CHAPTER 1

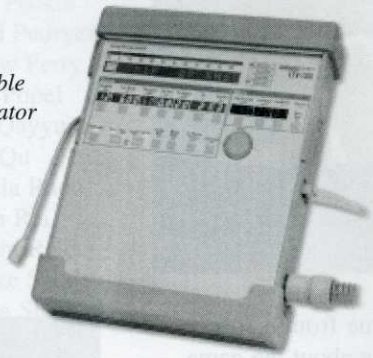
Introduction

1.1 DIGITAL SYSTEMS IN THE WORLD AROUND US

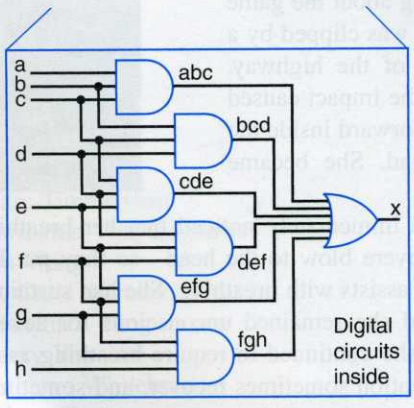
Meet Arianna. Arianna is a five-year-old girl who lives in California. She's a cheerful, outgoing kid who loves to read, play soccer, dance, and tell jokes that she makes up herself.

One day, Arianna's family was driving home from a soccer game. She was in the middle of excitedly talking about the game when suddenly the van in which she was riding was clipped by a car that had crossed over to the wrong side of the highway. Although the accident wasn't particularly bad, the impact caused a loose item from the rear of the van to project forward inside the van, striking Arianna in the back of the head. She became unconscious.

Arianna was rushed to a hospital. Doctors immediately noticed that her breathing was very weak—a common situation after a severe blow to the head—so they put her onto a ventilator, which is a medical device that assists with breathing. She had sustained brain trauma during the blow to the head, and she remained unconscious for several weeks. All her vital signs were stable, except she continued to require breathing assistance from the ventilator. Patients in such a situation sometimes recover, and sometimes they don't. When they do recover, sometimes that recovery takes many months.



Portable ventilator



Thanks to the advent of modern portable ventilators, Arianna's parents were given the option of taking her home while they hoped for her recovery, an option they chose. In addition to the remote monitoring of vital signs and the daily at-home visits by a nurse and respiratory therapist, Arianna was surrounded by her parents, brother, sister, cousins, other family, and friends. For the majority of the day, someone was holding her hand, singing to her, whispering in her ear, or encouraging her to recover. Her sister slept nearby. Some studies show that such human interaction can indeed increase the chances of recovery.

And recover she did. Several months later, with her mom sitting at her side, Arianna opened her eyes. Later that day, she was transported back to the hospital. She was weaned from the ventilator. Then, after a lengthy time of recovery and rehabilitation, Arianna finally went home. Today, six-year-old Arianna shows few signs of the accident that nearly took her life.

What does this story have to do with digital design? Arianna's recovery was aided by a portable ventilator device, whose invention was possible thanks to digital circuits. Over the past three decades, the amount of digital circuitry that can be stored on a single computer chip has increased dramatically—by nearly 100,000 times, believe it or not. Thus, ventilators, along with almost everything else that runs on electricity, can take advantage of incredibly powerful and fast yet inexpensive digital circuits. The ventilator in Arianna's case was the Pulmonetics LTV 1000 ventilator. Whereas a ventilator of the early 1990s might have been the size of a large copy machine and cost about $100,000, the LTV 1000 is not much bigger or heavier than this textbook and costs only a few thousand dollars—small enough, and inexpensive enough, to be carried in medical rescue helicopters and ambulances for life-saving situations, and even to be sent home with a patient. The digital circuits inside continually monitor the patient's breathing, and provide just the right amount of air pressure and volume to the patient. *Every breath* that the device delivers requires *millions* of computations for proper delivery, computations which are carried out by the digital circuits inside.

One indicator rate that new inventions are developed is th number of new patents grante nearly 200,00 2008 alone (fr about 500,000 applications).

Photo courtesy of Pulmonetics

Photo courtesy of Pulmonetics

Portable ventilators help not only trauma victims, but even more commonly help patients with debilitating diseases, like multiple sclerosis, to gain mobility. Today, such people can move about in a wheelchair, and hence do things like attend school, visit museums, and take part in a family picnic, experiencing a far better quality of life than was feasible just a decade ago when those people would have been confined to a bed connected to a large, heavy, expensive ventilator. For example, the young girl pictured on the left will likely require a ventilator for the rest of her life—but she will be able to move about in her wheelchair quite freely, rather than being mostly confined to her home.

The LTV 1000 ventilator described above was conceived and designed by a small group of people, pictured on the left, who sought to build a portable and reliable ventilator in order to help people like Arianna and thousands of others like her (as well as to make some good money doing so!). Those designers probably started off like you, reading textbooks and taking courses on digital design, programming, electronics, and/or other subjects.

The ventilator is just one of literally *tens of thousands* of useful devices that have come about and continue to be created thanks to the era of digital circuits. If you stop and think about how many devices in the world are made possible because of digital circuits, you may be quite surprised. A few such devices include:

> Antilock brakes, airbags, autofocus cameras, automatic teller machines, aircraft controllers and navigators, camcorders, cash registers, cell phones, computer networks, credit card readers, cruise controllers, defibrillators, digital cameras, DVD players, electric card readers, electronic games, electronic pianos, fax machines, fingerprint identifiers, hearing aids, home security systems, modems, pacemakers, pagers, personal computers, personal digital assistants, photocopiers, portable music players, robotic arms, scanners, televisions, thermostat controllers, TV set-top boxes, ventilators, video game consoles—the list goes on.

One indicator of the rate that new inventions are developed is the number of new patents granted: nearly 200,000 in 2008 alone (from about 500,000 total applications).

Those devices were created by hundreds of thousands of designers, including computer scientists, computer engineers, electrical engineers, mechanical engineers, and others, working with people like scientists, doctors, business people, and teachers. One thing that seems clear is that new devices will continue to be invented for the foreseeable future—devices that in another decade will be hundreds of times smaller, cheaper, and more powerful than today's devices, enabling new applications that we can barely dream of. Already, we see new applications that seem futuristic but that exist today, like tiny digital-circuit-controlled medicine dispensers implanted under the skin, voice-controlled appliances, robotic self-guiding household vacuum cleaners, laser-guided automobile cruise control, handheld phones with full Internet access, and more. What's not clear is what new and exciting applications will be developed in the future, or who those devices will benefit. Future designers, like yourself perhaps, will help determine that.

▶ 1.2 THE WORLD OF DIGITAL SYSTEMS

Digital versus Analog

A ***digital*** signal, also known as a discrete signal, is a signal that at any time can have one of a finite set of possible values. In contrast, an ***analog*** signal can have one of an infinite number of possible values, and is also known as a continuous signal. A signal is just some physical phenomenon that has a unique value at every instant of time. An everyday example of an analog signal is the temperature outside, because physical temperature is a continuous value—the temperature may be 92.356666... degrees. An everyday example of a digital signal is the number of fingers you hold up, because the value must be either 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, or 10—a finite set of values. In fact, the term "digital" comes from the Latin word for "digit" (digitus), meaning finger.

In computing systems, the most common digital signals are those that can have one of only two possible values, like on or off (often represented as `1` or `0`). Such a two-valued representation is known as a ***binary*** representation. A ***digital system*** is a system that takes digital inputs and generates digital outputs. A ***digital circuit*** is a connection of digital components that together comprise a digital system. In this textbook, the term "digital" will refer to systems with binary-valued signals. A single binary signal is known as a binary digit, or ***bit*** for short (*b*inary dig*it*). Digital electronics became extremely popular in the mid-1900s after the invention of the transistor, an electric switch that can be turned on or off using another electric signal. We'll describe transistors further in the next chapter.

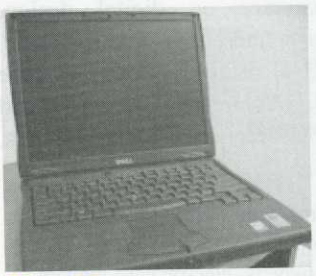


Figure 1.1 (a) General-purpose computer

(b) Embedded systems

About 100,000 unique new digital circuits were designed in 2008.

Digital Circuits are the Basis for Computers

The most well-known use of digital circuits in the world around us is probably to build the microprocessors that serve as the brain of general-purpose computers, like the personal computer or laptop computer that you might have at home, illustrated in Figure 1.1(a). General-purpose computers are also used as servers, which operate behind the scenes to implement banking, airline reservation, web search, payroll, and similar such systems. General-purpose computers take digital input data, such as letters and numbers received from files or keyboards, and output new digital data, such as new letters and numbers stored in files or displayed on a monitor. Learning about digital design is therefore useful in understanding how computers work "under the hood," and hence has been required learning for most computing and electrical engineering majors for decades. Based on material in upcoming chapters, we'll design a simple computer in Chapter 8.

Digital Circuits are the Basis for Much More

Increasingly, digital circuits are being used for much more than implementing general-purpose computers. More and more new applications convert analog signals to digital ones, and run those digital signals through customized digital circuits, to achieve numerous benefits. Such applications, such as those in Figure 1.1(b), include cell phones, automobile engine controllers, TV set-top boxes, music instruments, digital cameras and camcorders, video game consoles, and so on. Digital circuits found inside applications other than general-purpose computers are often called ***embedded systems,*** because those digital systems are embedded inside another electronic device.

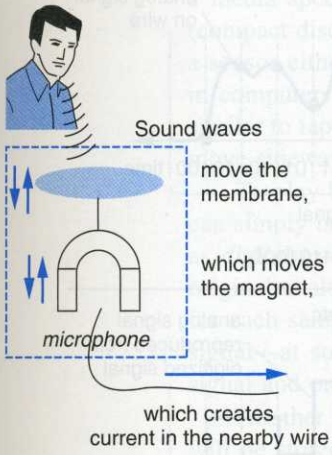


Figure 1.2 Analog audio with a microphone.

The world is mostly analog, and therefore many applications were previously implemented with analog circuits. However, many implementations have changed or are changing over to digital implementations. To understand why, notice that although the world is mostly analog, humans often benefit from converting analog signals to digital signals before "processing" that information. For example, a car horn is actually an analog signal—the volume can take on infinite possible values, and the volume varies over time due to variations in the battery strength, temperature, etc. But humans neglect those variations, and instead "digitize" the sound heard into one of two values: the car horn is "off," or the car horn is "on" (get out of the way!).

