for each output as follows:

```
y = a'bc + ab'c + abc' + abc
z = a'b'c + a'bc' + ab'c' + abc
```

We can simplify the first equation algebraically:

```
y = a'bc + ab'c + ab(c' + c) = a'bc + ab'c + ab
```

Step 2B: Implement as a gate-based circuit. We then create the final circuits for the two outputs, as shown in Figure 2.49.

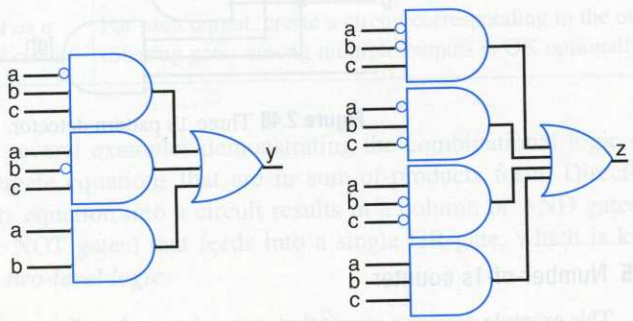


Figure 2.49 Number-of-1s counter gate-based circuit.

Simplifying Circuit Notations

Some new simplifying notations were used in the circuits in the previous example. One simplifying notation is to list the inputs multiple times, as in Figure 2.50(a). Such listing reduces lines in a drawing crossing one another. An input listed multiple times is assumed to have been branched from the same input.

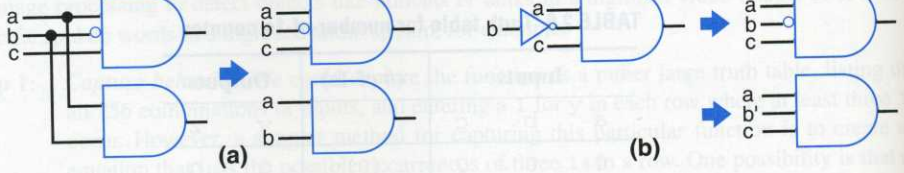


Figure 2.50 Simplifying circuit notations: (a) listing inputs multiple times to reduce drawing of crossing wires, (b) using inversion bubbles or complemented input to reduce NOT gates drawn.

Another simplifying notation is the use of an inversion bubble at the input of a gate, rather than the use of an inverter, as in Figure 2.50(b). An ***inversion bubble*** is a small circle drawn at the input of a gate as shown, indicating that the signal is inverted. An external input that has inversion bubbles at many gates is assumed to feed through a single inverter that is then branched out to those gates. An alternative simplification is to simply list the input as complemented, like `b'` shown in the figure.

Example 2.26 12-button keypad to 4-bit code converter

You've probably seen 12-button keypads in many different places, like on a telephone or at an ATM machine as shown in Figure 2.51. The first row has buttons 1, 2, and 3, the second row has 4, 5, and 6, the third row has 7, 8, and 9, and the last row has *, 0, and #. The outputs of such a keypad consist of seven signals—one for each of the four rows (`r1`, `r2`, `r3`, and `r4`), and one for each of the three columns (`c1`, `c2`, and `c3`). Pushing a particular button causes exactly two outputs to become `1`, corresponding to the row and column of that button. So pushing button "1" causes `r1=1` and `c1=1`, while pushing button "#" causes `r4=1` and `c3=1`.

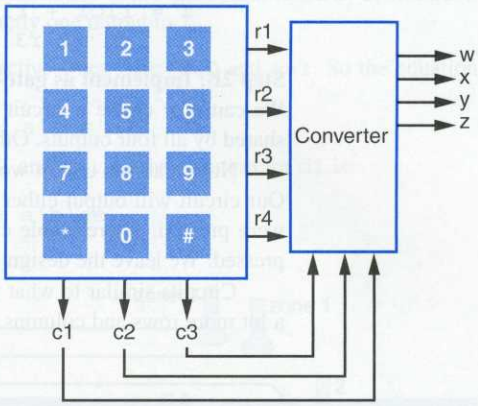


Figure 2.51 Converter for 12-button keypad.

A useful circuit converts the seven signals from the keypad into a 4-bit output `wxyz` that indicates which button is pressed, as in Figure 2.51; the output may be connected to a microprocessor or other device. Buttons "0" to "9" should be encoded as `0000` through `1001` (0 through 9 in binary), respectively. Button "*" should be encoded as `1010`, and # as `1011`. `1111` will mean that no button is pressed. Assume for now that only one button can ever be pressed at a given time.

For this example, starting from equations is a more natural choice than a truth table, although we used an informal table (not a truth table) to help us determine the equations.

Step 1: Capture behavior. We could capture the behavior for `w`, `x`, `y`, and `z` using a truth table, with the seven inputs on the left side of the table, and the four outputs on the right side, but that table would have $2^7 = 128$ rows, and most of those rows would correspond merely to multiple buttons being pressed. Let's try instead to capture the functions using equations. The informal table in Table 2.7 might help us get started.

TABLE 2.7 Informal table for the 12-button keypad to 4-bit code converter.

Button	Signals		4-bit code outputs			
			w	x	y	z
1	r1	c1	0	0	0	1
2	r1	c2	0	0	1	0
3	r1	c3	0	0	1	1
4	r2	c1	0	1	0	0
5	r2	c2	0	1	0	1
6	r2	c3	0	1	1	0
7	r3	c1	0	1	1	1
8	r3	c2	1	0	0	0
9	r3	c3	1	0	0	1
*	r4	c1	1	0	1	0
0	r4	c2	0	0	0	0
#	r4	c3	1	0	1	1
(none)			1	1	1	1

Guided by this table, we can create equations for each of the four outputs, as follows:

```
w = r3c2 + r3c3 + r4c1 + r4c3 + r1'r2'r3'r4'c1'c2'c3'
x = r2c1 + r2c2 + r2c3 + r3c1 + r1'r2'r3'r4'c1'c2'c3'
```

```
y = r1c2 + r1c3 + r2c3 + r3c1 + r4c1 + r4c3 +
    r1'r2'r3'r4'c1'c2'c3'
z = r1c1 + r1c3 + r2c2 + r3c1 + r3c3 + r4c3 +
    r1'r2'r3'r4'c1'c2'c3'
```

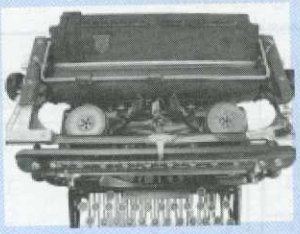
Step 2B: Implement as gate-based circuit. (We skip substep 2A, as we already created equations). We can now create a circuit for each output. Obviously, the last term of each equation could be shared by all four outputs. Other terms could be shared too (like `r2c3`).

Note that this circuit would not work well if multiple buttons can be pressed simultaneously. Our circuit will output either a valid or invalid code in that situation, depending on which buttons were pressed. A preferable circuit would treat multiple buttons being pressed as no button being pressed. We leave the design of that circuit as an exercise.

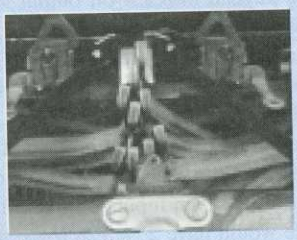
Circuits similar to what we designed above exist in computer keyboards, except that there are a lot more rows and columns.

▶ *SLOW DOWN! THE QWERTY KEYBOARD*

Inside a standard computer keyboard is a small microprocessor and a ROM. The microprocessor detects which key is being pressed, looks up the 8-bit code for that key (much like the 12-button keypad in Example 2.26) from the ROM, and sends that code to the computer. There's an interesting story behind the way the keys are arranged in a standard PC keyboard, which is known as a QWERTY keyboard because those are the keys that begin the top left row of letters. The QWERTY arrangement was made in the era of typewriters (shown in the picture below), which, in case you haven't seen one, had each key connected to an arm that would swing up and press an ink ribbon against paper.



Keys connected to arms.



Arms stuck!

