Inkjet printers come in two varieties: piezoelectric (used by Epson) and thermal (used by Canon, HP, and Lexmark). The piezoelectric inkjet printers have a special kind of crystal next to the ink chamber. When a voltage is applied to the crystal, it deforms slightly, forcing a droplet of ink out of the nozzle. The higher the voltage, the larger the droplet, allowing the software to control the droplet size.

Thermal inkjet printers (also called **bubblejet** printers) contain a tiny resistor inside each nozzle. When a voltage is applied to the resistor, it heats up extremely fast, instantly raising the temperature of the ink touching it to the boiling point until the ink vaporizes to form a gas bubble. The gas bubble takes up more volume than the ink that created it, producing pressure in the nozzle. The only place the ink can go is out the front of the nozzle onto the paper. The nozzle is then cooled and the resulting vacuum sucks in another ink droplet from the ink tank. The speed of the printer is limited by how fast the boil/cool cycle can be repeated. The droplets are all the same size, but smaller than what the piezoelectric printers use.

Inkjet printers typically have resolutions of at least 1200 **dpi** (**dots per inch**) and at the high end, 4800 dpi. They are cheap, quiet, and have good quality, although they are also slow, and use expensive ink cartridges. When the best of the high-end inkjet printers is used to print a professional high-resolution photograph on specially coated photographic paper, the results are indistinguishable from conventional photography, even up to 8×10 inch prints.

For best results, special ink and paper should be used. Two kinds of ink exist. **Dye-based inks** consist of colored dyes dissolved in a fluid carrier. They give bright colors and flow easily. Their main disadvantage is that they fade when exposed to ultraviolet light, such as that contained in sunlight. **Pigment-based inks** contain solid particles of pigment suspended in a fluid carrier that evaporates from the paper, leaving the pigment behind. These inks do not fade over time but are not as bright as dye-based inks and the pigment particles have a tendency to clog the nozzles, requiring periodic cleaning. Coated or glossy paper is required for printing photographs properly. These kinds of paper have been specially designed to hold the ink droplets and not let them spread out.

Specialty Printers

While laser and inkjet printers dominate the home and office printing markets, other kinds of printers are used in other situations that have other requirements in terms of color quality, price, and other characteristics.

A variant on the inkjet printer is the **solid ink printer**. This kind of printer accepts four solid blocks of a special waxy ink which are then melted into hot ink reservoirs. Startup times of these printers can be as much as 10 minutes, while the ink blocks are melting. The hot ink is sprayed onto the paper, where it solidifies and is fused with the paper by forcing it between two hard rollers. In a way, it combines the idea of ink spraying from inkjet printers and the idea of fusing the ink onto the paper with hard rubber rollers from laser printers.

Another printer is the **wax printer**. It has a wide ribbon of four-color wax that is segmented into page-size bands. Thousands of heating elements melt the wax as the paper moves under it. The wax is fused to the paper in the form of pixels using the CMYK system. Wax printers used to be the main color-printing technology, but they are being replaced by the other kinds with cheaper consumables.

Still another kind of color printer is the **dye sublimation printer**. Although it has Freudian undertones, sublimation is the scientific name for a solid changing into a gas without passing through the liquid state. Dry ice (frozen carbon dioxide) is a well-known material that sublimates. In a dye sublimation printer, a carrier containing the CMYK dyes passes over a thermal print head containing thousands of programmable heating elements. The dyes are vaporized instantly and absorbed by a special paper close by. Each heating element can produce 256 different temperatures. The higher the temperature, the more dye that is deposited and the more intense the color. Unlike all the other color printers, nearly continuous colors are possible for each pixel, so no halftoning is needed. Small snapshot printers often use the dye sublimation process to produce highly realistic photographic images on special (and expensive) paper.

Finally, we come to the **thermal printer**, which contains a small print head with some number of tiny heatable needles on it. When an electric current is passed through a needle, it gets very hot very fast. When a special thermally sensitive paper is pulled past the print head, dots are made on the paper when the needles are hot. In effect, a thermal printer is like the old matrix printers whose pins pressed against a typewriter ribbon to make dots on the paper behind the ribbon. Thermal printers are widely used to print receipts in stores, ATM machines, automated gas stations, etc.

2.4.6 Telecommunications Equipment

Most computers nowadays are connected to a computer network, often the Internet. Achieving this access requires special equipment. In this section we will see how this equipment works.

Modems

With the growth of computer usage in the past years, it is common for one computer to need to communicate with another computer. For example, many people have personal computers at home that they use for communicating with their computer at work, with an Internet Service Provider, or with a home banking system. In many cases, the telephone line provides the physical communication.

However, a raw telephone line (or cable) is not suitable for transmitting computer signals, which generally represent a 0 as 0 volts and a 1 as 3 to 5 volts as shown in Fig. 2-38(a). Two-level signals suffer considerable distortion when transmitted over a voice-grade telephone line, thereby leading to transmission errors. A

pure sine-wave signal at a frequency of 1000 to 2000 Hz, called a **carrier**, can be transmitted with relatively little distortion, however, and this fact is exploited as the basis of most telecommunication systems.

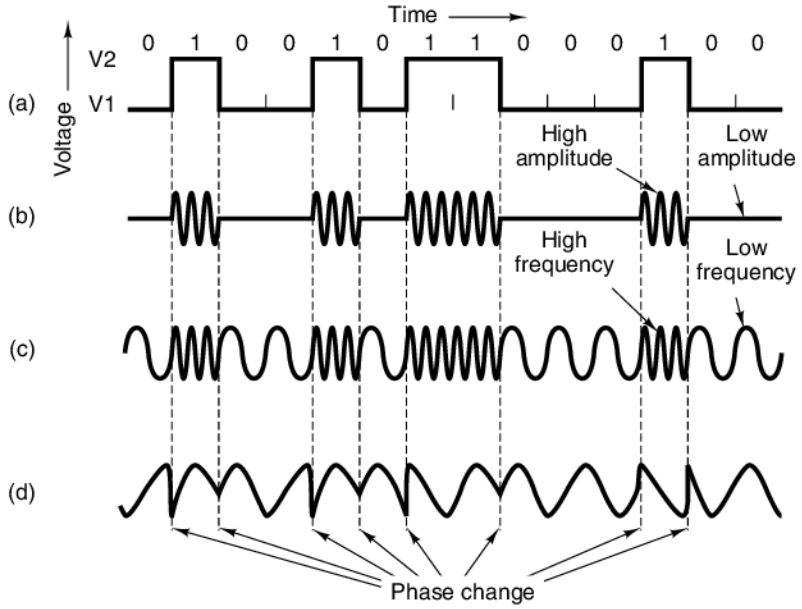


Figure 2-38. Transmission of the binary number 01001011000100 over a telephone line bit by bit. (a) Two-level signal. (b) Amplitude modulation. (c) Frequency modulation. (d) Phase modulation.

Because the pulsations of a sine wave are completely predictable, a pure sine wave transmits no information at all. However, by varying the amplitude, frequency, or phase, a sequence of 1s and 0s can be transmitted, as shown in Fig. 2-38. This process is called **modulation** and the device that does it is called a **modem**, which stands for **MOdulator DEModulator**. In **amplitude modulation** [see Fig. 2-38(b)], two different voltage levels are used, for 0 and 1, respectively. A person listening to digital data transmitted at a very low data rate would hear a loud noise for a 1 and no noise for a 0.

In **frequency modulation** [see Fig. 2-38(c)], the voltage level is constant but the carrier frequency is different for 1 and 0. A person listening to frequency modulated digital data would hear two tones, corresponding to 0 and 1. Frequency modulation is often referred to as **frequency shift keying**.

In simple **phase modulation** [see Fig. 2-38(d)], the amplitude and frequency do not change, but the phase of the carrier is reversed 180 degrees when the data switch from 0 to 1 or 1 to 0. In more sophisticated phase-modulated systems, at the start of each indivisible time interval, the phase of the carrier is abruptly shifted

by 45, 135, 225, or 315 degrees, to allow 2 bits per time interval, called **dibit** phase encoding. For example, a phase shift of 45 degrees could represent 00, a phase shift of 135 degrees could represent 01, and so on. Schemes for transmitting 3 or more bits per time interval also exist. The number of time intervals (i.e., the number of potential signal changes per second) is the **baud** rate. With 2 or more bits per interval, the bit rate will exceed the baud rate. Many people confuse these two terms. Again: the baud rate is the number of times the signal changes per second whereas the bit rate is the number of bits transmitted per second. The bit rate is generally a multiple of the baud rate, but theoretically it can be lower.

If the data to be transmitted consist of a series of 8-bit characters, it would be desirable to have a connection capable of transmitting 8 bits simultaneously—that is, eight pairs of wires. Because voice-grade telephone lines provide only one channel, the bits must be sent serially, one after another (or in groups of two if dibit encoding is being used). The device that accepts characters from a computer in the form of two-level signals, 1 bit at a time, and transmits the bits in groups of 1 or 2, in amplitude-, frequency-, or phase-modulated form, is the modem. To mark the start and end of each character, an 8-bit character is normally sent preceded by a start bit and followed by a stop bit, making 10 bits in all.

The transmitting modem sends the individual bits within one character at regularly spaced time intervals. For example, 9600 baud implies one signal change every 104 μsec. A second modem at the receiving end is used to convert a modulated carrier to a binary number. Because the bits arrive at the receiver at regularly spaced intervals, once the receiving modem has determined the start of the character, its clock tells it when to sample the line to read the incoming bits.

Modern modems run at 56 kbps, usually at much lower baud rates. They use a combination of techniques to send multiple bits per baud, modulating the amplitude, frequency, and phase. All are **full-duplex**, meaning they can transmit in both directions at the same time (on different frequencies). Modems or transmission lines that can transmit only in one direction at a time (like a single-track railroad that can handle northbound trains or southbound trains but not at the same time) are called **half-duplex**. Lines that can transmit in only one direction are **simplex**.

Digital Subscriber Lines

When the telephone industry finally got to 56 kbps, it patted itself on the back for a job well done. Meanwhile, the cable TV industry was offering speeds up to 10 Mbps on shared cables, and satellite companies were planning to offer upward of 50 Mbps. As Internet access became an increasingly important part of their business, the **telcos** (**telephone companies**) began to realize they needed a more competitive product than dialup lines. Their answer was to start offering a new digital Internet access service. Services with more bandwidth than standard telephone service are sometimes called **broadband**, although the term really is more of a marketing concept than anything else. From a very narrow technical perspective,

broadband means there are multiple signaling channels whereas baseband means there is only one. Thus in theory, 10-gigabit Ethernet, which is far faster than any telephone-company-provided "broadband" service, is not broadband at all since it has only one signaling channel.

Initially, there were many overlapping offerings, all under the general name of **xDSL** (**Digital Subscriber Line**), for various x. Below we will discuss what has become the most popular of these services, **ADSL** (**Asymmetric DSL**). ADSL is still being developed and not all the standards are fully in place, so some of the details given below may change in time, but the basic picture should remain valid. For more information about ADSL, see Summers (1999) and Vetter et al. (2000).

The reason that modems are so slow is that telephones were invented for carrying the human voice and the entire system has been carefully optimized for this purpose. Data have always been stepchildren. The wire, called the **local loop**, from each subscriber to the telephone company's office has traditionally been limited to about 3000 Hz by a filter in the telco office. It is this filter that limits the data rate. The actual bandwidth of the local loop depends on its length, but for typical distances of a few kilometers, 1.1 MHz is feasible.

The most common approach to offering ADSL is illustrated in Fig. 2-39. In effect, what it does is remove the filter and divide the available 1.1-MHz spectrum on the local loop into 256 independent channels of 4312.5 Hz each. Channel 0 is used for **POTS** (**Plain Old Telephone Service**). Channels 1–5 are not used, to keep the voice signal and data signals from interfering with each other. Of the remaining 250 channels, one is used for upstream control and one for downstream control. The rest are available for user data. ADSL is like having 250 modems.

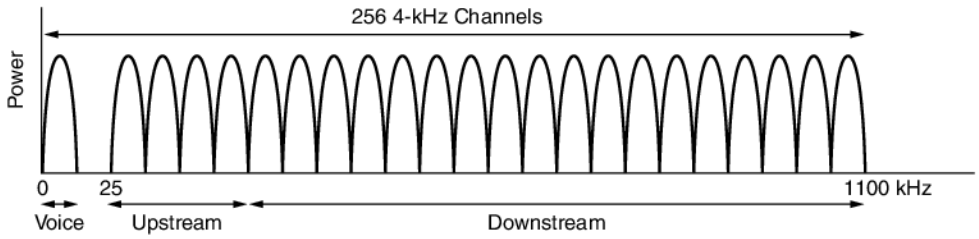


Figure 2-39. Operation of ADSL.

In principle, each of the remaining channels can be used for a full-duplex data stream, but harmonics, crosstalk, and other effects keep practical systems well below the theoretical limit. It is up to the provider to determine how many channels are used for upstream and how many for downstream. A 50–50 mix of upstream and downstream is technically possible, but most providers allocate something like 80%–90% of the bandwidth to the downstream channel since most users download more data than they upload. This choice gives rise to the "A" in ADSL. A common split is 32 channels for upstream and the rest for downstream.

Within each channel the line quality is constantly monitored and the data rate adjusted continuously as needed, so different channels may have different data rates. The actual data are sent using a combination of amplitude and phase modulation with up to 15 bits per baud. With, for example, 224 downstream channels and 15 bits/baud at 4000 baud, the downstream bandwidth is 13.44 Mbps. In practice, the signal-to-noise ratio is never good enough to achieve this rate, but 4–8 Mbps is possible on short runs over high-quality loops.

A typical ADSL arrangement is shown in Fig. 2-40. In this scheme, the user or a telephone company technician must install a **NID** (**Network Interface Device**) on the customer's premises. This small plastic box marks the end of the telephone company's property and the start of the customer's property. Close to the NID (or sometimes combined with it) is a **splitter**, an analog filter that separates the 0–4000 Hz band used by POTS from the data. The POTS signal is routed to the existing telephone or fax machine, and the data signal is routed to an ADSL modem. The ADSL modem is actually a digital signal processor that has been set up to act as 250 modems operating in parallel at different frequencies. Since most current ADSL modems are external, the computer must be connected to it at high speed. Usually, this is done by putting an Ethernet card in the computer and operating a very short two-node Ethernet containing only the computer and ADSL modem. (Ethernet is a popular and inexpensive local area network standard.) Occasionally the USB port is used instead of Ethernet. In the future, internal ADSL modem cards will no doubt become available.

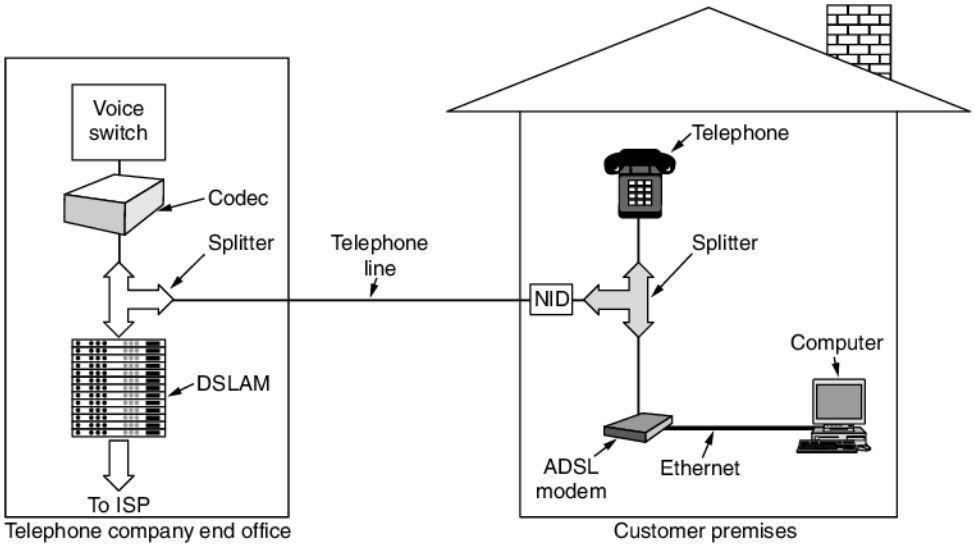


Figure 2-40. A typical ADSL equipment configuration.