Converting analog phenomena to digital for use with digital circuits can also yield benefits. Let's examine a particular example—audio recording. Audio is clearly an analog signal, with infinite possible frequencies and volumes. Consider recording an audio signal like music through a microphone, so that the music can later be played over speakers in an electronic stereo system. One type of microphone, a dynamic microphone, works based on a principle of electromagnetism—moving a magnet near a wire causes changing current (and hence voltage) in the wire, as illustrated in Figure 1.2. The more the magnet moves, the higher the voltage on the wire. A microphone thus has a small membrane attached to a magnet near a wire—when sound hits the membrane, the magnet moves, causing current in the wire. Likewise, a speaker works on the same principle in reverse—a changing current in a wire will cause a nearby magnet to move, which if attached to a membrane will create sound. (If you get a chance, open up an old speaker—you'll find a strong magnet inside.) If the microphone is attached directly to the speaker (through an amplifier that strengthens the microphone's output current), then no digitization is required for sound to be captured by the microphone and played by the speaker. But what if the sound should be saved on some sort of media so that a song can be recorded now and played back later? The sound can be recorded using analog methods or digital methods, but digital methods have many advantages.

One advantage of digital methods is lack of deterioration in quality over time. In the 1970s and 1980s, the audio cassette tape, an analog method, was a common method for recording and playing songs. Audio tape contains on its surface huge numbers of magnetic particles that can be moved to particular orientations using a magnet. Those particles hold that orientation even after the magnet is removed. Thus, magnetism can be used to change the tape's magnetic particles, some of them up, some higher, some down, etc. This is similar to how you can spike your hair, some up, some sideways, some down, using hair gel. The possible orientations of the tape's magnetic particles, and your hair, are infinite, so the tape is definitely analog. Recording onto a tape is done by passing the tape under a "head" that generates a magnetic field based on the electric current on the wire coming from a microphone. The tape's magnetic particles would thus be moved to particular orientations. To play a recorded song back, one would pass the tape under the head again, but this time the head operates in reverse, generating current on a wire based on the changing magnetic field of the moving tape. That current then gets amplified and sent to the speakers.

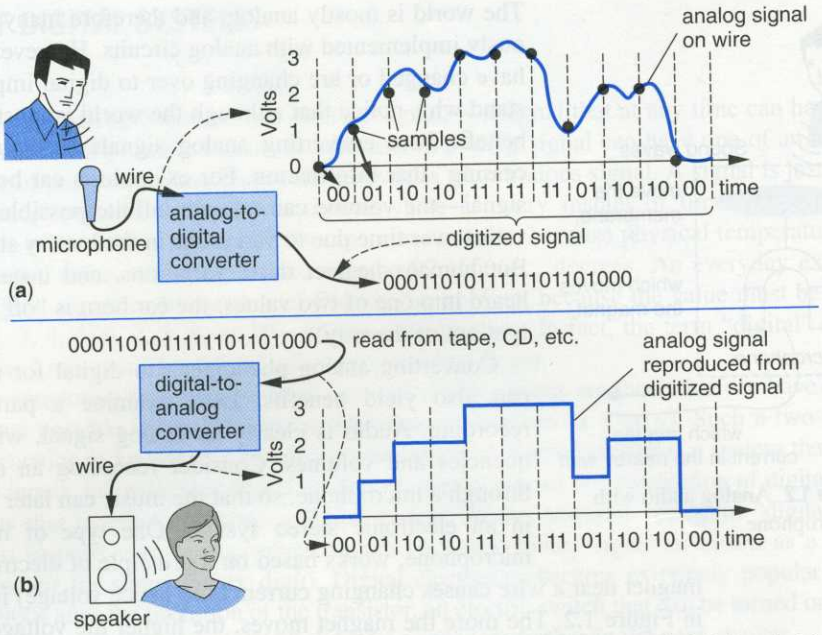


Figure 1.3 Analog-digital conversion: (a) Converting an analog signal to a digital signal, (b) converting a digital signal to an analog signal. Notice some quality loss in the reproduced signal—the signal (in blue) in (b) roughly follows but does not exactly match the signal in (a).

A problem with audio tape is that the orientations of the particles on the tape's surface change over time—just like a spiked hairdo in the morning eventually flattens out throughout the day. Thus, audio tape quality deteriorates over time. Such deterioration is a problem with many analog systems.

Digitizing the audio can reduce such deterioration. Digitized audio works as shown in Figure 1.3(a). The figure shows an analog signal on a wire during a period of time. We *sample* that signal at particular time intervals, shown by the dashed lines. Assuming the analog signal can range from 0 Volts to 3 Volts, and that we plan to store each sample using two bits, then we must round each sample to the nearest Volt (0, 1, 2, or 3). Each sample appears as a point in the figure. We can store 0 Volts as the two bits `00`, 1 Volt as the two bits `01`, 2 Volts as the two bits `10`, and 3 Volts as the two bits `11`. Thus, the shown analog signal would be converted into the following digital signal: `00011010111111101101000`. This process is called ***analog-to-digital conversion*** or just ***digitization***.

To record this digital signal, we just need to store `0`s and `1`s on the recording media. Regular audio tape could be used, recording a short beep to represent a `1` and no beep to represent a `0`, for example. While the audio signal on the tape will deteriorate over time, we can still certainly tell the difference between a beep and no beep, just like we can tell the difference between a car horn being on or off. A slightly quieter beep is still a beep. You've likely heard digitized data communicated using a manner similar to such beeps when you've picked up a phone being used by a computer modem or a fax machine. Even better than audio tape, the digital signal could be recorded using

a media specifically designed to store 0s and 1s. For example, the surface of a CD (compact disc) or DVD (digital video disc) can be configured to reflect a laser beam to a sensor either strongly or weakly, thus storing 1s and 0s easily. Likewise, hard disks in computers use magnetic particle orientation to store 0s and 1s, making such disks similar to tape, but enabling faster access to random parts of the disk since the head can move sideways across the top of the spinning disk.

To play back this digitized audio signal, the digital value of each sampling period can simply be converted to an analog signal, as in Figure 1.3(b). The process is known as ***digital-to-analog conversion***. The reproduced signal is not an exact replica of the original analog signal. The faster the analog signal is sampled and the more bits used for each sample, the closer the reproduced analog signal will be to the original analog signal—at some point, humans can't notice the difference between an original audio signal and one that has been digitized and then converted back to analog.

Another advantage of digitized audio is compression. Suppose that each sample will be stored with ten bits, rather than just two bits, to achieve better quality due to less rounding. The result is many more bits for the same audio—the signal in Figure 1.3 has eleven samples, and at ten bits per sample, one hundred ten bits would be required to store the audio. Sampling thousands of times a second results in huge numbers of bits. However, suppose that a particular audio recording has many samples that have the value 0000000000 or the value 1111111111. We could compress the digital file by creating new words using the following scheme: if the first bit of a word is *0*, the next bit being 0 means the word should be expanded to the sample 0000000000; the next bit being 1 means the word should be expanded to 1111111111. So *00* is shorthand for 0000000000 because the first bit is *0* and the next bit is 0, and *01* is shorthand for 1111111111. If instead the first bit of a word is a *1*, then the next ten bits represent the actual sample. So for *1*0000001111, the first bit is *1*, and thus the actual sample is the next ten bits, or 0000001111. Using this compression scheme, the digitized signal "0000000000 0000000000 0000001111 1111111111" would be compressed to "00 00 10000001111 01." The receiver, which must know the compression scheme, would decompress that compressed signal into the original uncompressed digitized signal. There are many other schemes that can be used to compress digitized audio. Perhaps the mostly widely known audio compression scheme is known as *MP3*, which is popular for compressing digitized songs. A typical song might require many tens of megabytes uncompressed, but compressed usually only requires about 3 or 4 megabytes. Thanks to compression (combined with higher-capacity disks), today's portable music players can store tens of thousands of songs—a capability undreamt of by most people in the 1990s.

Digitized audio is widely used not only in music recording, but also in voice communications. For example, digital cellular telephones, called cell phones or mobile phones, digitize a person's voice and then compress the digital signal before transmitting that signal. Such digitization enables far more cell phones to operate in a particular region than is possible using analog cell phones. Pictures and video can be digitized in a manner similar to that described for audio. Digital cameras and video recorders, for example, store pictures and video in compressed digital form.

Digitized audio, pictures, and video are just a few of the thousands of past and future applications that benefit from digitization of analog phenomena. As shown in

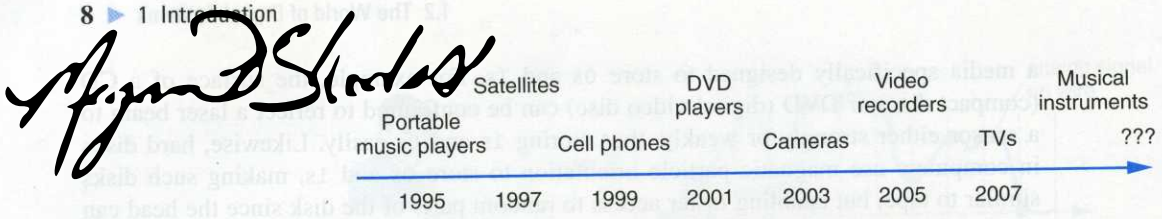


Figure 1.4 More and more analog products are becoming primarily digital.

Figure 1.4, over the past decade numerous products that were previously analog have converted primarily to digital technology. Portable music players, for example, switched from cassette tapes to digital CDs in the middle 1990s, and recently to MP3s and other digital formats. Early cell phones used analog communication, but in the late 1990s digital communication, similar in idea to that shown in Figure 1.3, became dominant. In the early 2000s, analog VHS video players gave way to digital video disc (DVD) players, and then to hard-drive-based digital video recorders (DVRs). Portable video cameras have begun to digitize video before storing the video onto tape or a hard drive, while still picture cameras have eliminated film and store photos on digital cards. Musical instruments are increasingly digital-based, with electronic drums, keyboards, and electric guitars including more digital processing. Analog TV is also giving way to digital TV. Hundreds of other devices have converted from analog to digital in past decades, such as clocks and watches, household thermostats, human temperature thermometers (which now work in the ear rather than under the tongue or other places), car engine controllers, gasoline pumps, hearing aids, and more. Many other devices were never analog, instead being introduced in digital form from the very start. For example, video games have been digital since their inception.

The above devices use digitization, and digitization requires that phenomena be encoded into `1`s and `0`s. Computations using digital circuits also require that numbers be digitized into `1`s and `0`s. The next section describes how to encode items digitally.

▶ *THE TELEPHONE.*

The telephone, patented by Alexander Graham Bell in the late 1800s (though invented by Antonio Meucci), operates using the electromagnetic principle described earlier—your speech creates sound waves that move a membrane, which moves a magnet, which creates current on a nearby wire. Run that wire to somewhere far away, put a magnet connected to a membrane near that wire, and the membrane will move, producing sound waves that sound like you talking. Much of the telephone system today digitizes the audio to improve quality and quantity of audio transmissions over long distances. A couple of interesting facts about the telephone:

- Believe it or not, Western Union actually turned down Bell's initial proposal to develop the telephone, perhaps thinking that the then-popular telegraph was all people needed.
- Bell and his assistant Watson disagreed on how to answer the phone: Watson wanted "Hello," which won, but Bell wanted "Hoy hoy" instead. (Fans of the TV show *The Simpsons* may have noticed that Homer's boss, Mr. Burns, answers the phone with a "hoy hoy.")

An early-style telephone.