An annoying problem with typewriters was that arms would often get jammed side-by-side up near the paper if you typed too fast—like too many people getting jammed side-by-side while they all try to simultaneously walk through a doorway. So typewriter keys were arranged in the QWERTY arrangement to *slow down* typing by *separating* common letters, since slower typing reduced the occurrences of jammed keys. When PCs were invented, the QWERTY arrangement was the natural choice for PC keyboards, as people were accustomed to that arrangement. Some say the differently-arranged Dvorak keyboard enables faster typing, but that type of keyboard isn't very common, as people are just too accustomed to the QWERTY keyboard.

Example 2.27 Sprinkler valve controller

Automatic lawn sprinkler systems use a digital system to control the opening and closing of water valves. A sprinkler system usually supports several different zones, such as the backyard, left side yard, right side yard, front yard, etc. Only one zone's valve can be opened at a time in order to maintain enough water pressure in the sprinklers in that zone. Suppose a sprinkler system supports up to 8 zones. Typically, a sprinkler system is controlled by a small, inexpensive microprocessor executing a program that opens each valve only at specific times of the day and for specific durations. Suppose the microprocessor only has 4 output pins available to control the valves, not 8 outputs as required for the 8 zones. We can instead program the microprocessor to use 1 pin to indicate whether a valve should be opened,

and use the 3 other pins to output the active zone (0, 1, ..., 7) in binary. Thus, we need to design a combinational circuit having 4 inputs, `e` (the enabler) and `a`, `b`, `c` (the binary value of the active zone), and having 8 outputs `d7`, `d6`, ..., `d0` (the valve controls), as shown in Figure 2.52. When `e=1`, the circuit should decode the 3-bit binary input by setting exactly one output to `1`.

Step 1: Capture behavior. Valve 0 should be active when `abc=000` and `e=1`. So the equation for `d0` is:

```
d0 = a'b'c'e
```

Likewise, valve 1 should be active when `abc=001` and `e=1`, so the equation for `d1` is:

```
d1 = a'b'ce
```

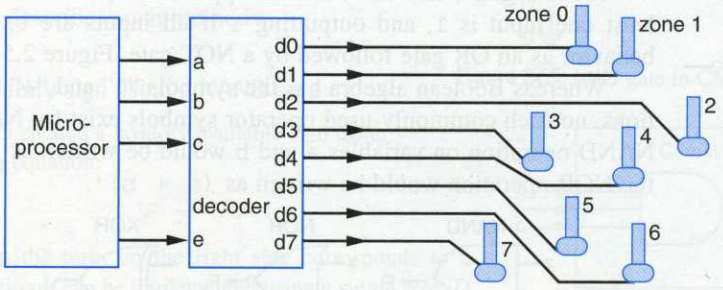


Figure 2.52 Sprinkler valve controller block diagram.

For this example, starting from equations is a more natural choice than a truth table.

The equations for the remaining outputs can be determined similarly:

```
d2 = a'bc'e
d3 = a'bce
d4 = ab'c'e
d5 = ab'ce
d6 = abc'e
d7 = abce
```

Step 2A: Create equations. Equations were already created.

Step 2B: Implement as a gate-based circuit. The circuit implementing the equations is shown in Figure 2.53. The circuit is actually a commonly used component known as a *decoder with enable*. Decoders as a building block will be introduced in an upcoming section.

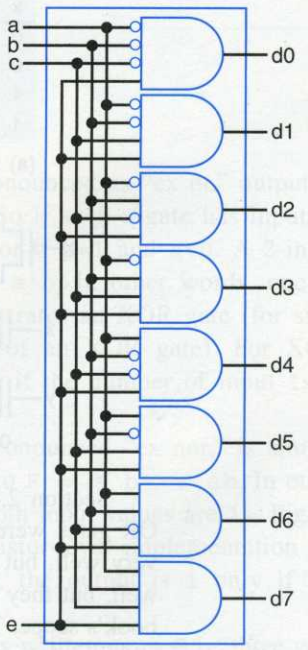
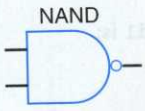


Figure 2.53 Sprinkler valve controller circuit (actually a 3x8 decoder with enable).

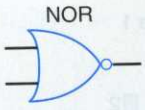
▶ 2.8 MORE GATES

Designers use several other types of gates beyond just AND, OR, and NOT. Those gates include NAND, NOR, XOR, and XNOR.

NAND & NOR



A ***NAND*** gate (short for "not AND") has the opposite output of an AND gate, outputting a `0` only when all inputs are `1`, and outputting a `1` otherwise (meaning at least one input is `0`). A NAND gate has the same behavior as an AND gate followed by a NOT gate. Figure 2.54(a) illustrates a NAND gate.



A ***NOR*** gate ("not OR") has the opposite output as an OR gate, outputting a `0` if at least one input is `1`, and outputting `1` if all inputs are `0`. A NOR gate has the same behavior as an OR gate followed by a NOT gate. Figure 2.54(b) shows a NOR gate.

Whereas Boolean algebra has the symbols "*" and "+" for the AND and OR operations, no such commonly-used operator symbols exist for NAND and NOR. Instead, the NAND operation on variables `a` and `b` would be written as `(a*b)'` or just `(ab)'`, and the NOR operation would be written as `(a + b)'`.

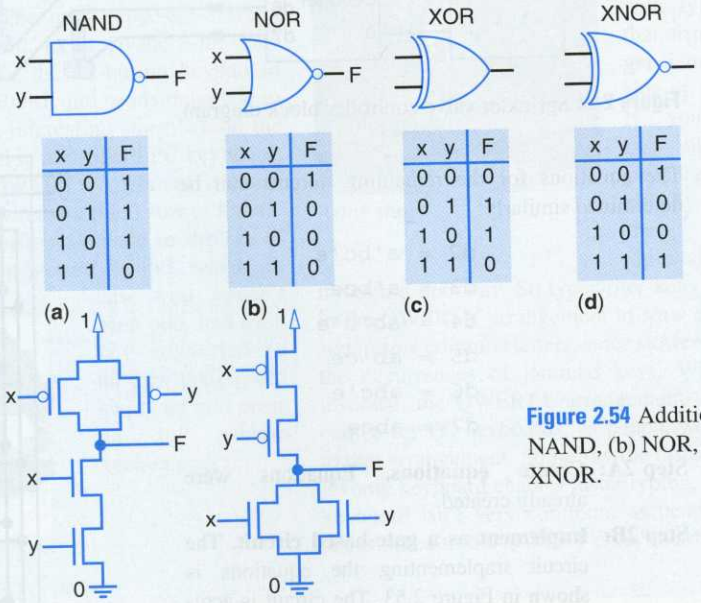


Figure 2.54 Additional gates: (a) NAND, (b) NOR, (c) XOR, (d) XNOR.

Section 2.4 warned that the shown CMOS transistor implementations of AND and OR gates were not realistic. The reason is because pMOS transistors don't conduct `0`s very well, but they conduct `1`s just fine. Likewise, nMOS transistors don't conduct `1`s well, but they conduct `0`s just fine. The reasons for these asymmetries are beyond this book's scope. The implications are that the AND and OR gates in Figure 2.8 are not feasible, as they rely on pMOS transistors to conduct `0`s (but pMOS conducts `0`s poorly) and nMOS transistors to conduct `1`s (but nMOS conducts `1`s poorly). However, if we switch the locations of power and ground in the AND and OR circuits of Figure 2.8, the results are the NAND and NOR gate circuits shown in Figure 2.54(a) and Figure 2.54(b).

An AND gate can still be implemented in CMOS, by appending a NOT gate at the output of a NAND gate (NAND followed by NOT computes AND), as in Figure 2.55. Likewise, an OR gate is implemented by appending a NOT gate at the output of a NOR gate. Those gates are obviously slower than NAND and NOR gates due to the extra NOT gate at the output. Fortunately, straightforward methods can convert any AND/OR/NOT circuit to a NAND-only circuit, or to a NOR-only circuit. Section 7.2 describes such methods.