At the other end of the wire, on the telco side, a corresponding splitter is installed. Here the voice portion of the signal is filtered out and sent to the normal

voice switch. The signal above 26 kHz is routed to a new kind of device called a **DSLAM** (**Digital Subscriber Line Access Multiplexer**), which contains the same kind of digital signal processor as the ADSL modem. Once the digital signal has been recovered into a bit stream, packets are formed and sent off to the ISP.

Internet over Cable

Many cable TV companies are now offering Internet access over their cables. Since the technology is quite different from ADSL, it is worth looking at briefly. The cable operator in each city has a main office and a large number of boxes full of electronics, called **headends**, spread all over its territory. The headends are connected to the main office by high-bandwidth cables or fiber optics.

Each headend has one or more cables that run from it past hundreds of homes and offices. Each cable customer taps onto the cable as it passes the customer's premises. Thus hundreds of users share the same cable to the headend. Usually, the cable has a bandwidth of about 750 MHz. This system is radically different from ADSL because each telephone user has a private (i.e., not shared) wire to the telco office. However, in practice, having your own 1.1-MHz channel to a telco office is not that different than sharing a 200-MHz piece of cable spectrum to the headend with 400 users, half of whom are not using it at any one instant. It does mean, however, that a cable Internet user will get much better service at 4 A.M. than at 4 P.M, whereas ADSL service is constant all day long. People intent on getting optimal Internet over cable service might wish to consider moving to a rich neighborhood (houses far apart so fewer customers per cable) or a poor neighborhood (nobody can afford Internet service).

Since the cable is a shared medium, determining who may send when and at which frequency is a big issue. To see how that works, we have to briefly describe how cable TV operates. Cable television channels in North America normally occupy the 54–550 MHz region (except for FM radio from 88 to 108 MHz). These channels are 6 MHz wide, including guard bands to prevent signal leakage between channels. In Europe the low end is usually 65 MHz and the channels are 6–8 MHz wide for the higher resolution required by PAL and SECAM, but otherwise the allocation scheme is similar. The low part of the band is not used for television transmission.

When introducing Internet over cable, the cable companies had two problems to solve:

1. How to add Internet access without interfering with TV programs.
2. How to have two-way traffic when amplifiers are inherently one way.

The solutions chosen are as follows. Modern cables have a bandwidth of at least 550 MHz, often as much as 750 MHz or more. The upstream (i.e., user to headend) channels go in the 5–42 MHz band (but slightly higher in Europe) and the

downstream (i.e., headend to user) traffic uses the frequencies at the high end, as illustrated in Fig. 2-41.

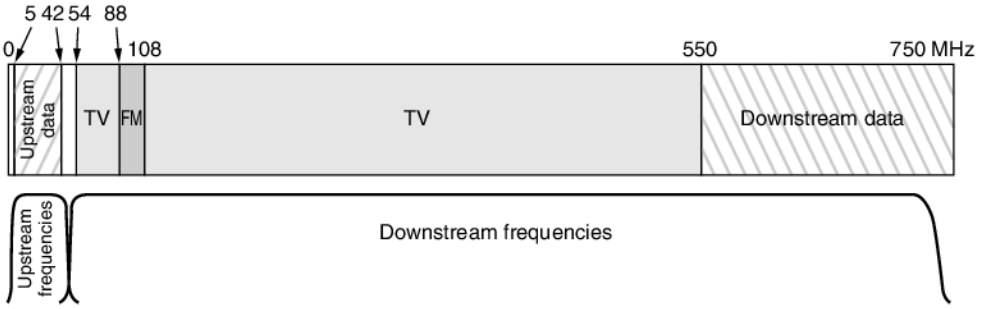


Figure 2-41. Frequency allocation in a typical cable TV system used for Internet access.

Note that since the television signals are all downstream, it is possible to use upstream amplifiers that work only in the 5–42 MHz region and downstream amplifiers that work only at 54 MHz and up, as shown in the figure. Thus, we get an asymmetry in the upstream and downstream bandwidths because more spectrum is available above television than below it. On the other hand, most of the traffic is likely to be downstream, so cable operators are not unhappy with this fact of life. As we saw earlier, telephone companies usually offer an asymmetric DSL service, even though they have no technical reason for doing so.

Internet access requires a cable modem, a device that has two interfaces on it: one to the computer and one to the cable network. The computer-to-cable-modem interface is straightforward. It is normally Ethernet, just as with ADSL. In the future, the entire modem might be a small card plugged into the computer, just as with the old telephone modems.

The other end is more complicated. A large part of the cable standard deals with radio engineering, a subject far beyond the scope of this book. The only part worth mentioning here is that cable modems, like ADSL modems, are always on. They make a connection when turned on and maintain that connection as long as they are powered up because cable operators do not charge for connect time.

To better understand how they work, let us see what happens when a cable modem is plugged in and powered up. The modem scans the downstream channels looking for a special packet periodically put out by the headend to provide system parameters to modems that have just come online. Upon finding this packet, the new modem announces its presence on one of the upstream channels. The headend responds by assigning the modem to its upstream and downstream channels. These assignments can be changed later if the headend deems it necessary to balance the load.

The modem then determines its distance from the headend by sending it a special packet and seeing how long it takes to get the response. This process is called

ranging. It is important for the modem to know its distance to accommodate the way the upstream channels operate and to get the timing right. The channels are divided in time in **minislots**. Each upstream packet must fit in one or more consecutive minislots. The headend announces the start of a new round of minislots periodically, but the starting gun is not heard at all modems simultaneously due to the propagation time down the cable. By knowing how far it is from the headend, each modem can compute how long ago the first minislot really started. Minislot length is network dependent. A typical payload is 8 bytes.

During initialization, the headend also assigns each modem to a minislot to use for requesting upstream bandwidth. As a rule, multiple modems will be assigned the same minislot, which leads to contention. When a computer wants to send a packet, it transfers the packet to the modem, which then requests the necessary number of minislots for it. If the request is accepted, the headend puts an acknowledgement on the downstream channel telling the modem which minislots have been reserved for its packet. The packet is then sent, starting in the minislot allocated to it. Additional packets can be requested using a field in the header.

On the other hand, if there is contention for the request minislot, there will be no acknowledgement and the modem just waits a random time and tries again. After each successive failure, the randomization time is doubled to spread out the load when there is heavy traffic.

The downstream channels are managed differently from the upstream channels. For one thing, there is only one sender (the headend) so there is no contention and no need for minislots, which is actually just time-division statistical multiplexing. For another, the traffic downstream is usually much larger than upstream, so a fixed packet size of 204 bytes is used. Part of that is a Reed-Solomon error-correcting code and some other overhead, leaving a user payload of 184 bytes. These numbers were chosen for compatibility with digital television using MPEG-2, so the TV and downstream data channels are formatted the same way. Logically, the connections are as depicted in Fig. 2-42.

Getting back to modem initialization, once the modem has completed ranging and gotten its upstream channel, downstream channel, and minislot assignments, it is free to start sending packets. These packets go to the headend, which relays them over a dedicated channel to the cable company's main office and then to the ISP (which may be the cable company itself). The first packet is one to the ISP requesting a network address (technically, an IP address), which is dynamically assigned. It also requests and gets an accurate time of day.

The next step involves security. Since cable is a shared medium, anybody who wants to go to the trouble to do so can read all the traffic zipping past him. To prevent everyone from snooping on their neighbors (literally), all traffic is encrypted in both directions. Part of the initialization procedure involves establishing encryption keys. At first one might think that having two strangers, the headend and the modem, establish a secret key in broad daylight with thousands of people watching would be impossible to accomplish. Turns out it is not, but the technique

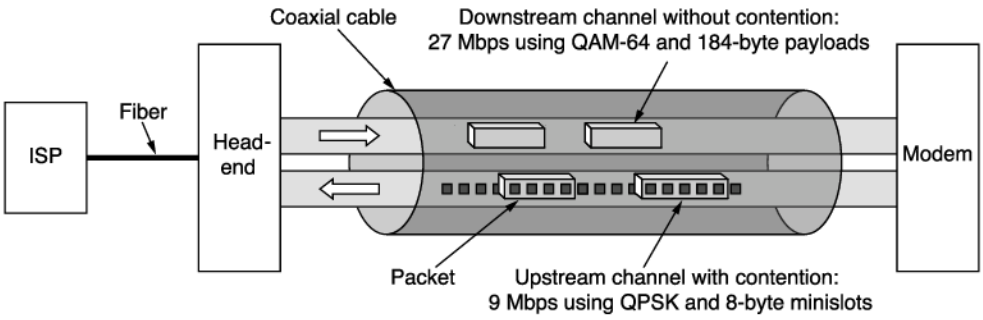


Figure 2-42. Typical details of the upstream and downstream channels in North America. QAM-64 (Quadrature Amplitude Modulation) allows 6 bits/Hz but works only at high frequencies. QPSK (Quadrature Phase Shift Keying) works at low frequencies but allows only 2 bits/Hz.

used (the Diffie-Hellman algorithm) is beyond the scope of this book. See Kaufman et al. (2002) for a discussion of it.

Finally, the modem has to log in and provide its unique identifier over the secure channel. At this point the initialization is complete. The user can now log in to the ISP and get to work.

There is much more to be said about cable modems. Some relevant references are Adams and Dulchinos (2001), Donaldson and Jones (2001), and Dutta-Roy (2001).

2.4.7 Digital Cameras

An increasingly popular use of computers is for digital photography, making digital cameras a kind of computer peripheral. Let us briefly see how that works. All cameras have a lens that forms an image of the subject in the back of the camera. In a conventional camera, the back of the camera is lined with film, on which a latent image is formed when light strikes it. The latent image can be made visible by the action of certain chemicals in the film developer. A digital camera works the same way except that the film is replaced by a rectangular array of **CCDs** (**Charge-Coupled Devices**) that are sensitive to light. (Some digital cameras use CMOS, but we will concentrate on the more common CCDs here.)

When light strikes a CCD, it acquires an electrical charge. The more light, the more charge. The charge can be read off by an analog-to-digital converter as an integer from 0 to 255 (on low-end cameras) or from 0 to 4095 (on digital single-lens-reflex cameras). The basic arrangement is shown in Fig. 2-43.

Each CCD produces a single value, independent of the color of light striking it. To form color images, the CCDs are organized in groups of four elements. A **Bayer filter** is placed on top of the CCD to allow only red light to strike one of the four CCDs in each group, blue light to strike another one, and green light to strike

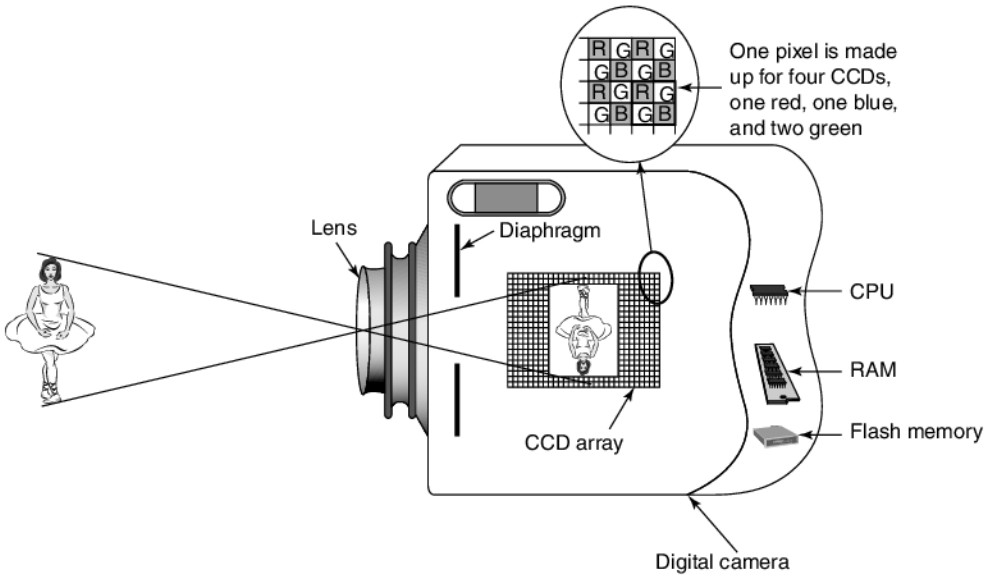


Figure 2-43. A digital camera.

the other two. Two greens are used because using four CCDs to represent one pixel is much more convenient than using three, and the eye is more sensitive to green light than to red or blue light. When a digital camera manufacturer claims a camera has, say, 6 million pixels, it is lying. The camera has 6 million CCDs, which together form 1.5 million pixels. The image will be read out as an array of 2828×2121 pixels (on low-end cameras) or 3000 times 2000 pixels (on digital SLRs), but the extra pixels are produced by interpolation by software inside the camera.

When the camera's shutter button is depressed, software in the camera performs three tasks: setting the focus, determining the exposure, and performing the white balance. The autofocus works by analyzing the high-frequency information in the image and then moving the lens until it is maximized, to give the most detail. The exposure is determined by measuring the light falling on the CCDs and then adjusting the lens diaphragm and exposure time to have the light intensity fall in the middle of the CCDs' range. Setting the white balance has to do with measuring the spectrum of the incident light to perform necessary color corrections in the postprocessing.

Then the image is read off the CCDs and stored as a pixel array in the camera's internal RAM. High-end digital SLRs used by photojournalists can shoot eight high-resolution frames per second for 5 seconds, and they need around 1 GB of internal RAM to store the images before processing and storing them permanently. Low-end cameras have less RAM, but still quite a bit.

In the post-capture phase, the camera's software applies the white-balance color correction to compensate for reddish or bluish light (e.g., from a subject in shadow or the use of a flash). Then it applies an algorithm to do noise reduction and another one to compensate for defective CCDs. After that, it attempts to sharpen the image (unless this feature has been disabled) by looking for edges and increasing the intensity gradient around them.

Finally, the image may be compressed to reduce the amount of storage required. A common format is **JPEG** (**Joint Photographic Experts Group**), in which a two-dimensional spatial Fourier transform is applied and some of the high-frequency components omitted. The result of this transformation is that the image requires fewer bits to store but fine detail is lost.

When all the in-camera processing is completed, the image is written to the storage medium, usually a flash memory or **microdrive**. The postprocessing and writing can take several seconds per image.

When the user gets home, the camera can be connected to a computer, usually using a USB or proprietary cable. The images are then transferred from the camera to the computer's hard disk. Using special software, such as Adobe Photoshop, the user can then crop the image, adjust brightness, contrast, and color balance, sharpen, blur, or remove portions of the image, and apply numerous filters. When the user is content with the result, the image files can be printed on a color printer, uploaded over the Internet to a photo-sharing Website or photofinisher, or written to CD-ROM or DVD for archival storage.

The amount of computing power, RAM, disk space, and software in a digital SLR camera is mind boggling. Not only does the computer have to do all the things mentioned above, but it also has to communicate with the CPU in the lens and the CPU in the flash, refresh the image on the LCD screen, and manage all the buttons, wheels, lights, displays, and gizmos on the camera in real time. This is an extremely powerful embedded system, often rivaling a desktop computer of only a few years earlier.

2.4.8 Character Codes

Each computer has a set of characters that it uses. As a bare minimum, this set includes the 26 uppercase letters, the 26 lowercase letters, the digits 0 through 9, and a set of special symbols, such as space, period, minus sign, comma, and carriage return.

In order to transfer these characters into the computer, each one is assigned a number: for example, a = 1, b = 2, ..., z = 26, + = 27, − = 28. The mapping of characters onto integers is called a **character code**. It is essential that communicating computers use the same code or they will not be able to understand one another. For this reason, standards have been developed. Below we will examine three of the most important ones.