(Source of some of the above material: www.pbs.org, transcript of "The Telephone").

Digital Encodings and Binary Numbers—0s and 1s

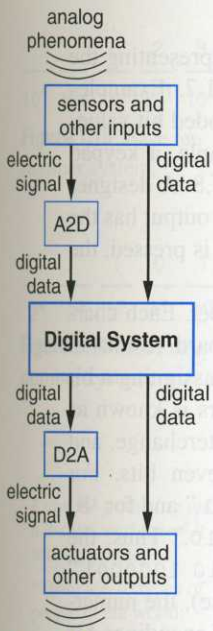


Figure 1.5 A typical digital system.

The previous section showed an example of a digital system, which involved digitizing an audio signal into bits that could then be processed using a digital circuit to achieve several benefits. Those bits *encoded* the data of interest. Encoding data into bits is a central task in digital systems. Some of the data to process may already be in digital form, while other data may be in analog form (e.g., audio, video, temperature) and thus require conversion to digital data first, as illustrated at the top of Figure 1.5. A digital system takes digital data as input, and produces digital data as output.

Encoding Analog Phenomena

Any analog phenomena can be digitized, and hence applications that digitize analog phenomena exist in a wide variety of domains. Automobiles digitize information about the engine temperature, car speed, fuel level, etc., so that an on-chip computer can monitor and control the vehicle. The ventilator introduced earlier digitizes the measure of the air flowing into the patient, so that a computer can make calculations on how much additional flow to provide. Digitizing analog phenomena requires:

- A ***sensor*** that measures the analog physical phenomena and converts the measured value to an analog electrical signal. One example is the microphone (which measures sound) in Figure 1.3. Other examples include video capture devices (which measure light), thermometers (which measure temperature), and speedometers (which measure speed).
- An ***analog-to-digital converter*** that converts the electrical signal into binary encodings. The converter must sample (measure) the electrical signal at a particular rate and convert each sample to some value of bits. Such a converter was featured in Figure 1.3, and is shown as the *A2D* component in Figure 1.5.

Likewise, a ***digital-to-analog converter*** (shown as *D2A* in Figure 1.5) converts bits back to an electrical signal, and an ***actuator*** converts that electrical signal back to physical phenomena. Sensors and actuators together represent types of devices known as ***transducers***—devices that convert one form of energy to another.

Many examples in this book will utilize idealized sensors that themselves directly output digitized data. For instance, an example might use a temperature sensor that reads the present temperature and sets its 8-bit output to an encoding representing the temperature as a binary number, as in Figure 1.6 (see next sections for binary number encodings).

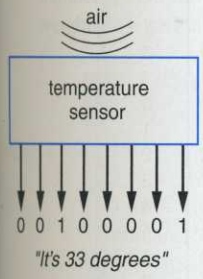


Figure 1.6 Idealized sensor that outputs digital data.

Encoding Digital Phenomena

Other phenomena are inherently digital. Such phenomena can only take on one value at a time from a finite set of values. Some digital phenomena can take on only one of two possible values at a time, and thus can be straightforwardly encoded as a single bit. For example, the following types of sensors may output an electrical signal that takes on one of two values at a time:

- Motion sensor: outputs a positive voltage (say +3 V) when motion is sensed, 0 volts when no motion is sensed.
- Light sensor: outputs a positive voltage when light is sensed, 0 V when dark.
- Button (sensor): outputs a positive voltage when the button is pressed, 0 V when not pressed.

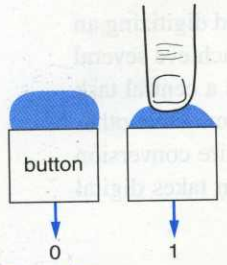


Figure 1.7 A button is easily encoded as a bit.

We can straightforwardly encode each sensor's output to a bit, with `1` representing the positive voltage and `0` representing 0 V, as for the button in Figure 1.7. Examples throughout this book utilize idealized sensors that directly output the encoded bit value. Other digital phenomena can assume several possible values. For example, a keypad may have four buttons, colored red, blue, green, and black, as in Figure 1.8. A designer might create a circuit such that when red is pressed, the keypad's three-bit output has the value `001`; blue outputs `010`, green `011`, and black `100`. If no button is pressed, the output is `000`.

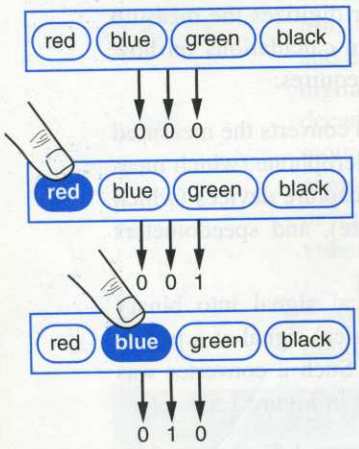


Figure 1.8 Keypad encodings.

An even more general digital phenomenon is the English alphabet. Each character comes from a finite set of characters, so typing on a keyboard results in digital, not analog, data. The digital data can be converted to bits by assigning a bit encoding to each character. A popular encoding of English characters is known as ASCII, which stands for American Standard Code for Information Interchange, and is pronounced as "askey." ASCII encodes each character into seven bits. For example, the ASCII encoding for the uppercase letter 'A' is "`1000001`," and for 'B' is "`1000010`." A lowercase 'a' is "`1100001`," and 'b' is "`1100010`." Thus, the name "ABBA" would be encoded as "`1000001 1000010 1000010 1000001`." ASCII defines 7-bit encodings for all 26 letters (upper- and lowercase), the numerical symbols 0 through 9, punctuation marks, and even a number of encodings for nonprintable "control" operations. There are 128 encodings total in ASCII. A subset of ASCII encodings is shown in Figure 1.9. Another encoding, Unicode, is increasing in popularity due to its support of international languages. Unicode uses 16 bits per character, instead of just the 7 bits used in ASCII, and represents characters from a diversity of languages in the world.

Encoding	Symbol
010 0000	<space>
010 0001	!
010 0010	"
010 0011	#
010 0100	$
010 0101	%
010 0110	&
010 0111	'
010 1000	(
010 1001	)
010 1010	*
010 1011	+
010 1100	,
010 1101	-
010 1110	.
010 1111	/

Encoding	Symbol
100 0001	A
100 0010	B
100 0011	C
100 0100	D
100 0101	E
100 0110	F
100 0111	G
100 1000	H
100 1001	I
100 1010	J
100 1011	K
100 1100	L
100 1101	M

Encoding	Symbol
100 1110	N
100 1111	O
101 0000	P
101 0001	Q
101 0010	R
101 0011	S
101 0100	T
101 0101	U
101 0110	V
101 0111	W
101 1000	X
101 1001	Y
101 1010	Z

Encoding	Symbol
110 0001	a
110 0010	b
...	
111 1001	y
111 1010	z
011 0000	0
011 0001	1
011 0010	2
011 0011	3
011 0100	4
011 0101	5
011 0110	6
011 0111	7
011 1000	8
011 1001	9

Figure 1.9 Sample ASCII encodings.

notion is sensed,

/ when dark.

ressed, 0 V when

representing the
re 1.7. Examples
ncoded bit value.
xample, a keypad
re 1.8. A designer
-bit output has the
ton is pressed, the

phabet. Each char-
eyboard results in
by assigning a bit
acters is known as
n Interchange, and
o seven bits. For
0001," and for 'B'
00010." Thus, the
00010 1000001."
ercase), the numer-
er of encodings for
in ASCII. A subset
ding, Unicode, is
ages. Unicode uses
l represents charac-

Encoding	Symbol
110 0001	a
110 0010	b
...	
111 1001	y
111 1010	z
011 0000	0
011 0001	1
011 0010	2
011 0011	3
011 0100	4
011 0101	5
011 0110	6
011 0111	7
011 1000	8
011 1001	9

Encoding Numbers as Binary Numbers

Perhaps the most important use of digital circuits is to perform arithmetic computations. In fact, a key driver of early digital computer design was the arithmetic computation of ballistic trajectories in World War II. To perform arithmetic computations, we need a way to encode numbers as bits—we need binary numbers.

To understand binary numbers, first refreshing our understanding of decimal numbers can help. Decimal numbers use a base ten numbering system. Base ten is a numbering system where the rightmost digit represents the number of ones (10^0) present, the next digit represents the number of groups of tens (10^1) present (meaning that the digit's place has a ***weight*** of 10^1), the next digit's place has a weight of (10^2), and so on, as illustrated in Figure 1.10. So the digits "523" in base 10 represent $5*10^2 + 2*10^1 + 3*10^0$.

		5	2	3
10^4	10^3	10^2	10^1	10^0

Figure 1.10 Base ten number system.

With an understanding of base ten numbers, we can introduce base two numbers, known as ***binary numbers***. Because digital circuits operate with values that are either "on" or "off," such circuits need only two symbols, rather than ten symbols. Let those two symbols be `0` and `1`. If we need to represent a quantity more than 1, we'll use another digit, whose weight will be 2^1. So "`10`" in base two represents 1 two and 0 ones. Be careful not to call that `10` "ten" or "ten, base two" (which makes no sense)—instead, you might say "one zero, base two." If we need a bigger quantity, we'll use another digit, whose weight will be 2^2. The weights for the first few digits in base two are shown in Figure 1.11. For a given binary number, the lowest-weight (rightmost) digit is called the ***least significant bit***, and the highest-weight (leftmost) digit is the ***most significant bit***.

		1	0	1
2^4	2^3	2^2	2^1	2^0

Figure 1.11 Base two number system.

I saw the following on a T-shirt, and found it rather funny:

"There are 10 types of people in the world: those who get binary, and those who don't."

For example, the number `101` in base two equals $1*2^2 + 0*2^1 + 1*2^0$, or 5 in base ten. `101` can be spoken as "one zero one, base two." Definitely do *not* say "one hundred one, base two." 101 is one hundred one if in base ten, but the leftmost 1 does not represent one hundred when in base two.

When we are writing numbers of different bases and the base of the number is not obvious, we can indicate the base with a subscript, as follows: $101_2 = 5_{10}$. We might speak this as "one zero one in base two equals five in base ten." This book usually displays binary numbers using a different font, e.g., `101`, to readily distinguish binary numbers from decimal numbers.

		1	0	1
16	8	4	2	1

Figure 1.12 Base two number system showing weights in base ten.

Note that because binary isn't as popular as decimal, people haven't created short names for its weights of 2^1, 2^2, and so on, like they have for weights in base ten (hundreds, thousands, millions, etc.). Instead, people just use the equivalent base ten name for each base two group—a source of confusion to some people just learning binary. Nevertheless, it may still be easier to think of each weight in base two using base ten names—one, two, four, eight—rather than increasing powers of two, as in Figure 1.12.

▶ *WHY BASE TEN?*

Humans have ten fingers, so they chose a numbering system where each digit can represent ten possible values. There's nothing magical about base ten. If humans had nine fingers, they would probably use a base nine numbering system. It turns out that base twelve was used somewhat in the past too, because by using our thumb, we can easily point to twelve different spots on the remaining four fingers on that thumbs's hand—the four tops of those fingers, the four middle parts of those fingers, and the four bottoms of those fingers. That may partly explain why twelve is common in human counting today, like the use of the term "dozen," and the twelve hours of a clock.