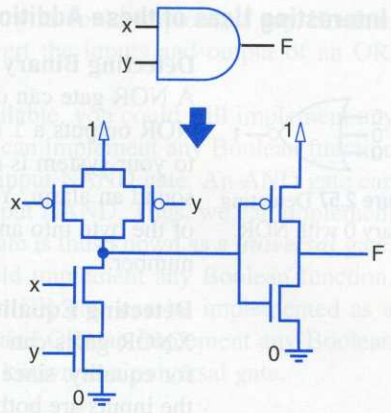


Figure 2.55 AND gate in CMOS.

Example 2.28 Aircraft lavatory sign using a NAND gate

Example 2.15 created a lavatory available sign using the following equation:

```
S = (abc)'
```

Noticing that the term on the right side corresponds to a NAND, the circuit can be implemented using a single NAND gate, as shown in Figure 2.56.

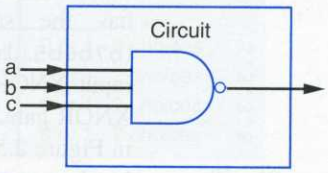
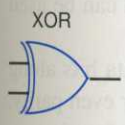
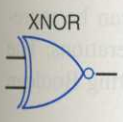


Figure 2.56 Circuit using NAND.

XOR & XNOR



A 2-input XOR gate, short for "exclusive or" and pronounced as "ex or," outputs a `1` if *exactly* one of the two inputs has a value of `1`. So if such a gate has inputs `a` and `b`, then the output `F` will be `1` if `a=1` and `b=0`, or if `b=1` and `a=0`. A 2-input XOR gate is equivalent to the function `F = ab' + a'b`. In other words, one or the other input is `1`, but not both. Figure 2.54(c) illustrates an XOR gate (for simplicity, we omit the transistor-level implementation of an XOR gate). For XOR gates with three or more inputs, the output is `1` only if the number of input `1`s is odd.



An XNOR gate, short for "exclusive nor" and pronounced "ex nor," is simply the opposite of XOR. A 2-input XNOR is equivalent to `F = a'b' + ab`. In other words, `F` will be `1` if both input values are `0`s, or if both input values are `1`s. Figure 2.54(d) illustrates an XNOR gate, omitting the transistor-level implementation for simplicity. For XNOR gates with 3 or more inputs, the output is `1` only if the number of input `1`s is even.

The XOR operation applied to variables `a` and `b` is written as $a \oplus b$; three variables would be $a \oplus b \oplus c$. There is no commonly used symbol for XNOR; instead, XNOR would be written as $(a \oplus b)'$

Interesting Uses of these Additional Gates

Detecting Binary 0 Using NOR

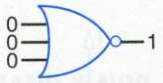


Figure 2.57 Detecting binary 0 with NOR.

A NOR gate can detect the situation of an n-bit binary number being equal to 0, because NOR outputs a `1` only when all n inputs are `0`. For example, suppose a byte (8-bit) input to your system is counting down from 99 to 0, and when the byte reaches 0, you wish to sound an alarm. You can detect the byte being equal to 0 by simply connecting the 8 bits of the byte into an 8-input NOR gate. Figure 2.57 shows such detection for a 3-bit binary number.

Detecting Equality Using XNOR

XNOR gates can be used to compare two n-bit data items for equality, since a 2-input XNOR outputs a `1` only when the inputs are both `0` or are both `1`. For example, suppose a byte input `A` (`a7a6a5...a0`) to your system is counting down from 99, and you want to sound an alarm when `A` has the same value as a second byte input `B` (`b7b6b5...b0`). You can detect such equality using eight 2-input XNOR gates, by connecting `a0` and `b0` to the first XNOR gate, `a1` and `b1` to the second XNOR gate, etc., as in Figure 2.58. Each XNOR gate indicates whether the bits in that particular position are equal. ANDing all the XNOR outputs indicates whether every position is equal.

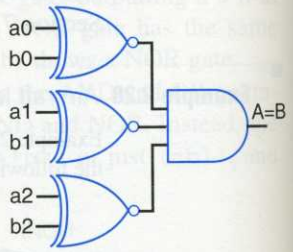


Figure 2.58 Detecting equality with 2-input XNORs.

Generating and Detecting Parity Using XOR

An XOR gate can be used to generate a parity bit for a set of data bits (see Example 2.19). XORing the data bits results in a `1` if there is an odd number of `1`s in the data, so XOR computes the correct parity bit for even parity, because the XOR's output `1` would make the total number of `1`s even. Notice that the truth table for generating an even parity bit in Table 2.3 does in fact represent a 3-bit XOR. Likewise, an XNOR gate can be used to generate an odd parity bit.

XOR can also be used to detect proper parity. XORing the incoming data bits along with the incoming parity bit will yield `1` if the number of `1`s is odd. Thus, for even parity, XOR can be used to indicate that an error has occurred, since the number of `1`s is supposed to be even. XNOR can be used to detect an error when odd parity is used.

Completeness of NAND and of NOR

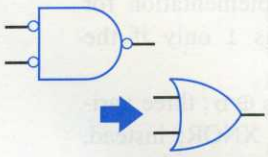


Figure 2.59 AND and NOT gates can form OR.

It should be fairly obvious that if you have AND gates, OR gates, and NOT gates, you can implement any Boolean function. This is because a Boolean function can be represented as a sum of products, which consists only of AND, OR, and NOT operations. The set of AND, OR, and NOT gates are thus ***complete*** with respect to implementing Boolean functions.

What might be slightly less obvious is that if you had only AND and NOT gates, you could still implement any Boolean function. Why? Here's a simple explanation—to obtain an OR, just put NOT gates at the inputs and output of an AND, as in Figure 2.59 (showing NOT gates as inversion bubbles). The resulting output computes OR, because `F = (a'b')' = a'' + b''` (by DeMorgan's Law) `= a + b`.

Likewise, if you had only OR and NOT gates, you could implement any Boolean function. To obtain an AND, you could simply invert the inputs and output of an OR, since `F = (a'+b')' = a''*b'' = ab`.

It follows that if you *only* had NAND gates available, you could still implement any Boolean function. Why? We just saw above that we can implement any Boolean function using just NOT and AND gates. A NOT gate is a 1-input NAND gate. An AND gate can be implemented as a NAND gate followed by a 1-input NAND. Thus, we can implement any Boolean function using just NAND. A NAND gate is thus known as a ***universal*** gate.

Likewise, if you had only NOR gates, you could implement any Boolean function, because a NOT gate is a 1-input NOR gate, and an OR gate can be implemented as a NOR gate followed by a 1-input NOR. Since NOT and OR can implement any Boolean function, so can just NOR. A NOR gate is thus also known as a universal gate.

Number of Possible Logic Gates

Having seen several different types of basic 2-input logic gates (AND, OR, NAND, NOR, XOR, XNOR), one might wonder how many possible 2-input logic gates exist. That question is the same as asking how many Boolean functions exist for two variables. To answer the question, first note that a two-variable function's truth table will have $2^2 = 4$ rows. For each row, the function could output one of two possible values (`0` or `1`). Thus, as illustrated in Figure 2.60, there are $2 * 2 * 2 * 2 = 2^4 = 16$ possible functions.

a	b	F		
0	0	0 or 1	2 choices	2
0	1	0 or 1	2 choices	* 2
1	0	0 or 1	2 choices	* 2
1	1	0 or 1	2 choices	* 2

$2^4 = 16$ possible functions

Figure 2.60 Counting the number of possible Boolean functions of two variables.

Figure 2.61 lists all 16 such functions. The figure labels the 6 familiar functions (AND, OR, NAND, NOR, XOR, XNOR). Some of the other functions are `0`, `a`, `b`, `a'`, `b'`, and `1`. The remaining functions are uncommon functions, but each could be useful for some application. Thus, logic gates may not be built to represent those functions, but instead those functions, when needed, might be built as a circuit of the basic logic gates.

a	b	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	f13	f14	f15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		0	a AND b		a		b	a XOR b	a OR b	a NOR b	a XNOR b	b'		a'		a NAND b	1

Figure 2.61 The 16 possible Boolean functions of two variables.