ASCII

One widely used code is called **ASCII** (**American Standard Code for Information Interchange**). Each ASCII character has 7 bits, allowing for 128 characters in all. However, because computers are byte oriented, each ASCII character is normally stored in a separate byte. Figure 2-44 shows the ASCII code. Codes 0 to 1F (hexadecimal) are control characters and do not print. Codes from 128 to 255 are not part of ASCII, but the IBM PC defined them to be special characters like smiley faces and most computers still support them.

Many of the ASCII control characters are intended for data transmission. For example, a message might consist of an SOH (Start of Header) character, a header, an STX (Start of Text) character, the text itself, an ETX (End of Text) character, and then an EOT (End of Transmission) character. In practice, however, the messages sent over telephone lines and networks are formatted quite differently, so the ASCII transmission control characters are not used much any more.

The ASCII printing characters are straightforward. They include the upper- and lowercase letters, digits, punctuation marks, and a few math symbols.

Unicode

The computer industry grew up mostly in the U.S., which led to the ASCII character set. ASCII is fine for English but less fine for other languages. French needs accents (e.g., système); German needs diacritical marks (e.g., für), and so on. Some European languages have a few letters not found in ASCII, such as the German ß and the Danish ø. Some languages have entirely different alphabets (e.g., Russian and Arabic), and a few languages have no alphabet at all (e.g., Chinese). As computers spread to the four corners of the globe and software vendors want to sell products in countries where most users do not speak English, a different character set is needed.

The first attempt at extending ASCII was IS 646, which added another 128 characters to ASCII, making it an 8-bit code called **Latin-1**. The additional characters were mostly Latin letters with accents and diacritical marks. The next attempt was IS 8859, which introduced the concept of a **code page**, a set of 256 characters for a particular language or group of languages. IS 8859-1 is Latin-1. IS 8859-2 handles the Latin-based Slavic languages (e.g., Czech, Polish, and Hungarian). IS 8859-3 contains the characters needed for Turkish, Maltese, Esperanto, and Galician, and so on. The trouble with the code-page approach is that the software has to keep track of which page it is currently on, it is impossible to mix languages over pages, and the scheme does not cover Japanese and Chinese at all.

A group of computer companies decided to solve this problem by forming a consortium to create a new system, called **Unicode**, and getting it proclaimed an International Standard (IS 10646). Unicode is now supported by programming languages (e.g., Java), operating systems (e.g., Windows), and many applications.

Hex	Name	Meaning	Hex	Name	Meaning
0	NUL	Null	10	DLE	Data Link Escape
1	SOH	Start Of Heading	11	DC1	Device Control 1
2	STX	Start Of TeXt	12	DC2	Device Control 2
3	ETX	End Of TeXt	13	DC3	Device Control 3
4	EOT	End Of Transmission	14	DC4	Device Control 4
5	ENQ	Enquiry	15	NAK	Negative AcKnowledgement
6	ACK	ACKnowledgement	16	SYN	SYNchronous idle
7	BEL	BELl	17	ETB	End of Transmission Block
8	BS	BackSpace	18	CAN	CANcel
9	HT	Horizontal Tab	19	EM	End of Medium
A	LF	Line Feed	1A	SUB	SUBstitute
B	VT	Vertical Tab	1B	ESC	ESCape
C	FF	Form Feed	1C	FS	File Separator
D	CR	Carriage Return	1D	GS	Group Separator
E	SO	Shift Out	1E	RS	Record Separator
F	SI	Shift In	1F	US	Unit Separator

Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char
20	(Space)	30	0	40	@	50	P	60	‘	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	’	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Figure 2-44. The ASCII character set.

The idea behind Unicode is to assign every character and symbol a unique 16-bit value, called a **code point**. No multibyte characters or escape sequences are used. Having every symbol be 16 bits makes writing software simpler.

With 16-bit symbols, Unicode has 65,536 code points. Since the world’s languages collectively use about 200,000 symbols, code points are a scarce resource that must be allocated with great care. To speed the acceptance of Unicode, the consortium cleverly used Latin-1 as code points 0 to 255, making conversion

between ASCII and Unicode easy. To avoid wasting code points, each diacritical mark has its own code point. It is up to software to combine diacritical marks with their neighbors to form new characters. While this puts more work on the software, it saves precious code points.

The code point space is divided up into blocks, each one a multiple of 16 code points. Each major alphabet in Unicode has a sequence of consecutive zones. Some examples (and the number of code points allocated) are Latin (336), Greek (144), Cyrillic (256), Armenian (96), Hebrew (112), Devanagari (128), Gurmukhi (128), Oriya (128), Telugu (128), and Kannada (128). Note that each of these languages has been allocated more code points than it has letters. This choice was made in part because many languages have multiple forms for each letter. For example, each letter in English has two forms—lowercase and UPPERCASE. Some languages have three or more forms, possibly depending on whether the letter is at the start, middle, or end of a word.

In addition to these alphabets, code points have been allocated for diacritical marks (112), punctuation marks (112), subscripts and superscripts (48), currency symbols (48), math symbols (256), geometric shapes (96), and dingbats (192).

After these come the symbols needed for Chinese, Japanese, and Korean. First are 1024 phonetic symbols (e.g., katakana and bopomofo) and then the unified Han ideographs (20,992) used in Chinese and Japanese, and the Korean Hangul syllables (11,156).

To allow users to invent special characters for special purposes, 6400 code points have been allocated for local use.

While Unicode solves many problems associated with internationalization, it does not (attempt to) solve all the world's problems. For example, while the Latin alphabet is in order, the Han ideographs are not in dictionary order. As a consequence, an English program can examine "cat" and "dog" and sort them alphabetically by simply comparing the Unicode value of their first character. A Japanese program needs external tables to figure out which of two symbols comes before the other in the dictionary.

Another issue is that new words are popping up all the time. Fifty years ago nobody talked about apps, chatrooms, cyberspace, emoticons, gigabytes, lasers, modems, smileys, or videotapes. Adding new words in English does not require new code points. Adding them in Japanese does. In addition to new technical words, there is a demand for adding at least 20,000 new (mostly Chinese) personal and place names. Blind people think Braille should be in there, and special interest groups of all kinds want what they perceive as their rightful code points. The Unicode consortium reviews and decides on all new proposals.

Unicode uses the same code point for characters that look almost identical but have different meanings or are written slightly differently in Japanese and Chinese (as though English word processors always spelled "blue" as "blew" because they sound the same). Some people view this as an optimization to save scarce code points; others see it as Anglo-Saxon cultural imperialism (and you thought

assigning 16-bit numbers to characters was not highly political?). To make matters worse, a full Japanese dictionary has 50,000 kanji (excluding names), so with only 20,992 code points available for the Han ideographs, choices had to be made. Not all Japanese people think that a consortium of computer companies, even if a few of them are Japanese, is the ideal forum to make these choices.

Guess what? 65,536 code points was not enough to satisfy everyone, so in 1996 an additional 16 **planes** of 16 bits each were added, expanding the total number of characters to 1,114,112.

UTF-8

Although better than ASCII, Unicode eventually ran out of code points and it also requires 16 bits per character to represent pure ASCII text, which is wasteful. Consequently, another coding scheme was developed to address these concerns. It is called **UTF-8 UCS Transformation Format** where **UCS** stands for **Universal Character Set**, which is essentially Unicode. UTF-8 codes are variable length, from 1 to 4 bytes, and can code about two billion characters. It is the dominant character set used on the World Wide Web.

One of the nice properties of UTF-8 is that codes 0 to 127 are the ASCII characters, allowing them to be expressed in 1 byte (versus 2 bytes in Unicode). For characters not in ASCII, the high-order bit of the first byte is set to 1, indicating that 1 or more additional bytes follow. In all, six different formats are used, as illustrated in Fig. 2-45. The bits marked "d" are data bits.

Bits	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
7	0ddddddd					
11	110ddddd	10dddddd				
16	1110dddd	10dddddd	10dddddd			
21	11110ddd	10dddddd	10dddddd	10dddddd		
26	111110dd	10dddddd	10dddddd	10dddddd	10dddddd	
31	1111110x	10dddddd	10dddddd	10dddddd	10dddddd	10dddddd

Figure 2-45. The UTF-8 encoding scheme.

UTF-8 has a number of advantages over Unicode and other schemes. First, if a program or document uses only characters that are in the ASCII character set, each can be represented in 8 bits. Second, the first byte of every UTF-8 character uniquely determines the number of bytes in the character. Third, the continuation bytes in an UTF-8 character always start with 10, whereas the initial byte never does, making the code self synchronizing. In particular, in the event of a communication or memory error, it is always possible to go forward and find the start of the next character (assuming it has not been damaged).

Normally UTF-8 is used to encode only the 17 Unicode planes, even though the scheme has far more than 1,114,112 code points. However, if anthropologists discover new tribes in New Guinea or elsewhere whose languages are not currently known (or if we make contact later with extraterrestrials), UTF-8 will be up to the job of adding their alphabets or ideographs.

2.5 SUMMARY

Computer systems are built up from three types of components: processors, memories, and I/O devices. The task of a processor is to fetch instructions one at a time from a memory, decode them, and execute them. The fetch-decode-execute cycle can always be described as an algorithm and, in fact, is sometimes carried out by a software interpreter running at a lower level. To gain speed, many computers now have one or more pipelines or have a superscalar design with multiple functional units that operate in parallel. A pipeline allows an instruction to be broken into steps and the steps for different instructions executed at the same time. Multiple functional units is another way to gain parallelism without affecting the instruction set or architecture visible to the programmer or compiler.

Systems with multiple processors are increasingly common. Parallel computers include array processors, on which the same operation is performed on multiple data sets at the same time, multiprocessors, in which multiple CPUs share a common memory, and multicomputers, in which multiple computers each have their own memories but communicate by message passing.

Memories can be categorized as primary or secondary. The primary memory is used to hold the program currently being executed. Its access time is short—a few tens of nanoseconds at most—and independent of the address being accessed. Caches reduce this access time even more. They are needed because processor speeds are much greater than memory speeds, meaning that having to wait for memory accesses all the time greatly slows down processor execution. Some memories are equipped with error-correcting codes to enhance reliability.

Secondary memories, in contrast, have access times that are much longer (milliseconds or more) and dependent on the location of the data being read or written. Tapes, flash memory, magnetic disks, and optical disks are the most common secondary memories. Magnetic disks come in many varieties, including IDE disks, SCSI disks, and RAIDs. Optical disks include CD-ROMs, CD-Rs, DVDs, and Blu-rays.

I/O devices are used to transfer information into and out of the computer. They are connected to the processor and memory by one or more buses. Examples are terminals, mice, game controllers, printers, and modems. Most I/O devices use the ASCII character code, although Unicode is also used and UTF-8 is gaining acceptance as the computer industry becomes more Web-centric.

PROBLEMS

1. Consider the operation of a machine with the data path of Fig. 2-2. Suppose that loading the ALU input registers takes 5 nsec, running the ALU takes 10 nsec, and storing the result back in the register scratchpad takes 5 nsec. What is the maximum number of MIPS this machine is capable of in the absence of pipelining?
2. What is the purpose of step 2 in the list of Sec. 2.1.2? What would happen if this step were omitted?
3. On computer 1, all instructions take 10 nsec to execute. On computer 2, they all take 5 nsec to execute. Can you say for certain that computer 2 is faster? Discuss.
4. Imagine you are designing a single-chip computer for an embedded system. The chip is going to have all its memory on chip and running at the same speed as the CPU with no access penalty. Examine each of the principles discussed in Sec. 2.1.4 and tell whether they are so important (assuming that high performance is still desired).
5. To compete with the newly invented printing press, a medieval monastery decided to mass-produce handwritten paperback books by assembling a vast number of scribes in a huge hall. The head monk would then call out the first word of the book to be produced and all the scribes would copy it down. Then the head monk would call out the second word and all the scribes would copy it down. This process was repeated until the entire book had been read aloud and copied. Which of the parallel processor systems discussed in Sec. 2.1.6 does this system resemble most closely?
6. As one goes down the five-level memory hierarchy discussed in the text, the access time increases. Make a reasonable guess about the ratio of the access time of optical disk to that of register memory. Assume that the disk is already online.
7. Sociologists can get three possible answers to a typical survey question such as "Do you believe in the tooth fairy?"—namely, yes, no, and no opinion. With this in mind, the Sociomagnetic Computer Company has decided to build a computer to process survey data. This computer has a trinary memory—that is, each byte (tryte?) consists of 8 trits, with a trit holding a 0, 1, or 2. How many trits are needed to hold a 6-bit number? Give an expression for the number of trits needed to hold n bits.
8. Compute the data rate of the human eye using the following information. The visual field consists of about 10^6 elements (pixels). Each pixel can be reduced to a superposition of the three primary colors, each of which has 64 intensities. The time resolution is 100 msec.
9. Compute the data rate of the human ear from the following information. People can hear frequencies up to 22 kHz. To capture all the information in a sound signal at 22 kHz, it is necessary to sample the sound at twice that frequency, that is, at 44 kHz. A 16-bit sample is probably enough to capture most of the auditory information (i.e., the ear cannot distinguish more than 65,535 intensity levels).
10. Genetic information in all living things is coded as DNA molecules. A DNA molecule is a linear sequence of the four basic nucleotides: A, C, G, and T. The human genome contains approximately 3×10^9 nucleotides in the form of about 30,000 genes. What

is the total information capacity (in bits) of the human genome? What is the maximum information capacity (in bits) of the average gene?

11. A certain computer can be equipped with 1,073,741,824 bytes of memory. Why would a manufacturer choose such a peculiar number, instead of an easy-to-remember number like 1,000,000,000?

12. Devise a 7-bit even-parity Hamming code for the digits 0 to 9.

13. Devise a code for the digits 0 to 9 whose Hamming distance is 2.

14. In a Hamming code, some bits are "wasted" in the sense that they are used for checking and not information. What is the percentage of wasted bits for messages whose total length (data + check bits) is $2^n - 1$? Evaluate this expression numerically for values of n from 3 to 10.

15. An extended ASCII character is represented by an 8-bit quantity. The associated Hamming encoding of each character can then be represented by a string of three hex digits. Encode the following extended five-character ASCII text using an even-parity Hamming code: Earth. Show your answer as a string of hex digits.

16. The following string of hex digits encodes extended ASCII characters in an even-parity Hamming code: 0D3 DD3 0F2 5C1 1C5 CE3. Decode this string and write down the characters that are encoded.

17. The disk illustrated in Fig. 2-19 has 1024 sectors/track and a rotation rate of 7200 RPM. What is the sustained transfer rate of the disk over one track?

18. A computer has a bus with a 5-nsec cycle time, during which it can read or write a 32-bit word from memory. The computer has an Ultra4-SCSI disk that uses the bus and runs at 160 Mbytes/sec. The CPU normally fetches and executes one 32-bit instruction every 1 nsec. How much does the disk slow down the CPU?

19. Imagine you are writing the disk-management part of an operating system. Logically, you represent the disk as a sequence of blocks, from 0 on the inside to some maximum on the outside. As files are created, you have to allocate free sectors. You could do it from the outside in or the inside out. Does it matter which strategy you choose on a modern disk? Explain your answer.

20. How long does it take to read a disk with 10,000 cylinders, each containing four tracks of 2048 sectors? First, all the sectors of track 0 are to be read starting at sector 0, then all the sectors of track 1 starting at sector 0, and so on. The rotation time is 10 msec, and a seek takes 1 msec between adjacent cylinders and 20 msec for the worst case. Switching between tracks of a cylinder can be done instantaneously.

21. RAID level 3 is able to correct single-bit errors using only one parity drive. What is the point of RAID level 2? After all, it also can correct only one error and takes more drives to do so.

22. What is the exact data capacity (in bytes) of a mode-2 CD-ROM containing the now-standard 80-min media? What is the capacity for user data in mode 1?

23. To burn a CD-R, the laser must pulse on and off at a high speed. When running at 10x speed in mode 1, what is the pulse length, in nanoseconds?

24. To be able to fit 133 minutes worth of video on a single-sided single-layer DVD, a fair amount of compression is required. Calculate the compression factor required. Assume that 3.5 GB of space is available for the video track, that the image resolution is 720×480 pixels with 24-bit color (RGB at 8 bits each), and images are displayed at 30 frames/sec.