(*Source*: *Ideas and Information*, Arno Penzias, W.W. Norton and Company.)

▶ NAMES IN BASE TEN.

Indian English has a name for 10^5: lakh. In 2008, the Indian car company Tata Motors unveiled the "one lakh car," costing a mere 100,000 rupees, or about $2,500.

*The Web search tool name **Google** comes from the word "googol" —a 1 followed by 100 zeroes, apparently implying that the tool can search a lot of information.*

English speakers use names for various quantities in base ten, names that are useful but can hamper gaining an intuitive understanding of base ten. 10^2 has its own name: *hundred*. 10^3 has the name *thousand*. There is no name (in American English) for 10^4 or 10^5. 10^6 has the name *million*, and subsequent groups that are multiples of 1,000 have the names *billion*, *trillion*, *quadrillion*, etc. English speakers also use abbreviated names for groups of tens—the numbers 10, 20, 30, ..., 90 could be called one ten, two ten, up to nine ten, but instead have abbreviated names: one ten as just "ten," two ten is "twenty," up to nine ten being "ninety." You can see how "ninety" is a shortening of "nine ten." Special names are also used for the numbers between 10 and 20. 11 could be "one ten one," but is instead "eleven," while 19 could be "one ten nine" but is instead "nineteen." Table 1.1 indicates how one might count in base ten without these various names, to emphasize the nature of base ten. 523 might be spoken as "five hundred two ten three" rather than "five hundred twenty-three." Kids may have a harder time learning math because of the arbitrary base ten names—for example, carrying a one from the ones column to the tens column makes more sense if the ones column sums to "one ten seven" rather than to "seventeen"—"one ten seven" obviously adds one to the tens column. Likewise, learning binary may be slightly harder for some students due to a lack of a solid understanding of base ten. To help remedy the situation, perhaps when a store clerk tells you "That will be ninety-nine cents," you might say "You mean nine ten nine cents." If enough of us do this, perhaps it will catch on?

Table 1.1 Counting in base ten without the abbreviated or short names.

0 to 9	As usual: "zero," "one," "two," ..., "nine."
10 to 99	10, 11, 12, ... 19: "one ten," "one ten one," "one ten two," ... "one ten nine" 20, 21, 22, ..., 29: "two ten," "two ten one," "two ten two," ... "two ten nine" 30, 40, ... 90: "three ten," "four ten," ... "nine ten"
100 to 900	As usual: "one hundred," "two hundred," ... "nine hundred." Even clearer would be to replace the word "hundred" by "ten to the power of 2."
1000 and up	As usual. Even clearer for understanding bases: replace "thousand" by "ten to the (power of) 3", "ten thousand" by "ten to the 4," etc., eliminating the various names.

Example 1.1 Using digital data in a digital system

A digital system is desired that reads the value of a temperature sensor and shows the letter "F" (for "freezing") on a display if the temperature is 32 degrees Fahrenheit or below, shows "N" (for "normal") if the temperature is between 32 and 212 degrees, and shows the letter "B" (for "boiling") if the temperature is 212 or greater. The temperature sensor has an 8-bit output representing the temperature as a binary number between 0 and 255. The display has a 7-bit input that accepts an ASCII bit encoding and displays the corresponding symbol.

Figure 1.13 shows the temperature sensor output connected to the input of the desired digital system. Each wire can have

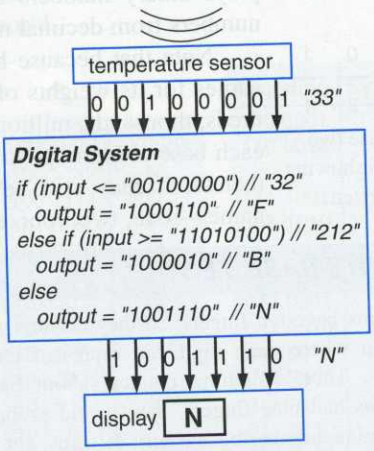


Figure 1.13 Digital system with bit encoded input (an 8-bit binary number) and 7-bit output (ASCII). The desired behavior of the digital system is shown.

the value of 1 or 0. The figure also shows a 7-bit output from the digital system connected to the display's 7-bit input.

The desired behavior for the digital system is shown in the figure: if the input is less than or equal to binary 00100000, which is 32 in base ten, then the output should be set to 1000110, which is the letter "F" in ASCII, as seen from Figure 1.9. Likewise, if the input is greater than or equal to binary 11010100, which is 212 in base ten, then the output should be 1000010, which is "B" in ASCII. For any other input value (which means the value is between 32 and 212), the output should be 1001110, which is "N" in ASCII. An example input of 00100001, which is 33 in base ten, is shown. For that input, the digital system outputs "N," as shown.

This example demonstrates how a digital system operates on digital input bits—0s and 1s—and creates digital output bits. We'll later see how to build circuits to implement desired digital system behavior.

Converting from Binary to Decimal

Because humans deal primarily with decimal numbers, a common digital design task is to convert a binary number to decimal, or to convert a decimal number to binary. Converting a binary number to decimal is straightforward: we simply sum the weights of each digit having a 1, as in the following example.

Example 1.2 Binary to decimal

Convert these binary numbers to decimal numbers: 1, 110, 10000, 10000111, and 00110.

1_2 is just $1*2^0$, or 1_{10}.

110_2 is $1*2^2 + 1*2^1 + 0*2^0$, or 6_{10}. We might think of this using the weights shown in Figure 1.12: 1*4 + 1*2 + 0*1, or 6.

10000_2 is 1*16 + 0*8 + 0*4 + 0*2 + 0*1, or 16_{10}.

10000111_2 is 1*128 + 1*4 + 1*2 + 1*1 = 135_{10}. Notice this time that we didn't bother to write the weights having a 0 bit.

00110_2 is the same as 110_2 above — the leading 0's don't change the value.

1
2
4
8
16
32
64
128
256
512
1024
2048

Figure 1.14 Memorizing powers of two helps in working with binary.

When converting from binary to decimal, people often find it useful to be comfortable knowing the powers of two, shown in Figure 1.14, because each successive place to the left in a binary number is two times the previous place. In binary, the first, rightmost place is 1, the second place is 2, then 4, then 8, 16, 32, 64, 128, 256, 512, 1024, 2048, and so on. You might stop at this point to practice counting up by powers of two: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, etc., a few times. Now, when you see the number 10000111, you might move along the number from right to left and count up by powers of two for each bit to determine the weight of the leftmost bit: 1, 2, 4, 8, 16, 32, 64, *128*. The next highest 1 has a weight of (counting up again) 1, 2, *4*; adding 4 to 128 gives 132. The next 1 has a weight of 2; adding that to 132 gives 134. The rightmost 1 has a weight of 1; adding that to 134 gives 135. Thus, 10000111 equals 135 in base ten.

Being comfortable counting up in binary can also be helpful when working with binary. Counting up in binary goes as follows (using three digits): 000, 001, 010, 011, 100, 101, 110, 111. You might practice writing out this sequence several times to become more comfortable with it, doing so for four or even five digits also. Note that a binary number whose digits are all 1s has a base ten value exactly one less than the value of the next higher digit; for example, 111 is 7 in base ten, which is one less than 1000.

An interesting fact about binary numbers is that you can quickly determine whether a binary number is odd just by checking if the least-significant (i.e., rightmost) digit has a `1`. If the rightmost digit is a `0`, the number must be even, because the number is the sum of even numbers, as the only odd-weighted digit is the rightmost digit with a weight of 1.

Converting from Decimal to Binary Using the Addition Method

As seen earlier, converting a binary number to decimal is easy—just add the weights of each digit having a `1`. Converting a decimal number to binary takes slightly more effort. One method for converting a decimal number to a binary number by hand is the ***addition method***, in which we put a `1` in the highest place whose weight doesn't exceed the number, add that number to a sum, and repeat until the sum equals the desired number. For example, we can convert the decimal number 12 to binary as shown in Figure 1.15.

	Desired decimal number: **12**	Current sum	Binary number
(a)	16 > 12, too big; Put 0 in 16's place	0	**0** / ***16*** 8 4 2 1
(b)	8 <= 12, so put 1 in 8's place, current sum is 8	8	0 / 16 **1** / ***8*** 4 2 1
(c)	8+4=12 <= 12, so put 1 in 4's place, current sum is 12	12	0 / 16 1 / 8 **1** / ***4*** 2 1
(d)	Reached desired 12, so put 0s in remaining places	*done*	0 / 16 1 / 8 1 / 4 0 / 2 0 / 1

Figure 1.15 Converting the decimal number 12 to binary using the addition method: (a) putting a `1` in place 16 would exceed 12, so we put a `0` there, (b) putting a `1` in place 8 gives us a sum of 8 so far, (c) putting a `1` in place 4 gives a sum of 8+4=12, the desired value, (d) because 12 has already been reached, we put `0`s in the remaining places. The answer is thus `01100`, or just `1100`.

We can check our work by converting `1100` back to decimal: 1*8 + 1*4 + 0*2 + 0*2 = 12.

As another example, Figure 1.16 illustrates the addition method for converting the decimal number 23 to binary, using a more compact representation of the calculations. We can check our work by converting the result, `10111`, back to decimal: 1*16 + 0*8 + 1*4 + 1*2 + 1*1 = 23.

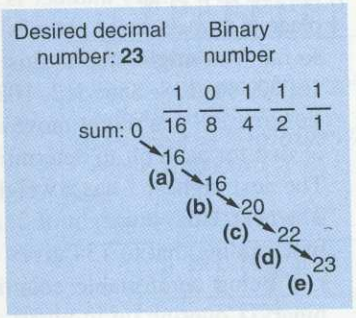


Figure 1.16 Decimal 23 to binary with the addition method: (a) 32 too big, put `1` in place 16, (b) 16+8=24 is too much, put `0` in place 8, (c) 16+4=20, (d) 20+2=22, (e) 22+1=23.

Example 1.3 Decimal to binary

Convert the following decimal numbers to binary using the addition method: 8, 14, 99.

To convert 8 to binary, we start by putting a `1` in the 8's place, yielding `1000` (putting `0`s in the lower places not yet considered). The current sum is 8, so we are done—the answer is `1000`.

To convert 14 to binary, we start by putting a 1 in the 8's place (16 is too much), yielding 1000 and a sum of 8. We put a 1 in the 4's place, yielding 1100 and a sum of 8+4=12. We put a 1 in the 2's place, yielding 1110, and a sum of 12+2=14, so we are done—the answer is 1110. We can check our work by converting back to decimal: 8 + 4 + 2 = 14.