A more general question of interest is how many Boolean functions exist for a Boolean function of *N* variables. This number can be determined by first noting that an

N-variable function will have 2^N rows in its truth table. Then, note that each row can output one of two possible values. Thus, the number of possible functions will be 2 * 2 * 2 *—2^N times. Therefore, the total number of functions is:

$$2^{2^N}$$

So there are: $2^{2^3} = 2^8 = 256$ possible Boolean functions of 3 variables, and $2^{2^4} = 2^{16} = 65536$ possible functions of 4 variables.

▶ 2.9 DECODERS AND MUXES

Two additional components, a decoder and a multiplexer, are also commonly used as digital circuit building blocks, though they themselves can be built from logic gates.

Decoders

A decoder is a higher-level building block commonly used in digital circuits. A ***decoder*** decodes an input n-bit binary number by setting exactly one of the decoder's 2^n outputs to `1`. For example, a 2-input decoder, illustrated in Figure 2.62(a), would have $2^2 = 4$ outputs, `d3`, `d2`, `d1`, `d0`. If the two inputs `i1i0` are `00`, `d0` would be `1` and the remaining outputs would be `0`. If `i1i0=01`, `d1` would be `1`. If `i1i0=10`, `d2` would be `1`. If `i1i0=11`, `d3` would be `1`. One and only one output of a decoder will ever be `1` at a given time, corresponding to the particular current value of the inputs, as shown in Figure 2.62(a).

The internal design of a decoder is straightforward. Consider a 2x4 decoder. Each output `d0`, `d1`, `d2`, and `d3` is a distinct function. `d0` should be `1` only when `i1=0` and `i0=0`, so `d0 = i1'i0'`. Likewise, `d1=i1'i0`, `d2=i1i0'`, and `d3=i1i0`. Thus, we build the decoder with one AND gate for each output, connecting the true or complemented values of `i1` and `i0` to each gate, as shown in Figure 2.62.

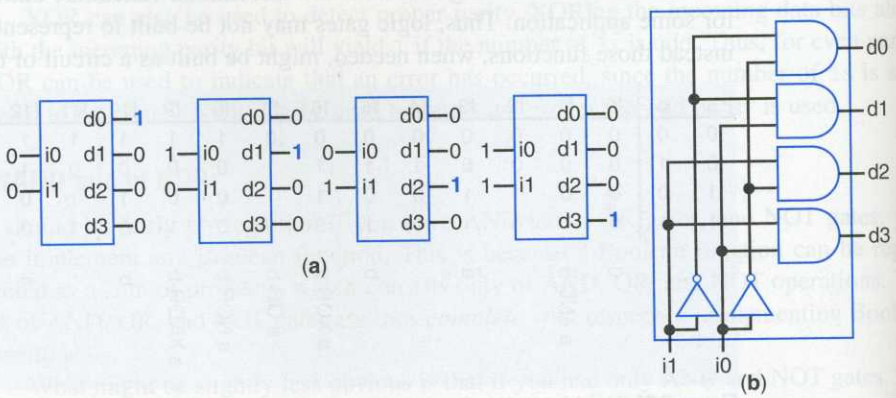


Figure 2.62 2x4 decoder: (a) outputs for possible input combinations, (b) internal design.

The internal design of a 3x8 decoder is similar: `d0=i2'i1'i0'`, `d1=i2'i1'i0`, etc.

A decoder often comes with an extra input called ***enable***. When enable is `1`, the decoder acts normally. But when enable is `0`, the decoder outputs all `0`s—no output is a `1`. The enable is useful when sometimes you don't want to activate any of the outputs. Without an enable, one output of the decoder *must* be a `1`, because the decoder has an output for every possible value of the decoder's n-bit input. We earlier created and used a decoder with enable in Figure 2.53. A block diagram and illustrated behavior of a decoder with enable appear in Figure 2.63.

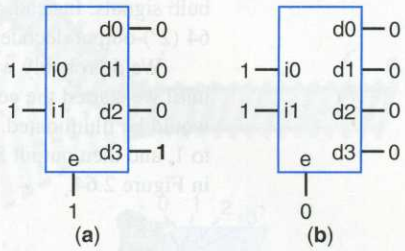


Figure 2.63 Decoder with enable: (a) `e=1`: normal decoding, (b) `e=0`: all outputs `0`.

When designing a particular system, we check whether part (or all) of the system's functionality could be carried out by a decoder. Using a decoder reduces the amount of required combinational logic design, as you'll see in Example 2.30.

Example 2.29 Basic questions about decoders

1. What would be a 2x4 decoder's output values when the inputs are `00`? *Answer:* `d0=1`, `d1=0`, `d2=0`, `d3=0`.
2. What would be a 2x4 decoder's output values when the inputs are `11`? *Answer:* `d0=0`, `d1=0`, `d2=0`, `d3=1`.
3. What input values of a 2x4 decoder cause more than one of the decoder's outputs to be `1` at the same time? *Answer:* No such input values exist. Only one of a decoder's outputs can be `1` at a given time.
4. What would the input values of a decoder be if the output values are `d0=0`, `d1=1`, `d2=0`, `d3=0`? *Answer:* The input values must be `i1=0`, `i0=1`.
5. What would the input values of a decoder be if the output values are `d0=1`, `d1=1`, `d2=0`, `d3=0`? *Answer:* This question is not valid. A decoder only has one output equal to 1 at any time.
6. How many outputs would a 5-input decoder have? *Answer:* 2^5, or 32.
7. A 2-input decoder with enable having inputs `i1=0`, `i0=1`, and `e=0`, would have what output values? *Answer:* All outputs would be `0`.

Example 2.30 New Year's Eve countdown display

A New Year's Eve countdown display could make use of a decoder. The display may have 60 light bulbs going up a tall pole, as in Figure 2.64. We want one light per second to illuminate (with the previous one turning off), starting from bulb 59 at the bottom of the pole, and ending with bulb 0 at the top. We could use a microprocessor to count down from 59 to 0, but the microprocessor probably doesn't have 60 output pins that we could use to control each light. Our microprocessor program could instead output the numbers 59, 58, ..., 2, 1, 0 in binary on a 6-bit output port (thus outputting `111011`, `111010`, ..., `000010`, `000001`, `000000`). Assume each light bulb has a signal that illu-

minates the bulb when set to 1. Thus, the problem is to design a circuit that, for each binary number that could be input, illuminates the appropriate light bulb.

Designing a circuit using gates could be done, but would require a design for each of the 60 bulb signals. Instead, we could connect those six bits coming from the microprocessor to a 6-input, 64 (2^6)-output decoder, with decoder output d59 lighting bulb 59, d58 lighting bulb 58, etc.

We'd probably want an enable on our decoder in this example, since all the lights should be off until we started the countdown. The microprocessor would initially set enable to 0 so that no lights would be illuminated. When the 60 second countdown begins, the microprocessor would set enable to 1, and then output 59, then 58 (1 second later), then 57, etc. The final system would look like that in Figure 2.64.

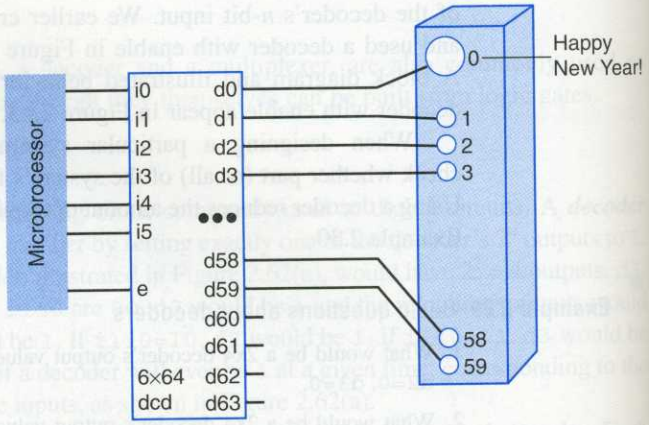


Figure 2.64 Using a 6x64 decoder to connect a microprocessor and a column of lights for a New Year's Eve display. The microprocessor sets e = 1 when the last minute countdown begins, and then counts down from 59 to 0 in binary on the pins i5..i0. Note that the microprocessor should never output 60, 61, 62, or 63 on i5..i0, and thus those outputs of the decoder go unused.