25. Blu-ray runs at 4.5 MB/sec and has a capacity of 25 GB. How long does it take to read the entire disk?

26. A manufacturer advertises that its color bit-map terminal can display 2^{24} different colors. Yet the hardware only has 1 byte for each pixel. How can this be done?

27. You are part of a top-secret international scientific team which has just been assigned the task of studying a being named Herb, an extra-terrestrial from Planet 10 who has recently arrived here on Earth. Herb has given you the following information about how his eyes work. His visual field consists of about 10^8 pixels. Each pixel is basically a superposition of five "colors" (i.e., infrared, red, green, blue, and ultraviolet), each of which has 32 intensities. The time resolution of Herb's visual field is 10 msec. Calculate the data rate, in GB/sec, of Herb's eyes.

28. A bit-map terminal has a 1920×1080 display. The display is redrawn 75 times a second. How long is the pulse corresponding to one pixel?

29. In a certain font, a monochrome laser printer can print 50 lines of 80 characters per page. The average character occupies a box 2 mm $\times$ 2 mm, about 25% of which is toner. The rest is blank (i.e., no toner) The toner layer is 25 microns thick. The printer's toner cartridge measures $25 \times 8 \times 2$ cm. How many pages is one toner cartridge good for?

30. The Hi-Fi Modem Company has just designed a new frequency-modulation modem that uses 64 frequencies instead of just 2. Each second is divided into n equal time intervals, each of which contains one of the 64 possible tones. How many bits per second can this modem transmit, using synchronous transmission?

31. An Internet user has subscribed to a 2-Mbps ADSL service. Her neighbor has subscribed to a cable Internet service that has a shared bandwidth of 12 MHz. The modulation scheme in use is QAM-64. There are n houses on the cable, each with one computer. A fraction f of these computers are online at any one time. Under what conditions will the cable user get better service than the ADSL user?

32. A digital camera has a resolution of 3000×2000 pixels, with 3 bytes/pixel for RGB color. The manufacturer of the camera wants to be able to write a JPEG image at a 5x compression factor to the flash memory in 2 sec. What data rate is required?

33. A high-end digital camera has a sensor with 24 million pixels, each with 6 bytes/pixel. How many pictures can be stored on an 8-GB flash memory card if the compression factor is 5x? Assume that 1 GB means 2^{30} bytes.

34. Estimate how many characters, including spaces, a typical computer-science textbook contains. How many bits are needed to encode a book in ASCII with parity? How many CD-ROMs are needed to store a computer-science library of 10,000 books? How many single-sided, dual-layer DVDs are needed for the same library?

35. Write a procedure *hamming*(*ascii*, *encoded*) that converts the low-order 7 bits of *ascii* into an 11-bit integer codeword stored in *encoded*.

36. Write a function *distance*(*code*, *n*, *k*) that takes an array *code* of *n* characters of *k* bits each as input and returns the distance of the character set as output.

3

THE DIGITAL LOGIC LEVEL

At the bottom of the hierarchy of Fig. 1-2 we find the digital logic level, the computer's real hardware. In this chapter, we will examine many aspects of digital logic, as a building block for the study of higher levels in subsequent chapters. This subject is on the boundary of computer science and electrical engineering, but the material is self-contained, so no previous hardware or engineering experience is needed to follow it.

The basic elements from which all digital computers are constructed are amazingly simple. We will begin our study by looking at these basic elements and also at the special two-valued algebra (Boolean algebra) used to analyze them. Next we will examine some fundamental circuits that can be built using gates in simple combinations, including circuits for doing arithmetic. The following topic is how gates can be combined to store information, that is, how memories are organized. After that, we come to the subject of CPUs and especially how single-chip CPUs interface with memory and peripheral devices. Numerous examples from industry will be discussed later in this chapter.

3.1 GATES AND BOOLEAN ALGEBRA

Digital circuits can be constructed from a small number of primitive elements by combining them in innumerable ways. In the following sections we will describe these primitive elements, show how they can be combined, and introduce a powerful mathematical technique that can be used to analyze their behavior.

3.1.1 Gates

A digital circuit is one in which only two logical values are present. Typically, a signal between 0 and 0.5 volt represents one value (e.g., binary 0) and a signal between 1 and 1.5 volts represents the other value (e.g., binary 1). Voltages outside these two ranges are not permitted. Tiny electronic devices, called **gates**, can compute various functions of these two-valued signals. These gates form the hardware basis on which all digital computers are built.

The details of how gates work inside is beyond the scope of this book, belonging to the **device level**, which is below our level 0. Nevertheless, we will now digress ever so briefly to take a quick look at the basic idea, which is not difficult. All modern digital logic ultimately rests on the fact that a transistor can be made to operate as a very fast binary switch. In Fig. 3-1(a) we have shown a bipolar transistor (the circle) embedded in a simple circuit. This transistor has three connections to the outside world: the **collector**, the **base**, and the **emitter**. When the input voltage, V_{in}, is below a certain critical value, the transistor turns off and acts like an infinite resistance. This causes the output of the circuit, V_{out}, to take on a value close to V_{cc}, an externally regulated voltage, typically +1.5 volts for this type of transistor. When V_{in} exceeds the critical value, the transistor switches on and acts like a wire, causing V_{out} to be pulled down to ground (by convention, 0 volts).

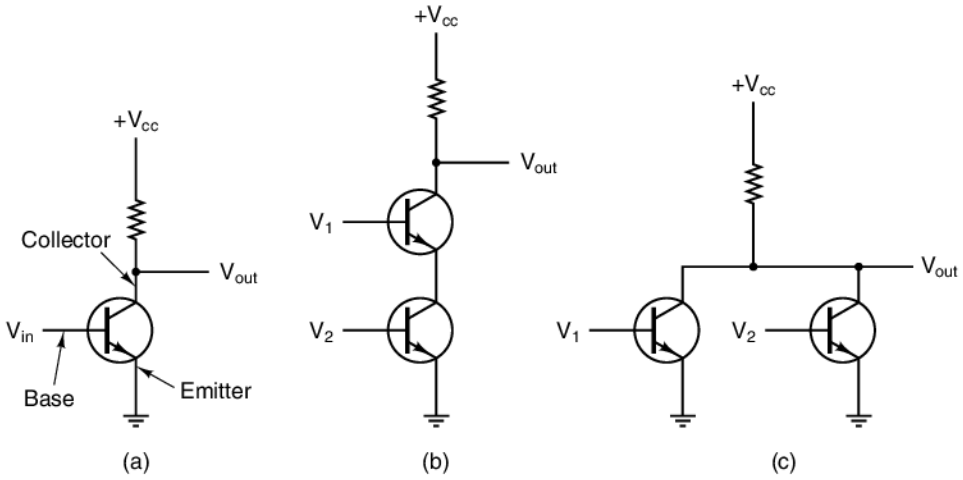


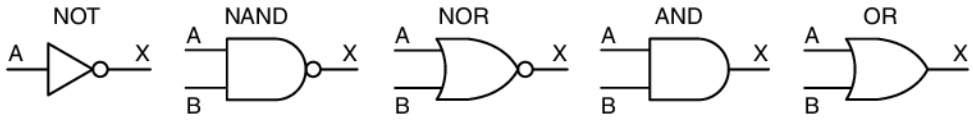
Figure 3-1. (a) A transistor inverter. (b) A NAND gate. (c) A NOR gate.

The important thing to notice is that when V_{in} is low, V_{out} is high, and vice versa. This circuit is thus an inverter, converting a logical 0 to a logical 1, and a logical 1 to a logical 0. The resistor (the jagged line) is needed to limit the amount of current drawn by the transistor so it does not burn out. The time required to switch from one state to the other is typically a nanosecond or less.

In Fig. 3-1(b) two transistors are cascaded in series. If both V_1 and V_2 are high, both transistors will conduct and V_{out} will be pulled low. If either input is low, the corresponding transistor will turn off, and the output will be high. In other words, V_{out} will be low if and only if both V_1 and V_2 are high.

In Fig. 3-1(c) the two transistors are wired in parallel instead of in series. In this configuration, if either input is high, the corresponding transistor will turn on and pull the output down to ground. If both inputs are low, the output will remain high.

These three circuits, or their equivalents, form the three simplest gates. They are called NOT, NAND, and NOR gates, respectively. NOT gates are often called **inverters**; we will use the two terms interchangeably. If we now adopt the convention that "high" (V_{cc} volts) is a logical 1, and that "low" (ground) is a logical 0, we can express the output value as a function of the input values. The symbols used to depict these three gates are shown in Fig. 3-2(a)–(c), along with the functional behavior for each circuit. In these figures, A and B are inputs and X is the output. Each row specifies the output for a different combination of the inputs.



A	X
0	1
1	0

(a)

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

(b)

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

(c)

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

(d)

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

(e)

Figure 3-2. The symbols and functional behavior for the five basic gates.

If the output signal of Fig. 3-1(b) is fed into an inverter circuit, we get another circuit with precisely the inverse of the NAND gate—namely, a circuit whose output is 1 if and only if both inputs are 1. Such a circuit is called an AND gate; its symbol and functional description are given in Fig. 3-2(d). Similarly, the NOR gate can be connected to an inverter to yield a circuit whose output is 1 if either or both inputs are 1 but 0 if both inputs are 0. The symbol and functional description of this circuit, called an OR gate, are given in Fig. 3-2(e). The small circles used as part of the symbols for the inverter, NAND gate, and NOR gate are called **inversion bubbles**. They are often used in other contexts as well to indicate an inverted signal.

The five gates of Fig. 3-2 are the principal building blocks of the digital logic level. From the foregoing discussion, it should be clear that NAND and NOR gates require two transistors each, whereas the AND and OR gates require three each. For

this reason, many computers are based on NAND and NOR gates rather than the more familiar AND and OR gates. (In practice, all the gates are implemented somewhat differently, but NAND and NOR are still simpler than AND and OR.) In passing it is worth noting that gates may well have more than two inputs. In principle, a NAND gate, for example, may have arbitrarily many inputs, but in practice more than eight inputs is unusual.

Although the subject of how gates are constructed belongs to the device level, we would like to mention the major families of manufacturing technology because they are referred to frequently. The two major technologies are **bipolar** and **MOS** (Metal Oxide Semiconductor). The major bipolar types are **TTL** (Transistor-Transistor Logic), which had been the workhorse of digital electronics for years, and **ECL** (Emitter-Coupled Logic), which was used when very high-speed operation was required. For computer circuits, MOS has now largely taken over.

MOS gates are slower than TTL and ECL but require much less power and take up much less space, so large numbers of them can be packed together tightly. MOS comes in many varieties, including PMOS, NMOS, and CMOS. While MOS transistors are constructed differently from bipolar transistors, their ability to function as electronic switches is the same. Most modern CPUs and memories use CMOS technology, which runs on a voltage in the neighborhood of +1.5 volts. This is all we will say about the device level. Readers interested in pursuing their study of this level should consult the readings suggested on the book's Website.

3.1.2 Boolean Algebra

To describe the circuits that can be built by combining gates, a new type of algebra is needed, one in which variables and functions can take on only the values 0 and 1. Such an algebra is called a **Boolean algebra**, after its discoverer, the English mathematician George Boole (1815–1864). Strictly speaking, we are really referring to a specific type of Boolean algebra, a **switching algebra**, but the term "Boolean algebra" is so widely used to mean "switching algebra" that we will not make the distinction.

Just as there are functions in "ordinary" (i.e., high school) algebra, so are there functions in Boolean algebra. A Boolean function has one or more input variables and yields a result that depends only on the values of these variables. A simple function, f, can be defined by saying that $f(A)$ is 1 if A is 0 and $f(A)$ is 0 if A is 1. This function is the NOT function of Fig. 3-2(a).

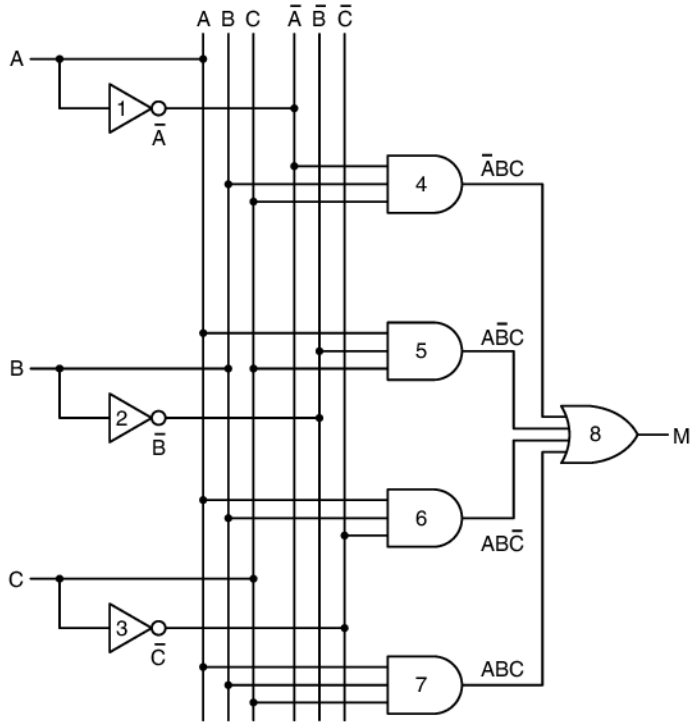
Because a Boolean function of n variables has only 2^n possible combinations of input values, the function can be completely described by giving a table with 2^n rows, each row telling the value of the function for a different combination of input values. Such a table is called a **truth table**. The tables of Fig. 3-2 are all examples of truth tables. If we agree to always list the rows of a truth table in numerical order (base 2), that is, for two variables in the order 00, 01, 10, and 11, the function can be completely described by the 2^n-bit binary number obtained by reading the

result column of the truth table vertically. Thus, NAND is 1110, NOR is 1000, AND is 0001, and OR is 0111. Obviously, only 16 Boolean functions of two variables exist, corresponding to the 16 possible 4-bit result strings. In contrast, ordinary algebra has an infinite number of functions of two variables, none of which can be described by giving a table of outputs for all possible inputs because each variable can take on any one of an infinite number of possible values.

Figure 3-3(a) shows the truth table for a Boolean function of three variables: $M = f(A, B, C)$. This function is the majority logic function, that is, it is 0 if a majority of its inputs are 0 and 1 if a majority of its inputs are 1. Although any Boolean function can be fully specified by giving its truth table, as the number of variables increases, this notation becomes increasingly cumbersome. Instead, another notation is frequently used.

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a)



(b)

Figure 3-3. (a) The truth table for the majority function of three variables. (b) A circuit for (a).

To see how this other notation comes about, note that any Boolean function can be specified by telling which combinations of input variables give an output value of 1. For the function of Fig. 3-3(a) there are four combinations of input variables that make M equal to 1. By convention, we will place a bar over an input

variable to indicate that its value is inverted. The absence of a bar means that it is not inverted. Furthermore, we will use implied multiplication or a dot to mean the Boolean AND function and + to mean the Boolean OR function. Thus, for example, $A\bar{B}C$ takes the value 1 only when $A = 1$ and $B = 0$ and $C = 1$. Also, $A\bar{B} + B\bar{C}$ is 1 only when ($A = 1$ and $B = 0$) or ($B = 1$ and $C = 0$). The four rows of Fig. 3-3(a) producing 1 bits in the output are: $\bar{A}BC$, $A\bar{B}C$, $AB\bar{C}$, and ABC. The function, M, is true (i.e., 1) if any of these four conditions is true, so we can write

$$M = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

as a compact way of giving the truth table. A function of n variables can thus be described by giving a "sum" of at most 2^n n-variable "product" terms. This formulation is especially important, as we will see shortly, because it leads directly to an implementation of the function using standard gates.