To convert 99 to binary, we start by putting a 1 in the 64's place (the next higher place, 128, is too big—notice that being able to count by powers of two is quite handy in this problem), yielding 1000000 and a sum of 64. We put a 1 in the 32's place, yielding 1100000 and a sum of 64+32=96. Putting a 1 in the 16's place yield a sum of 96+16=112, which is too much, so we put a 0 in the 16's place. Likewise, we put 0s in the 8's place and the 4's place. Putting a 1 in the 2's place yields 1100010 and a sum of 96+2=98. Finally, we put a 1 in the 1's place, yielding the final answer of 1100011 and a sum of 98+1=99. We can check our work by converting back to decimal: 64 + 32 + 2 + 1 = 99.

Note that the addition method must be conducted starting from the highest weight place rather than starting from the lowest weight—in other words from left to right. Starting from the lowest weight, or from right to left, does not work. For example, for the decimal number 8, starting from the lowest weight would put a 1 in the 1's place, then a 1 in the 2's place yielding a sum of 3, and then a 1 in the 4's place yielding a sum of 7. Putting a 1 in the 8's place would then yield a sum of 15, which is too much, and thus the procedure would fail.

Example 1.4 Converting from decimal to binary to set a DIP-switch controlled channel

This example illustrates converting decimal to binary to configure a digital household appliance. A ceiling fan is a common household appliance that often comes with a remote controller that can be used to turn the fan on or off, as illustrated in Figure 1.17(a). All the fans and remote controllers may operate on the same wireless frequency. Because one house may have multiple such ceiling fans, a method is needed to prevent the remote controller of one fan from affecting another fan. A common method is to encode a channel number in the wireless signal, such that each ceiling-fan/remote-controller pair in a home shares a unique channel. When a ceiling fan's module detects the remote control wireless frequency, it checks whether the detected channel matches its own channel before responding. A common means for setting the channel is to use a DIP switch inside the remote controller and another DIP switch inside the ceiling fan module. An 8-pin DIP switch has eight switches that each can be either in an up or down position, and eight outputs that each will be either a 1 if its corresponding switch is up or a 0 if its switch is down. Thus, such a DIP switch can represent $2^8 = 256$ distinct values.

Suppose the ceiling fan manufacturer wishes to set a particular ceiling-fan/remote-controller channel to 73. The manufacturer first converts 73 to binary, shown in Figure 1.17(b) to be 01001001. The manufacturer can then set the DIP switch inside the ceiling fan module, as well as inside the remote controller, to the up/down settings shown in Figure 1.17(c). Then, that ceiling fan module will only respond (by turning the fan on or off, in this case shown merely as setting its output to 1), if it detects the remote controller's frequency AND the encoded channel matches its DIP switch's value.

In case a homeowner happens to purchase two fan/controller pairs that are set to the same channel (the chances of which are 1 in 256), noticing this when one remote controller affects two fans, then the homeowner can remedy the problem without having to remove and exchange one of the fans.

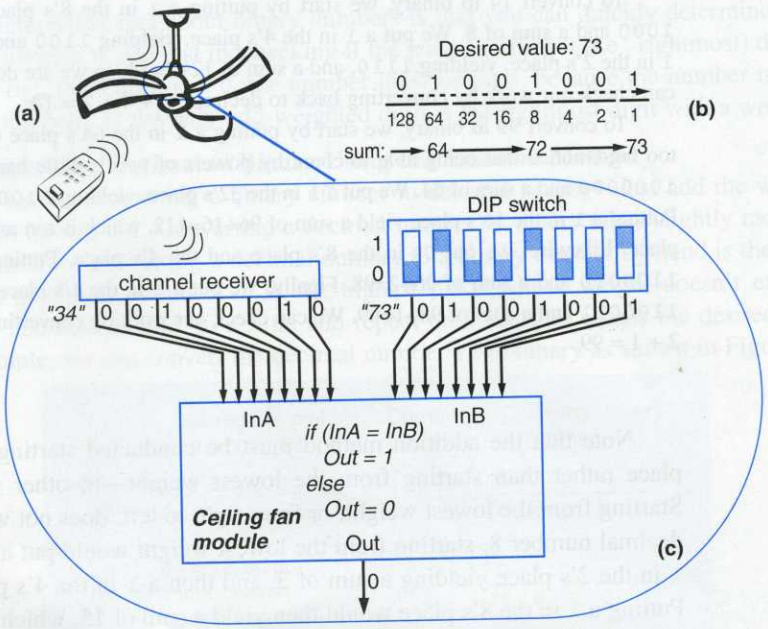


Figure 1.17 Decimal to binary conversion for a DIP switch: (a) ceiling fan with remote control, both having DIP switches to set their communication channel, (b) setting the fan's channel to "73" requires first converting 73 to binary, then setting the DIP switch to represent that binary value, (c) ceiling fan module only outputs 1 if the received channel matches DIP switch setting.

Instead, he/she can open the ceiling fan module and the remote controller of one pair, and simply change the DIP switch settings for that pair, ensuring that both DIP switches match after the change.

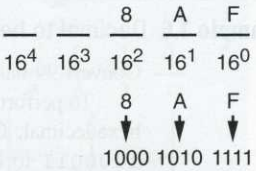
While this section introduced the addition method for converting from decimal to binary, many books and web resources introduce the ***subtraction method***, wherein we start by setting a current number to the desired decimal number, put a 1 in the highest binary number place that doesn't exceed the current number, subtract that place's weight from the current number, and repeat until the current number reaches zero. The two methods are fundamentally the same; the addition method may be more intuitive when converting by hand.

Hexadecimal and Octal Numbers

Base sixteen numbers, known as ***hexadecimal numbers*** or just ***hex***, are also popular in digital design, mainly because one base sixteen digit is equivalent to four base two digits, making hexadecimal numbers a nice shorthand representation for binary numbers. In base sixteen, the first digit represents up to fifteen ones—the sixteen symbols commonly used are 0, 1, 2, ..., 9, A, B, C, D, E, F (so A = ten, B = eleven, C = twelve, D = thirteen, E = fourteen, and F = fifteen). The next digit represents the number of groups of 16^1, the next digit the number of groups of 16^2, etc., as shown in Figure 1.18. So $8AF_{16}$ equals $8*16^2 + 10*16^1 + 15*16^0$, or 2223_{10}.

Because one digit in base 16 represents 16 values, and four digits in base two represents 16 values, then each digit in base 16 represents four digits in base two, as shown at the bottom of Figure 1.18. Thus, to convert $8AF_{16}$ to binary, we convert 8_{16} to 1000_2, A_{16} to 1010_2, and F_{16} to 1111_2, resulting in $8AF_{16} = 100010101111_2$. You can see why hexadecimal is a popular shorthand for binary: 8AF is a lot easier on the eye than 100010101111.

To convert a binary number to hexadecimal, we just substitute every four bits with the corresponding hexadecimal digit. Thus, to convert 101101101_2 to hex, we group the bits into groups of four starting from the right, yielding 1 0110 1101. We then replace each group of four bits with a single hex digit. 1101 is D, 0110 is 6, and 1 is 1, resulting in the hex number $16D_{16}$.



hex	binary	hex	binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Figure 1.18 Base sixteen number system.

Example 1.5 Hexadecimal to/from binary

Convert the following hexadecimal numbers to binary: FF, 1011, A0000. You may find it useful to refer to Figure 1.18 to expand each hexadecimal digit to four bits.

FF_{16} is `1111` (for the left F) and `1111` (for the right F), or 11111111_2.

1011_{16} is `0001, 0000, 0001, 0001`, or 0001000000010001_2. Don't be confused by the fact that 1011 didn't have any symbols but 1 and 0 (which makes the number look like a binary number). We said it was base 16, so it was. If we said it was base ten, then 1011 would equal one thousand and eleven.

$A0000_{16}$ is `1010, 0000, 0000, 0000, 0000`, or 10100000000000000000_2.

Convert the following binary numbers to hexadecimal: `0010, 01111110, 111100`.

0010_2 is 2_{16}.

01111110_2 is `0111` and `1110`, meaning 7 and E, or $7E_{16}$. 111100_2 is `11` and `1100`, which is `0011` and `1100`, meaning 3 and C, or $3C_{16}$. Notice that we start grouping bits into groups of four from the right, not the left.

If a decimal number needs to be converted to hexadecimal, one can use the addition method. Sometimes, though, it is easier to first convert from decimal to binary using the addition method, and then converting from binary to hexadecimal by grouping sets of four bits.

Example 1.6 Decimal to hexadecimal

Convert 99 base 10 to base 16.

To perform this conversion, we can first convert 99 to binary and then convert the binary result to hexadecimal. Converting 99 to binary was done in Example 1.3, yielding `1100011`. Converting `1100011` to hexadecimal can be done by grouping sets of four bits (starting from the right), so `1100011` is `110` and `0011`, meaning 6 and 3, or 63_{16}. We can check our work by converting 63_{16} to decimal: $6*16^1 + 3*16^0 = 96 + 3 = 99$.

Example 1.7 RFID tag with identifier in hexadecimal

An RFID tag (radio frequency identification tag) is a chip that automatically responds to a radio signal by sending back a signal containing a unique identification number. RFID tags have been used since the 1990s with automobile toll transponders, dairy cows (as in Figure 1.19(a)), and dogs (in which the tag may be implanted under the skin). An RFID tag typically has an electronic circuit that uses the radio signal to power the chip, thus eliminating the need for a battery, decreasing the tag size, and increasing tag longevity.

In 2004, a Barcelona club began allowing customers to implant RFID tags under their skin to gain access to VIP lounges and to pay their bill. The U.S. F.D.A. approved human RFID implants in 2004. Source: www.theregister.co.uk

Figure 1.19(b) illustrates how the identifier in a cow tag might be structured as a 32-bit stored value where the first 8 bits correspond to a province (or state) number and the next 8 bits to a city number (thus representing the cow's birthplace), and the last 16 bits correspond to the animal's unique number. The device that wirelessly programs that identifier into the tag may require that the identifier be input in hex, even though the programming device will actually write `0`s and `1`s into the tag's 32-bit storage—typing 8 hex digits is less error-prone than typing 32 bits. Figure 1.19(c) provides sample desired values in decimal, Figure 1.19(d) shows those values converted to binary, and Figure 1.19(e) shows the binary values converted to hex. Finally, Figure 1.19(f) shows the 8-digit hex value that would be entered into the programming device, which would then program a specific tag with the corresponding 32-bit value. Once that value is programmed into the tag, the tag can be attached to the cow, and subsequently read countless times by an RFID reader, perhaps to help ensure that each cow is milked no more than once per day.

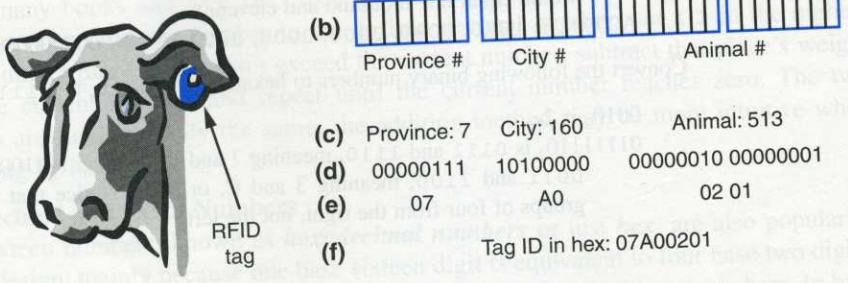


Figure 1.19 Hex used to configure RFID tag: (a) RFID tag attached to cow's ear, (b) 32-bit value inside tag stores unique identifier, the first 8 bits indicating a province number, the second 8 bits indicating a city number, and the remaining 16 bits being the animal's unique number, (c) sample values in decimal, (d) binary equivalents, (e) hex equivalents obtained from binary, (f) final 32-bit identifier in hex.