Notice that we implemented this system without having to design any gate-level combinational logic—we merely used a decoder and connected it to the appropriate inputs and outputs.

Whenever you have outputs such that exactly one of those outputs should be set to 1 based on the value of inputs representing a binary number, think about using a decoder.

Multiplexers (Muxes)

A multiplexer ("mux" for short) is another higher-level building block in digital circuits. An *M*x1 ***multiplexer*** has *M* data inputs and 1 output, and allows only one input to pass through to that output. A set of additional inputs, known as select inputs, determines which input to pass through. Multiplexers are sometimes called ***selectors*** because they select one input to pass through to the output.

A mux is like a railyard switch that connects multiple input tracks to a single output track, as shown in Figure 2.65. The switch's control lever causes the connection of the appropriate input track to the output track. Whether a train appears at the output depends on whether a train exists on the presently selected input track. For a mux, the switch's control is not a lever, but rather select inputs, which represent the desired connection in binary. Rather than a train appearing or not appearing at the output, a mux outputs a 1 or a 0 depending on whether the connected input has a 1 or a 0.

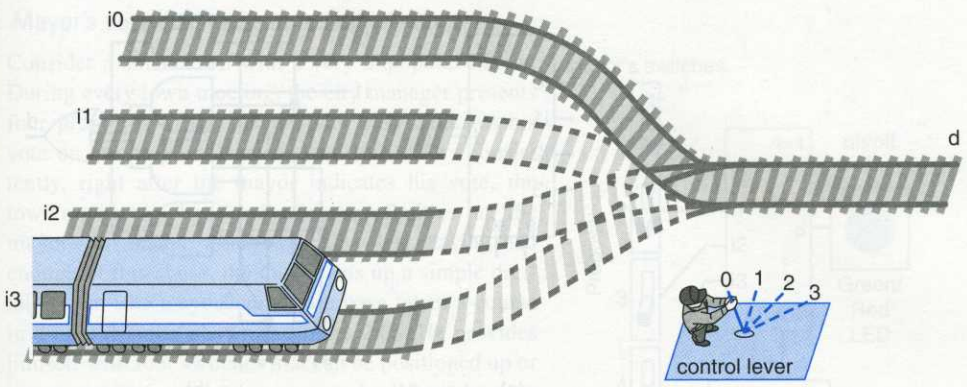


Figure 2.65 A multiplexer is like a railyard switch, determining which input track connects to the single output track, according to the switch's control lever.

A 2-input multiplexer, known as a 2x1 multiplexer, has two data inputs `i1` and `i0`, one select input `s0`, and one data output `d`, as shown in Figure 2.66(a). As shown in Figure 2.66(b), if `s0=0`, `i0`'s value passes through; if `s0=1`, `i1`'s value passes through.

The internal design of a 2x1 multiplexer is shown in Figure 2.66(c). When `s0=0`, the top AND gate outputs `1*i0=i0`, and the bottom AND gate outputs `0*i1=0`. Thus, the OR gate outputs `i0+0=i0`. So `i0` passes through as desired. Likewise, when `s0=1`, the bottom gate passes `i1` while the top gate outputs `0`, resulting in the OR gate passing `i1`.

A 4-input multiplexer, known as a 4x1 multiplexer, has four data inputs `i3`, `i2`, `i1`, and `i0`, two select inputs `s1` and `s0`, and one data output `d`. A mux *always* has just one data output, no matter how many inputs. A 4x1 mux block diagram is shown in Figure 2.67(a).

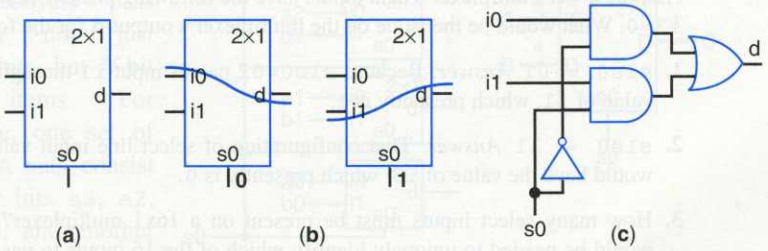


Figure 2.66 2x1 multiplexer: (a) block symbol, (b) connections for `s0=0`, and `s0=1`, and (c) internal design.

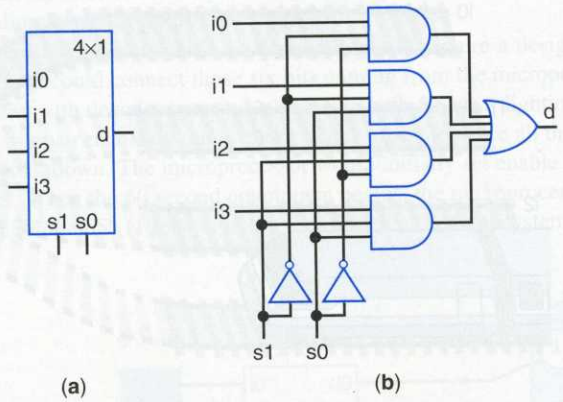


Figure 2.67 4x1 multiplexer: (a) block symbol and (b) internal design.

A common misconception is that a multiplexor is a decoder in reverse. It is not.

The internal design of a 4x1 multiplexer is shown in Figure 2.67(b). When `s1s0=00`, the top AND gate outputs `i0*1*1=i0`, the next AND gate outputs `i1*0*1=0`, the next gate outputs `i2*1*0=0`, and the bottom gate outputs `i3*0*0=0`. The OR gate outputs `i0+0+0+0=i0`. Thus, `i0` passes through, as desired. Likewise, when `s1s0=01`, the second AND gate passes `i1`, while the remaining AND gates all output `0`. When `s1s0=10`, the third AND gate passes `i2`, and the other AND gates output `0`. When `s1s0=11`, the bottom AND gate passes `i3`, and the other AND gates output `0`. For any value on `s1s0`, only 1 AND gate will have two `1`s for its select inputs and will thus pass its data input; the other AND gates will have at least one `0` for its select inputs and will thus output `0`.

An 8x1 multiplexer would have 8 data inputs (`i7...i0`), 3 select inputs (`s2`, `s1`, and `s0`), and one data output. More generally, an Mx1 multiplexer has M data inputs, $\log_2(M)$ select inputs, and one data output. Remember, a multiplexer always has just one output.

Example 2.31 Basic questions about multiplexers

Assume a 4x1 multiplexer's data inputs have the following present values: `i0=1`, `i1=1`, `i2=0`, and `i3=0`. What would be the value on the multiplexer's output `d` for the following select input values?

1. `s1s0 = 01`. *Answer:* Because `s1s0=01` passes input `i1` through to `d`, then `d` would have the value of `i1`, which presently is `1`.
2. `s1s0 = 11`. *Answer:* That configuration of select line input values passes `i3` through, so `d` would have the value of `i3`, which presently is `0`.
3. How many select inputs must be present on a 16x1 multiplexer? *Answer:* Four select inputs would be needed to uniquely identify which of the 16 inputs to pass through to the output since $\log_2(16)=4$.
4. How many select lines are there on a 4x2 multiplexer? *Answer:* This question is not valid—there is no such thing as a 4x2 multiplexer. A multiplexer has exactly one output.
5. How many data inputs are there on a multiplexer having five select inputs? *Answer:* Five select inputs can uniquely identify one of $2^5=32$ inputs to pass through to the output.