It is important to keep in mind the distinction between an abstract Boolean function and its implementation by an electronic circuit. A Boolean function consists of variables, such as A, B, and C, and Boolean operators such as AND, OR, and NOT. A Boolean function is described by giving a truth table or a Boolean function such as

$$F = A\bar{B}C + AB\bar{C}$$

A Boolean function can be implemented by an electronic circuit (often in many different ways) using signals that represent the input and output variables and gates such as AND, OR, and NOT. We will generally use the notation AND, OR, and NOT when referring to the Boolean operators and AND, OR, and NOT when referring to the gates, even though it is sometimes ambiguous as to whether we mean the functions or the gates.

3.1.3 Implementation of Boolean Functions

As mentioned above, the formulation of a Boolean function as a sum of up to 2^n product terms leads directly to a possible implementation. Using Fig. 3-3 as an example, we can see how this implementation is accomplished. In Fig. 3-3(b), the inputs, A, B, and C, are shown at the left edge and the output function, M, is shown at the right edge. Because complements (inverses) of the input variables are needed, they are generated by tapping the inputs and passing them through the inverters labeled 1, 2, and 3. To keep the figure from becoming cluttered, we have drawn in six vertical lines, of which three are connected to the input variables, and three connected to their complements. These lines provide a convenient source for the inputs to subsequent gates. For example, gates 5, 6, and 7 all use A as an input. In an actual circuit these gates would probably be wired directly to A without using any intermediate "vertical" wires.

The circuit contains four AND gates, one for each term in the equation for M (i.e., one for each row in the truth table having a 1 bit in the result column). Each

AND gate computes one row of the truth table, as indicated. Finally, all the product terms are ORed together to get the final result.

The circuit of Fig. 3-3(b) uses a convention that we will use repeatedly throughout this book: when two lines cross, no connection is implied unless a heavy dot is present at the intersection. For example, the output of gate 3 crosses all six vertical lines but it is connected only to $\bar{C}$. Be warned that some authors use other conventions.

From the example of Fig. 3-3 it should be clear how we can derive a general method to implement a circuit for any Boolean function:

1. Write down the truth table for the function.
2. Provide inverters to generate the complement of each input.
3. Draw an AND gate for each term with a 1 in the result column.
4. Wire the AND gates to the appropriate inputs.
5. Feed the output of all the AND gates into an OR gate.

Although we have shown how any Boolean function can be implemented using NOT, AND, and OR gates, it is often convenient to implement circuits using only a single type of gate. Fortunately, it is straightforward to convert circuits generated by the preceding algorithm to pure NAND or pure NOR form. All we need is a way to implement NOT, AND, and OR using a single gate type. The top row of Fig. 3-4 shows how all three of these can be implemented using only NAND gates; the bottom row shows how it can be done using only NOR gates. (These are straightforward, but there are other ways, too.)

One way to implement a Boolean function using only NAND or only NOR gates is first follow the procedure given above for constructing it with NOT, AND, and OR. Then replace the multi-input gates with equivalent circuits using two-input gates. For example, $A + B + C + D$ can be computed as $(A + B) + (C + D)$, using three two-input OR gates. Finally, the NOT, AND, and OR gates are replaced by the circuits of Fig. 3-4.

Although this procedure does not lead to the optimal circuits, in the sense of the minimum number of gates, it does show that a solution is always feasible. Both NAND and NOR gates are said to be **complete**, because any Boolean function can be computed using either of them. No other gate has this property, which is another reason they are often preferred for the building blocks of circuits.

3.1.4 Circuit Equivalence

Circuit designers often try to reduce the number of gates in their products to reduce the chip area needed to implement them, minimize power consumption, and increase speed. To reduce the complexity of a circuit, the designer must find another circuit that computes the same function as the original but does so with fewer

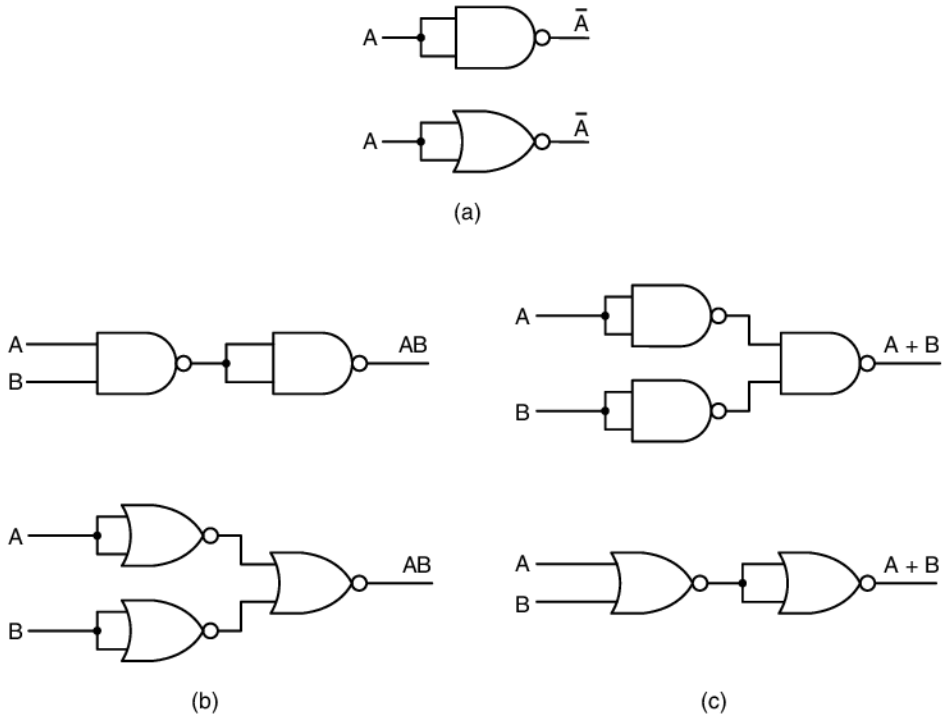


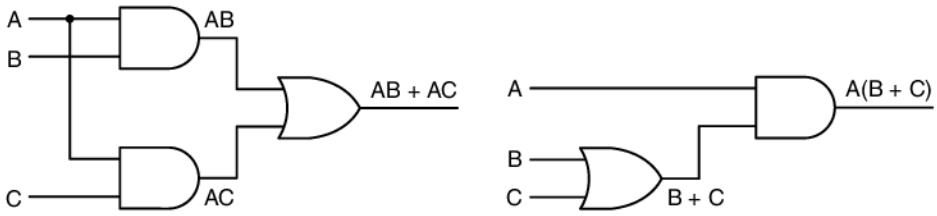
Figure 3-4. Construction of (a) NOT, (b) AND, and (c) OR gates using only NAND gates or only NOR gates.

gates (or perhaps with simpler gates, for example, two-input gates instead of four-input gates). In the search for equivalent circuits, Boolean algebra can be a valuable tool.

As an example of how Boolean algebra can be used, consider the circuit and truth table for $AB + AC$ shown in Fig. 3-5(a). Although we have not discussed them yet, many of the rules of ordinary algebra also hold for Boolean algebra. In particular, $AB + AC$ can be factored into $A(B + C)$ using the distributive law. Figure 3-5(b) shows the circuit and truth table for $A(B + C)$. Because two functions are equivalent if and only if they have the same output for all possible inputs, it is easy to see from the truth tables of Fig. 3-5 that $A(B + C)$ is equivalent to $AB + AC$. Despite this equivalence, the circuit of Fig. 3-5(b) is clearly better than that of Fig. 3-5(a) because it contains fewer gates.

In general, a circuit designer starts with a Boolean function and then applies the laws of Boolean algebra to it in an attempt to find a simpler but equivalent one. From the final function, a circuit can be constructed.

To use this approach, we need some identities from Boolean algebra. Figure 3-6 shows some of the major ones. It is interesting to note that each law has two



A	B	C	AB	AC	AB + AC
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

(a)

A	B	C	A	B + C	A(B + C)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

(b)

Figure 3-5. Two equivalent functions. (a) $AB + AC$. (b) $A(B + C)$.

forms that are **duals** of each other. By interchanging AND and OR and also 0 and 1, either form can be produced from the other one. All the laws can be easily proven by constructing their truth tables. Except for De Morgan's law, the absorption law, and the AND form of the distributive law, the results should be understandable with some study. De Morgan's law can be extended to more than two variables, for example, $\overline{ABC} = \bar{A} + \bar{B} + \bar{C}$.

De Morgan's law suggests an alternative notation. In Fig. 3-7(a) the AND form is shown with negation indicated by inversion bubbles, both for input and output. Thus, an OR gate with inverted inputs is equivalent to a NAND gate. From Fig. 3-7(b), the dual form of De Morgan's law, it should be clear that a NOR gate can be drawn as an AND gate with inverted inputs. By negating both forms of De Morgan's law, we arrive at Fig. 3-7(c) and (d), which show equivalent representations of the AND and OR gates. Analogous symbols exist for the multiple-variable forms of De Morgan's law (e.g., an n input NAND gate becomes an OR gate with n inverted inputs).

Using the identities of Fig. 3-7 and the analogous ones for multi-input gates, it is easy to convert the sum-of-products representation of a truth table to pure NAND or pure NOR form. As an example, consider the EXCLUSIVE OR function of Fig. 3-8(a). The standard sum-of-products circuit is shown in Fig. 3-8(b). To

Name	AND form	OR form
Identity law	1A = A	0 + A = A
Null law	0A = 0	1 + A = 1
Idempotent law	AA = A	A + A = A
Inverse law	$A\overline{A} = 0$	$A + \overline{A} = 1$
Commutative law	AB = BA	A + B = B + A
Associative law	(AB)C = A(BC)	(A + B) + C = A + (B + C)
Distributive law	A + BC = (A + B)(A + C)	A(B + C) = AB + AC
Absorption law	A(A + B) = A	A + AB = A
De Morgan's law	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}\,\overline{B}$

Figure 3-6. Some identities of Boolean algebra.

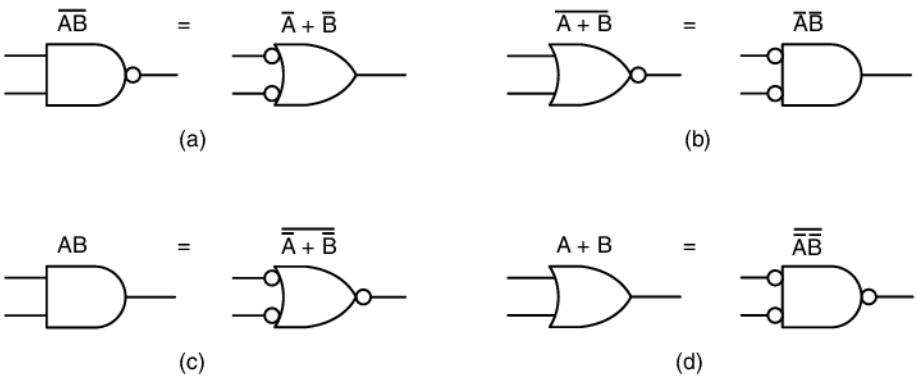
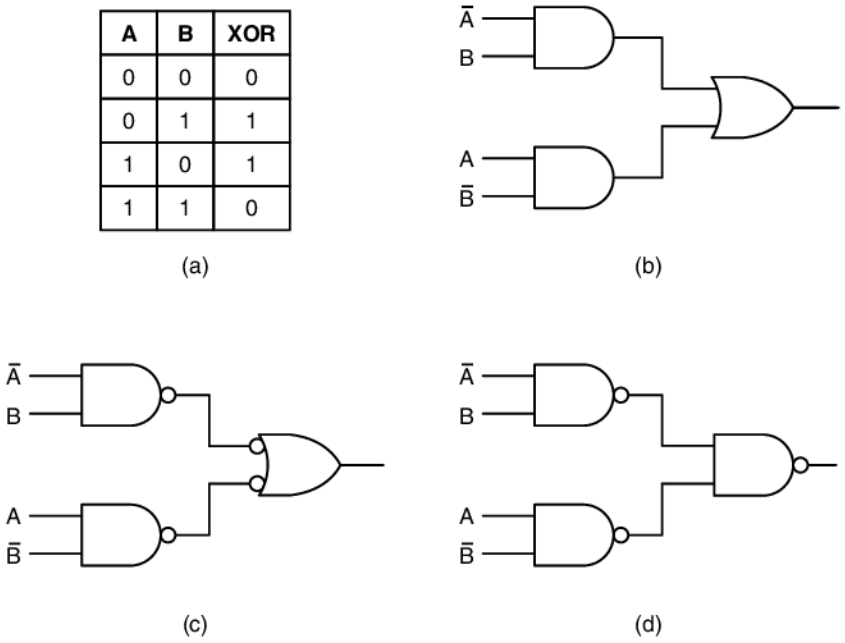


Figure 3-7. Alternative symbols for some gates: (a) NAND (b) NOR (c) AND (d) OR.

convert to NAND form, the lines connecting the output of the AND gates to the input of the OR gate should be redrawn with two inversion bubbles, as shown in Fig. 3-8(c). Finally, using Fig. 3-7(a), we arrive at Fig. 3-8(d). The variables $\bar{A}$ and $\bar{B}$ can be generated from A and B using NAND or NOR gates with their inputs tied together. Note that inversion bubbles can be moved along a line at will, for example, from the outputs of the input gates in Fig. 3-8(d) to the inputs of the output gate.

As a final note on circuit equivalence, we will now demonstrate the surprising result that the same physical gate can compute different functions, depending on the conventions used. In Fig. 3-9(a) we show the output of a certain gate, F, for different input combinations. Both inputs and outputs are shown in volts. If we



A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Figure 3-8. (a) The truth table for the XOR function. (b)–(d) Three circuits for computing it.

adopt the convention that 0 volts is logical 0 and 1.5 volts is logical 1, called **positive logic**, we get the truth table of Fig. 3-9(b), the AND function. If, however, we adopt **negative logic**, which has 0 volts as logical 1 and 1.5 volts as logical 0, we get the truth table of Fig. 3-9(c), the OR function.

A	B	F
0^V	0^V	0^V
0^V	5^V	0^V
5^V	0^V	0^V
5^V	5^V	5^V

(a)

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

(b)

A	B	F
1	1	1
1	0	1
0	1	1
0	0	0

(c)

Figure 3-9. (a) Electrical characteristics of a device. (b) Positive logic. (c) Negative logic.

Thus, the convention chosen to map voltages onto logical values is critical. Except where otherwise specified, we will henceforth use positive logic, so the terms logical 1, true, and high are synonyms, as are logical 0, false, and low.

3.2 BASIC DIGITAL LOGIC CIRCUITS

In the previous sections we saw how to implement truth tables and other simple circuits using individual gates. In practice, few circuits are actually constructed gate-by-gate anymore, although this once was common. Nowadays, the usual building blocks are modules containing a number of gates. In the following sections we will examine these building blocks more closely and see how they are used and how they can be constructed from individual gates.

3.2.1 Integrated Circuits

Gates are not manufactured or sold individually but rather in units called **Integrated Circuits**, often called **ICs** or **chips**. An IC is a rectangular piece of silicon of varied size depending on how many gates are required to implement the chip's components. Small dies will measure about 2 mm × 2 mm, while larger dies can be as large as 18 mm × 18 mm. ICs are mounted into plastic or ceramic packages that can be much larger than the dies they house, if many pins are required to connect the chip to the outside world. Each pin connects to the input or output of some gate on the chip or to power or to ground.

Figure 3-10 shows a number of common IC packages used for chips today. Smaller chips, such as those used to house microcontrollers or RAM chips, will use **Dual Inline Packages** or **DIPs**. A DIP is a package with two rows of pins that fit into a matching socket on the motherboard. The most common DIP packages have 14, 16, 18, 20, 22, 24, 28, 40, 64, or 68 pins. For large chips, square packages with pins on all four sides or on the bottom are often used. Two common packages for larger chips are **Pin Grid Arrays** or **PGAs** and **Land Grid Arrays** or **LGAs**. PGA have pins on the bottom of the package, which fit into a matching socket on the motherboard. PGA sockets often utilize a zero-insertion-force mechanism in which the PGA can be placed into the socket without force, then a lever can be thrown which will apply lateral pressure to all of the PGA's pins, holding it firmly in the PGA socket. LGAs, on the other hand, have small flat pads on the bottom of the chip, and an LGA socket will have a cover that fits over the LGA and applies a downward force on the chip, ensuring that all of the LGA pads make contact with the LGA socket pads.