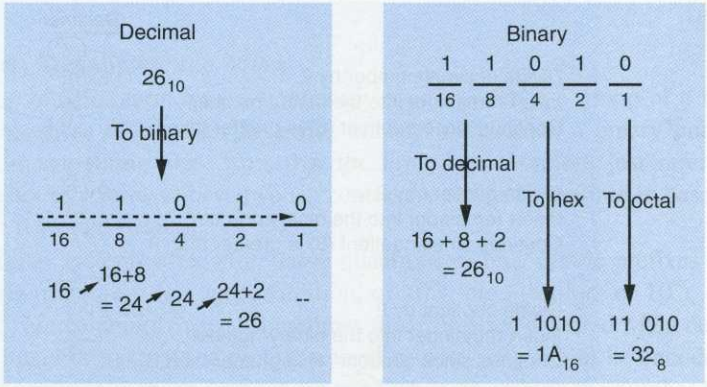


Figure 1.20 Methods for converting to binary and from binary by hand.

Base eight numbers, known as ***octal numbers***, are sometimes used as a binary shorthand too, because one base eight digit equals three binary digits. 503_8 equals $5*8^2 + 0*8^1 + 3*8^0 = 323_{10}$. We can convert 503_8 directly to binary simply by expanding each digit into three bits, resulting in 503_8 = `101 000 011`, or 101000011_2. Likewise, we can convert binary to octal by grouping the binary number into groups of three bits starting from the right, and then replacing each group with the corresponding octal digit. Thus, 1011101_2 yields `1 011 101`, or 135_8.

Figure 1.20 summarizes the methods for converting decimal to binary, and for converting binary to decimal, hex, or octal. Converting from decimal to hex or octal can be done by first converting to binary and then from binary to hex or octal. Converting from hex or octal to binary is a straightforward expansion of each digit to its four- or three-bit equivalent, respectively.

Automatic Conversion from Decimal to Binary Using the Divide-by-2 Method

The addition method is intuitive when converting from decimal to binary by hand, but if we wish to perform the conversion automatically with a computer program, another method, known as the divide-by-2 method, is well-suited. The ***divide-by-2 method*** involves repeatedly dividing the decimal number by 2; the remainder at each step, which will be either 0 or 1, becomes a bit in the binary number, starting from the least significant (rightmost) digit. For example, the process of converting the decimal number 12 to binary using the divide-by-2 method is shown in Figure 1.21.

Example 1.8 Decimal to binary using the divide-by-2 method

Convert the following numbers to binary using the divide-by-2 method: 8, 14, 99.

To convert 8 to binary, we start by dividing 8 by 2: 8/2=4, remainder 0. Then we divide the quotient, 4, by 2: 4/2=2, remainder 0. Then we divide 2 by 2: 2/2=1, remainder 0. Finally, we divide 1 by 2: 1/2=0, remainder 1. We stop dividing because the quotient is now 0. Combining all the remainders, least significant digit first, yields the binary number `1000`. We can check this answer by multiplying each binary digit by its weight and adding the terms: $1*2^3 + 0*2^2 + 0*2^1 + 0*2^0 = 8$.

To convert 14 to binary, we follow a similar process: 14/2=7, remainder 0. 7/2=3, remainder 1. 3/2=1, remainder 1. 1/2=0, remainder 1. Combining the remainders gives us the binary number

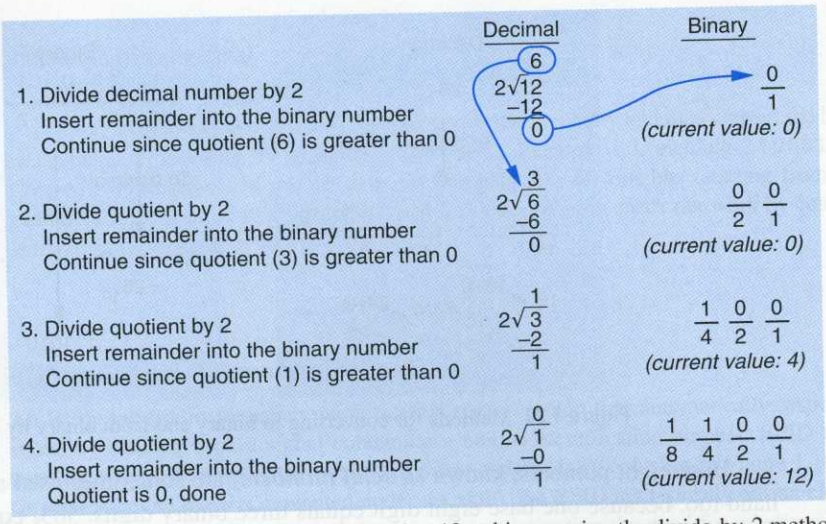


Figure 1.21 Converting the decimal number 12 to binary using the divide-by-2 method.

`1110`. Checking the answer shows that `1110` is correct: $1*2^3 + 1*2^2 + 1*2^1 + 0*2^0 = 8 + 4 + 2 + 0 = 14$.

To convert 99 to binary, the process is the same but naturally takes more steps: 99/2 = 49 remainder 1. 49/2 = 24, remainder 1. 24/2 = 12, remainder 0. 12/2 = 6, remainder 0. 6/2 = 3, remainder 0. 3/2 = 1, remainder 1. 1/2 = 0, remainder 1. Combining the remainders together gives us the binary number `1100011`.We know from Example 1.3 that this is the correct answer.

We can use the same basic method to convert a base 10 number to a number in *any* base. To convert a number from base 10 to base n, we repeatedly divide the number by n and place the remainder in the new base n number, starting from the least significant digit. The method is called the ***divide-by-n method***.

Example 1.9 Decimal to arbitrary bases using the divide-by-*n* method

Convert the number 3439 to base 10 and to base 7.

We know the number 3439 is 3439 in base 10, but let's use the divide-by-n method (where n is 10) to illustrate that the method works for any base. We start by dividing 3439 by 10: 3439/10 = 343, remainder 9. We then divide the quotient by 10: 343/10 = 34, remainder 3. We do the same with the new quotient: 34/3 = 3, remainder 4. Finally, we divide 3 by 10: 3/10 = 0, remainder 3. Combining the remainders, least significant digit first, gives us the base 10 number 3439.

To convert 3439 to base 7, the approach is similar, except we now divide by 7. We begin by dividing 3439 by 7: 3439/7 = 491, remainder 2. Continuing our calculations, we get 491/7 = 70, remainder 1. 70/7 = 10, remainder 0. 10/7 = 1, remainder 3. 1/7 = 0, remainder 1. Thus, 3439 in base 7 is 13012. Checking the answer verifies that we have the correct result: $1*7^4 + 3*7^3 + 0*7^2 + 1*7^1 + 2*7^0 = 2401 + 1029 + 7 + 2 = 3439$.

Conversion between any two bases can be done by first converting to base ten, then converting the base ten number to the desired base using the divide-by-n method.

Bytes, Kilobytes, Megabytes, and More

When discussing bits, the term ***byte*** is commonly used to refer to a group of 8 bits. The term is used regardless of whether those bits encode a character, a binary number, an audio sample, or something else. Note that the term *byte* therefore just refers to the number of bits, not to a value in base two (in contrast to the term *hundred* in base ten, for example, which refers to a value).

Because digital systems often store large quantities of bits, metric prefixes like *kilo* (referring to a thousand, or 10^3), *mega* (million, or 10^6), *giga* (billion, or 10^9), and *tera* (trillion, or 10^{12}) are commonly used. Sometimes those prefixes are used when describing the number of bits. For example, a typical several minute song might be encoded as 24 megabits. More commonly, those prefixes are used when describing the number of bytes. So that same song would be encoded as 24/8 = 3 megabytes. A feature length movie might be encoded as about 2 gigabytes. A computer hard drive might have a capacity of 1 terabyte. Kilobytes, megabytes, gigabytes, and terabytes are commonly written as Kbytes or KB, Mbytes or MB, Gbytes or GB, and Tbytes or TB, respectively.

Unfortunately, an inaccurate naming convention developed in digital system terminology. Quantities in digital systems, especially memory sizes, are often powers of two, such as 1024 (2^{10}), 2048 (2^{11}), or 4096 (2^{12}). Engineers began referring to 1024 as 1K, 2048 as 2K, 4096 as 4K, and so on. For example, one of the first personal computers was the Commodore 64, named due to its having 2^{16} = 65,536 bytes of memory, or about 64 kilobytes of memory. This inaccurate use of metric prefixes is common today when referring to computer memory sizes, so a 1 megabyte memory actually has 1,048,576 (2^{20}) bytes rather than 1,000,000 bytes, and a 1 gigabyte memory actually has 1,073,741,824 (2^{30}) bytes rather than 1,000,000,000 bytes.

Seeking to solve the problem, the International Electrotechnical Commission (IEC) in 1999 introduced standard names, known as binary prefixes, for these power-of-two sizes: "kibi" for 1024 (2^{10}), "mebi" for 1,048,576 (2^{20}), then "gibi," "tebi," and more. Those prefixes haven't quite caught on yet.

Another unfortunate convention is the use of Kb to represent kilobits, Mb for megabits, etc. Note the lower-case "b" to represents *bits*, in contast to the upper-case "B" to represent bytes as in KB or MB. The difference is easy to overlook.

In case you are curious, the metric prefixes that follow *tera* are *peta* (10^{15}), *exa* (10^{18}), *zetta* (10^{21}), and *yotta* (10^{24}). Do you think your computer will ever have a yotta-byte-sized hard drive?

Appendix A discusses number representations further.

▶ *INTERNET PROTOCOL (IP) ADDRESSES*

Internet domain names have an IP (Internet Protocol) address that is 32 bits long (for IPv4, the most common IP addressing used today). To aid in human readability, the 32 bits are divided into four bytes, and each byte is written as its decimal number equivalent, with each number separated by a period. You may have therefore seen an address like 192.168.1.1. That address in binary is 11000110 10101000 00000001 00000001 (spaces are added for readability; they don't exist in the actual binary address). Given a website's domain name, a web browser looks up the domain name's IP address and then contacts that address.

32 bits can represent 2^{32}, or about 4 billion, unique addresses, which you might imagine soon may not be sufficient for all the computers and sites on the Internet. The newer IPv6 therefore uses 128 bit addresses, and is slowly being adopted.

▶ 1.3 IMPLEMENTING DIGITAL SYSTEMS: MICROPROCESSORS VERSUS DIGITAL CIRCUITS

Designers can implement a digital system for an application by choosing one of two common digital system implementation methods—programming a microprocessor, or creating a custom digital circuit, which is known as digital design.