Example 2.32 Mayor's vote display using a multiplexer

Consider a small town with a very unpopular mayor. During every town meeting, the city manager presents four proposals to the mayor, who then indicates his vote on the proposal (approve or deny). Very consistently, right after the mayor indicates his vote, the town's citizens boo and shout profanities at the mayor—no matter which way he votes. Having had enough of this abuse, the mayor sets up a simple digital system (the mayor happens to have taken a course in digital design), shown in Figure 2.68. He provides himself with four switches that can be positioned up or down, outputting `1` or `0`, respectively. When the time comes during the meeting for him to vote on the first proposal, he places the first switch either in the up (accept) or down (deny) position—but nobody else can see the position of the switch. When the time comes to vote on the second proposal, he votes on the second proposal by placing the second switch up or down. And so on. When he has finished casting all his votes, he leaves the meeting and heads home. With the mayor gone, the city manager powers up a large green/red light. When the input to the light is `0`, the light lights up red. When the input is `1`, the light lights up green. The city manager controls two "select" switches that can route any of the mayor's switch outputs to the light, and so the manager steps through each configuration of the switches, starting with configuration `00` (and calling out "The mayor's vote on this proposal is ..."), then `01`, then `10`, and finally `11`, causing the light to light either green or red for each configuration depending on the positions of the mayor's switches. The system can easily be implemented using a 4x1 multiplexer, as shown in Figure 2.68.

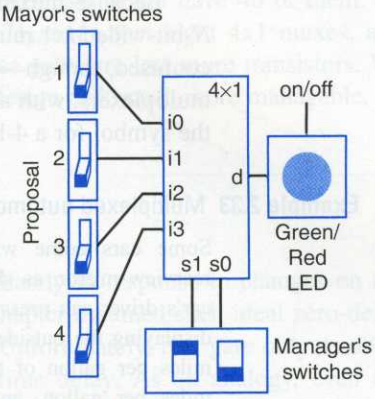


Figure 2.68 Mayor's vote display system implemented using a 4x1 mux.

N-bit *M*x1 Multiplexer

Muxes are often used to selectively pass through not just single bits, but *N*-bit data items. For example, one set of inputs `A` may consist of four bits `a3`, `a2`, `a1`, `a0`, and another set of inputs `B` may also consist of four bits `b3`, `b2`, `b1`, `b0`. We want to multiplex those inputs to a four-bit output `C`, con-

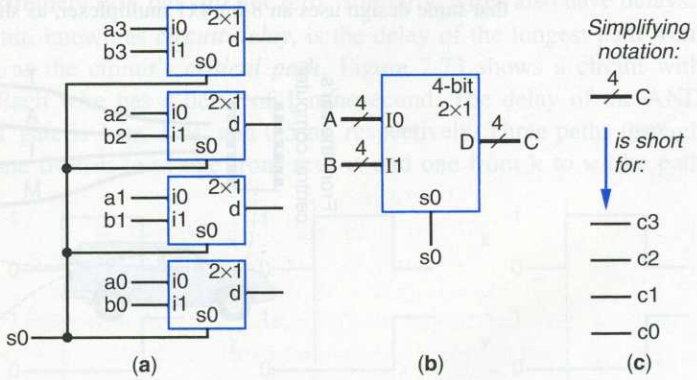


Figure 2.69 4-bit 2x1 mux: (a) internal design using four 2x1 muxes for selecting among 4-bit data items A or B, and (b) block diagram of a 4-bit 2x1 mux component. (c) The block diagram uses a common simplifying notation, using one thick wire with a slanted line and the number 4 to represent 4 single wires.

sisting of c3, c2, c1, c0. Figure 2.69(a) shows how to accomplish such multiplexing using four 2x1 muxes.

Because muxing data is so common, another common building block is that of an *N*-bit-wide *M*x1 multiplexer. So in our example, we would use a 4-bit 2x1 mux. Don't get confused, though—an *N*-bit *M*x1 multiplexer is really just the same as *N* separate *M*x1 multiplexers, with all those muxes sharing the same select inputs. Figure 2.69(b) provides the symbol for a 4-bit 2x1 mux.

Example 2.33 Multiplexed automobile above-mirror display

Some cars come with a display above the rearview mirror, as shown in Figure 2.70. The car's driver can press a button to select among displaying the outside temperature, the average miles per gallon of the car, the instantaneous miles per gallon, and the approximate miles remaining until the car runs out of gasoline. Assume the car's central computer sends the data to the display as four 8-bit binary numbers, T (the temperature), A (average mpg), I (instantaneous mpg), and M (miles remaining). T consists of 8 bits: t7, t6, t5, t4, t3, t2, t1, t0. Likewise for A, I, and M. Assume the display system has two additional inputs x and y, which always change according to the following sequence—00, 01, 10, 11—whenever the button is pressed (we'll see in a later chapter how to create such a sequence). When xy=00, we want to display T. When xy=01, we want to display A. When xy=10, we want to display I, and when xy=11, we want to display M. Assume the outputs D go to a display that knows how to convert the 8-bit binary number on D to a human-readable displayed number like that in Figure 2.70.

Figure 2.70 Above-mirror display.

We can design the display system using eight 4x1 multiplexers. A simpler representation of that same design uses an 8-bit 4x1 multiplexer, as shown in Figure 2.71.

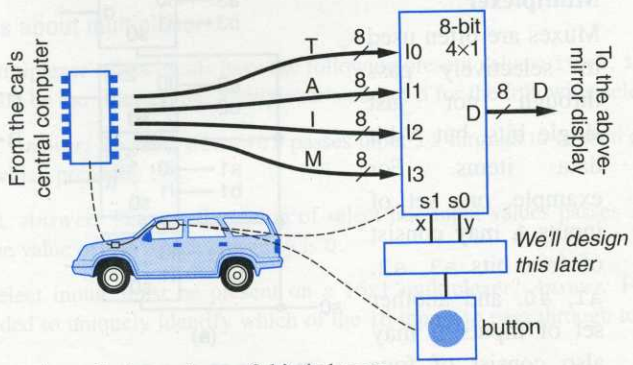


Figure 2.71 Above-mirror display using an 8-bit 4x1 mux.

Notice how many wires must be run from the car's central computer, which may be under the hood, to the above-mirror display—8*4=32 wires. That's a lot of wires. We'll see in a later chapter how to reduce the number of wires.

Notice in the previous example how simple a design can be when we can utilize higher-level building blocks. If we had to use regular 4x1 muxes, we would have 8 of them, and lots of wires drawn. If we had to use gates, we would have 40 of them. Of course, underlying our simple design in Figure 2.71 are in fact eight 4x1 muxes, and underlying those are 40 gates. And underlying those gates are lots more transistors. We see that the higher-level building blocks make our design task much more managable.

▶ 2.10 ADDITIONAL CONSIDERATIONS

Nonideal Gate Behavior—Delay

Ideally, a logic gate's output would change immediately in response to changes on the gate's inputs. The timing diagrams earlier in this chapter assumed such ideal zero-delay gates, as shown in Figure 2.72(a) for an OR gate. Unfortunately, real gate outputs don't change immediately, but rather after some short time delay. As an analogy, even the fastest automobiles can't go from 0 to 60 miles per hour in 0 seconds. A gate's delay is due in part to the fact that transistors don't switch from nonconducting to conducting (or vice versa) immediately—it takes some time for electrons to accumulate in the channel of an nMOS transistor, for example. Furthermore, electric current travels at the speed of light, which, while extremely fast, is still not infinitely fast. Additionally, wires aren't perfect and can slow down electric current because of "parasitic" characteristics like capacitance and inductance.

For example, the timing diagram in Figure 2.72(a) shows how an OR gate's output would ideally change from `0` to `1` when an input becomes `1`. Figure 2.72(b) depicts how the output would actually rise slowly from 0 Volts, representing logic `0`, to its higher voltage of 1.8 Volts, representing logic `1`.

The maximum time for a gate's output to change (from `0` to `1` or `1` to `0`) in response to an input change is the gate's ***delay***. Delays for modern CMOS gates can be less than 1 nanosecond, which is extremely fast, but still not zero. Similarly, wires also have delays.