Because many IC packages are symmetric in shape, figuring out which orientation is correct is a perennial problem with IC installation. DIPs typically have a notch in one end which matches a corresponding mark on the DIP socket. PGAs typically have one pin missing, so if you attempt to insert the PGA into the socket incorrectly, the PGA will not insert. Because LGAs do not have pins, correct installation is enforced by placing a notch on one or two sides of the LGA, which matches a notch in the LGA socket. The LGA will not enter the socket unless the two notches match.

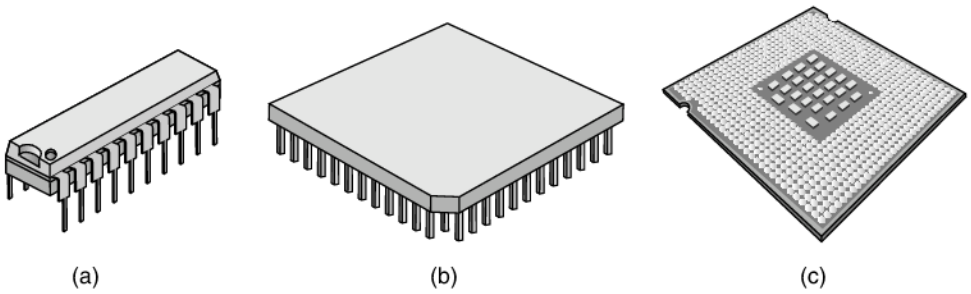


Figure 3-10. Common types of integrated-circuit packages, including a dual-in-line package (a), pin grid array (b), and land grid array (c).

For our purposes, all gates are ideal in the sense that the output appears as soon as the input is applied. In reality, chips have a finite **gate delay**, which includes both the signal propagation time through the chip and the switching time. Typical delays are 100s of picoseconds to a few nanoseconds.

It is within the current state of the art to put more than 1 billion transistors on a single chip. Because any circuit can be built up from NAND gates, you might think that a manufacturer could make a very general chip containing 500 million NAND gates. Unfortunately, such a chip would need 1,500,000,002 pins. With the standard pin spacing of 1 millimeter, an LGA would have to be 38 meters on a side to accommodate all of those pins, which might have a negative effect on sales. Clearly, the only way to take advantage of the technology is to design circuits with a high gate/pin ratio. In the following sections we will look at simple circuits that combine a number of gates internally to provide a useful function requiring only a limited number of external connections (pins).

3.2.2 Combinational Circuits

Many applications of digital logic require a circuit with multiple inputs and outputs in which the outputs are uniquely determined by the current input values. Such a circuit is called a **combinational circuit**. Not all circuits have this property. For example, a circuit containing memory elements may generate outputs that depend on the stored values as well as the input variables. A circuit implementing a truth table, such as that of Fig. 3-3(a), is a typical example of a combinational circuit. In this section we will examine some frequently used combinational circuits.

Multiplexers

At the digital logic level, a **multiplexer** is a circuit with 2^n data inputs, one data output, and n control inputs that select one of the data inputs. The selected data input is "gated" (i.e., sent) to the output. Figure 3-11 is a schematic diagram

for an eight-input multiplexer. The three control lines, *A*, *B*, and *C*, encode a 3-bit number that specifies which of the eight input lines is gated to the OR gate and thence to the output. No matter what value is on the control lines, seven of the AND gates will always output 0; the other one may output either 0 or 1, depending on the value of the selected input line. Each AND gate is enabled by a different combination of the control inputs. The multiplexer circuit is shown in Fig. 3-11.

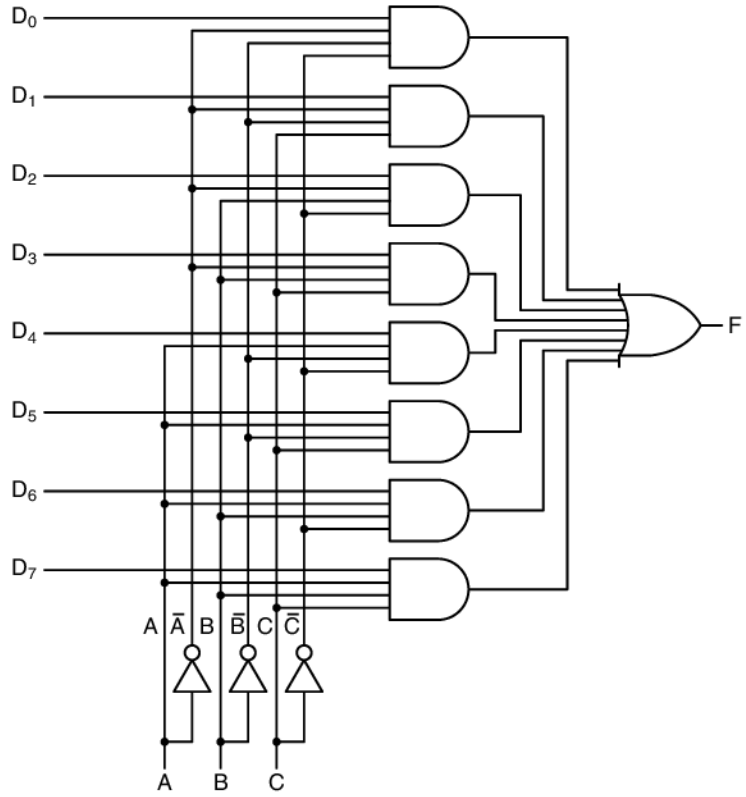


Figure 3-11. An eight-input multiplexer circuit.

Using the multiplexer, we can implement the majority function of Fig. 3-3(a), as shown in Fig. 3-12(b). For each combination of *A*, *B*, and *C*, one of the data input lines is selected. Each input is wired to either V_{cc} (logical 1) or ground (logical 0). The algorithm for wiring the inputs is simple: input D_i is the same as the value in row *i* of the truth table. In Fig. 3-3(a), rows 0, 1, 2, and 4 are 0, so the corresponding inputs are grounded; the remaining rows are 1, so they are wired to logical 1. In this manner any truth table of three variables can be implemented using the chip of Fig. 3-12(a).

We just saw how a multiplexer chip can be used to select one of several inputs and how it can implement a truth table. Another of its many applications is as a

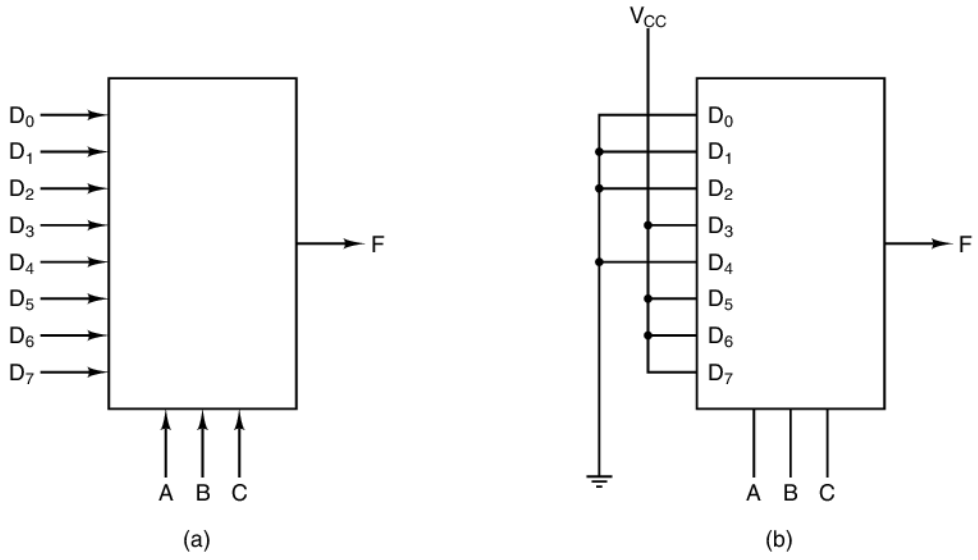


Figure 3-12. (a) An eight-input multiplexer. (b) The same multiplexer wired to compute the majority function.

parallel-to-serial data converter. By putting 8 bits of data on the input lines and then stepping the control lines sequentially from 000 to 111 (binary), the 8 bits are put onto the output line in series. A typical use for parallel-to-serial conversion is in a keyboard, where each keystroke implicitly defines a 7- or 8-bit number that must be output over a serial link, such as USB.

The inverse of a multiplexer is a **demultiplexer**, which routes its single input signal to one of 2^n outputs, depending on the values of the n control lines. If the binary value on the control lines is k, output k is selected.

Decoders

As a second example, we will now look at a circuit that takes an n-bit number as input and uses it to select (i.e., set to 1) exactly one of the 2^n output lines. Such a circuit, illustrated for $n = 3$ in Fig. 3-13, is called a **decoder**.

To see where a decoder might be useful, imagine a small memory consisting of eight chips, each containing 256 MB. Chip 0 has addresses 0 to 256 MB, chip 1 has addresses 256 MB to 512 MB, and so on. When an address is presented to the memory, the high-order 3 bits are used to select one of the eight chips. Using the circuit of Fig. 3-13, these 3 bits are the three inputs, A, B, and C. Depending on the inputs, exactly one of the eight output lines, $D_0, \ldots, D_7$, is 1; the rest are 0. Each output line enables one of the eight memory chips. Because only one output line is set to 1, only one chip is enabled.

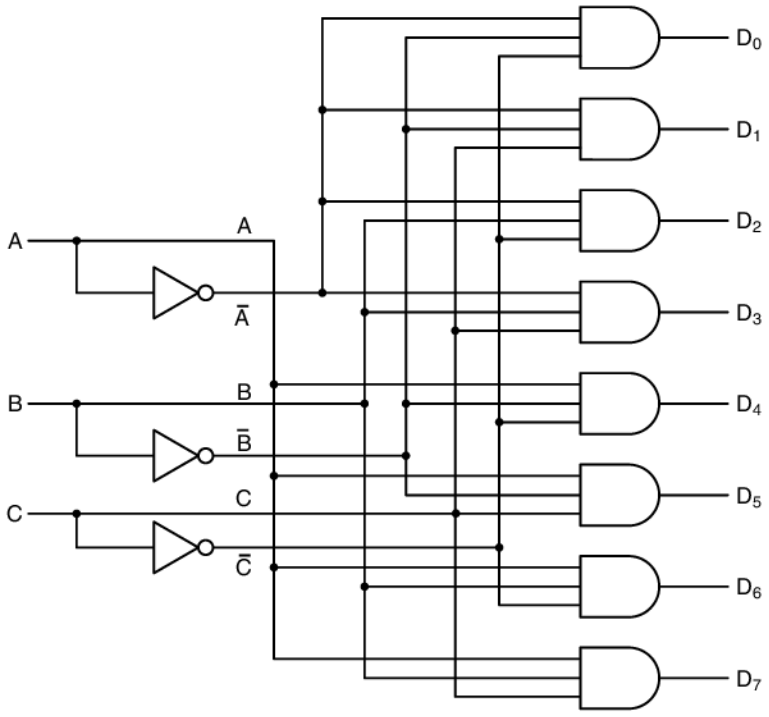


Figure 3-13. A three-to-eight decoder circuit.

The operation of the circuit of Fig. 3-13 is straightforward. Each AND gate has three inputs, of which the first is either A or $\bar{A}$, the second is either B or $\bar{B}$, and the third is either C or $\bar{C}$. Each gate is enabled by a different combination of inputs: D_0 by $\bar{A}\ \bar{B}\ \bar{C}$, D_1 by $\bar{A}\ \bar{B}\ C$, and so on.

Comparators

Another useful circuit is the **comparator**, which compares two input words. The simple comparator of Fig. 3-14 takes two inputs, A and B, each of length 4 bits, and produces a 1 if they are equal and a 0 otherwise. The circuit is based on the XOR (EXCLUSIVE OR) gate, which puts out a 0 if its inputs are equal and a 1 otherwise. If the two input words are equal, all four of the XOR gates must output 0. These four signals can then be ORed together; if the result is 0, the input words are equal, otherwise not. In our example we have used a NOR gate as the final stage to reverse the sense of the test: 1 means equal, 0 means unequal.

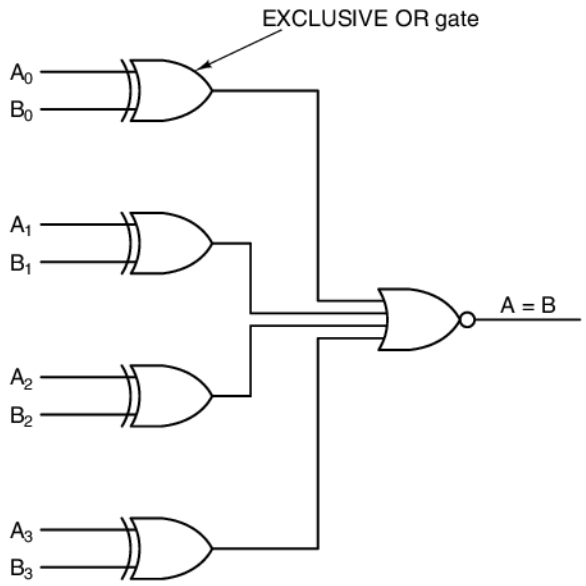


Figure 3-14. A simple 4-bit comparator.

3.2.3 Arithmetic Circuits

It is now time to move on from the general-purpose circuits discussed above to combinational circuits. As a reminder, combinational circuits have outs that are functions of their inputs, but circuits used for doing arithmetic do not have this property. We will begin with a simple 8-bit shifter, then look at how adders are constructed, and finally examine arithmetic logic units, which play a central role in any computer.

Shifters

Our first arithmetic circuit is an eight-input, eight-output shifter (see Fig. 3-15). Eight bits of input are presented on lines $D_0, \ldots, D_7$. The output, which is just the input shifted 1 bit, is available on lines $S_0, \ldots, S_7$. The control line, C, determines the direction of the shift, 0 for left and 1 for right. On a left shift, a 0 is inserted into bit 7. Similarly, on a right shift, a 1 is inserted into bit 0.

To see how the circuit works, notice the pairs of AND gates for all the bits except the gates on the end. When $C = 1$, the right member of each pair is turned on, passing the corresponding input bit to output. Because the right AND gate is wired to the input of the OR gate to its right, a right shift is performed. When $C = 0$, it is the left member of the AND gate pair that turns on, doing a left shift.

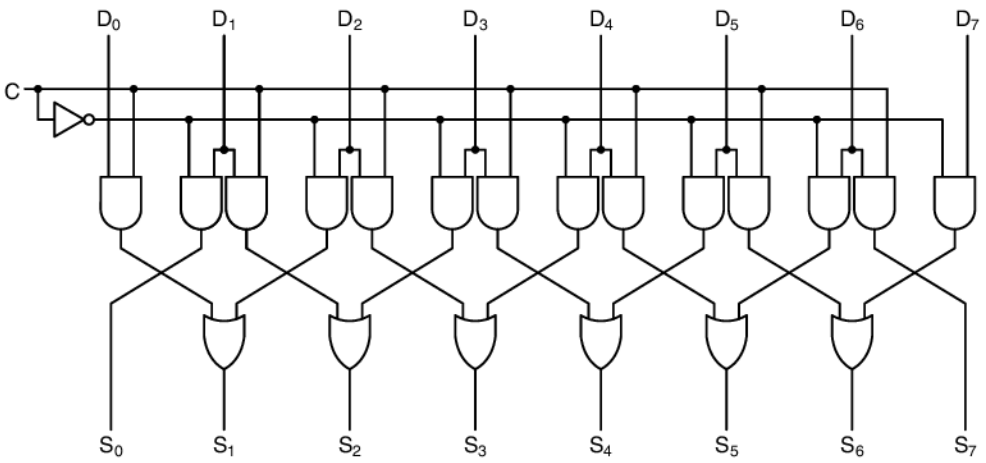


Figure 3-15. A 1-bit left/right shifter.