As an example of this choice, consider a simple application that turns on a lamp whenever there is motion in a dark room. Assume a motion detector has an output wire named *a* that outputs a `1` bit when motion is detected, and a `0` bit otherwise. Assume a light sensor has an output wire *b* that outputs a `1` bit when light is sensed, and a `0` bit otherwise. And assume a wire *F* turns on the lamp when *F* is `1`, and turns off the lamp when `0`. A drawing of the system is shown in Figure 1.22(a).

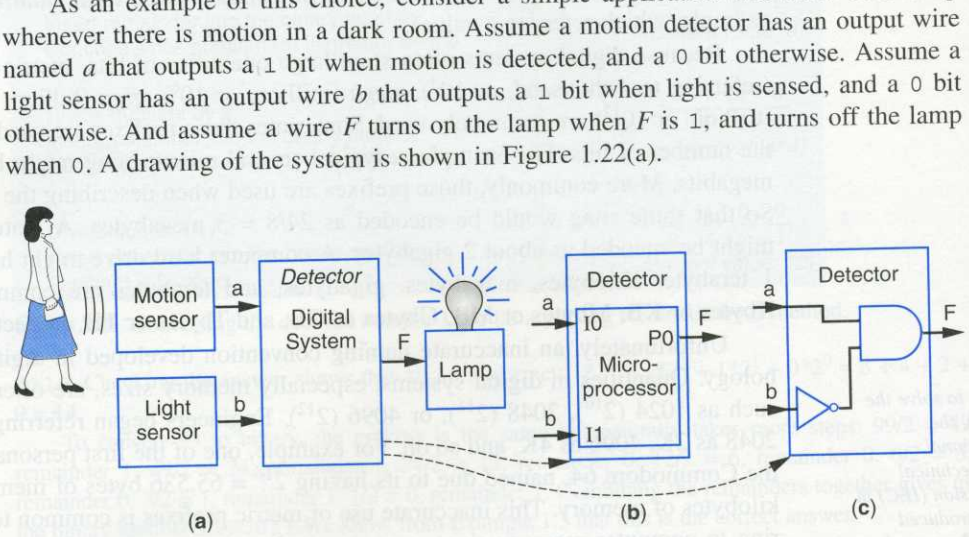


Figure 1.22 Motion-in-the-dark-detector system: (a) system block diagram, (b) implementation using a microprocessor, (c) implementation using a custom digital circuit.

The design problem is to determine what to put inside the block named *Detector*. The *Detector* block takes wires *a* and *b* as inputs, and generates a value on *F*, such that the light should turn on when motion is detected when dark. The *Detector* application is readily implemented as a digital system, because the application's inputs and outputs obviously are digital, having only two possible values each. A designer can implement the *Detector* block by programming a microprocessor (Figure 1.22(b)) or by creating a custom digital circuit (Figure 1.22(c)).

Software on Microprocessors: The Digital Workhorse

Designers who work with digital phenomena commonly buy an off-the-shelf microprocessor and write software for that microprocessor, rather than design a custom digital circuit. Microprocessors are really the "workhorse" of digital systems, handling most digital processing tasks. A ***microprocessor*** is a programmable digital device that executes a user-specified sequence of instructions, known as a *program* or as *software*. Some of those instructions read the microprocessor's inputs, others write to the microprocessor's outputs, and other instructions perform computations on the input data.

A "processor" processes, or transforms, data. A "microprocessor" is a programmable processor implemented on a single computer chip—the "micro" just means small here. The term "microprocessor" became popular in the 1980s when processors shrank down from multiple chips to just one. The first single-chip microprocessor was the Intel 4004 chip in 1971.

Figure 1.23(a) illustrates a basic microprocessor with eight input pins named *I0, I1, ..., I7*, and eight output pins named *P0, P1, ..., P7*. A photograph of a microprocessor package with such pins is shown in Figure 1.23(b) (the ninth pin on this side is for power, and on the other side for ground).

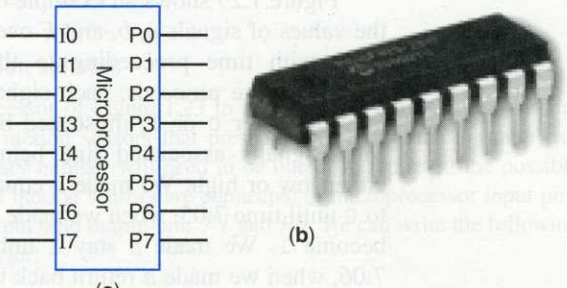


Figure 1.23 A basic microprocessor: (a) with eight outputs *P0-P7* that each can be set to a `0` or `1`, and eight inputs *I0-I7* too, (b) photograph of a real microprocessor package.

A microprocessor-based solution to the motion-in-the-dark detector application is illustrated in Figure 1.22(b), and a photograph of a physical implementation is shown in Figure 1.24. The designer connects the `a` wire to the microprocessor input pin *I0*, the `b` wire to input pin *I1*, and output pin *P0* to the *F* wire. The designer could then specify the instructions for the microprocessor by writing the following C code:

```
void main()
{
   while (1) {
     P0 = I0 && !I1; // F = a and !b,
   }
}
```

C is one of several popular languages for describing the desired instructions to execute on the microprocessor. The above C code works as follows. The microprocessor, after being powered up and reset, executes the instructions within `main`'s curly brackets { }. The first instruction is "`while (1)`" which means to forever repeat the instructions in the while's curly brackets. Inside those brackets is one instruction, "`P0 = I0 && !I1`," which assigns the microprocessor's output pin *P0* with a `1` if the input pin *I0* is `1` *and* (written as `&&`) the input pin *I1* is not `1` (meaning *I1* is `0`). Thus, the output pin *P0*, which turns the lamp on or off, forever gets assigned the appropriate value based on the input pin values, which come from the motion and light sensors.

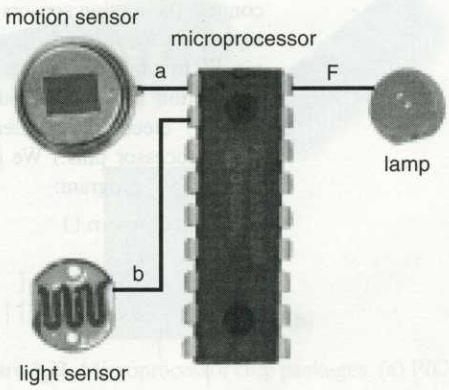


Figure 1.24 Physical motion-in-the-dark detector implementation using a microprocessor.

Figure 1.25 shows an example of the values of signals *a*, *b*, and *F* over time, with time proceeding to the right. As time proceeds, each signal may be either `0` or `1`, illustrated by each signal's associated line being either low or high. We made *a* equal to `0` until time 7:05, when we made *a* become `1`. We made *a* stay `1` until 7:06, when we made *a* return back to `0`. We made *a* stay `0` until 9:00, when we made *a* become `1` again, and then we made *a* become `0` at 9:01. On the other hand, we made *b* start as `0`, and then become `1` between 7:06 and 9:00. The diagram shows what the value of *F* would be given the C program executing on the microprocessor—when *a* is `1` and *b* is `0` (from 7:05 to 7:06), *F* will be `1`. A diagram with time proceeding to the right, and the values of digital signals shown by high or low lines, is known as a ***timing diagram***. We draw the input lines (*a* and *b*) to be whatever values we want, but then the output line (*F*) must describe the behavior of the digital system.

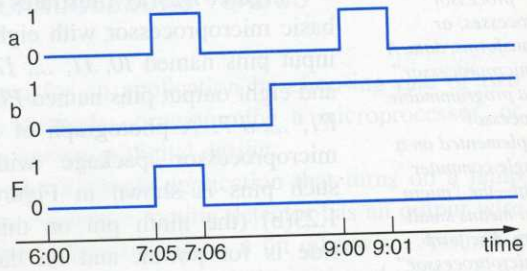


Figure 1.25 Timing diagram of motion-in-the-dark detector system.

Example 1.10 Outdoor motion notifier using a microprocessor

Let's use the basic microprocessor of Figure 1.23 to implement a system that sounds a buzzer when motion is detected at any of three motion sensors outside a house. We connect the motion sensors to microprocessor input pins *I0*, *I1*, and *I2*, and connect output pin *P0* to a buzzer (Figure 1.26). (We assume the motion sensors and buzzers have appropriate electronic interfaces to the microprocessor pins.) We can then write the following C program:

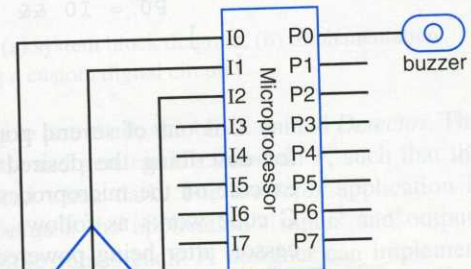


Figure 1.26 Motion sensors connected to microprocessor.

```
void main()
{
   while (1) {
     P0 = I0 || I1 || I2;
   }
}
```

The program executes the statement inside the while loop repeatedly. That statement will set *P0* to `1` if *I0* is `1` *or* (written as `||` in the C language) *I1* is `1` or *I2* is `1`, otherwise the statement sets *P0* to `0`.

Example 1.11 Counting the number of active motion sensors

This example uses the basic microprocessor of Figure 1.23 to implement a simple digital system that outputs in binary the number of motion sensors that presently detect motion. Assume two motion sensors, meaning a two-bit binary number will need to be output to represent the possible counts 0 (00), 1 (01), and 2 (10). The motion sensors are connected to microprocessor input pins *I0* and *I1*, and the binary number is output onto output pins *P1* and *P0*. We can write the following C program to achieve the desired behavior:

```
void main()
{
   while (1) {
      if (!I0 && !I1) {
         P1 = 0; P0 = 0; // output 00, meaning zero
      }
      else if ( (I0 && !I1) || (!I0 && I1) ) {
         P1 = 0; P0 = 1; // output 01, meaning one
      }
      else if (I0 && I1) {
         P1 = 1; P0 = 0; // output 10, meaning two
      }
   }
}
```

Intel named their evolving 1980s/90s desktop processors using numbers: 80286, 80386, 80486. As PCs became popular, Intel switched to catchier names: the 80586 was called a Pentium ("penta" means 5), followed by the Pentium Pro, the Pentium II, Core 2 Due, and others. Eventually, the names dominated over the numbers.