The delay of a circuit, known as ***circuit delay***, is the delay of the longest path from input to output, known as the circuit's ***critical path***. Figure 2.73 shows a circuit with sample delays shown. Each wire has a delay of 1 nanosecond. The delay of the AND gate, OR gate, and NOT gate is 1 ns, 1 ns, and 0.5 ns, respectively. Three paths through the circuit are shown, one from `t` to `w`, one from `s` to `w`, and one from `k` to `w` (the path

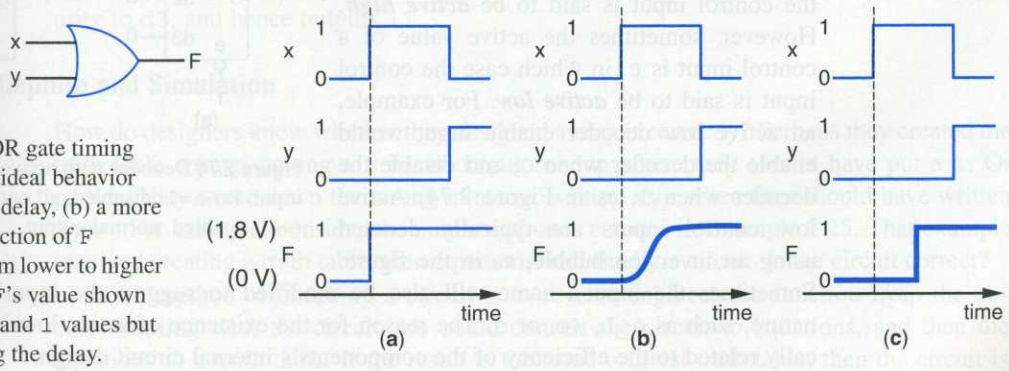


Figure 2.72 OR gate timing diagram: (a) ideal behavior without gate delay, (b) a more realistic depiction of `F` changing from lower to higher voltage, (c) `F`'s value shown with logic `0` and `1` values but incorporating the delay.

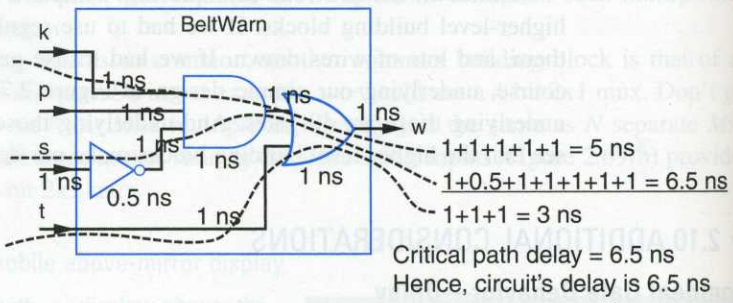


Figure 2.73 Computing the delay of the *BeltWarn* circuit from Example 2.8.

from p to w is the same length and thus not shown). The path from k to w passes through a wire (1 ns), the AND gate (1 ns), another wire (1 ns), the OR gate (1 ns), and finally the wire to the output (1 ns), for a total path delay of 5 ns. The path from s to w passes through a wire (1 ns), the NOT gate (0.5 ns), a wire (1 ns), the AND gate (1 ns), a wire (1 ns), the OR gate (1 ns), and finally the wire to the output (1 ns), for a path delay of 6.5 ns. Finally, the path from t to w passes through a wire (1 ns), the OR gate (1 ns), and a wire (1 ns), for a path delay of 3 ns. The path from s to w is thus the critical path, and hence the circuit's delay is said to be 6.5 ns. Even though the output would change in less than 6.5 ns in response to t's input changing, such information is not usually considered by digital designers; a designer using this circuit should expect to have to wait 6.5 ns for the output to change in response to *any* change on the inputs.

Active Low Inputs

Component inputs can generally be divided into two types. ***Control inputs*** influence the behavior of the component, such as the two select inputs of a 4x1 mux, or the enable input of a 3x8 decoder. In contrast, ***data inputs*** flow through the component, such as the 4 data inputs of a 4x1 mux, the 3 data inputs of a 3x8 decoder, or the inputs of any logic gate. Some control inputs involve the notion of being ***active***—when the input is at one of its two levels, the input is carrying out its purpose; at the other level, the input is inactive.

For example, the enable input of a decoder is *active* when its value is set to enable the decoder. Normally the active value of a control input is 1, in which case the control input is said to be ***active high***. However, sometimes the active value of a control input is 0, in which case the control input is said to be ***active low***. For example, an active low decoder enable input would enable the decoder when 0, and disable the decoder when 1, as in Figure 2.74. Active low control inputs are typically denoted using an inversion bubble, as in the figure. Sometimes the input's name will also be modified to suggest the input's active low nature, such as e_L, /e, or $\overline{e}$. The reason for the existence of active low inputs is typically related to the efficiency of the components's internal circuit design.

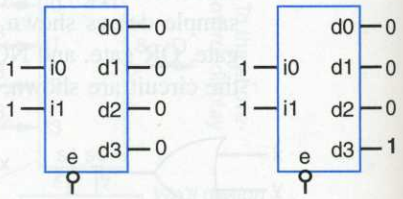


Figure 2.74 Decoder with active low enable input: (a) e=1: disabled, all outputs 0, (b) e=0: enabled, normal output.

When discussing the behavior of a component, designers will often use the term ***assert*** to mean setting a control input to the value that activates the associated operation. Thus, we might say that one must "assert" the enable input of a decoder to enable the decoder's outputs to be active. Using the term *assert* avoids possible confusion that could occur when some control inputs are active-high and others are active-low.

Demultiplexers and Encoders

Two additional components, demultiplexers and encoders, can also be considered combinational building blocks. However, those components are far less commonly used than their counterparts of multiplexers and decoders. Nevertheless, for completeness, we'll briefly introduce those additional components here. You may notice throughout this book that demultiplexers and encoders don't appear in many examples, if in any examples at all.

Demultiplexer

A demultiplexer has roughly the opposite functionality of a multiplexer. Specifically, a 1xM ***demultiplexer*** has one data input, and based on the values of $\log_2(M)$ select lines, passes that input through to one of M outputs. The other outputs stay `0`.

Encoder

An ***encoder*** has the opposite functionality of a decoder. Specifically, an n x log2(n) encoder has n inputs and $\log_2(n)$ outputs. Of the n inputs, exactly one is assumed to be `1` at any given time (such would be the case if the input consisted of a sliding or rotating switch with n possible positions, for example). The encoder outputs a binary value over the $\log_2(n)$ outputs, indicating which of the n inputs was a `1`. For example, a 4x2 encoder would have four inputs `d3`, `d2`, `d1`, `d0`, and two outputs `e1`, `e0`. For an input `0001`, the output is `00`. `0010` yields `01`, `0100` yields `10`, and `1000` yields `11`. In other words, `d0=1` results in an output of `0` in binary, `d1=1` results in an output of `1` in binary, `d2=1` results in an output of 2 in binary, and `d3=1` results in an output of 3 in binary.

A ***priority encoder*** has similar behavior, but handles situations where more than one input is `1` at the same time. A priority encoder gives priority to the highest input that is a `1`, and outputs the binary value of that input. For example, if a 4x2 priority encoder has inputs `d3` and `d1` both equal to `1` (so the inputs are `1010`), the priority encoder gives priority to `d3`, and hence outputs `11`.

Schematic Capture and Simulation

How do designers know whether they designed a circuit correctly? Perhaps they created the truth table wrong, putting a `0` in an output column where they should have put a `1`. Or perhaps they wrote down the wrong minterm, writing `xyz` when they should have written `xyz'`. For example, consider the number-of-ones counter in Example 2.25. That example involved creating a truth table, then equations, and finally a circuit. Is the circuit correct?