Adders

A computer that cannot add integers is almost unthinkable. Consequently, a hardware circuit for performing addition is an essential part of every CPU. The truth table for addition of 1-bit integers is shown in Fig. 3-16(a). Two outputs are present: the sum of the inputs, *A* and *B*, and the carry to the next (leftward) position. A circuit for computing both the sum bit and the carry bit is illustrated in Fig. 3-16(b). This simple circuit is generally known as a **half adder**.

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

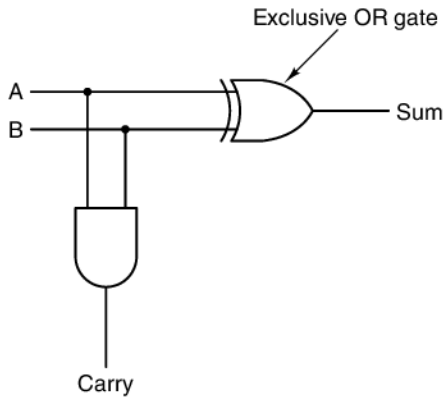
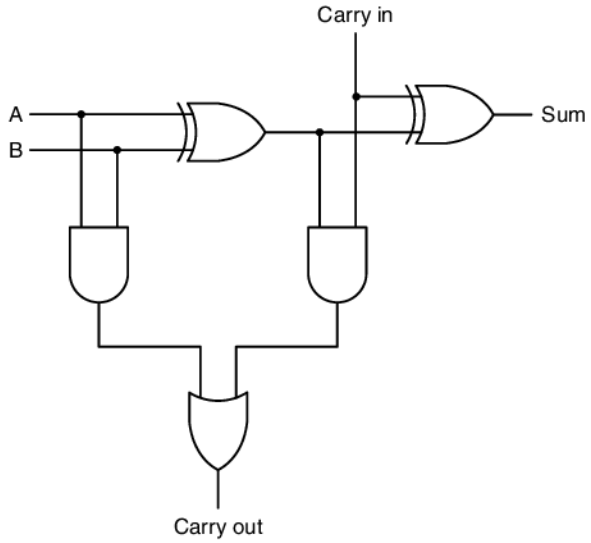


Figure 3-16. (a) Truth table for 1-bit addition. (b) A circuit for a half adder.

Although a half adder is adequate for summing the low-order bits of two multi-bit input words, it will not do for a bit position in the middle of the word because it does not handle the carry into the position from the right. Instead, the **full adder** of Fig. 3-17 is needed. From inspection of the circuit it should be clear that a full adder is built up from two half adders. The *Sum* output line is 1 if an odd number of *A*, *B*, and the *Carry in* are 1. The *Carry out* is 1 if either *A* and *B* are both 1 (left input to the OR gate) or exactly one of them is 1 and the *Carry in* bit is also 1. Together the two half adders generate both the sum and the carry bits.

A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a)



(b)

Figure 3-17. (a) Truth table for full adder. (b) Circuit for a full adder.

To build an adder for, say, two 16-bit words, one just replicates the circuit of Fig. 3-17(b) 16 times. The carry out of a bit is used as the carry into its left neighbor. The carry into the rightmost bit is wired to 0. This type of adder is called a **ripple carry adder**, because in the worst case, adding 1 to 111...111 (binary), the addition cannot complete until the carry has rippled all the way from the rightmost bit to the leftmost bit. Adders that do not have this delay, and hence are faster, also exist and are usually preferred.

As a simple example of a faster adder, consider breaking up a 32-bit adder into a 16-bit lower half and a 16-bit upper half. When the addition starts, the upper adder cannot yet get to work because it will not know the carry into it for 16 addition times.

However, consider this modification to the circuit. Instead of having a single upper half, give the adder two upper halves in parallel by duplicating the upper

half's hardware. Thus, the circuit now consists of three 16-bit adders: a lower half and two upper halves, *U0* and *U1* that run in parallel. A 0 is fed into *U0* as a carry; a 1 is fed into *U1* as a carry. Now both of these can start at the same time the lower half starts, but only one will be correct. After 16 bit-addition times, it will be known what the carry into the upper half is, so the correct upper half can now be selected from the two available answers. This trick reduces the addition time by a factor of two. Such an adder is called a **carry select adder**. This trick can then be repeated to build each 16-bit adder out of replicated 8-bit adders, and so on.

Arithmetic Logic Units

Most computers contain a single circuit for performing the AND, OR, and sum of two machine words. Typically, such a circuit for n-bit words is built up of n identical circuits for the individual bit positions. Figure 3-18 is a simple example of such a circuit, called an **Arithmetic Logic Unit** or **ALU**. It can compute any one of four functions—namely, A AND B, A OR B, $\bar{B}$, or $A + B$, depending on whether the function-select input lines F_0 and F_1 contain 00, 01, 10, or 11 (binary). Note that here $A + B$ means the arithmetic sum of A and B, not the Boolean OR.

The lower left-hand corner of our ALU contains a 2-bit decoder to generate enable signals for the four operations, based on the control signals F_0 and F_1. Depending on the values of F_0 and F_1, exactly one of the four enable lines is selected. Setting this line allows the output for the selected function to pass through to the final OR gate for output.

The upper left-hand corner has the logic to compute A AND B, A OR B, and $\bar{B}$, but at most one of these results is passed onto the final OR gate, depending on the enable lines coming out of the decoder. Because exactly one of the decoder outputs will be 1, exactly one of the four AND gates driving the OR gate will be enabled; the other three will output 0, independent of A and B.

In addition to being able to use A and B as inputs for logical or arithmetic operations, it is also possible to force either one to 0 by negating ENA or ENB, respectively. It is also possible to get $\bar{A}$, by setting INVA. We will see uses for INVA, ENA, and ENB in Chap. 4. Under normal conditions, ENA and ENB are both 1 to enable both inputs and INVA is 0. In this case, A and B are just fed into the logic unit unmodified.

The lower right-hand corner of the ALU contains a full adder for computing the sum of A and B, including handling the carries, because it is likely that several of these circuits will eventually be wired together to perform full-word operations. Circuits like Fig. 3-18 are actually available and are known as **bit slices**. They allow the computer designer to build an ALU of any desired width. Figure 3-19 shows an 8-bit ALU built up of eight 1-bit ALU slices. The INC signal is useful only for addition operations. When present, it increments (i.e., adds 1 to) the result, making it possible to compute sums like $A + 1$ and $A + B + 1$.

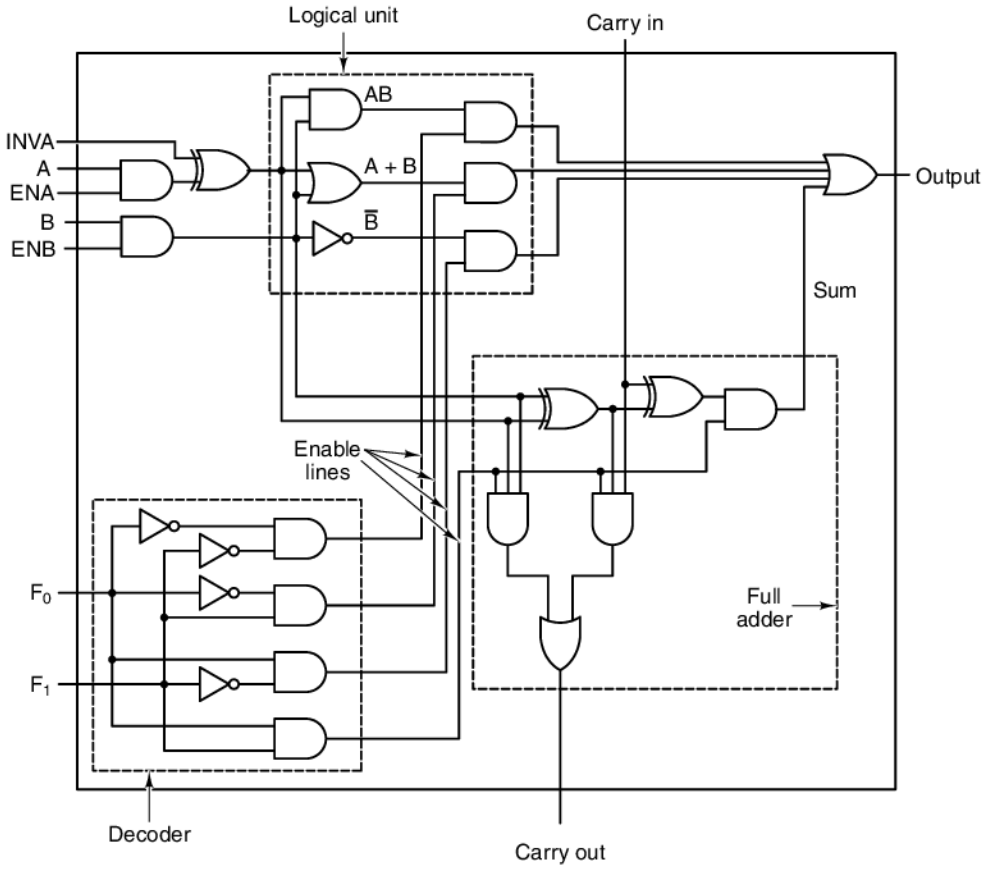


Figure 3-18. A 1-bit ALU.

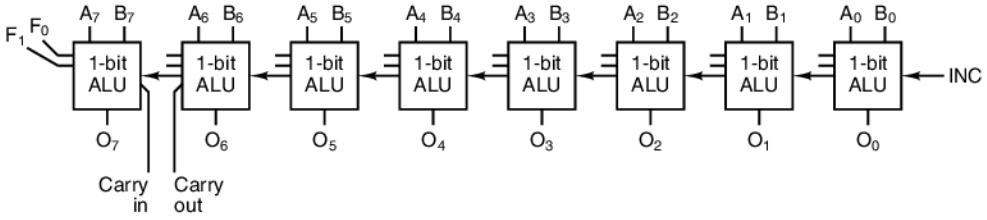


Figure 3-19. Eight 1-bit ALU slices connected to make an 8-bit ALU. The enables and invert signals are not shown for simplicity.

Years ago, a bit slice was an actual chip you could buy. Nowadays, a bit slice is more likely to be a library a chip designer can replicate the desired number of times in a computer-aided-design program that produces an output file that drives the chip-production machines. But the idea is essentially the same.

3.2.4 Clocks

In many digital circuits the order in which events happen is critical. Sometimes one event must precede another, sometimes two events must occur simultaneously. To allow designers to achieve the required timing relations, many digital circuits use clocks to provide synchronization. A **clock** in this context is a circuit that emits a series of pulses with a precise pulse width and precise interval between consecutive pulses. The time interval between the corresponding edges of two consecutive pulses is known as the **clock cycle time**. Pulse frequencies are commonly between 100 MHz and 4 GHz, corresponding to clock cycles of 10 nsec to 250 psec. To achieve high accuracy, the clock frequency is usually controlled by a crystal oscillator.

In a computer, many events may happen during a single clock cycle. If these events must occur in a specific order, the clock cycle must be divided into subcycles. A common way of providing finer resolution than the basic clock is to tap the primary clock line and insert a circuit with a known delay in it, thus generating a secondary clock signal that is phase-shifted from the primary, as shown in Fig. 3-20(a). The timing diagram of Fig. 3-20(b) provides four time references for discrete events:

1. Rising edge of C1.
2. Falling edge of C1.
3. Rising edge of C2.
4. Falling edge of C2.

By tying different events to the various edges, the required sequencing can be achieved. If more than four time references are needed within a clock cycle, more secondary lines can be tapped from the primary, with one with a different delay if necessary.

In some circuits, one is interested in time intervals rather than discrete instants of time. For example, some event may be allowed to happen whenever C1 is high, rather than precisely at the rising edge. Another event may happen only when C2 is high. If more than two different intervals are needed, more clock lines can be provided or the high states of the two clocks can be made to overlap partially in time. In the latter case four distinct intervals can be distinguished: $\overline{\text{C1}}$ AND $\overline{\text{C2}}$, $\overline{\text{C1}}$ AND C2, C1 AND $\overline{\text{C2}}$, and C1 AND C2.

As an aside, clocks are symmetric, with time spent in the high state equal to the time spent in the low state, as shown in Fig. 3-20(b). To generate an asymmetric pulse train, the basic clock is shifted using a delay circuit and ANDed with the original signal, as shown in Fig. 3-20(c) as C.

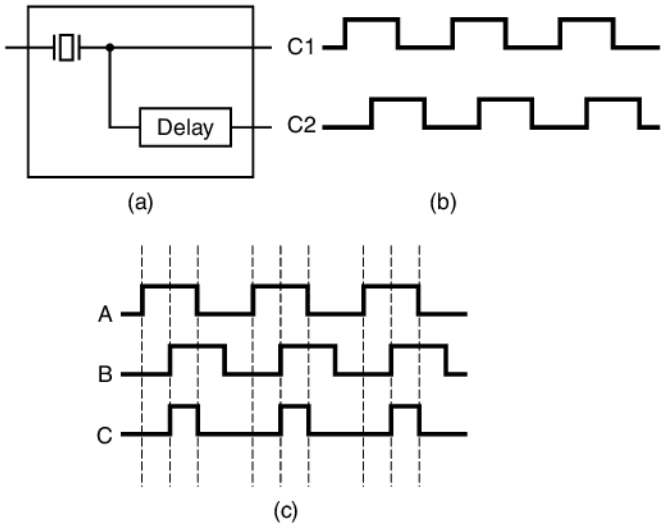


Figure 3-20. (a) A clock. (b) The timing diagram for the clock. (c) Generation of an asymmetric clock.

3.3 MEMORY

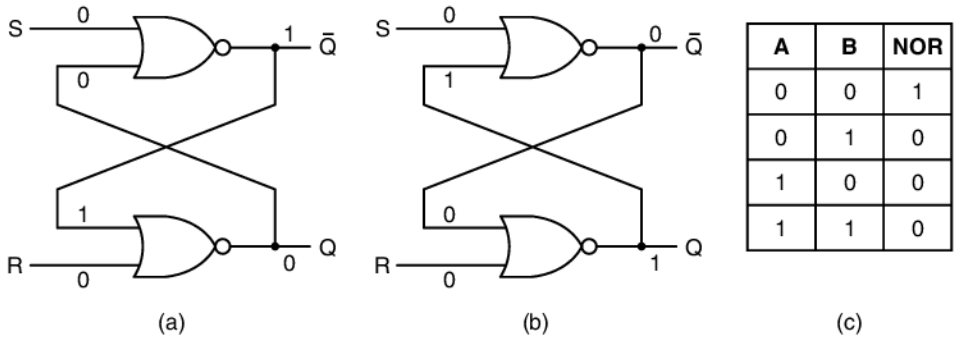
An essential component of every computer is its memory. Without memory there could be no computers as we now know them. Memory is used for storing both instructions to be executed and data. In the following sections we will examine the basic components of a memory system starting at the gate level to see how they work and how they are combined to produce large memories.

3.3.1 Latches

To create a 1-bit memory, we need a circuit that somehow "remembers" previous input values. Such a circuit can be constructed from two NOR gates, as illustrated in Fig. 3-21(a). Analogous circuits can be built from NAND gates. We will not mention these further, however, because they are conceptually identical to the NOR versions.

The circuit of Fig. 3-21(a) is called an **SR latch**. It has two inputs, S, for Setting the latch, and R, for Resetting (i.e., clearing) it. It also has two outputs, Q and $\bar{Q}$, which are complementary, as we will see shortly. Unlike a combinational circuit, the outputs of the latch are not uniquely determined by the current inputs.

To see how this comes about, let us assume that both S and R are 0, which they are most of the time. For argument's sake, let us further assume that $Q = 0$. Because Q is fed back into the upper NOR gate, both of its inputs are 0, so its output,



A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

Figure 3-21. (a) NOR latch in state 0. (b) NOR latch in state 1. (c) Truth table for NOR.

$\bar{Q}$, is 1. The 1 is fed back into the lower gate, which then has inputs 1 and 0, yielding $Q = 0$. This state is at least consistent and is depicted in Fig. 3-21(a).