Designers like to use microprocessors in their digital systems because microprocessors are readily available, inexpensive, easy to program, and easy to reprogram. It may surprise you to learn that you can buy certain microprocessor chips for under $1. Such microprocessors are found in places like telephone answering machines, microwave ovens, cars, toys, certain medical devices, and even in shoes with blinking lights. Examples include the 8051 (originally designed by Intel), the 68HC11 (made by Motorola), the PIC (made by MicroChip), and the AVR (made by Atmel). Other microprocessors may cost tens of dollars, found in places like cell phones, portable digital assistants, office automation equipment, and medical equipment. Such processors include the ARM (made by the ARM corporation),

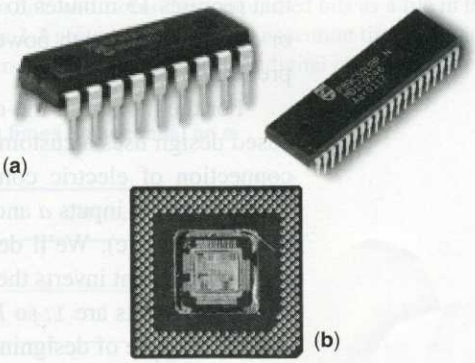


Figure 1.27 Microprocessor chip packages: (a) PIC and 8051 microprocessors, costing about $1 each, (b) a Pentium processor with part of its package cover removed, showing the silicon chip inside.

the MIPS (made by the MIPS corporation), and others. Other microprocessors, like the well-known Pentium or Core 2 Quad processors from Intel, may cost several hundred dollars and may be found in desktop computers. Some microprocessors may cost several thousand dollars and are found in a mainframe computer running, perhaps, an airline reservation system. There are many hundreds of different microprocessor types available, differing in performance, cost, power, and other metrics. And many of the small low-power processors cost under $1.

Some readers of this book may be familiar with software programming of microprocessors, others may not. Knowledge of microprocessor programming is not essential to learning the material in this book. We will occasionally compare custom digital circuits with their corresponding microprocessor implementations—the conclusions of those comparisons can be understood without knowledge of programming itself.

Digital Design—When Microprocessors Aren't Good Enough

With a variety of microprocessors readily available, why would anyone ever need to design new digital circuits, other than those relatively few people designing microprocessor digital circuits themselves? The reason is that software running on a microprocessor isn't always good enough for a particular application. In many cases, software may be too slow. Microprocessors only execute one instruction or a few instructions at a time. But a custom digital circuit can execute hundreds or thousands of computations in parallel. Many applications, like picture or video compression, face recognition, voice command detection, or graphics display, require huge numbers of computations to be done in a short period of time in order to be practical—after all, who wants a voice-controlled phone that requires 5 minutes to decode your voice command, or a digital camera that requires 15 minutes to take each picture? In other cases, microprocessors are too big, or consume too much power, or would be too costly, thus making custom digital circuits preferable.

For the motion-in-the-dark-detector application, an alternative to the microprocessor-based design uses a custom digital circuit inside the *Detector* block. A ***circuit*** is an interconnection of electric components. We must design a circuit that, for each different combination of inputs a and b, generates the proper value on F. One such circuit is shown in Figure 1.22(c). We'll describe the components in that circuit later; briefly, the triangular component inverts the value on b, and the bullet-shaped component outputs a `1` only if both its inputs are `1`, so F will be `1` only if a is `1` and b is `0`. But you've now seen one simple example of designing a digital circuit to solve a design problem. A microprocessor also has a circuit inside, but because that circuit is designed to execute programs rather than just detect motion at night, a small microprocessor's circuit may contain about ten thousand components, compared to just two components in our custom digital circuit of Figure 1.22(c). Thus, our custom digital circuit may be smaller, cheaper, and faster, and consume less power than an implementation on a microprocessor.

Example 1.12 Deciding among a microprocessor and custom digital circuit

We are asked to design a digital system to control a fighter jet's aircraft wing. In order to properly control the aircraft, the digital system must execute a computation task 100 times per second that adjusts the wing's position based on the aircraft's present and desired speeds, pitch, yaw, and other flight factors. Suppose we estimate that software on a microprocessor would require 50 ms (milliseconds) for each execution of the computation task, whereas a custom digital circuit would require 5 ms per execution.

Executing the computation task 100 times on the microprocessor would require 100 * 50 ms = 5000 ms, or 5 seconds. But we require those 100 executions to be done in 1 second, so the microprocessor is not fast enough. Executing the task 100 times with the custom digital circuit would require 100 * 5 ms = 500 ms, or 0.5 seconds. As 0.5 seconds is less than 1 second, the custom digital circuit can satisfy the system's performance constraint. We thus choose to implement the digital system as a custom digital circuit rather than by using a microprocessor.

Many applications use both microprocessors and custom digital circuits to attain a system that achieves just the right balance of performance, cost, power, size, design time, flexibility, etc.

Example 1.13 Partitioning tasks in a digital camera

A digital camera captures pictures digitally using several steps. When the shutter button is pressed, a grid of a few million light-sensitive electronic elements capture the image, each element storing a binary number (perhaps 16 bits) representing the intensity of light hitting the element. The camera then performs several tasks: the camera *reads* the bits of each of these elements, *compresses* the tens of millions of bits into perhaps a few million bits, and *stores* the compressed bits as a file in the camera's flash memory, among other tasks. Table 1.2 provides sample task execution times running on an inexpensive low-power microprocessor versus executing as a custom digital circuit.

Table 1.2 Sample digital camera task execution times (in seconds) on a microprocessor versus a digital circuit.

Task	Microprocessor	Custom digital circuit
Read	5 s	0.1 s
Compress	8 s	0.5 s
Store	1 s	0.8 s

We need to decide which tasks to implement on the microprocessor and which to implement as a custom digital circuit, subject to the constraint that we should strive to minimize the amount of custom digital circuitry in order to reduce chip costs. Such decisions are known as ***partitioning***. Three partitioning options are shown in Figure 1.28. If we implement all three tasks on the microprocessor, the camera will require 5 + 8 + 1 = 14 seconds to take a picture—too much time for the camera to be popular with consumers. We could implement all the tasks as custom digital circuits, resulting in 0.1 + 0.5 + 0.8 = 1.4 seconds. We could instead implement the read and compress tasks

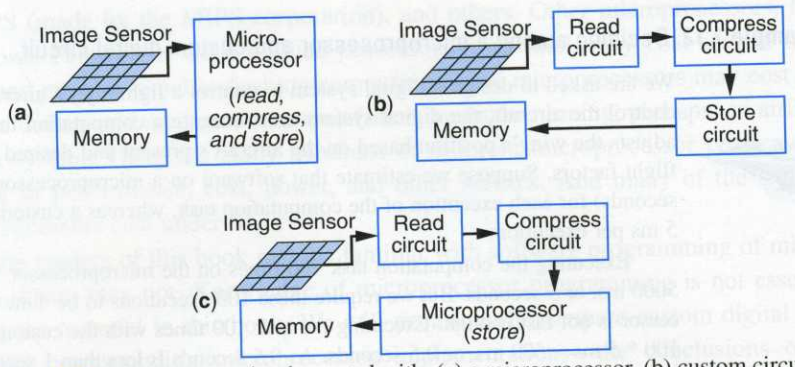


Figure 1.28 Digital camera implemented with: (a) a microprocessor, (b) custom circuits, and (c) a combination of custom circuits and a microprocessor.

with custom digital circuits, while leaving the store task to the microprocessor, resulting in 0.1 + 0.5 + 1, or 1.6 seconds.

We might decide on this last implementation option, to save cost without much noticeable time overhead. ■

▶ 1.4 ABOUT THIS BOOK

Section 1.1 discussed how digital systems now appear nearly everywhere and significantly impact the way we live. Section 1.2 highlighted how learning digital design accomplishes two goals: showing us how microprocessors work "under the hood," and enabling us to implement systems using custom digital circuits instead of (or along with) microprocessors to achieve better implementations. This latter goal is becoming increasingly significant since so many analog phenomena, like music and video, are becoming digital. That section also introduced a key method of digitizing analog signals, namely binary numbers, and described how to convert between decimal and binary numbers, as well as between numbers of any two bases. Section 1.3 described how designers may prefer to implement digital systems by writing software that executes on a microprocessor, yet designers often use custom digital circuits to meet an application's performance requirements or other requirements.

In the remainder of this book you will learn about the exciting and challenging field of digital design, wherein we convert desired system functionality into a custom digital circuit. Chapter 2 will introduce the most basic form of digital circuit, combinational circuits, whose outputs are each a function of the present values on the circuit's inputs. That chapter will show how to use a form of math called Boolean algebra to describe our desired circuit functionality, and will provide clear steps for converting Boolean equations to circuits. Chapter 3 will introduce a more advanced type of circuit, sequential circuits, whose outputs are a function not only of the present input values, but also of previous input values—in other words, sequential circuits have memory. Such circuits are commonly referred to as controllers. That chapter will show us how to use another mathematical abstraction, known as a finite-state machine, to represent desired sequential

functionality, and will provide clear steps for converting finite-state machines to circuits. As with any form of design, we often use pre-designed building blocks, or components, to make our design task easier. Chapter 4 describes several such components, known as datapath components, including registers (for storing digital data), adders, comparators, multipliers, and small memories called register files, among other blocks. Chapter 5 introduces the modern approach to digital design, known as register-transfer level design, wherein we design systems consisting of datapath components controlled by controllers, to implement interesting and useful custom digital circuits. In fact, that chapter shows how to convert a C program to a custom digital circuit—clearly demonstrating that any desired function can be implemented as software on a microprocessor or as a custom digital circuit. That chapter also introduces some additional components, including ROM and RAM memories, and queues. Chapters 1 through 5 form the core of this book—after those five chapters, the reader can specify a wide variety of desired functionality and can convert that functionality to a working custom digital circuit.

Chapter 6 introduces methods for designing *better* digital circuits. The chapter describes methods for improving basic combinational circuits, basic sequential circuits, datapath components, and register-transfer level designs. That chapter emphasizes the important notion of *tradeoffs*, wherein we might make one aspect of the design better, but at the expense of worsening another aspect. Tradeoffs are the essence of design.

Chapter 7 describes different physical devices on which we can implement our digital circuits, including application-specific integrated circuits, field-programmable gate-arrays (FPGAs), simple programmable logic devices, and cheap off-the-shelf ICs.

Chapter 8 applies the digital design methods of earlier chapters to build a common type of digital circuit—a programmable microprocessor. The chapter demystifies the workings of a microprocessor, using a very simple design to illustrate the concepts.

Chapter 9 introduces hardware description languages, which are widely used in modern digital design for describing desired circuit functionality as well as for representing the final custom digital circuit design. Hardware description languages, looking much like software programming languages but with important extensions and differences, serve as the input to most modern digital design tools.

▶ 1.5 EXERCISES

SECTION 1.2: THE WORLD OF DIGITAL SYSTEMS

1.1 What is a digital signal, and how does it differ from an analog signal? Give two everyday examples of digital phenomena (e.g., a window can be open or closed) and two everyday examples of analog phenomena.

1.2 Suppose an analog audio signal comes in over a wire, and the voltage on the wire can range from 0 Volts (V) to 3 V. You want to convert the analog signal to a digital signal. You decide to encode each sample using two bits, such that 0 V would be encoded as `00`, 1 V as `01`, 2 V as `10`, and 3 V as `11`. You sample the signal every 1 millisecond and detect the following sequence of voltages: 0V 0V 1V 2V 3V 2V 1V. Show the signal converted to digital as a stream of `0`s and `1`s.