One method of checking a circuit is to reverse engineer the function from the circuit—starting with the circuit, we could convert the circuit to equations, and then the equations to a truth table. If the result is the same original truth table, then the circuit is

likely to be correct. However, sometimes designers start with an equation rather than a truth table, as in Example 2.24. A designer can reverse engineer the circuit to an equation, but that equation may be different than the original equation, especially if the designer algebraically manipulated the original equation when designing the circuit. Furthermore, checking that two equations are equivalent may require converting to canonical form (sum-of-minterms), which may result in huge equations if the function has a large number of inputs.

In fact, even if a designer didn't make any mistakes in converting a mental understanding of the desired function into a truth table or equation, how does the designer know that the original understanding was correct?

A commonly used method for checking that a circuit works as expected is called simulation. ***Simulation*** of a circuit is the process of providing sample inputs to the circuit and running a computer program that computes the circuit's output for the given inputs. A designer can then check that the output matches what is expected. The computer program that performs simulation is called a ***simulator***.

To use simulation to check a circuit, a designer must describe the circuit using a method that enables computer programs to read the circuit. One method of describing a circuit is to draw the circuit using a schematic capture tool. A ***schematic capture tool*** allows a user to place logic gates on a computer screen and to draw wires connecting those gates. The tool allows users to save their circuit drawings as computer files. All the circuit drawings in this chapter have represented examples of schematics—for example, the circuit drawing in Figure 2.62(b), which showed a 2x4 decoder, was an example of a schematic. Figure 2.75 shows a schematic for the same design, drawn using a popular commercial schematic capture tool. Schematic capture is used not only to capture circuits for simulator tools, but also for tools that map our circuits to physical implementations, which will be discussed in Chapter 7.

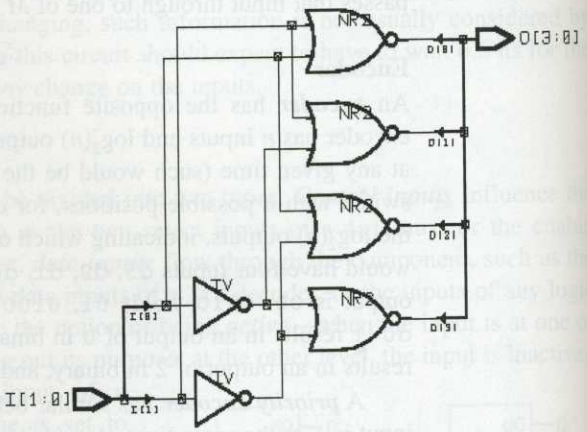


Figure 2.75 Display snapshot of a commercial schematic capture tool.

Once a designer has created a circuit using schematic capture, the designer must provide the simulator with a set of inputs that will be used to check for proper output. One way of providing the inputs is by drawing waveforms for the circuit's inputs. An input's ***waveform*** is a line that goes from left to right, representing the value of the input as time proceeds to the right. The line is drawn high to represent `1` and low to represent `0`

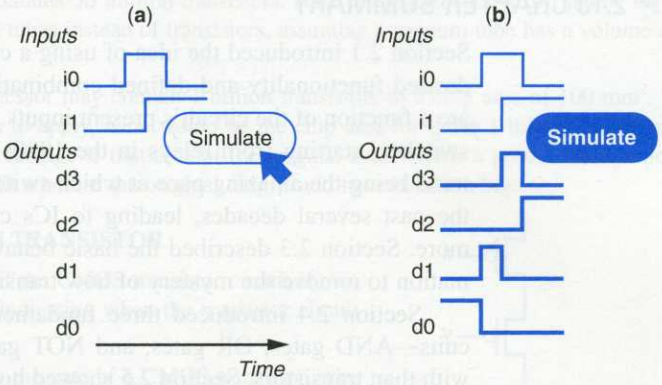


Figure 2.76 Simulation: (a) begins with us defining the inputs signal over time, (b) automatically generates the output waveforms when we ask the simulator to simulate the circuit.

for periods of time, as shown in Figure 2.76(a). After a designer is satisfied with the input waveforms, the designer instructs the simulator to simulate the circuit for the given input waveforms. The simulator determines what the circuit outputs would be for each unique combination of inputs, and generates waveforms for the outputs, as illustrated in Figure 2.76(b). The designer can then check that the output waveforms match the output values that are expected. Such checking can be done visually, or by providing certain checking statements (often called *assertions*) to the simulator.

Simulation still does not guarantee that a circuit is correct, but rather increases a designer's *confidence* that the circuit is correct.

▶ 2.11 COMBINATIONAL LOGIC OPTIMIZATIONS AND TRADEOFFS (SEE SECTION 6.2)

The earlier sections in this chapter described how to create basic combinational circuits. This section physically appears in this book as Section 6.2, and describes how to make those circuits better (smaller, faster, etc.)—namely, how to make optimizations and tradeoffs. One use of this book involves studying combinational logic optimizations and tradeoffs immediately after studying basic combinational logic design, meaning covering that section now (as Section 2.11). An alternative use of the book studies that section later (as Section 6.2), after also studying basic sequential design, datapath components, and register-transfer level design—namely, after Chapters 3, 4, and 5.

▶ 2.12 COMBINATIONAL LOGIC DESCRIPTION USING HARDWARE DESCRIPTION LANGUAGES (SEE SECTION 9.2)

Hardware description languages (HDLs) allow designers to describe their circuits using a textual language rather than as circuit drawings. This section introduces the use of HDLs to describe combinational logic. The section physically appears in the book as Section 9.2. One use of this book studies HDLs now (as Section 2.12), immediately after studying basic combinational logic. An alternative use of the book studies HDLs later (as Section 9.2), after mastery of basic combinational, sequential, and register-transfer level design.

▶ 2.13 CHAPTER SUMMARY

Section 2.1 introduced the idea of using a custom digital circuit to implement a system's desired functionality and defined combinational logic as a digital circuit whose outputs are a function of the circuit's present inputs. Section 2.2 provided a brief history of digital switches, starting from relays in the 1930s to today's CMOS transistors, with the main trend being the amazing pace at which switch size and delay have continued to shrink for the past several decades, leading to ICs capable of containing a billion transistors or more. Section 2.3 described the basic behavior of a CMOS transistor, just enough information to remove the mystery of how transistors work.

Section 2.4 introduced three fundamental building blocks for building digital circuits—AND gates, OR gates, and NOT gates (inverters), which are far easier to work with than transistors. Section 2.5 showed how Boolean algebra could be used to represent circuits built from AND, OR, and NOT gates, enabling us to build and manipulate circuits by using math—an extremely powerful concept. Section 2.6 introduced several different representations of Boolean functions, namely equations, circuits, and truth tables.

Section 2.7 described a straightforward three-step process for designing combinational circuits, and gave several examples of building real circuits using the three-step process.

Section 2.8 described why NAND and NOR gates are actually more commonly used than AND and OR gates in CMOS technology, and showed that any circuit built from AND, OR, and NOT gates could be built with NAND gates alone or NOR gates alone. That section also introduced two other commonly used gates, XOR and XNOR. Section 2.9 introduced two additional commonly used combinational building blocks, decoders and multiplexers.

Section 2.10 discussed how real gates actually have a small delay between the time that inputs change and the time that the gate's output changes. The section introduced active low inputs, and it also introduced some less commonly used combinational building blocks, demultiplexers and encoders. The section introduced schematic capture tools, which allow designers to draw circuits such that computer programs can read those circuits. The section also introduced simulation, which generates the output waveforms for designer-provided input waveforms, to help a designer verify that a circuit is correct.

▶ 2.14 EXERCISES

An asterisk (*) indicates an especially challenging problem.

SECTION 2.2: SWITCHES

2.1 A microprocessor in 1980 used about 10,000 transistors. How many of those microprocessors would fit in a modern chip having 3 billion transistors?

2.2 The first Pentium microprocessor had about 3 million transistors. How many of those microprocessors would fit in a modern chip having 3 billion transistors?

2.3 Describe the concept known as Moore's Law.

2.4 Assume for a particular year that a particular size chip using state-of-the-art technology can contain 1 billion transistors. Assuming Moore's Law holds, how many transistors will the same size chip be able to contain in ten years?