Now let us imagine that Q is not 0 but 1, with R and S still 0. The upper gate has inputs of 0 and 1, and an output, $\bar{Q}$, of 0, which is fed back to the lower gate. This state, shown in Fig. 3-21(b), is also consistent. A state with both outputs equal to 0 is inconsistent, because it forces both gates to have two 0s as input, which, if true, would produce 1, not 0, as output. Similarly, it is impossible to have both outputs equal to 1, because that would force the inputs to 0 and 1, which yields 0, not 1. Our conclusion is simple: for $R = S = 0$, the latch has two stable states, which we will refer to as 0 and 1, depending on Q.

Now let us examine the effect of the inputs on the state of the latch. Suppose that S becomes 1 while $Q = 0$. The inputs to the upper gate are then 1 and 0, forcing the $\bar{Q}$ output to 0. This change makes both inputs to the lower gate 0, forcing the output to 1. Thus, setting S (i.e., making it 1) switches the state from 0 to 1. Setting R to 1 when the latch is in state 0 has no effect because the output of the lower NOR gate is 0 for inputs of 10 and inputs of 11.

Using similar reasoning, it is easy to see that setting S to 1 when in state $Q = 1$ has no effect but that setting R drives the latch to state $Q = 0$. In summary, when S is set to 1 momentarily, the latch ends up in state $Q = 1$, regardless of what state it was previously in. Likewise, setting R to 1 momentarily forces the latch to state $Q = 0$. The circuit "remembers" whether S or R was last on. Using this property, we can build computer memories.

Clocked SR Latches

It is often convenient to prevent the latch from changing state except at certain specified times. To achieve this goal, we modify the basic circuit slightly, as shown in Fig. 3-22, to get a **clocked SR latch**.

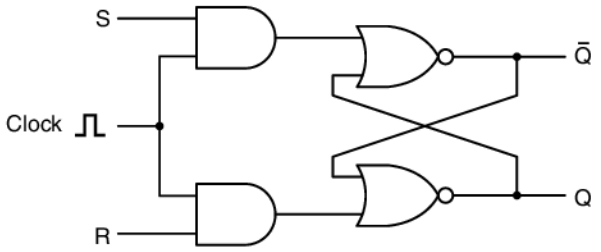


Figure 3-22. A clocked SR latch.

This circuit has an additional input, the clock, which is normally 0. With the clock 0, both AND gates output 0, independent of S and R, and the latch does not change state. When the clock is 1, the effect of the AND gates vanishes and the latch becomes sensitive to S and R. Despite its name, the clock signal need not be driven by a clock. The terms **enable** and **strobe** are also widely used to mean that the clock input is 1; that is, the circuit is sensitive to the state of S and R.

Up until now we have carefully swept under the rug the problem of what happens when both S and R are 1. And for good reason: the circuit becomes nondeterministic when both R and S finally return to 0. The only consistent state for $S = R = 1$ is $Q = \bar{Q} = 0$, but as soon as both inputs return to 0, the latch must jump to one of its two stable states. If either input drops back to 0 before the other, the one remaining 1 longest wins, because when just one input is 1, it forces the state. If both inputs return to 0 simultaneously (which is very unlikely), the latch jumps to one of its stable states at random.

Clocked D Latches

A good way to resolve the SR latch's instability (caused when $S = R = 1$) is to prevent it from occurring. Figure 3-23 gives a latch circuit with only one input, D. Because the input to the lower AND gate is always the complement of the input to the upper one, the problem of both inputs being 1 never arises. When $D = 1$ and the clock is 1, the latch is driven into state $Q = 1$. When $D = 0$ and the clock is 1, it is driven into state $Q = 0$. In other words, when the clock is 1, the current value of D is sampled and stored in the latch. This circuit, called a **clocked D latch**, is a true 1-bit memory. The value stored is always available at Q. To load the current value of D into the memory, a positive pulse is put on the clock line.

This circuit requires 11 transistors. More sophisticated (but less obvious) circuits can store 1 bit with as few as six transistors. In practice, such designs are normally used. This circuit can remain stable indefinitely as long as power (not shown) is applied. Later we will see memory circuits that quickly forget what state they are in unless constantly "reminded" somehow.

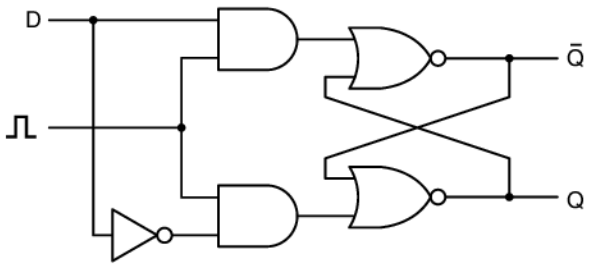


Figure 3-23. A clocked D latch.

3.3.2 Flip-Flops

In many circuits it is necessary to sample the value on a certain line at a particular instant in time and store it. In this variant, called a **flip-flop**, the state transition occurs not when the clock is 1 but during the clock transition from 0 to 1 (rising edge) or from 1 to 0 (falling edge) instead. Thus, the length of the clock pulse is unimportant, as long as the transitions occur fast.

For emphasis, we will repeat the difference between a flip-flop and a latch. A flip-flop is **edge triggered**, whereas a latch is **level triggered**. Be warned, however, that in the literature these terms are often confused. Many authors use "flip-flop" when they are referring to a latch, and vice versa.

There are various approaches to designing a flip-flop. For example, if there were some way to generate a very short pulse on the rising edge of the clock signal, that pulse could be fed into a D latch. There is, in fact, such a way, and the circuit for it is shown in Fig. 3-24(a).

At first glance, it might appear that the output of the AND gate would always be zero, since the AND of any signal with its inverse is zero, but the situation is a bit more subtle than that. The inverter has a small, but nonzero, propagation delay through it, and that delay is what makes the circuit work. Suppose that we measure the voltage at the four measuring points *a*, *b*, *c*, and *d*. The input signal, measured at *a*, is a long clock pulse, as shown in Fig. 3-24(b) on the bottom. The signal at *b* is shown above it. Notice that it is both inverted and delayed slightly, typically hundreds of picoseconds, depending on the kind of inverter used.

The signal at *c* is delayed, too, but only by the signal propagation time (at the speed of light). If the physical distance between *a* and *c* is, for example, 20 microns, then the propagation delay is 0.0001 nsec, which is certainly negligible compared to the time for the signal to propagate through the inverter. Thus, for all intents and purposes, the signal at *c* is as good as identical to the signal at *a*.

When the inputs to the AND gate, *b* and *c*, are ANDed together, the result is a short pulse, as shown in Fig. 3-24(b), where the width of the pulse, Δ, is equal to the gate delay of the inverter, typically 5 nsec or less. The output of the AND gate is just this pulse shifted by the delay of the AND gate, as shown at the top of

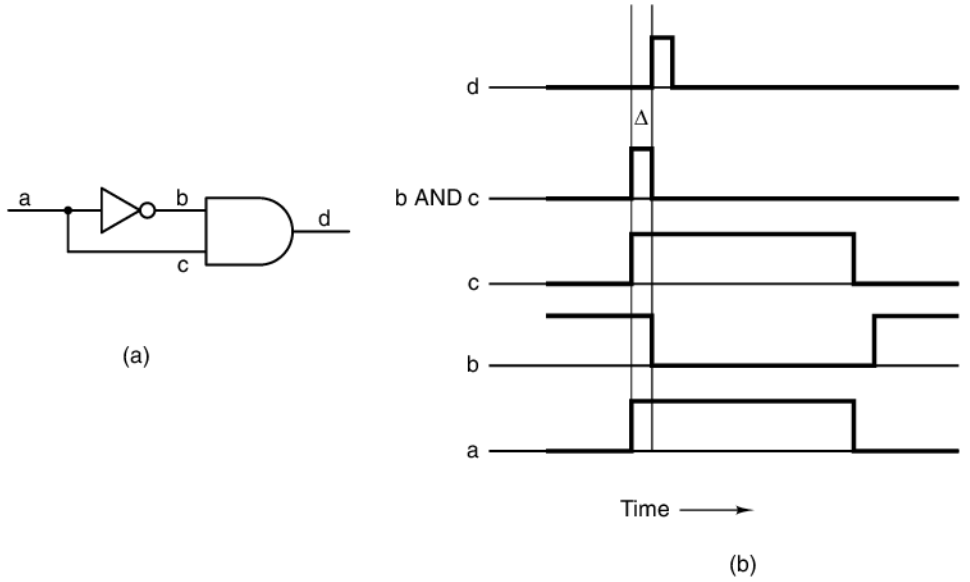


Figure 3-24. (a) A pulse generator. (b) Timing at four points in the circuit.

Fig. 3-24(b). This time shifting just means that the D latch will be activated at a fixed delay after the rising edge of the clock, but it has no effect on the pulse width. In a memory with a 10-nsec cycle time, a 1-nsec pulse telling it when to sample the *D* line may be short enough, in which case the full circuit can be that of Fig. 3-25. It is worth noting that this flip-flop design is nice because it is easy to understand, but in practice more sophisticated flip-flops are normally used.

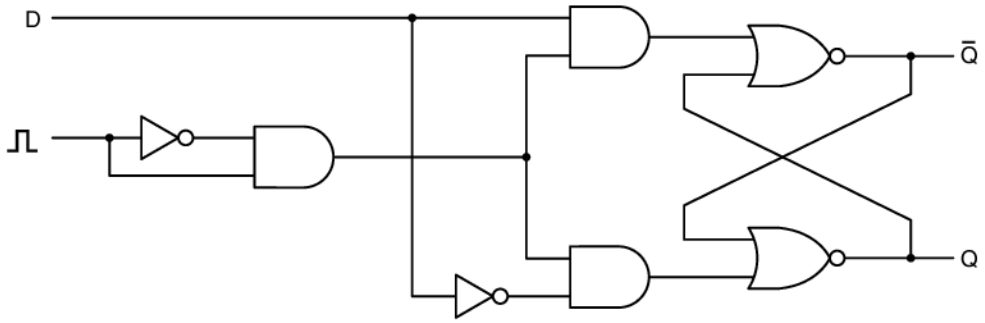


Figure 3-25. A D flip-flop.

The standard symbols for latches and flip-flops are shown in Fig. 3-26. Figure 3-26(a) is a latch whose state is loaded when the clock, *CK*, is 1. It is in contrast to Fig. 3-26(b) which is a latch whose clock is normally 1 but which drops to 0

momentarily to load the state from *D*. Figure 3-26(c) and (d) are flip-flops rather than latches, which is indicated by the pointy symbol on the clock inputs. Figure 3-26(c) changes state on the rising edge of the clock pulse (0-to-1 transition), whereas Fig. 3-26(d) changes state on the falling edge (1-to-0 transition). Many, but not all, latches and flip-flops also have $\bar{Q}$ as an output, and some have two additional inputs *Set* or *Preset* (force state to $Q = 1$) and *Reset* or *Clear* (force state to $Q = 0$).

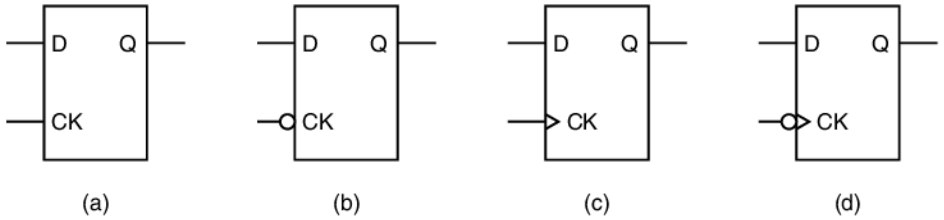


Figure 3-26. D latches and flip-flops.

3.3.3 Registers

Flip-flops can be combined in groups to create registers, which hold data types larger than 1 bit in length. The register in Fig. 3-27 shows how eight flip-flops can be ganged together to form an 8-bit storage register. The register accepts an 8-bit input value (*I*0 to *I*7) when the clock *CK* transitions. To implement a register, all the clock lines are connected to the same input signal *CK*, such that when the clock transitions, each register will accept the new 8-bit data value on the input bus. The flip-flops themselves are of the Fig. 3-26(d) type, but the inversion bubbles on the flip-flops are canceled by the inverter tied to the clock signal *CK*, such that the flip-flops are loaded on the rising transition of the clock. All eight clear signals are also ganged, so when the clear signal *CLR* goes to 0, all the flip-flops are forced to their 0 state. In case you are wondering why the clock signal *CK* is inverted at the input and then inverted again at each flip-flop, an input signal may not have enough current to drive all eight flip-flops; the input inverter is really being used as an amplifier.

Once we have designed an 8-bit register, we can use it as a building block to create larger registers. For example, a 32-bit register could be created by combining two 16-bit registers by tying their clock signals *CK* and clear signals *CLR*. We will look at registers and their uses more closely in Chap. 4.

3.3.4 Memory Organization

Although we have now progressed from the simple 1-bit memory of Fig. 3-23 to the 8-bit memory of Fig. 3-27, to build large memories a fairly different organization is required, one in which individual words can be addressed. A widely used

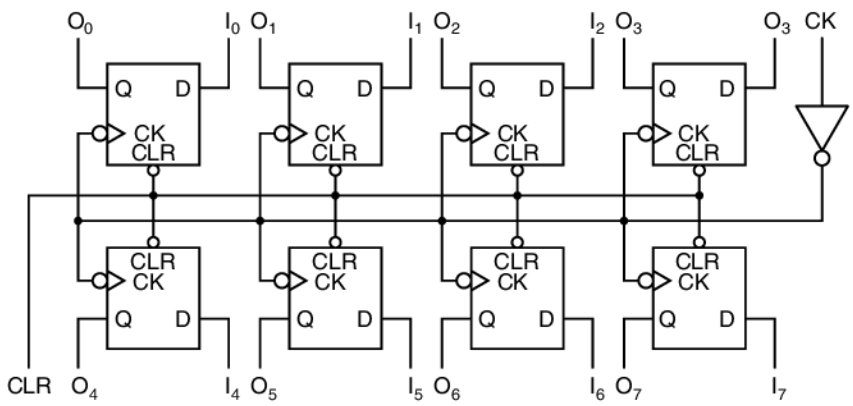


Figure 3-27. An 8-bit register constructed from single-bit flip-flops.

memory organization that meets this criterion is shown in Fig. 3-28. This example illustrates a memory with four 3-bit words. Each operation reads or writes a full 3-bit word. While the total memory capacity of 12 bits is hardly more than our octal flip-flop, it requires fewer pins and, most important, the design extends easily to large memories. Note the number of words is always a power of 2.

While the memory of Fig. 3-28 may look complicated at first, it is really quite simple due to its regular structure. It has eight input lines and three output lines. Three inputs are data: I_0, I_1, and I_2; two are for the address: A_0 and A_1; and three are for control: CS for Chip Select, RD for distinguishing between read and write, and OE for Output Enable. The three outputs are for data: O_0, O_1, and O_2. It is interesting to note that this 12-bit memory requires fewer signals than the previous 8-bit register. The 8-bit register requires 20 signals, including power and ground, while the 12-bit memory requires only 13 signals. The memory block requires fewer signals because, unlike the register, memory bits share an output signal. In this memory, 4 memory bits each share one output signal. The value of the address lines determine which of the 4 memory bits is allowed to input or output a value.

To select this memory block, external logic must set CS high and also set RD high (logical 1) for read and low (logical 0) for write. The two address lines must be set to indicate which of the four 3-bit words is to be read or written. For a read operation, the data input lines are not used, but the word selected is placed on the data output lines. For a write operation, the bits present on the data input lines are loaded into the selected memory word; the data output lines are not used.

Now let us look at Fig. 3-28 closely to see how it works. The four word-select AND gates at the left of the memory form a decoder. The input inverters have been placed so that each gate is enabled (output is high) by a different address. Each gate drives a word select line, from top to bottom, for words 0, 1, 2, and 3. When the chip has been selected for a write, the vertical line labeled CS $\cdot$ $\overline{\text{RD}}$ will be high,